CommCareHQ Documentation

Release 1.0

Dimagi

Jan 30, 2024

OVERVIEW

CommCare is a multi-tier mobile, server, and messaging platform. The platform enables users to build and configure content and a user interface, deploy that application to Android devices or to an end-user-facing web interface for data entry, and receive that data back in real time. In addition, content may be defined that leverages bi-directional messaging to end-users via API interfaces to SMS gateways, e-mail systems, or other messaging services. The system uses multiple persistence mechanisms, analytical frameworks, and open source libraries.

Data on CommCare mobile is stored encrypted-at-rest (symmetric AES256) by keys that are secured by the mobile user's password. User data is never written to disk unencrypted, and the keys are only ever held in memory, so if a device is turned off or logged out the data is locally irretrievable without the user's password. Data is transmitted from the phone to the server (and vis-a-versa) over a secure and encrypted HTTPS channel.

Contents:

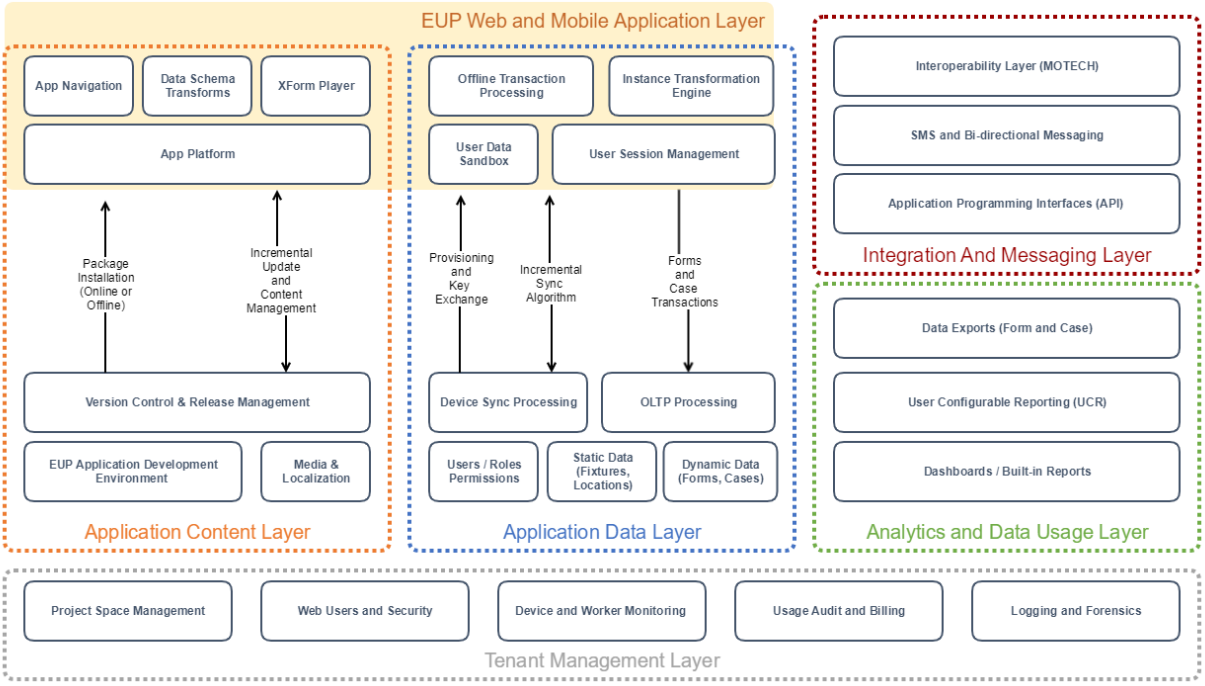
CHAPTER

ONE

COMMCARE HQ PLATFORM OVERVIEW

The major functional components are:

- Application Building and Content Management
- Application Data Layer
- Tenant Management
- Analytics and Usage
- Messaging Layer
- Integration



1.1 Application Content Layer

1.1.1 Application Building and Deployment Management

The Application Builder provides an interface for users to create and structure an application's content and workflow. Questions can be added by type (text, integer, multiple answer, date, etc.) and logic conditions can be applied to determine whether the question should be displayed or if the answer is valid.

This environment also provides critical support for detailed management of content releases. CommCare's deployment management provides a staging-to-deploy pipeline, profile-based releases for different regions, and supports incremental rollout and distribution for different regions.

1.1.2 Android Mobile App Runner and Web App Engine

Applications developed in the end user programming (EUP) content builder are deployed to users and then executed within the CommCare application engine, which is built on a shared Java codebase. The application configurations can be run on both a native Android client and a Spring web client, to allow access for users in the field as well as those accessing the application from a computer on the web.

1.2 Application Data Layer

1.2.1 Data Management

There are two data models that underpin the CommCare data model:

Form A form is the basic building block of Applications. Forms are represented as XForms (XML Forms) which contain data, logic and rules. Users interact with forms on the mobile device to capture data and perform logic. This data is then sent back to CommCare as a *form submission* which is an XML document containing only the data portion of the XForm.

Forms may include *case blocks* which can be used to create, update and close cases.

Case Cases are used to track interactions with objects, often people. Cases provide longitudinal records which can track the ongoing interactions with a case through form submissions and facilitate the complex sharding and reconciliation required from synchronizing offline clients.

Each case has a type, such as "patient", "contact", "household" which distinguishes it from cases of other types. Cases may also be structured in a hierarchy using uni-directional relationships between cases.

The full specification for cases can be found here.

1.2.2 Transaction Processing

CommCare provides a transaction processing layer which acts as the first step in the underlying data and storage pipeline. This layer manages the horizontal workload of the mobile and web applications submitting forms, which are archived into a chunked object storage, and extracts the transactional 'case' logic which is used to facilitate data synchronization through more live storage in the table based storage layer. The transaction processor then appropriately queues transactions into the real time data pipeline for processing into the reporting databases through the Kakfa Change Feed, or triggering asynchronous business rules in the Celery queue.

The data processing service is flexible to store any content sent or received via mobile form submissions or SMS services as long as it adheres to the XForms specification. It also saves all logging and auditing information necessary

for data security compliance. The data processing service saves all data at the transactional level so that histories can be audited and reconstructed if necessary.

1.2.3 Synchronization

The synchronization process allows for case and user data to be kept up-to-date through incremental syncs of information from the backend server for offline use cases. To ensure consistency, the backend keeps a shadow record of each user's application state hashed to a minimal format, when users submit data or request synchronization, this shadow record hash is kept up to date to identify issues with what local data is on device.

Syncs request a diff from the server by providing their current hashed state and shadow record token. The server then establishes what cases have been manipulated outside of the local device's storage (along with reports or other static data) which may be relevant to the user, such as a new beneficiary or household registered in their region. After all of those cases are established, the server produces an XML payload similar to the ones generated by filling out forms on the local device, which is used to update local device storage with the new data.

1.3 Tenant Management Layer

1.3.1 Project Spaces

Every project has its own site on CommCare HQ. Project spaces can house one, or more than one inter-related applications. Data is not shared among project spaces.

Content can be centrally managed with a master project space housing a master application that can be replicated in an unlimited number of additional project spaces. CommCare enables fine grained release management along with roll-back that can be controlled from each project space. These project spaces can be managed under an Enterprise Subscription that enables centralized control and administration of the project spaces.

1.3.2 User Management

There are two main user types in CommCare: Project Users and Application Users.

Project Users are meant to view data, edit data, manage exports, integrations, and application content. Project Users can belong to one or more project spaces and are able to transition between project spaces without needing to login/logout by simply selecting from a drop-down.

Application Users are expected to primarily use CommCare as an end-user entering data and driving workflows through an application.

Project Users and Application Users are stored with separate models. These models include all permission and project space membership information, as well as some metadata about the user such as their email address, phone number, etc. Additionally, authentication stubs are synchronized in real time to SQL where they are saved as Django Users, allowing us to use standard Django authentication, as well as Django Digest, a third-party Django package for supporting HTTP Digest Authentication.

1.3.3 Device and Worker Monitoring

Mobile devices which are connected to the CommCare server communicate maintenance and status information through a lightweight HTTP 'heartbeat' layer, which receives up-to-date information from devices like form throughput and application health, and can transmit back operational codes for maintenance operations, allowing for remote management of the application directly outside of a full-fledged MDM.

1.4 Analytics and Usage

There are several standard reports available in CommCare. The set of standard reports available are organized into four categories: Monitor Workers, Inspect Data, Messaging Reports and Manage Deployments.

Monitor Workers

Includes reports that allow you to view and compare activity and performance of end workers against each other.

Inspect Data

Reports for finding and viewing in detail individual cases and form submissions.

Messaging Reports

Domains that leverage CommCare HQ's messaging capabilities have an additional reporting section for tracking SMS messages sent and received through their domain

Manage Deployments

Provides tools for looking at applications deployed to users' phones and device logging information.

1.4.1 User Defined Reports

In addition to the set of standard reports users may also configure reports based on the data collected by their users. This reporting framework allows users to define User Configurable Reports (UCR) which store their data in SQL tables.

1.4.2 Mobile Reports

UCRs may also be used to send report data to the mobile devices. This data can then be displayed on the device as a report or graph.

1.5 Messaging Layer

CommCare Messaging integrates with a SMS gateway purchased and maintained by the client as the processing layer for SMS messages. This layer manages the pipeline from a Case transaction to matching business logic rules to message scheduling and validation.

1.5.1 Conditional Scheduled Messages

Every time a case is created, updated, or closed in a form it is placed on the asynchronous processing queue. Asynchronous processors review any relevant business logic rules to review whether the case has become (or is no longer) eligible for the rule, and schedules a localized message which can contain information relevant to the case, such as an individual who did not receive a scheduled visit.

1.5.2 Broadcast Messages

Broadcast messaging is used to send ad-hoc messages to users or cases. These messages can either be sent immediately, or at a later date and time, and can also be configured to send to groups of users in the system.

1.5.3 Gateway Connectivity and Configuration, Logging, and Audit Tracking

All SMS traffic (inbound and outbound) is logged in the CommCare Message Log, which is also available as a report. In addition to tracking the timestamp, content, and contact the message was associated with, the Message Log also tracks the SMS backend that was used and the workflow that the SMS was a part of (broadcast message, reminder, or keyword interaction).

The messaging layer is also used to provide limits and controls on messaging volume, restricting the number of messages which can be sent in a 24hr period, and restricting the time of day which messages will be sent, to comply with regulations. These restrictions may apply to both ad-hoc and scheduled messages. Messages are still processed and queued 24hrs per day, but only submitted when permitted.

1.5.4 Messaging Dashboards

Charts and other kinds of visualizations are useful for getting a general overview of the data in your system. The dashboards in CommCare display various graphs that depict case, user, and SMS activity over time. These graphs provide visibility into when new cases and users were created, how many SMS messages are being sent daily, and the breakdown of what those messages were used for (reminders, broadcasts, etc.).

1.6 Integration

CommCare has robust APIs as well as a MOTECH integration engine that is embedded in CommCare. APIs allow for direct programmatic access to CommCare. The MOTECH integration engine allows for custom business rules to be implemented that allow for real-time or batch integration with external systems. This engine does not have an application or content management environment, and so requires custom engineering to be added to a CommCare instance.

1.6.1 APIs

CommCare has extensive APIs to get data in and out for bidirectional integration with other systems. This method of data integration requires familiarity with RESTful HTTP conventions, such as GET and POST and url parameters.

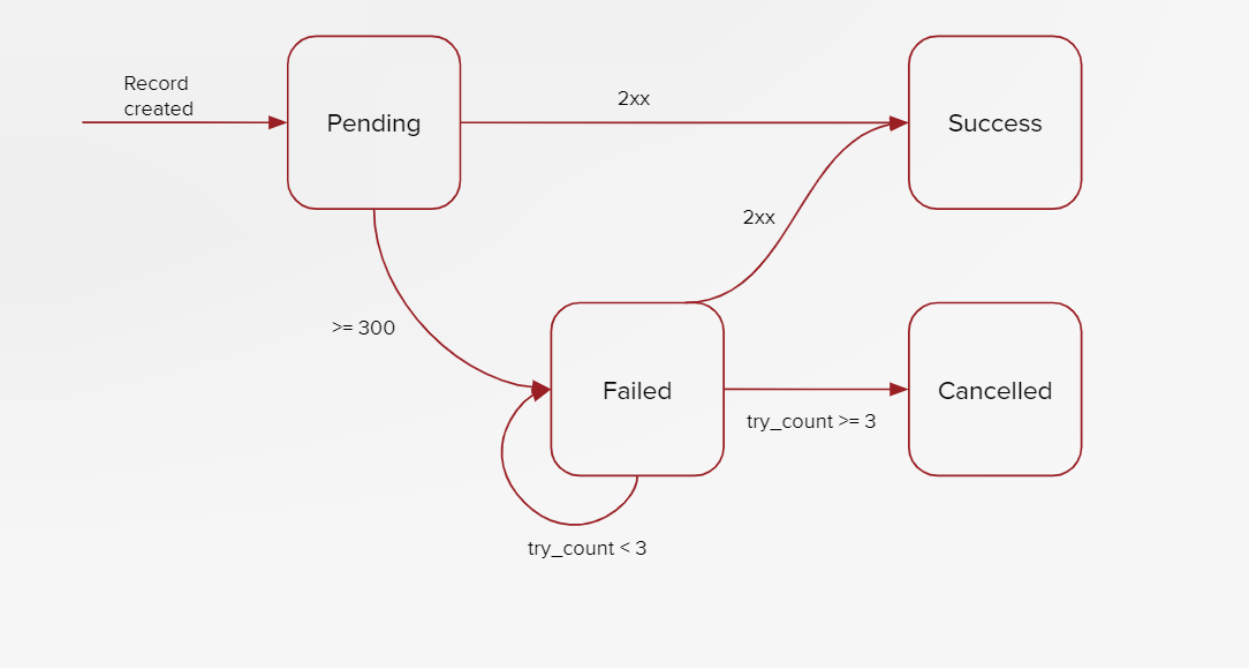
There are APIs both for reading and writing data to CommCare. This can be updated data related to forms or cases in the system and enable highly-sophisticated integrations with CommCare.

More details on CommCare's API can be found in the API documentation.

1.6.2 MOTECH Repeaters

For interoperability with external systems which process transactional data, CommCare has a MOTECH repeater layer, which manages the pipeline of case and form transactions received and manages the lifecycle of secure outbound messages to external systems.

This architecture is designed to autonomously support the scale and volume of transactional data up to hundreds of millions of transactions in a 24hr period.



New transformation code for this subsystem can be authored as Python code modules for each outbound integration. These modules can independently transform the transactional data for the repeater layer, or rely on other data from the application layer when needed by integration requirements.

CHAPTER

TWO

COMMCARE ARCHITECTURE OVERVIEW

2.1 CommCare Backend Services

The majority of the code runs inside the server process. This contains all of the data models and services that power the CommCare website.

Each module is a collection of one or more Django applications that each contain the relevant data models, url mappings and view controllers, templates, and Database views necessary to provide that module's functionality.

2.2 Data flow for forms and cases

CommCare deals with many different types of data but the primary data that is generated by users form and case data. This data most often comes from a mobile device running the CommCare mobile application. It may also come from Web Apps, via an integration API or from a case import.

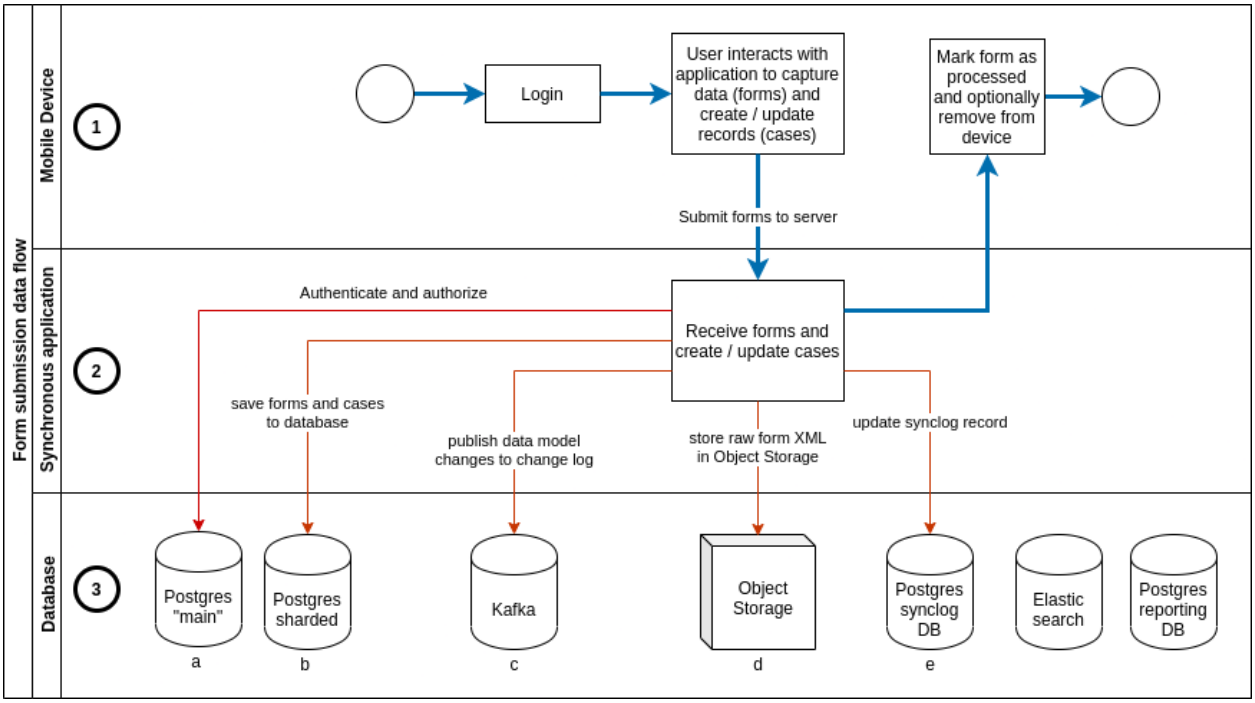
The data processing for form and case data is split into two portions, synchronous processing and asynchronous processing.

2.2.1 Synchronous processing

Form and case data in CommCare always follows the same pathway through the system regardless of whether the data originated from a mobile device (as in the diagram below), from Web Apps, an external integration or from an internal process such as case update rules.

In all instances a form (which may contain zero or more case changes) is received by CommCare and processed synchronously and atomically to the point of persisting it in the primary databases and recording the change in the change log.

The diagram below shows this synchronous processing:

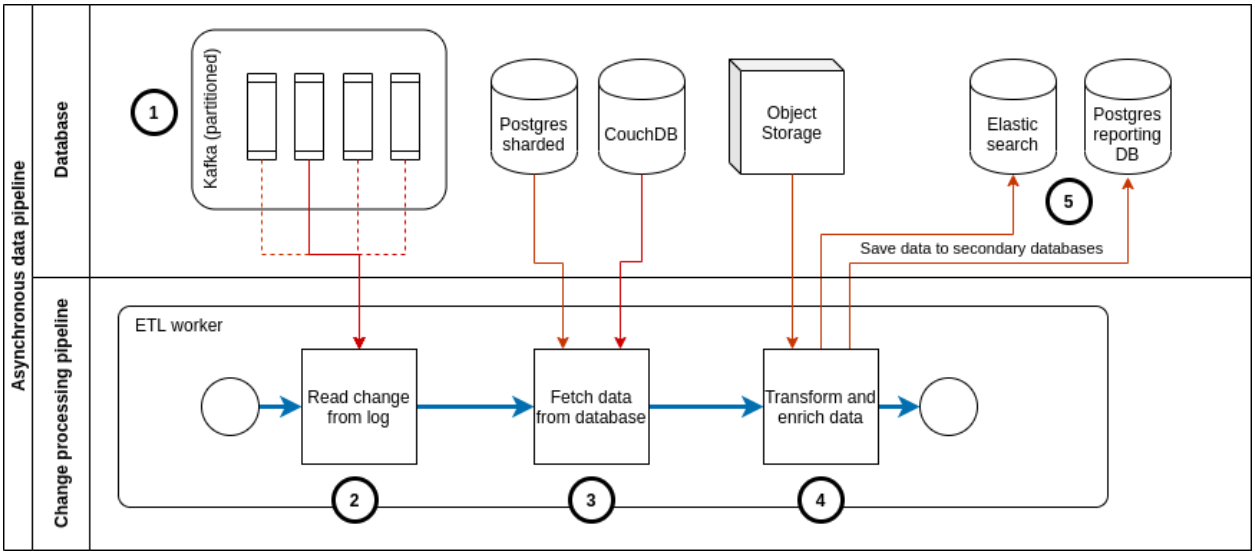


1. A form is created on a mobile device (this could also be Web Apps, an external system).
2. The form is received by CommCare and once the request has been authenticated and authorized it is fully processed before responding with a success or error response.
3. The data is persisted to the various backend data sources.

1. This SQL database is used for authentication and authorization.
2. A set of partitioned SQL databases form the primary data store for form and case data.
3. Once processing is complete a metadata record is published to the change log for each data model that was created or updated.
4. The raw form XML is persisted to object storage (only metadata about the form is saved to the primary database in (b).
5. If this submission is from a mobile device the sync record for the device is updated to allow efficient synchronization when the mobile device next makes a sync request.
4. A successful response is sent to the sender.

2.2.2 Asynchronous data pipeline

Separately from the synchronous workflow described above there is a data processing pipeline which runs asynchronously. This pipeline is responsible for populating the secondary databases which are used for reporting and as the datasource for some of the APIs that CommCare offers.



1. Kafka stores the metadata about data model changes. Kafka partitions the data based on the data model ID (case ID / form ID) and stores each partition separately. Data is sent to Kafka during the synchronous request processing as described above.
2. A pool of ETL workers (a.k.a. pillows) subscribe to Kafka and receive the metadata records from the partitions they are subscribed to.

a. Each ETL worker subscribes to a unique set of partitions.

b. Since each worker is independent of the others the rate of processing can vary between workers or can get delayed due to errors or other external factors.

c. The impact of this is that data liveness in the secondary database may vary based on the specific components in the pipeline. I.e. two cases which got updated in the same form may be updated in Elasticsearch at different times due to variations in the processing delay between pillow workers.

3. The ETL workers fetch the data record from the primary database.

a. For forms and cases this data comes from PostgreSQL

b. For users and applications this data comes from CouchDB

4. If the data model is a form then the form XML is also retrieved from object storage. This data together with the record from the primary database are used to produce the final output which is written to the secondary databases.

a. In the case of UCRs there may be other data that is fetched during the processing stage e.g. locations, users.

2.3 Change Processors (Pillows)

Change processors (known in the codebase as pillows) are events that trigger when changes are introduced to the database. CommCare has a suite of tools that listen for new database changes and do additional processing based on those changes. These include the analytics engines, as well as secondary search indices and custom report utilities. All change processors run in independent threads in a separate process from the server process, and are powered by Apache Kafka.

2.4 Task Queue

The task queue is used for asynchronous work and periodic tasks. Processes that require a long time and significant computational resources to run are put into the task queue for asynchronous processing. These include data exports, bulk edit operations, and email services. In addition the task queue is used to provide periodic or scheduled functionality, including SMS reminders, scheduled reports, and data forwarding services. The task queue is powered by Celery, an open-source, distributed task queueing framework.

2.5 Data Storage Layer

CommCare HQ leverages the following databases for its persistence layer.

2.5.1 PostgreSQL

A large portion of our data is stored in the PostgreSQL database, including case data, form metadata, and user account information.

Also stored in a relational database, are tables of domain-specific transactional reporting data. For a particular reporting need, our User Configurable Reporting framework (UCR) stores a table where each row contains the relevant indicators as well as any values necessary for filtering.

For larger deployments the PostgreSQL database is sharded. Our primary data is sharded using a library called PL/Proxy as well as application logic written in the Python.

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.

See *Configuring SQL Databases in CommCare*

2.5.2 CouchDB

CommCare uses CouchDB as the primary data store for some of its data models, including the application builder metadata and models around multitenancy like domains and user permissions. CouchDB is an open source database designed to be used in web applications. In legacy systems CouchDB was also used to store forms, cases, and SMS records, though these models have moved to PostgreSQL in recent applications.

CouchDB was primarily chosen because it is completely schema-less. All data is stored as JSON documents and views are written to index into the documents to provide fast map-reduce-style querying.

In addition CommCare leverages the CouchDB changes feed heavily to do asynchronous and post processing of our data. This is outlined more fully in the "change processors" section above.

2.5.3 Elasticsearch

Elasticsearch is a flexible and powerful open source, distributed real-time search and analytics engine for the cloud. CommCare uses Elasticsearch for several distinct purposes:

Much of CommCare's data is defined by users in the application configuration. In order to provide performant reporting and querying of user data CommCare makes use of Elasticsearch.

CommCare also serves portions of the REST API from a read-only copy of form and case data that is replicated in real time to an Elasticsearch service.

This also allows independent scaling of the transactional data services and the reporting services.

2.6 Devops Automation

2.6.1 Fabric / Ansible

Fabric and Ansible are deployment automation tools which support the efficient management of cloud resources for operations like deploying new code, rolling out new server hosts, or running maintenance processes like logically resharding distributed database. CommCare uses these tools as the foundation for our cloud management toolkit, which allows us to have predictable and consistent maintenance across a large datacenter.

Dimagi's tool suite, commcare-cloud is also available on Github

2.7 Other services

2.7.1 Nginx (proxy)

CommCare's main entry point for all traffic to CommCare HQ goes through Nginx. Nginx performs the following functions:

- SSL termination
- Reverse proxy and load balancing
- Request routing to CommCare and Formplayer
- Serving static assets
- Request caching
- Rate limiting (optional)

2.7.2 Redis

Redis is an open source document store that is used for caching in CommCare HQ. Its primary use is for general caching of data that otherwise would require a query to the database to speed up the performance of the site. Redis also is used as a temporary data storage of large binary files for caching export files, image dumps, and other large downloads.

2.7.3 Apache Kafka

Kafka is a distributed streaming platform used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, fast, and runs in production in thousands of companies. It is used in CommCare to create asynchronous feeds that power our change processors (pillows) as part of the reporting pipeline.

2.7.4 RabbitMQ

RabbitMQ is an open source Advanced Message Queuing Protocol (AMQP) compliant server. As mentioned above CommCare uses the Celery framework to execute background tasks. The Celery task queues are managed by RabbitMQ.

2.7.5 Gunicorn

Gunicorn is an out-of-the-box multithreaded HTTP server for Python, including good integration with Django. It allows CommCare to run a number of worker processes on each worker machine with very little additional setup. CommCare is also using a configuration option that allows each worker process to handle multiple requests at a time using the popular event-based concurrency library Gevent. On each worker machine, Gunicorn abstracts the concurrency and exposes our Django application on a single port. After deciding upon a machine through its load balancer, our proxy is then able to forward traffic to this machine's port as if forwarding to a naive single-threaded implementation such as Django's built-in "runserver".

CHAPTER

THREE

COMMCARE ENHANCEMENT PROPOSAL PROCESS

This process outlines a mechanism for proposing changes to CommCare HQ. The process is intentionally very lightweight and is not intended as a gateway that must be passed through. The main goal of the process is to communicate intended changes or additions to CommCare HQ and facilitate discussion around those changes.

The CommCare Enhancement Proposal (CEP) process is somewhat analogous to the Request For Comments process though much simpler:

1. Create a CEP

   Create a Github Issue using the CEP template. Once you have completed the template submit the issue an notify relevant team members or **@dimagi/dimagi-dev**.

2. Respond to any questions or comments that arise

CHAPTER

FOUR

APPLICATION TERMINOLOGY

4.1 Applications (and builds)

An application typically has many different `Application` documents: one for the current/primary/canonical application, plus one for each saved build. The current app's id is what you'll see in the URL on most app manager pages. Most pages will redirect to the current app if given the id of an older build, since saved builds are essentially read-only.

In saved builds, `copy_of` contains the primary app's id, while it's `None` for the primary app itself. If you need to be flexible about finding primary app's id on an object that might be either an app or a build, use the property `origin_id`.

Within code, "build" should always refer to a saved build, but "app" is used for both the current app and saved builds. The ambiguity of "app" is occasionally a source of bugs.

Every time an app is saved, the primary doc's `version` is incremented. Builds have the `version` from which they were created, which is never updated, even when a build doc is saved (e.g., the build is released or its build comment is updated).

When a user makes a build of an application, a copy of the primary application document is made. These documents are the "versions" you see on the deploy page. Each build document will have a different id, and the `copy_of` field will be set to the ID of the primary application document. Both builds and primary docs contain `built_on` and `built_with` information - for a primary app doc, these fields will match those of the most recent build. Additionally, a build creates attachments such as `profile.xml` and `suite.xml` and saves then to the build doc (see `create_all_files`).

When a build is released, its `is_released` attribute will be set to `True`. `is_released` is always false for primary application docs.

4.2 Modules

An application contains one or more modules, which are called "menus" in user-facing text. These modules roughly map to menus when using the app on a device. In apps that use case management, each module is associated with a case type.

Each module has a `unique_id` which is guaranteed unique only within the application.

4.3 Forms

A "form" in HQ may refer to either a form *definition* within an application or a form *submission* containing data. App manager code typically deals with form definitions.

A module contains one or more form definitions. Forms, at their most basic, are collections of questions. Forms also trigger case changes and can be configured in a variety of ways.

Each form has a `unique_id` which is guaranteed unique only within the domain. For the most part, being unique within an application is sufficient, but uniqueness across the domain is required for the sake of multimaster linked applications. See this commit for detail.

Forms also have an xml namespace, abbreviated `xmlns`, which is part of the form's XML definition. Reports match form submissions to form definitions using the xmlns plus the app id, which most apps pass along to secure_post. For reports to identify forms accurately, xmlns must be unique within an app.

Duplicate xmlnses in an app will throw an error when a new version of the app is built. When an app is copied, each form in the copy keeps the same XMLNS as the corresponding form in the original. When a form is copied within an app - or when a user uploads XML using an xmlns already in use by another form in the same app - the new form's xmlns will be set to a new value in save_xform.

4.3.1 Exceptions

Linked apps use similar workflows to app copy for creating and pulling. See docs for more detail on how they handle form unique ids and xmlnses.

Shadow forms are a variation of advanced forms that "shadow" another form's XML but can have their own settings and actions. Because they don't have their own XML, shadow forms do not have an xmlns but instead inherit their source form's xmlns. In reports, submissions from shadow forms show up as coming from their source form.

CHAPTER

FIVE

FEATURES

5.1 Template apps

HQ currently has two versions of "template apps."

5.1.1 Onboarding apps

The first set of template apps are simple apps in different sectors that we've experimented with adding to a user's project when they first sign up for an account. These template apps are stored as json in the code base, in the template_apps directory. They are imported using the view `app_from_template`.

5.1.2 COVID app library

This is a set of template applications that exist in a real project space and are made publicly visible to other domains. A record of each app is stored as an `ExchangeApplication` model.

These applications are visible via the `app_exchange` view.

To add a new app, add a new `ExchangeApplication` model via django admin. You must supply a domain and an app id. Use the "canonical" app id used in app manager URLs, not a specific build id. You may also provide a help link and/or a link to a version history. All released versions of the app will be available via the app library, with the versions labeled by date of release, *not* by version number. The application title displayed in the library will be from the latest version of the app.

CHAPTER

SIX

BULK APPLICATION TRANSLATIONS

HQ supports a file download and re-upload to update all application-specific translations.

The download has two variants, a multi-sheet and a single-sheet format. Both are tolerant of partial uploads:

missing sheets, missing language columns (as opposed to the columns needed to identify a row), and missing rows

(with some exceptions for module translations, which depend on case properties being present and correctly ordered).

The default multi-sheet format contains a first "menus and forms" sheet for editing module and form names and menu media. It then contains a sheet for each module and each form.

The single-sheet format allows editing all of the same content, just with all rows in the same sheet. Depending on the type of row (module/form name, module content, or form content) certain columns will be blank.

The UI defaults to the multi-sheet download. There's a feature flagged ability to select a single language, which downloads the single sheet format for that language. There's no technical reason that the single sheet download is also a single language download.

For domains with Transifex integration, Transifex generates Excel files in the multi-sheet format that HQ accepts.

Code is organized into - `download.py` Generation of Excel downloads - `upload_app.py` Entry point to upload code - `upload_module.py` Helper functions to update module content - `upload_form.py` Helper functions to update form content - `utils.py` Helper functions shared by download and upload, such as header generators

CHAPTER

SEVEN

MULTIMEDIA

7.1 General multimedia handling

Multimedia metadata is stored in couch, using `CommCareMultimedia` and its subclasses: `CommCareImage`, `CommCareAudio`, etc.

Each file is stored once, regardless of how many applications or domains use it. This is enforced via the by_hash view, which stores a hash of each multimedia file's content. When new multimedia is uploaded, the contents are hashed and looked up to determine wheether or not to create a new document. See `CommCareMultimedia.get_by_data` and its calling code in places like `BaseProcessFileUploadView.process_upload`.

These documents are never deleted.

7.2 Multimedia in applications

7.2.1 The app's multimedia_map

The biggest use case for multimedia in CommCare is displaying multimedia in applications. The application's `multimedia_map` property stores information about where it uses multimedia.

`multimedia_map` is a dict where each key is a path and each value is a dict corresponding to an `HQMediaMapItem`. The map has one entry for every path, also called a **reference**, in the application. If an application uses the same image as an icon for two forms, it will have two entries in the map, *unless* a user has edited the app so that both forms use the same path.

Sample multimedia_map entry:

```
"jr://file/commcare/image/module0_en.jpg": {
    "doc_type": "HQMediaMapItem",
    "media_type": "CommCareImage",
    "multimedia_id": "7f14f2dc29fa7c406dc1b0d40603e742",
    "unique_id": "b6ee70e5e61ea550b26d36cc185dde23",
    "version": 17
}
```

The **path** is auto-generated by HQ based on where the file is being used. It can be manually edited - everywhere that a multimedia file can be uploaded, its path can be edited. There is also a tool to edit multimedia paths in bulk via Excel.

The **doc_type** is always `HQMediaMapItem`.

The **media_type** is `CommCareMultimedia` or one of its subclasses.

The **multimedia_id** is the id of the corresponding `CommCareMultimedia` document.

The **unique_id** is a hash of the concatenated multimedia id and path at the time the item was created. That means this id will be unique within the multimedia map, but will not be globally unique.

The **version** is the app version where the multimedia was added or most recently changed. The version can be `None` for applications that are unbuilt, and then any blank versions will be assigned when a new build is made. More on this below.

Versioning multimedia

CommCare (on mobile) uses the version to determine when to re-download an updated multimedia file. The version will always be an app version. This correspondence isn't necessary; all the phone checks is if the current version is greater than the version already on the phone - if so, the phone will re-download the multimedia. This logic is based on the logic used to download updated forms.

The version is updated (set to the latest app version) in the following situations:

- Multimedia is added to the app.
- The content of multimedia is changed, by using the "Replace" button in the uploader. This also causes the multimedia id to change. It typically does not change the unique*id, because *set*media*versions* looks up the path in the previous version of the app and uses that item's unique id. However, it may change the unique*id for form question media, because form questions include a random string at the end of their path which changes when a user uploads a new file.
- The path alone is changed using the "Manage Multimedia Paths" bulk uploader. This does not change the unique id, even though the path changes.
- The path and content are changed together in the UI. This also replaces the multimedia id and re-generates the unique id.

Linked apps

Linked apps, when pulled, copy the multimedia map directly from the upstream app, so all the attributes of each item will match those in the upstream app. Because linked apps are always pulled from a released version of an upstream app, each item should have a version set.

7.2.2 media_suite.xml

The media suite contains a list of `media` elements, one for each value in the multimedia map.

```
<media path="../../commcare/image">
    <resource descriptor="Image File: FILENAME" id="media-UNIQUE_ID-FILENAME" version=
↪"VERSION">
        <location authority="local">PATH</location>
        <location authority="remote">
            URL
        </location>
    </resource>
</media>
```

- FILENAME is originally generated by HQ and will be fairly readable, something like `module0_en.jpg`, and may be modified by users.
- PATH will be the path to the file on the device, so similar to the path used as a key in the multimedia map, though in a format like `./commcare/image/module0*en.jpg` rather than the `jr://commcare/image/module0*en.jpg` format used in the map.

- UNIQUE_ID and VERSION are from the multimedia map.
- URL is the url to access the file on the server and incorporates the media type, multimedia id, and filename, e.g., `http://www.commcarehq.org/hq/multimedia/file/CommCareImage/7f14f2dc29fa7c406dc1b0d40603e742/module0_en.jpg`.

If the same file is uploaded to two different paths, it will be included twice on the device, but on the server it will be stored only once and will have only one `CommCareMultimedia` document. Although there will be two different URLs in the media suite, they will point to the same page: HQ only uses the multimedia id to identify the file, it ignores the URL's filename suffix.

CHAPTER

EIGHT

ADDING A NEW COMMCARE SETTING

A new setting can be defined in commcare-app-settings.yml or commcare-profile-settings.yml depending on whether the setting is HQ only or should also go to mobile xml files. The spec for setting is given in below section.

CHAPTER

NINE

COMMCARE SETTINGS CONFIG SPEC

This page documents the YAML configuration found in these locations:

- commcare-app-settings.yml: Settings that are specific to CommCare HQ's configurations
- commcare-profile-settings.yml: Settings that are 1-to-1 with CommCare mobile profile features/properties
- commcare-settings-layout.yml: Determines how these settings are grouped and laid out on the app settings page

Each of `commcare-app-settings.yml` and `commcare-profile-settings.yml` contain a yaml list with each element containing the following properties:

9.1 Required properties

- `id` - The "`key`" attribute to be used in the CommCare profile (or "feature" name)
- `name` - Human readable name of the property
- `description` - A longer human readable description of what the property does
- `default` - The default value for the property
- `values` - All the possible values for the property
- `value_names` - The human readable names corresponding to the `values`

9.2 Optional

- `requires` - Should be set if this property is only enabled when another property has a certain value. Syntax is `"{SCOPE.PROPERTY}='VALUE'"`, where `SCOPE` can be `hq`, `properties`, `features`, or `$parent`.
- `requires_txt` - Optional text explaining the dependency enforced by `requires`
- `contingent_default` - What value to force this property to if it's disabled. E.g. `[{"condition": "{features.sense}='true'", "value": "cc-su-auto"}]`, means "if the feature `sense` is `'true'`, then this property should be forced to take the value `"cc-su-auto"`.
- `since` - The CommCare version in which this property was introduced. E.g. `2.1`.
- `type` - Less common. To render as a "feature" set this to `"features"`.
- `commcare_default` - The default used by CommCare, if it differs from the default we want it to have on HQ.
- `disabled_default` - The default to be used if the app's build version is less than the `since` parameter. `contingent_default` takes precedence over this setting.
- `values_txt` - Extra help text describing what values can be entered

- `group` - Presentational; defines how the properties get grouped on HQ
- `disabled` - Set to `true` for deprecated values we don't want to show up in the UI anymore
- `force` - Set to `true` to have the `force` attribute of the setting set when building the profile. Only applies when type='properties' (default).
- `toggle` - If specified, the property will only be shown if the given toggle is enabled. The value should be an identifier for a toggle in `corehq/toggles.py`, e.g. "CUSTOM_APP_BASE_URL"
- `warning` - Text displayed if the value of the setting is invalid

Only static setting options can be defined in settings yaml files, any app or domain specific context to render the setting on HQ Edit Application settings page can be provided in `corehq.apps.app_manager.view.apps:get_app_view_context`

9.3 Example

```
- name: "Auto Update Frequency"
  description: "How often CommCare mobile should attempt to check for a new, released␣
↪application version."
  id: "cc-autoup-freq"
  values: ["freq-never", "freq-daily", "freq-weekly"]
  value_names: ["Never", "Daily", "Weekly"]
  default: "freq-never"
  values_txt: "After login, the application will look at the profile's defined reference␣
↪for the authoritative location of the newest version. This check will occur with some␣
↪periodicity since the last successful check based on this property. freq-never␣
↪disables the automatic check."
  since: "1.3"
```

CHAPTER

TEN

APP NAVIGATION FEATURES

Navigation in CommCare is oriented around form entry. The goal of each CommCare session is to complete a form. Menus and case lists are tools for gathering the necessary data required to enter a particular form. App manager gives app builders direct control over many parts of their app's UI, but the exact sequence of screens a user sees is *indirectly* configured.

Based on app configuration, HQ builds a `suite.xml` file that acts as a blueprint for the app. It outlines what forms are available to fill out, where they fit, and what data (like cases) they will need to function. CommCare interprets this configuration to decide which screens to show to the user and in what order.

Much of the complexity of app manager code, and of building apps, comes from inferences HQ makes while building the suite file, especially around determining the set of data required for each form. These features that influence, but don't directly control, the suite also influence each other, in ways that may not be obvious. The following features are particularly prone to interact unexpectedly and should be tested together when any significant change is made to any of them:

1. Display Only Forms
2. Select Parent First
3. End of Form Navigation and Form Linking
4. Child Modules
5. Shadow Modules

Several of these features are simple from an app builder's perspective, but they require HQ to "translate" UI concepts into suite concepts. Other features force the app builder to understand suite concepts, so they may be challenging for app builders to learn but are less prone to interacting poorly with other features:

1. Case search, which maps fairly cleanly to the `<remote-request>` element (except when using the `USH_INLINE_SEARCH` flag).
2. Advanced modules

10.1 Display Only Forms

Display only forms is deceptively simple. This setting causes a module's forms to be displayed directly in the parent menu (either the parent module's menu or the root CommCare menu), instead of the user needing to explicitly select the menu's name. This can be a UX efficiency gain.

However, quite a lot of suite generation is structured around modules, and using display only forms means that modules no longer map cleanly to `<menu>` elements. This means that modules using display only forms can't be "destinations" in their own right, so they don't work with end of form navigation, form linking, or smart links. It also complicates menu construction, raising issues like how to deal with module display conditions when the module doesn't have a dedicated `<menu>`.

10.2 Select Parent First

When the "select parent first" setting is turned on for a module, the user is presented with a case list for the **parent** case type. The user selects a case from this list and is then given another case list limited to children of that parent. The user can select any other module in the app that uses the parent case type to use as the configuration for this parent case list.

This setting is controlled by `ModuleBase.parent_select` and has a dedicated model, `ParentSelect`. The suite implementation is small: HQ adds a `parent_id` datum to the module's `<entry>` blocks and a filter to the main `case_id` datum's nodeset to filter it to children of the parent: `[index/parent=instance('commcaresession')/session/data/parent_id]`.

This is easy to confuse with parent/child modules (see below), which affect the suite's `<menu>` elements and can affect datum generation.

The feature flag `NON_PARENT_MENU_SELECTION` allows the user to use any module as the "parent" selection, and it does not use the additional nodeset filter. This allows for more generic two-case-list workflows.

10.3 End of Form Navigation and Form Linking

These features allow the user to select a destination for the user to be automatically navigated to after filling out a particular form. To support this, HQ needs to figure out how to get to the requested destination, both the actions taken (user selecting a form or menu) and the data needed (which needs to be pulled from somewhere, typically the session, in order to automatically navigate the user instead of asking them to provide it).

End of form navigation ("EOF nav") allows for a couple of specific locations, such as going back to the form's module or its parent module. EOF nav also has a "previous screen" option this is particularly fragile, since it requires HQ to replicate CommCare's UI logic.

Form linking, which is behind the `FORM_LINK_WORKFLOW` flag, allows the user to select a form as the destination. Form linking allows the user to link to multiple forms, depending on the value of an XPath expression.

Most forms can be linked "automatically", meaning that it's easy for HQ to determine what datums are needed. See the auto_link logic for implementation. For other forms, HQ pushes the burden of figuring out datums towards the user, requiring them to provide an XPath expression for each datum.

EOF nav and form linking config is stored in `FormBase.post_form_workflow`. In the suite, it's implemented as a stack in the form's `<entry>` block. For details, see docs on `WorkflowHelper`.

10.4 Child Modules

In principle child modules is very simple. Making one module a child of another simply changes the `menu` elements in the *suite.xml* file. For example in the XML below module `m1` is a child of module `m0` and so it has its `root` attribute set to the ID of its parent.

```
<menu id="m0">
    <text>
        <locale id="modules.m0"/>
    </text>
    <command id="m0-f0"/>
</menu>
<menu id="m1" root="m0">
    <text>
        <locale id="modules.m1"/>
```

(continues on next page)

(continued from previous page)

```
    </text>
    <command id="m1-f0"/>
</menu>
```

HQ's app manager only allows users to configure one level of nesting; that is, it does not allow for "grandchild" modules. Although CommCare mobile supports multiple levels of nesting, beyond two levels it quickly gets prohibitively complex for the user to understand the implications of their app design and for for HQ to determine a logical set of session variables for every case. The modules could have all different case types, all the same, or a mix, and for modules that use the same case type, that case type may have a different meanings (e.g., a "person" case type that is sometimes a mother and sometimes a child), which all makes it difficult for HQ to determine the user's intended application design. See below for more on how session variables are generated with child modules.

10.4.1 Menu structure

As described above the basic menu structure is quite simple however there is one property in particular that affects the menu structure: *module.put_in_root*

This property determines whether the forms in a module should be shown under the module's own menu item or under the parent menu item:

put_in_root	Resulting menu
True	id="<parent menu id>"
False	id="<module menu id>" root="<parent menu id>"

Notes:

- If the module has no parent then the parent is *root*.
- *root="root"* is equivalent to excluding the *root* attribute altogether.

10.4.2 Session Variables

This is all good and well until we take into account the way the Session works on the mobile which "prioritizes the most relevant piece of information to be determined by the user at any given time".

This means that if all the forms in a module require the same case (actually just the same session IDs) then the user will be asked to select the case before selecting the form. This is why when you build a module where *all forms require a case* the case selection happens before the form selection.

From here on we will assume that all forms in a module have the same case management and hence require the same session variables.

When we add a child module into the mix we need to make sure that the session variables for the child module forms match those of the parent in two ways, matching session variable names and adding in any missing variables. HQ will also update the references in expressions to match the changes in variable names. See `corehq.apps.app_manager.suite_xml.sections.entries.EntriesHelper.add_parent_datums` for implementation.

Matching session variable names

For example, consider the session variables for these two modules:

module A:

```
case_id:            load mother case
```

module B child of module A:

```
case_id_mother:     load mother case
case_id_child:      load child case
```

You can see that they are both loading a mother case but are using different session variable names.

To fix this we need to adjust the variable name in the child module forms otherwise the user will be asked to select the mother case again:

case_id_mother -> case_id

module B final:

```
case_id:            load mother case
case_id_child:      load child case
```

Note: If you have a case_id in both module A and module B, and you wish to access the ID of the case selected in parent module within an expression like the case list filter, then you should use `parent_id` instead of `case_id`

Inserting missing variables

In this case imagine our two modules look like this:

module A:

```
case_id:            load patient case
case_id_new_visit:  id for new visit case ( uuid() )
```

module B child of module A:

```
case_id:            load patient case
case_id_child:      load child case
```

Here we can see that both modules load the patient case and that the session IDs match so we don't have to change anything there.

The problem here is that forms in the parent module also add a `case_id_new_visit` variable to the session which the child module forms do not. So we need to add it in:

module B final:

```
case_id:            load patient case
case_id_new_visit:  id for new visit case ( uuid() )
case_id_child:      load child case
```

Note that we can only do this for session variables that are automatically computed and hence does not require user input.

10.5 Shadow Modules

A shadow module is a module that piggybacks on another module's commands (the "source" module). The shadow module has its own name, case list configuration, and case detail configuration, but it uses the same forms as its source module.

This is primarily for clinical workflows, where the case detail is a list of patients and the clinic wishes to be able to view differently-filtered queues of patients that ultimately use the same set of forms.

Shadow modules are behind the feature flag **Shadow Modules**.

10.5.1 Scope

The shadow module has its own independent:

- Name
- Menu mode (display module & forms, or forms only)
- Media (icon, audio)
- Case list configuration (including sorting and filtering)
- Case detail configuration

The shadow module inherits from its source:

- case type
- commands (which forms the module leads to)
- end of form behavior

10.5.2 Limitations

A shadow module can neither **be** a parent module nor **have** a parent module

A shadow module's source can **be** a parent module. The shadow will automatically create a shadow version of any child modules as required.

A shadow module's source can **have** a parent module. The shadow will appear as a child of that same parent.

Shadow modules are designed to be used with case modules. They may behave unpredictably if given an advanced module or reporting module as a source.

Shadow modules do not necessarily behave well when the source module uses custom case tiles. If you experience problems, make the shadow module's case tile configuration exactly matches the source module's.

10.5.3 Entries

A shadow module duplicates all of its parent's entries. In the example below, m1 is a shadow of m0, which has one form. This results in two unique entries, one for each module, which share several properties.

```
<entry>
    <form>
        http://openrosa.org/formdesigner/86A707AF-3A76-4B36-95AD-FF1EBFDD58D8
    </form>
    <command id="m0-f0">
```

(continues on next page)

(continued from previous page)

```
        <text>
            <locale id="forms.m0f0"/>
        </text>
    </command>
</entry>
<entry>
    <form>
        http://openrosa.org/formdesigner/86A707AF-3A76-4B36-95AD-FF1EBFDD58D8
    </form>
    <command id="m1-f0">
        <text>
            <locale id="forms.m0f0"/>
        </text>
    </command>
</entry>
```

10.5.4 Menu structure

In the simplest case, shadow module menus look exactly like other module menus. In the example below, m1 is a shadow of m0. The two modules have their own, unique menu elements.

```
<menu id="m0">
    <text>
        <locale id="modules.m0"/>
    </text>
    <command id="m0-f0"/>
</menu>
<menu id="m1">
    <text>
        <locale id="modules.m1"/>
        </text>
    <command id="m1-f0"/>
</menu>
```

Menus get more complex when shadow modules are mixed with parent/child modules. In the following example, m0 is a basic module, m1 is a child of m0, and m2 is a shadow of m0. All three modules have *put_in_root=false* (see **Child Modules > Menu structure** above). The shadow module has its own menu and also a copy of the child module's menu. This copy of the child module's menu is given the id *m1.m2* to distinguish it from *m1*, the original child module menu.

```
<menu id="m0">
    <text>
        <locale id="modules.m0"/>
    </text>
    <command id="m0-f0"/>
</menu>
<menu root="m0" id="m1">
    <text>
        <locale id="modules.m1"/>
    </text>
    <command id="m1-f0"/>
</menu>
```

(continues on next page)

(continued from previous page)

```
<menu root="m2" id="m1.m2">
↪                                    <text>
        <locale id="modules.m1"/>
    </text>
↪                                    <command id="m1-f0"/>
</menu>
<menu id="m2">
↪                                    <text>
        <locale id="modules.m2"/>
    </text>
↪                                    <command id="m2-f0"/>
</menu>
```

10.5.5 Legacy Child Shadow Behaviour

Prior to August 2020 shadow modules whose source was a parent had inconsistent behaviour.

The child-shadows were not treated in the same manner as other shadows - they inherited everything from their source, which meant they could never have their own case list filter, and were not shown in the UI. This was confusing. A side-effect of this was that display-only forms were not correctly interpreted by the phone. The ordering of child shadow modules also used to be somewhat arbitrary, and so some app builders had to find workarounds to get the ordering they wanted. Now in V2, what you see is what you get.

Legacy (V1) style shadow modules that have children can be updated to the new behaviour by clicking "Upgrade" on the settings page. This will create any real new shadow-children, as required. This will potentially rename the identifier for all subsequent modules (i.e. *m3* might become *m4* if a child module is added above it), which could lead to issues if you have very custom XML references to these modules anywhere. It might also change the ordering of your child shadow modules since prior to V2, ordering was inconsistent. All of these things should be easily testable once you upgrade. You can undo this action by reverting to a previous build.

If the old behaviour is desired for any reason, there is a feature flag "V1 Shadow Modules" that allows you to make old-style modules.

CHAPTER

ELEVEN

THE SUITE

An application's **suite.xml file** controls its structure.

The full XML spec for the suite is availabe on the commcare-core wiki.

11.1 Overview

Suite generation starts with `Application.create_suite`, which delegates to `SuiteGenerator`.

Suite generation is organized based on its major XML elements: resources, entries, details, etc. The suite is generated in two passes:

1. `corehq.apps.app_manager.suite_xml.sections` generates independent parts. A "section" is one of the major elements that goes into the suite: resources, entries, details, etc. This logic relies on the app document itself.
2. `corehq.apps.app_manager.suite_xml.post_process` handles logic that depends on the first pass being complete. Some of this logic adds new elements, some manipulates existing elements. This logic relies on the app document and also on the XML models generated by the first pass. Anything that deals with stacks must be a post processor, to guarantee that all menus have already been generated.

Challenges for developers in suite generation code:

- Language mixes CommCare concepts, such as "datum" and "menu", with HQ concepts, such as "modules"
- Lots of branching
- Has evolved one feature at a time, sometimes without attention to how different features interact
- CommCare's suite spec supports plenty of behavior that HQ doesn't allow the app builder to configure. In some areas, 20% of the HQ logic handles 80% of what's actually supported, so the code is more complex than the developer might expect. As an example of this, the suite code generally supports an arbitrary number of datums per form, even though the vast majority of forms only require one or two cases.

A bright spot: test coverage for suite generation is good, and adding new tests is typically straightforward.

11.2 Sections

11.2.1 DetailContributor

Details represent the configuration for case lists and case details. The reuse of the word "Detail" here is unfortunate. Details **can** be used for other purposes, such as the `referral_detail`, but 99% of the time they're used for case list/detail.

The case list is the "short" detail and the case detail is the "long" detail. A handful of configurations are only supported for one of these, e.g., actions only get added to the short detail.

The detail element can be nested. HQ never nests short details, but it nests long details to produce tabbed case details. Each tab has its own `<detail>` element.

The bulk of detail configuration is in the display properties, called "fields" and sometimes "columns" in the code. Each field has a good deal of configuration, and the code transforms them into named tuples while processing them. Each field has a format, one of about a dozen options. Formats are typically either UI-based, such as formatting a phone number to display as a link, or calculation-based, such as configuring a property to display differently when it's "late", i.e., is too far past some reference date.

Most fields map to a particular case property, with the exception of calculated properties. These calculated properties are identified only by number. A typical field might be called `case_dob_1` in the suite, indicating both its position and its case property, but a calculation would be called `case_calculated_property_1`.

11.2.2 EntriesContributor

This is the largest and most complex of the suite sections, responsible for generating an `<entry>` element for each form, including the datums required for form entry. The `EntriesHelper`, which does all of the heavy lifting here, is imported into other places in HQ that need to know what datums a form requires, such as the session schema generator for form builder and the UI for form linking.

When forms work with multiple datums, they need to be named in a way that is predictable for app builders, who reference them inside forms. This is most relevant to the "select parent first" feature and to parent/child modules. See `update_refs` and `rename_other_id`, both inner functions in `add_parent_datums`, plus this comment on matching parent and child datums.

11.2.3 FixtureContributor

This contributor adds a tiny fixture with a demo user group.

It's also the parent class for `SchedulerFixtureContributor`, a flagged feature.

11.2.4 MenuContributor

Menus *approximately* correspond to HQ modules.

Menus *almost* correspond to command lists, the screens in CommCare that ask the user to select a form or sub-menu. However, if the suite contains multiple `<menu>` elements with the same `id`, they will be concatenated and displayed as a single screen.

Menu ids will typically map to the module's position in the application: the first menu is `m0`, second is `m1`, etc.

Highlights of menu configuration:

- Display conditions, which become `relevant` attributes

- Display-only forms, which becomes the `put_in_root` attribute
- Grid style, to determine whether the command list should be displayed as a flat list or as a grid that emphasizes the menu icons

11.2.5 Resource Contributors

These contributors let the suite know where to find external resources,. These external resources are text files that are also part of the application's CCZ.

- `FormResourceContributor` handles XForms
- `LocaleResourceContributor` handles the text files containing translations
- `PracticeUserRestoreContributor` handles a dummy restore used for Practice Mode

11.3 Post Processors

11.3.1 EndpointsHelper

This is support for session endpoints, which are a flagged feature for mobile that also form the basis of smart links in web apps.

Endpoints define specific locations in the application using a stack, so they rely on similar logic to end of form navigation. The complexity of generating endpoints is all delegated to `WorkflowHelper`.

11.3.2 InstancesHelper

Every instance referenced in an xpath expression needs to be added to the relevant entry or menu node, so that CommCare knows what data to load when. This includes case list calculations, form/menu display conditions, assertions, etc.

HQ knows about a particular set of instances (locations, reports, etc.). There's factory-based code dealing with these "known" instances. When a new feature involves any kind of XPath calculation, it needs to be scanned for instances.

Instances are used to reference data beyond the scope of the current XML document. Examples are the commcare session, casedb, lookup tables, mobile reports, case search data etc.

Instances are added into the suite file in `<entry>` or `<menu>` elements and directly in the form XML. This is done in post processing of the suite file in `corehq.apps.app_manager.suite_xml.post_process.instances`.

11.3.3 How instances work

When running applications instances are initialized for the current context using an instance declaration which ties the instance ID to the actual instance model:

> <instance id="my-instance" ref="jr://fixture/my-fixture" />

This allows using the fixture with the specified ID:

> instance('my-instance')path/to/node

From the mobile code point of view the ID is completely user defined and only used to 'register' the instance in current context. The index 'ref' is used to determine which instance is attached to the given ID.

11.3.4 Instances in CommCare HQ

In CommCare HQ we allow app builders to reference instance in many places in the application but don't require that the app builder define the full instance declaration.

When 'building' the app we rely on instance ID conventions to enable the build process to determine what 'ref' to use for the instances used in the app.

For static instances like 'casedb' the instance ID must match a pre-defined name. For example

- casedb
- commcaresession
- groups

Other instances use a namespaced convention: "type:sub-type". For example:

- commcare-reports:<uuid>
- item-list:<fixture name>

11.3.5 Custom instances

App builders can define custom instances in a form using the 'CUSTOM_INSTANCES' plugin

11.3.6 RemoteRequestsHelper

The `<remote-request>` descends from the `<entry>`. Remote requests provide support for CommCare to request data from the server and then allow the user to select an item from that data and use it as a datum for a form. In practice, remote requests are only used for case search and claim workflows.

This case search config UI in app manager is a thin wrapper around the various elements that are part of `<remote-request>`, which means `RemoteRequestsHelper` is not especially complicated, although it is rather long.

Case search and claim is typically an optional part of a workflow. In this use case, the remote request is accessed via an action, and the rewind construct is used to go back to the main flow. However, the flag `USH_INLINE_SEARCH` supports remote requests being made in the main flow of a session. When using this flag, a `<post>` and query datums are added to a normal form `<entry>`. This makes search inputs available after the search, rather than having them destroyed by rewinding.

This module includes `SessionEndpointRemoteRequestFactory`, which generates remote requests for use by session endpoints. This functionality exists for the sake of smart links: whenever a user clicks a smart link, any cases that are part of the smart link need to be claimed so the user can access them.

11.3.7 ResourceOverrideHelper

This is dead code. It supports a legacy feature, multi-master linked applications.

The actual flag has been removed, but a lot of related code still exists.

11.3.8 WorkflowHelper

This is primarily used for end of form navigation and form linking. It contains logic to determine the proper sequence of commands to navigate a particular place in an app, such as a specific case list. It also needs to provide any datums required to reach that place in the app.

Because CommCare's UI logic is driven by the data currently in the user's session and the data needed by forms, rather than being directly configured, this means HQ needs to predict how CommCare's UI logic will behave, which is difficult and results in code that's easily disturbed by new features that influence navigation.

Understanding stacks in the CommCare Session is useful for working with `WorkflowHelper`.

Some areas to be aware of:

- Datums can either require manual selection (from a case list) or can be automatically selected (such as the user-case id).
- HQ names each datum, defaulting to `case_id` for datums selected from case lists. When HQ determines that a form requires multiple datums, it creates a new id for the new datum, which will often incorporate the case type. It also may need to rename datums that already exist - see `_replace_session_references_in_stack`.
- To determine which datums are distinct and which represent the same piece of information, HQ has matching logic in `_find_best_match`.
- `get_frame_children` generates the list of frame children that will navigate to a given form or module, mimicking CommCare's navigation logic
- Shadow modules complicate this entire area, because they use their source module's forms but their own module configuration.
- There are a bunch of advanced features with their own logic, such as advanced modules, but even the basic logic is fairly complex.
- Within end of form navigation and form linking, the "previous screen" option is the most fragile. Form linking has simpler code, since it pushes the complexity of the feature onto app builders.

CHAPTER

TWELVE

SYNCING LOCAL HQ INSTANCE WITH AN ANDROID PHONE

12.1 No syncing or submitting, easy method

If you would like to use a url or barcode scanner to download the application to your phone here is what you need to setup. You won't be able to submit or sync using this method, but it is easier.

12.1.1 Make sure your local django application is accessible over the network

The django server will need to be running on an ip address instead of localhost. To do this, run the application using the following command, substituting your local IP address.

```
./manage.py runserver 192.168.1.5:8000
```

Try accessing this url from the browser on your phone to make sure it works.

12.1.2 Make CommCare use this IP address

The url an application was created on gets stored for use by the app builder during site creation. This means if you created a site and application previously, while using a 'localhost:8000' url, you will have to make a code tweak to have the app builder behave properly.

The easiest way to check this is to see what url is shown below the barcode on the deploy screen.

If it is currently displaying a `localhost:8000/a/yourapp/...` url then open `localsettings.py` and set `BASE_ADDRESS = "192.168.1.5:8000"` substituting `192.168.1.5` with your local IP address.

12.1.3 Try it out

With this set up, you should be able to scan the barcode from your phone to

download and install your own locally built CommCare application!

12.2 Submitting and syncing from your local HQ instance (harder method)

12.2.1 Install nginx

`sudo apt-get install nginx` or

`brew install nginx`

12.2.2 Install the configuration file

In `/etc/nginx/nginx.conf`, at the bottom of the `http{}` block, above any other site includes, add the line:

`include /path/to/commcarehq/deployment/nginx/cchq_local_nginx.conf;`

12.2.3 Start nginx

`sudo nginx`

12.2.4 Make sure your local django application is accessible over the network

`./manage.py runserver`

Try accessing `http://localhost/a/domain` and see if it works. nginx should

proxy all requests to localhost to your django server.

12.2.5 Make Commcare use your local IP address

Set the `BASE_ADDRESS` setting in `localsettings.py` to your IP address (e.g.

`192.168.0.10`), without a port.

Additionally, modify `deployment/nginx/cchq_local_nginx.conf` to replace localhost with your IP address as `server_name`. For example, set server_name as `192.168.0.10`. Then run `sudo nginx -s reload` or `brew services restart nginx` to reload configuration.

You should now be able to access `http://your_ip_address/a/domain` from a phone or other device on the same network.

Note: You'll have to update these if you ever change networks or get a new IP address.

12.2.6 Rebuild and redeploy your application

You'll have to rebuild and redeploy your application to get it to sync.

CHAPTER

THIRTEEN

DIRECTLY MODIFYING APP BUILDS (CCZ FILES)

During development, it's occasionally useful to directly edit app files.

CommCare apps are bundled as `.ccz` files, which are just zip files with a custom extension.

See ccz.sh for utilities for unzipping, editing, and rezipping CCZ files. Doing this via the command line is often cleaner than doing it an in OS, which may add additional hidden files.

CHAPTER

FOURTEEN

ADDING COMMCARE BUILDS TO COMMCARE HQ

14.1 Using a management command

- *./manage.py add_commcare_build –latest* To fetch the latest released build from github
- *./manage.py add_commcare_build –build_version 2.53.0* To manually specify the build number to use

14.2 In the web UI

- Go to *http://HQ_ADDRESS/builds/edit_menu/*
- In the second section *Import a new build from the build server*
  1. In the Version field input the version in *x.y.z* format
  2. Click *Import Build*
- In the first section *Menu Options* add the version to HQ to make sure the build is available in the app settings.

CHAPTER

FIFTEEN

WEB APPS JAVASCRIPT

This document is meant to orient developers to working with Web Apps. Its primary audience is developers who are familiar with CommCare HQ but not especially familiar with CommCare mobile or formplayer.

15.1 System Architecture

High-level pieces of the system:

- **Web Apps** is a piece of CommCare HQ that allows users to enter data in a web browser, providing a web equivalent to CommCare mobile. Like the rest of HQ, web apps is built on django, but it is much heavier on javascript and lighter on python than most areas of HQ. While it is hosted on HQ, its major "backend" is formplayer.
- Formplayer is a Java-based service for entering data into XForms. Web apps can be thought of as a UI for this service. In this vein, the bulk of web apps javascript implements a javascript application called "Formplayer-Frontend". This makes the word "formplayer" sometimes ambiguous in this document: usually it describes the Java-based service, but it also shows up in web apps code references.
- **CloudCare** is a legacy name for web apps. Web apps code is in the `cloudcare` django app. It should not be used in documentation or anything user-facing. It shouldn't be used in code, either, unless needed for consistency. It mostly shows up in filenames and URLs.

Web apps is tightly coupled with formplayer, so check out the formplayer README.

15.1.1 Is Web Apps Part of HQ? Yes and No.

Web apps is a part of HQ, but once you move into an app, its communication with the rest of HQ is quite limited.

Ways in which web apps is a typical piece of HQ code:

- The `cloudcare` django app contains the HQ side of web apps.
- The `cloudcare.views` module contains views. Note that there's just one major view for web apps, `FormplayerMain`, and another for app preview, `PreviewAppView`.
- When you look at the web apps home page, where there's a tile for each app, those apps come from HQ.
- Web apps does have some interactions with HQ once you're in an app:
  - The Log In As action works via HQ
  - HQ provides some system information, like the current user's username and the mapbox API key, via the original context and initial page data
  - HQ directly serves multimedia files

- Web apps calls HQ analytics code (Google Analytics, Kissmetrics, etc.)

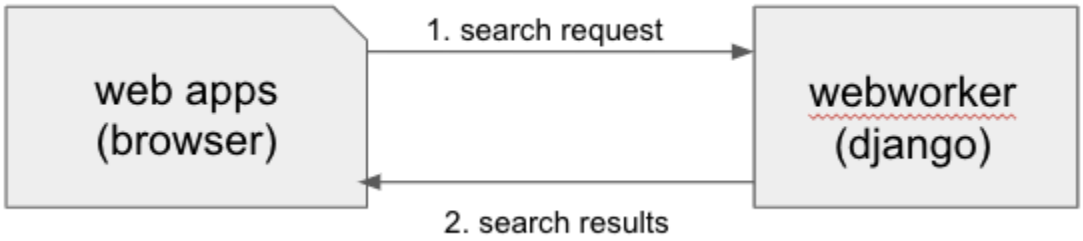
However, in most ways, once you move into an app, web apps only interacts with formplayer and is just a thin UI layer.

Also, before going into an app, on the web apps home page, the sync and saved forms options are formplayer requests.

Example: Case Search

As an example, consider case search, where the user triggers a search that runs against all cases in the domain, not just cases in their casedb, which requires a query to postgres.

If web apps were a typical area of HQ, this might be implemented as a single ajax request:



Instead, formplayer acts as an intermediary. Web apps sends a navigation request to formplayer, which constructs and sends a search request to HQ, which returns a search results response to formplayer, which transforms the results into a case list response and sends that back to web apps.

Note that formplayer does a good deal of processing here. It's formplayer that determines a search request is needed, and it's formplayer that processes the results and turns them into a table. Web apps doesn't even know that a search happened.

This approach clearly isn't minimizing network requests. However, this architecture is what allows CommCare mobile and web apps to share the majority of their logic, which is huge for developing and maintaining features to work on both platforms.

This architecture also makes formplayer responsible for security. Formplayer authorizes the user, via a request to HQ. It also means that formplayer mediates all access to data, so the user never has access to the full restore. This means that in-app limitations, like case list filters, are genuinely firm boundaries. You could imagine a javascript implementation of formplayer, which would reduce network requests, but would involve the browser making a request for the full restore, which the user could then inspect.

15.2 Anatomy of a Web Apps Feature

The relationships between HQ, formplayer, and mobile mean that web apps work frequently involves working in multiple languages, in multiple repositories, which may have different release processes.

New features require some or all of the following:

	Repository	Language
App manager UI where the the feature is enabled & configured	commcare-hq	Python / HTML / JavaScript
App build logic, typically changes to suite generation	commcare-hq	Python
New model for the configuration	commcare-core	Java
Formplayer processing to add the new feature to a response	formplayer	Java
Web apps UI for the feature	commcare-hq	JavaScript / HTML
CommCare Mobile UI for the new feature	commcare-android	Java

Not all features have all of these pieces:

- Some features don't require any Java
  - They might use existing flexible configuration, like adding a new appearance attribute value to support a new data entry widget
  - They might rearrange existing constructs in a new way. CommCare supports a much broader set of functionality than what HQ allows users to configure.
- Some features don't get implemented on mobile.
- Some features, like case search, have additional HQ work because they interact with HQ in ways beyond what's described above.

15.2.1 Example: Registration from Case List

As an example, consider registration from the case list:

- A CommCare HQ user goes to the module settings page in app builder and turns on the feature, selecting the registration form they want to be accessible from the case list.
  - This adds a new attribute to their `Application` document - specifically, it populates `case_list_form` on a `Module`.
- When the user makes a new build of their app, the app building code reads the `Application` doc and writes out all of the application files, including the `suite.xml`.
  - The module's case list configuration is transformed into a detail element, which includes an action element that represents the case list form.
- When a Web Apps user clicks the menu's name to access the case list, web apps sends a `navigate_menu` request to formplayer that includes a set of `selections` (see navigation and replaying of sessions).
  - The formplayer response tells web apps what kind of sceen to display:
    - The `type` is `entities` which tells web apps to display a case list UI
    - The `entities` list contains the cases and their properties
    - The `actions` list includes an action for the case list registration form, which tells web apps to display a button at the bottom of the case list with the given label, that when clicked will add the string `action`

`0` to the `selections` list and then send formplayer another navigation request, which will cause formplayer to send back a form response for the registration form, which web apps will then display for the user.

Note how generic the concepts web apps deals with are: "entities" can be cases, fixture rows, ledger values, etc. Web apps doesn't know what cases are, and it doesn't know the difference between an action that triggers a case list registration form and an action that triggers a case search.

15.3 JavaScript Overview

The remainder of this document discusses the web apps front end, which is the javascript in `corehq.apps.cloudcare.static.cloudcare.js`. As described above, in many ways this code is independent of the rest of HQ.

Think of the web apps code as split into two major pieces: form entry and everything else.

Form entry contains all interaction while filling out a form: all the different types of questions, the logic for validating answers as the user fills them out, etc. This code is written in a combination of knockout and vanilla JS, and it's quite old (pre-2014).

Wrapped around the form entry code is everything else, which is controlled by the `FormplayerFrontend` javascript application. The single-page application (SPA) approach is unique in HQ. This is also the only area of HQ that uses Backbone and Marionette. Most of this code was written, or substantially re-written, around 2016. `FormplayerFrontend` controls:

- In-app navigation, case lists, case search, etc.
- Web apps home screen displaying all of a domain's apps
- Syncing
- Saved forms
- Log In As

15.4 JavaScript Vocabulary

Tight coupling with formplayer means web apps tends to use formplayer/mobile/CommCare vocabulary rather than HQ vocabulary: "entities" instead of "cases", etc.

The major CommCare/HQ concepts FormplayerFrontend deals with are apps, users, menus, and sessions. "Apps" and "users" are the same concepts they are in the rest of HQ, while a "menu" is a UI concept that covers the main web apps screens, and "sessions" means incomplete forms.

15.4.1 Apps

These are HQ apps. Most of the logic around apps has to do with displaying the home screen of web apps, where you see a tiled list of apps along with buttons for sync, settings, etc.

This home screen has access to a subset of data from each app's couch document, similar but not identical to the "brief apps" used in HQ that are backed by the `applications_brief` couch view.

Once you enter an app, web apps no longer has access to this app document. All app functionality in web apps is designed as it is in mobile, with the feature's configuration encoded in the form XML or suite.xml. That config is then used to generate the web apps UI and to formulate requests to formplayer.

15.4.2 Users

These are HQ users, although the model has very few of the many attributes of CouchUser.

Most of the time you're only concerned with the current user, who is accessible by requesting `currentUser` from the FormplayerFrontEnd's channel (see below for more on channels).

The users code also deals with the Log In As workflow. Log In As is often described as "restore as" in the code: the user has a `restoreAs` attribute with the username of the current Log In As user, the `RestoreAsBanner` is the yellow banner up top that shows who you're logged in as, and the `RestoreAsView` is the Log In As screen. The current Log In As user is stored in a cookie so that users do not need to repeat the workflow often.

15.4.3 Menus

This is where the bulk of new web apps development happens. This contains the actual "menu" screen that lists forms & sub-menus, but it also contains case lists, case details, and case search screens.

menus/views.js contains the views for case list and case detail, while views/query.js contains the case search view.

15.4.4 Sessions

These are incomplete forms - the same incomplete forms workflow that happens on mobile, but on web apps, incomplete forms are created automatically instead of at the user's request. When a user is in form entry, web apps creates an incomplete form in the background and stores the current answers frequently so they can be accessed if the user closes their browser window, etc. These expire after a few days, maybe a week, exact lifespan might be configurable by a project setting. They're accessible from the web apps home screen.

15.5 JavaScript Directory Structure

All of this code is stored in `corehq.apps.cloudcare.static.cloudcare.js`

It has top-level directories for the two major areas described above: `form_entry` for in-form behavior and `formplayer` for the `FormplayerFrontend` application. There are also a few top-level directories and files for miscellaneous behavior.

15.5.1 form_entry

The form_entry directory contains the logic for viewing, filling out, and submitting a form.

This is written in knockout, and it's probably the oldest code in this area.

Major files to be aware of:

- form_ui.js defines `Question` and `Container`, the major abstractions used by form definitions. `Container` is the base abstraction for groups and for forms themselves.
- entries.js defines `Entry` and its many subclasses, the widgets for entering data. The class hierarchy of entries has a few levels. There's generally a class for each question type: `SingleSelectEntry`, `TimeEntry`, etc. Appearance attributes can also have their own classes, such as `ComboboxEntry` and `GeoPointEntry`.
- web_form_session.js defines the interaction for filling out a form. Web apps sends a request to formplayer every time a question is answered, so the session manages a lot of asynchronous requests, using a task queue. The session also handles loading forms, loading incomplete forms, and within-form actions like changing the form's language.

Form entry has a fair amount of test coverage. There are entry-specific tests and also tests for web_form_session.

15.5.2 formplayer

The formplayer directory contains logic for selecting an app, navigating through modules, displaying case lists, and almost everything besides filling out a form.

This is written using Backbone and Marionette. Backbone is an MVC framework for writing SPAs, and Marionette is a library to simplify writing Backbone views.

`FormplayerFrontend` is the "application" in this SPA.

15.5.3 Miscellany

This is everything not in either the `form_entry` or `formplayer` directory.

debugger

This controls the debugger, the "Data Preview" bar that shows up at the bottom of app preview and web apps and lets the user evaluate XPath and look at the form data and the submission XML.

preview_app

This contains logic specific to app preview.

There isn't much here: some initialization code and a plugin that lets you scroll by grabbing and dragging the app preview screen.

The app preview and web apps UIs are largely identical, but a few places do distinguish between them, using the `environment` attribute of the current user. Search for the constants `PREVIEW_APP_ENVIRONMENT` and `WEB_APPS_ENVIRONMENT` for examples.

hq_events.js, although not in this directory, is only really relevant to app preview. It controls the ability to communicate with HQ, which is used for the "phone icons" on app preview: back, refresh, and switching between the standard "phone" mode and the larger "tablet" mode.

config.js

This controls the UI for the Web Apps Permissions page, in the Users section of HQ. Web apps permissions are not part of the standard roles and permissions framework. They use their own model, which grants/denies permissions to apps based on user groups.

formplayer_inline.js

Inline formplayer is for the legacy "Edit Forms" behavior, which allowed users to edit submitted forms using the web apps UI. This feature has been a deprecation path for quite a while, largely replaced by data corrections. However, there are still a small number of clients using it for workflows that data corrections doesn't support.

utils.js

This contains miscellaneous utilities, mostly around error/success/progress messaging:

- Error and success message helpers
- Progress bar: the thin little sliver at the very top of both web apps and app preview
- Error and success messaging for syncing and the "settings" actions: clearing user data and breaking locks
- Sending formplayer errors to HQ so they show up in sentry

markdown.js

Code for initializing the markdown renderer including a bunch of code, `injectMarkdownAnchorTransforms` and its helpers, related to some custom feature flags that integrate web apps with external applications.

15.6 JavaScript Architectural Concepts

There are a few ways that web apps is architecturally different from most HQ javascript, generally related to it being a SPA and being implemented in Backbone and Marionette.

It's heavily asynchronous, since it's a fairly thin UI on top of formplayer. Want to get the a case's details? Ask formplayer. Want to validate a question? Ask formplayer. Adding functionality? It will very likely require a formplayer PR - see "Anatomy of a Web Apps Feature" above.

Web apps is also a relatively large piece of functionality to be controlled by a single set of javascript. It doesn't exactly use globals, but `FormplayerFrontend` is basically a god object, and it uses a global message bus - see "Events" below.

15.6.1 Persistence

Web apps has only transient data. All persistent data is handled by formplayer and/or HQ. The data that's specific to web apps consists mostly of user-related settings and is handled by the browser: cookies, local storage, or session storage.

The Log In As user is stored in a cookie. Local storage is used for the user's display options, which are the settings for language, one question per screen, etc. Session storage is also used to support some location handling and case search workflows.

Note that these methods aren't appropriate for sensitive data, which includes all project data. This makes it challenging to implement features like saved searches.

15.6.2 Application

`FormplayerFrontend` is a Marionette Application, which ties together a bunch of views and manages their behavior. It's defined in formplayer/app.js.

For day-to-day web apps development, it's just useful to know that `FormplayerFrontend` controls basically everything, and that the initial hook into its behavior is the `start` event, so we have a `before:start` handler and a `start` handler.

15.6.3 Regions

Marionette's regions are UI containers, defined in the FormplayerFrontend's `before:start` handler.

We rarely touch the region-handling code, which defines the high-level structure of the page: the "main" region, the progress bar, breadcrumbs, and the restore as banner. The persistent case tile also has a region. Most web apps development happens within the `main` region.

It is sometimes useful to know how the breadcrumbs work. The breadcrumbs are tightly tied to formplayer's selections-based navigation. See Navigation and replaying of sessions for an overview and examples. The breadcrumbs use this same selections array, which is also an attribute of `CloudcareURL`, with one breadcrumb for each selection.

15.6.4 Backbone.Radio and Events

Marionette integrates with Backbone.Radio to support a global message bus.

Although you can namespace channels, web apps uses a single `formplayer` channel for all messages, which is accessed using `FormplayerFrontend.getChannel()`. You'll see calls to get the channel and then call `request` to get at a variety of global-esque data, especially the current user. All of these requests are handled by `reply` callbacks defined in `FormplayerFrontend`.

`FormplayerFrontend` also supports events, which behave similarly. Events are triggered directly on the `FormplayerFrontend` object, which defines `on` handlers. We tend to use events for navigation and do namespace some of them with `:`, leading to events like `menu:select`, `menu:query`, and `menu:show:detail`. Some helper events are not namespaced, such as `showError` and `showSuccess`.

15.6.5 Routing, URLs, and Middleware

As in many SPAs, all of web apps' "URLs" are hash fragments appended to HQ's main cloudcare URL, `/a/<DOMAIN>/cloudcare/apps/v2/`

Navigation is handled by a javascript router, `Marionette.AppRouter`, which extends Backbone's router.

Web apps routes are defined in router.js.

Routes **outside** of an application use human-readable short names. For example:

- `/a/<DOMAIN>/cloudcare/apps/v2/#apps` is the web apps home screen, which lists available apps and actions like sync.
- `/a/<DOMAIN>/cloudcare/apps/v2/#restore_as` is the Log In As screen

Routes **inside** an application serialize the `CloudcareURL` object.

`CloudcareURL` contains the current state of navigation when you're in an application. It's basically a js object with getter and setter methods.

Most app-related data that needs to be passed to or from formplayer ends up as an attribute of CloudcareURL. It interfaces almost directly with formplayer, and most of its attributes are properties of formplayer's SessionNavigationBean.

CloudcareURL is defined in formplayer/utils/utils.js although it probably justifies its own file.

URLs using `CloudcareURL` are not especially human-legible due to JSON serialization, URL encoding, and the obscurity of the attributes. Example URL for form entry:

`/a/<DOMAIN>/cloudcare/apps/v2/#%7B%22appId%22%3A%226<APP_ID>%22%2C%22steps%22%3A%5B%221%22%2C%22<CASE_I`

The router also handles actions that may not sound like traditional navigation in the sense that they don't change which screen the user is on. This includes actions like pagination or searching within a case list.

Other code generally interacts with the router by triggering an event (see above for more on events). Most of `router.js` consists of event handlers that then call the router's API.

Every call to one of the router's API functions also runs each piece of web apps middleware, defined in middleware.js. This middleware doesn't do much, but it's a useful place for reset-type logic that should be called on each screen change: scrolling to the top of the page, making sure any form is cleared out, etc. It's also where the "User navigated to…" console log messages come from.

15.6.6 Tests

There are tests in the `spec` directory. There's decent test coverage for js-only workflows, but not for HTML interaction.

15.7 Marionette Views

Web apps development frequently happens in `FormplayerFrontend` views. These views are javascript classes that inherit from Marionette.View. This section describes the `View` attributes that web apps most frequently uses.

For code references, take a look at the query views, which control the case search screen, or the menus views, which control menus, case lists, and case details.

15.7.1 `template` and `getTemplate`

These attributes link view code with the relevant HTML template.

We typically use `template` and just fetch a template by its id, then run it through underscore's `_.template` function. The QueryListView, which controls the case search screen, is a good example, defining `template` as `_.template($("#query-view-list-template").html() || "")`.

`getTemplate` is a callback, so it has access to `this` and allows for more complex logic. We use it in the <MenuView *https://github.com/dimagi/commcare-hq/blob/9baa5a05181e3e74cdf8608223eeff69aca5c0d7/corehq/apps/cloudcare/static/cloudcare/js/formplayer/menus/views.js#L35-L43*>__ to determine whether to display the menu in a list style or in a grid style.

15.7.2 `tagName`, `className`, and `attributes`

All views have a single encompassing container, which is added by Marionette, so it doesn't show up in the view's HTML template. These attributes influence that container.

`tagName`, which can be a string or a callback, defines the HTML node type, typically `div` or `tr`.

`className` allows setting a CSS class on the container.

`attributes` allows setting HTML attributes. We mostly use this for accessibility, to set attributes like `tabindex`.

15.7.3 `initialize`, `templateContext`, and `onRender`

`initialize` is for any setup, particularly for storing any options that were passed into the view (although `this.options` is available throughout the view).

`templateContext` is for building an object of context to pass to the template, as with `_.template` and django views.

`onRender` is called every time Marionette renders the view. We use this primarily for attaching events to content. Note that Marionette has its own attributes for event handling, discussed below, but `onRender` is useful for non-standard events provided by third-party widgets like select2 and jQuery UI.

15.7.4 `ui`, `events`, and `modelEvents`

These attributes are for event handling.

`ui` is an object where keys are identifiers and values are jQuery selectors. Elements defined in `ui` are available to other code in the view using `this.ui`. For an example, see how QueryListView defines ui elements for the case search screen's submit and clear buttons.

`events` ties elements from `ui` with standard HTML events. Events references the event, the ui element, and the callback to invoke. Again, QueryListView is a good example.

`modelEvents` attaches callbacks to events on the Backbone model, as opposed to ui events. We don't use this often, but QueryView, which controls an individual search field on the case search screen, uses it to force the view to re-render whenever the underlying model changes, so that select2 behaves properly.

15.7.5 `childView`, `childViewContainer`, and `childViewOptions`

These options apply to views that extend Marionette.CollectionView. These views are structured to display a list of child views. As an example, QueryListView controls the case search screen and has a child QueryView for each individual search field. The case list's CaseListView is a more complex example, with a CaseView child view that has several subclasses.

`childView` names the view that is a child of this view.

`childViewContainer` tells Marionette where in the parent view to render the children. This can be an HTML node name, analagous to `tagName`, or it can be a jQuery selector identifying a specific element in the view that should contain the children.

`childViewOptions` allows the parent view to pass data to the children views. Some use cases:

- DetailTabListView * uses it to pass information about the parent to the child views.
- QueryListView uses it to give the child views access to the entire parent view.
- MenuListView uses it to pass information that the parent view calculates, namely, the child's index position in the collection.

CHAPTER

SIXTEEN

FORMPLAYER IN HQ

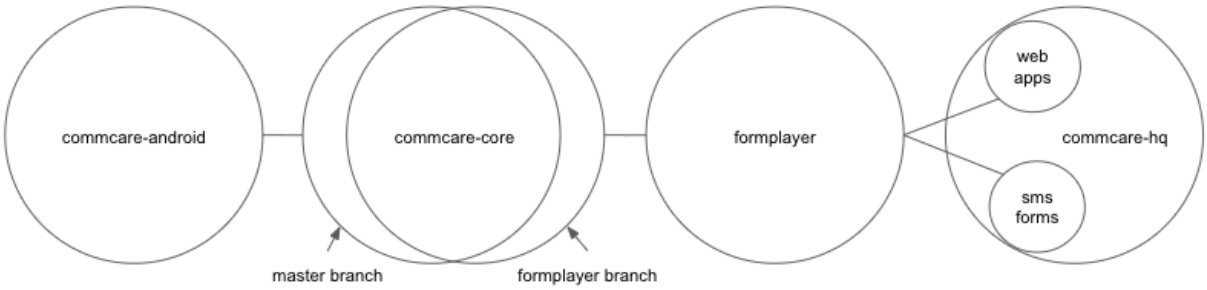
This documentation describes how formplayer fits into the larger CommCare system, especially how it relates to CommCare HQ development. For details on building, running, and contributing to formplayer, see the formplayer repository.

16.1 What Is Formplayer?

Formplayer is a Java Spring Boot server that wraps commcare-core and presents its main features as an HTTP API. CommCare HQ's Web Apps, App Preview, and SMS Forms features are built on top of it:

- Web Apps is a single-page application, inlined into a CommCare HQ template, that provides a web UI backed by the formplayer API.
- App Preview is essentially the same as web apps, but embedded as a cell-phone-shaped iframe within the App Builder.
- SMS Forms serializes a form filling session over SMS in a question / answer sequence that is handled by the main HQ process, which hits formplayer's API to send answers and get the next question.

16.2 Repository Overview



- commcare-android: The UI layer of CommCare mobile.
- commcare-core: The CommCare engine, this powers both CommCare mobile and formplayer. Mobile uses the `master` branch, while formplayer uses the `formplayer` branch. The two branches have a fairly small diff.
- formplayer
- commcare-hq: HQ hosts web apps and the processes that run SMS forms.

16.3 Relevant Architectural Decisions

While a full detailing of formplayer's architecture is beyond the scope of this document, a few architectural decisions are particularly useful for HQ devs who are new to formplayer to understand.

16.3.1 Sandboxes

Sharing the commcare-core code between mobile and formplayer allows us to keep CommCare Android and web apps essentially compatible. However, because commcare-core was first written for mobile some of the paradigms it uses make more sense on mobile than on the web. Mobile is offline-first, so submitting up newly entered data and syncing back down changes others have made are intentional steps designed not to block someone who was unable to reach the server for hours, days, or longer. That model makes very little sense on the always-online Web Apps, but the sync/restore process is still a core part of the working model. There's even a "Sync" button shown to the user in the web apps UI.

Rather than always fetching the latest data from the source of truth, formplayer works off of locally synced subsets of data like those that would be on people's phones if every user had their own phone. These "sandboxes" are stored as Sqlite DB files, as they are on the phone. A phone typically has one db file and one user, whereas on formplayer, there are as many db files as there are users, i.e. tens of thousands. Each file has its own slice of the data synced down from the source of truth, but always just a little bit out of date if anyone's updated it after their last sync.

16.3.2 Request routing

Each user is tied by a `formplayer_session` cookie directly to a machine. The cookie is just a routing hint that contains the user id but doesn't constitute authentication. That sticky association only changes if we add or remove machines, and in that case, the minimum number of associations are changed to rebalance it because we use "consistent hashing". In steady state, one user's requests will always be served by the same machine.

An obvious side effect of that is that if a machine is down, all users assigned to that machine will not be able to do anything until the machine is back up. During a formplayer deploy, when we have to restart all formplayer processes, a rolling restart doesn't help uptime, since for every individual user their machine's process will be down while it restarts.

Routing implications for adding and removing machines

It's expensive to delete a user's sqlite sandbox, because rebuilding it requires requesting a full restore from HQ, but it's always **safe** to delete it, because that rebuild from the source of truth will get the user back to normal. This property makes removing machines a safe operation. Similarly, adding new machines doesn't pose an issue because the subset of users that get routed to them will just have their sqlite db file rebuilt on that machine the next time it's needed. These sandbox db files effectively serve as a cache.

What **does** cause a problem is if a user is associated with machine A, and then gets switched over to machine B, and then goes back to machine A. In that situation, any work done on machine A wouldn't get synced to machine B until the next time the user did a "sync" on machine B. Until then, they would be working from stale data. This is especially a problem for SMS Forms, where the user doesn't have an option to explicitly sync, and where if the underlying case database switches mid-form or between consecutive forms to a stale one, the user will see very unintuitive behavior. Formplayer currently doesn't have a concept of "this user has made a request handled by a different formplayer machine since the last time this machine saw this user"; if it did and it forced a sync in that situation, that would mostly solve this problem. This problem can show up if you expand the cluster and then immediately scale it back down by removing the new machines.

Lastly, sqlite db files don't hang around forever. So that stale files don't take up ever more disk, all formplayer sqlite db files not modified in the last 5 days are regularly deleted. The "5 days" constant is set by formplayer_purge_time_spec.

Balancing issues for large numbers of machines

Each user has a widely variable amount of traffic, and the more machines there are in the system, the wider the spread becomes between the least-traffic machine and the most-traffic machine, both statistically and in practice.

If you randomly select 10,000 values from *[1, 10, 100, 100]* and then divide them into *n* chunks, the sum of the values in each chunk have a wider distribution the larger *n* is. Here the values represent each user and how much traffic they generate, so this is meant to show that the more machines you have for a fixed number of users using this rigid load balancing method, the wider the spread is between the least-used and most-used machine.

This means that fewer, larger machines is better than more smaller machines. However, we have also found that formplayer performance drops sharply when you go from running on machines with 64G RAM and 30G java heap to machines with 128G RAM and (still) 30G java heap. So for the time being our understanding is that the max machine size is 64G RAM to run formplayer on. This, of course, limits our ability to mitigate the many-machines load imbalance problem.

16.4 Navigation

The purpose of this section is to introduce formplayer navigation *in the context of CommCare HQ*. CommCare allows for a wide variety of behavior, but applications built in HQ use a subset of this behavior and a few common workflows.

For a full picture of CommCare, see the commcare-core wiki, in particular

- CommCare Session
- CommCare Session External Instance Definition
- CommCare 2.0 Suite Definition

Note: This document uses case-centric language, because that is the entity most often used in HQ. Any references to cases could be changed to any model that is backed by a similar XML structure.

16.4.1 The CommCare Session

A single CommCare session is (loosely) defined as the series of **actions** taken by a user from the time that they view the home screen until the time that they press "Submit" in a form, plus the **data** that is collected and persisted along the way as those those actions are taken.

The end goal of a session is to complete a form. This implies:

- Every CommCare form has specific pieces of data that it needs to have access to in order to function properly.
- Forms always get that data by referencing the session, i.e. `instance('commcaresession')/session/data/blahblahblah`
- The flow of a CommCare session is always structured to ensure that a user has "collected" all of the data that a certain form needs before allowing the user to enter that form.

The session is implemented by the class CommCareSession, with its data stored in `CommCareSession.collectedDatums`. The session also keeps track of the current menu or form id, in `CommCaseSession.currentCmd`. CommCareSession.getNeededData determines what information is needed next, based on the current command on the data needed by entries associated with that command, and `MenuSessionRunnerService` (see below) uses that need to determine what screen to show.

Each piece of **data** in the session is either:

- "Action history" - information about the actions that a user has taken in the session so far. This is useful to implement "back" navigation, and it is also a necessary part of formplayer being a RESTful service (see the section below on replaying sessions).
- "Collected data" - pieces of raw app data that will be used later within some form in the app, like case ids

The **actions** a user can take in the session are:

- Select a menu - this adds a "command id", which identifies the menu, to the session
- Select a case (or confirm selection of a case) - this adds a "datum" to the session, the case's id, which both serves as a record of the selection action and identifies the case.
- Select a form - this adds a "command id", which identifies the form, to the session

16.4.2 Screens

This section answers the question, "After each user action in a CommCare app, how does CommCare decide what screen to show next?"

There are three principles used to answer this question:

1. Order matters: CommCare will never instruct the user to collect a datum that is listed later in a <session> block before one listed earlier in that same block. This allows <datum>s that come later in the list to refer to ones that came earlier, which is useful in workflows such as selecting a case that must be the child of a previously selected case.

1. Equality of datums: CommCare is at all times aware of a universe of all datums which are required by at least one form in the app - some of which may overlap. The most notable effect of this is that if all of the possible actions a user is considering all require the same datum, CommCare will ask the user to select that datum before moving on to select the action.

1. Never collect unnecessary data

At any given time, there is one piece of data that the app is focused on acquiring, and the screen that CommCare shows is determined by the 'type' of that piece of data:

- If CommCare is looking for a "datum", it will show a case list
- If CommCare is looking for a "command id", it will show a menu screen

Before the "Start" button is pressed, CommCare is always looking for a command id (module or form), which is why the app's root module menu is always the first screen to be shown.

`commcare-core`, the engine shared by CommCare mobile, the CommCare CLI, and formplayer, has the following types of screens:

- `MenuScreen` - Displays a list of menus and/or forms.
- `EntityScreen` - Displays a case list.
- `QueryScreen` - Used for case search and claim, see section below. This is the screen the displays search fields. Search results are displayed using an `EntityScreen`.
- `SyncScreen` - Used for case search and claim, see section below. This screen isn't visible to the user, but it controls the sending of the claim request and then syncing.

`formplayer` uses these same screens, but `FormplayerQueryScreen` and `FormplayerSyncScreen` extend `QueryScreen` and `SyncScreen`. This means that formplayer and the CLI use different logic for case search & claim.

A screen's job is to handle input, which often includes updating the session - either setting the `currentCmd` or adding an item to *collectedDatums* `.

The `EntityScreen` is a special case, since it handles what, from the user's perspective, are two screens: the case list and the case detail confirmation. `EntityScreen` acts as a "host" screen, extending `CompoundScreenHost`. The `EntityDetailSubscreen`, which handles the case detail, is not a full `Screen` but rather a `Subscreen` that updates its host, the entity screen, which is then in charge of updating the session.

Case lists that allow for the selection of multiple entities have further special handling, described in formplayer docs.

16.4.3 Selections

User activity in CommCare is oriented around navigating to and then submitting forms. User actions are represented as a series of "selections" that begin at the app's initial list of menus and eventually end in form entry.

The selections list keeps track of actions the user has taken in the current session. Every time a user takes a navigation action (selecting a menu, case, or form), web apps updates the `selections` list and sends it to formplayer as part of a `navigate_menu` request.

A single selection can be:

- An integer index. This is used for lists of menus and/or forms and represents the position of the selected item.
- A case id. This indicates that the user selected the given case.
- The keyword `action` and an integer index, such as `action 0`. This represents the user selecting an action on a detail screen. The index represents the position of the action in the detail's list of actions.

Replaying sessions

For an example, consider the selections `[1, 'abc123', 0]`. These indicate that a user selected the second visible menu, then selected case `abc123`, then selected the first visible menu (or form). This might have mapped to the following requests:

- `navigate_menu_start` to view the first screen, a list of menus
- `navigate_menu` with selections `[1]` to select the first menu, which leads to a case list
- `get_details` with selections `[1]` to select a case and show its details
- `navigate_menu` with selections `[1, 'abc123']` to confirm the case selection, which leads to a list of forms
- `navigate_menu` with selections `[1, 'abc123', 0]` to select the first form
- `submit-all` to submit the form when complete, which sends the user back to the first list of menus

Because formplayer is a RESTful service, each of these individual request plays through all of the given selections, even those that were already completed earlier. If an early selection contained an expensive operation, that operation can slow down requests for the rest of the session. Selections that cause side effects will cause them repeatedly.

MenuSessionRunnerService controls formplayer navigation. This largely happens in `advanceSessionWithSelections`, which loops over the selections list, replaying the full session as described above.

On each iteration, `advanceSessionWithSelections` determines the current screen based on the state of the `MenuSession` and then adds the next selection. It handles special navigation, which mostly relates to case search and claim (see below). When it runs out of selections, it returns the current menu, which is a response bean.

16.4.4 Case Search and Claim

Case search and claim allows a user to gain access to a case not already in their casedb. Case search and claim are implemented using a "remote request", which is an extension of an entry. While an entry's purpose is to get the user into an XForm, a remote request's purpose is to send a request to the server (HQ).

From the case list, the user takes a case search action. This presents them with a multi-field search screen, the `QueryScreen`. Their search inputs are sent as a request to HQ, which queries ElasticSearch for all cases in the domain and sends an XML document back with the results. Formplayer displays these results as a case list, an `EntityScreen`. When the user selects and then confirms a case, formplayer sends a POST request to HQ. This request, configured as part of the app, creates an extension case for the selected case. When this request returns, formplayer syncs, causing the selected case to be added to the user's casedb. CommCare then "rewinds" to the case list where the user started, selecting the case they claimed and moving them on to the next form or menu, using a mark/rewind mechanism discussed elsewhere.

CommCare treats case search and claim as pieces of data to be gathered. Just as CommCare typically is expecting either a command (a menu or form) or a datum (a case), it can instead expect a `QUERY_REQUEST` or a `SYNC_REQUEST`, which indicate it should display a `QueryScreen` or handle a `SyncScreen` (send the post request and subsequent sync).

Alternate Case Search Workflows

For projects using CommCare mobile, case search and claim is typically an unusual workflow. However, projects that use web apps, and therefore have guaranteed connectivity, may use it much more heavily, even to the point that the user is unaware of their casedb and always uses case search to find cases.

To support this approach, HQ allows apps to be configured with several alternate navigation flows. These workflows are gated by the `USH_CASE_CLAIM_UPDATES` feature flag.

The default case search and claim workflow shows the user the following screens:

- A menu screen, where the user selects a form/menu that requires a case
- A case list screen displaying the user's casedb, where the user elects to go into case search
- A case search screen, with search inputs for various fields
- A case list screen displaying the results of the search

The alternate case search workflows allow the user to skip the casedb case list, the case search screen, or both.

To handle this skipping behavior, every iteration over the selections list in `MenuSessionRunnerService.advanceSessionWithSelections` checks to see if there are any "automatic" actions needed, in `autoAdvanceSession`.

CHAPTER

SEVENTEEN

DEVICE RESTORE OPTIMIZATION

This document is based on the definitions and requirements for restore logic outlined in new-idea-for-extension-cases.md.

Important terms from that document that are also used in this document:

A case is **available** if

- it is **open** and not an **extension** case
- it is **open** and is the **extension** of an **available** case.

A case is **live** if any of the following are true:

- it is **owned** and **available**
- it has a **live child**
- it has a **live extension**
- it is **open** and is the **exension** of a **live** case

17.1 Dealing with shards

Observation: the decision to shard by case ID means that the number of levels in a case hierarchy impacts restore performance. The minimum number of queries needed to retrieve all *live* cases for a device can equal the number of levels in the hierarchy. The maximum is unbounded.

Since cases are sharded by case ID…

- Quotes from How Sharding Works
  - Non-partitioned queries do not scale with respect to the size of cluster, thus they are discouraged.
  - Queries spanning multiple partitions … tend to be inefficient, so such queries should be done sparingly.
  - A particular cross-partition query may be required frequently and efficiently. In this case, data needs to be stored in multiple partitions to support efficient reads.
- Potential optimizations to allow PostgreSQL to do more of the heavy lifting for us.
  - Shard `case_index` by domain.
    - Probably not? Some domains are too large.
  - Copy related case index data into all relevant shards to allow a query to run on a single shard.
    - Nope. Effectively worse than sharding by domain: would copy entire case index to every shard because in a given set of *live* cases that is the same size as or larger than the number of shards, each case will probably live in a different shard.

  - Re-shard based on ownership rather than case ID
    * Maybe use hierarchical keys since ownership is strictly hierarchical. This may simplify the sharding function.
    * Copy or move data between shards when ownership changes.

17.2 Data Structure

Simplified/minimal table definitions used in sample queries below.

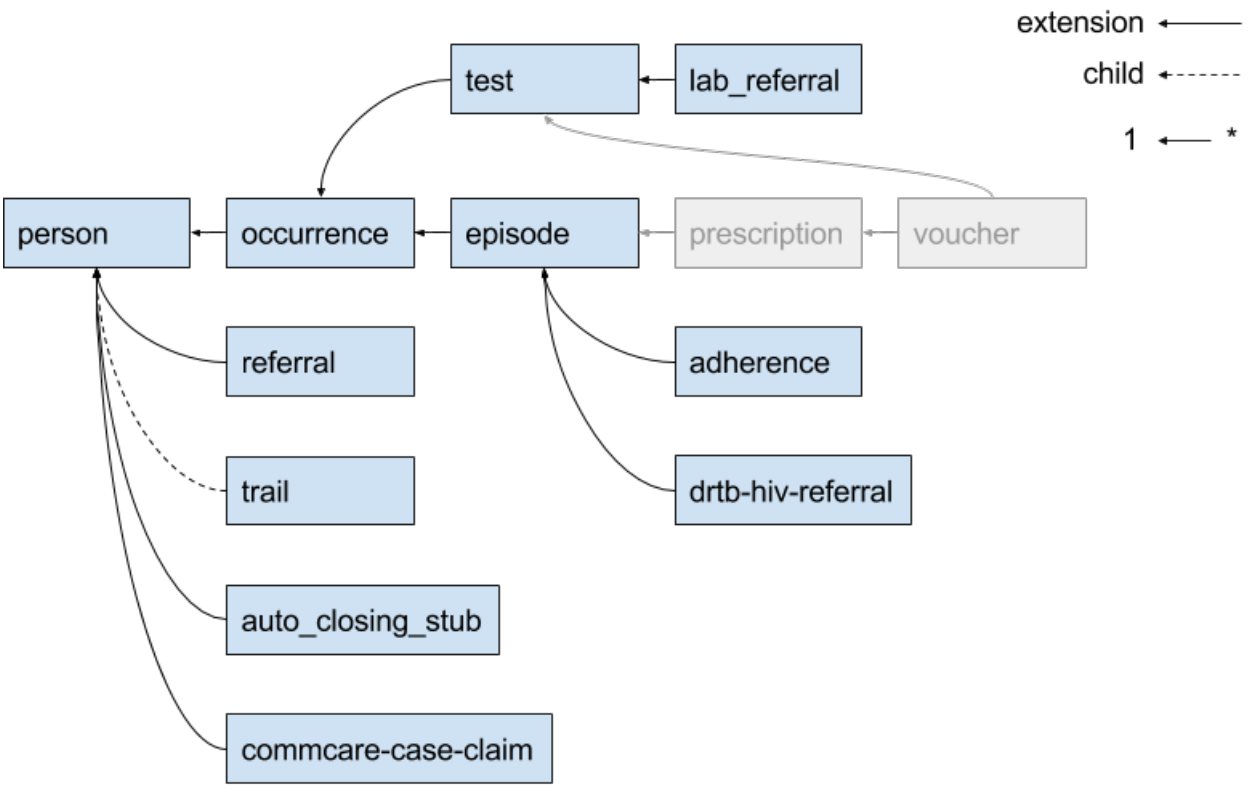
```
cases
  domain        char
  case_id       char
  owner_id      char
  is_open       bool

case_index
  domain        char
  parent_id     char
  child_id      char
  child_type    enum (CHILD|EXTENSION)
```

Presence of a row in the `case_index` adjacency list table implies that the referenced cases are *available*. The `case_index` is updated when new data is received during a device sync: new case relationships are inserted and relationships for closed cases are deleted. All information in the `case_index` table is also present in the `CommCareCaseIndex` and `CommCareCase` tables. Likewise for the `cases` table, which is a subset of `CommCareCase`.

17.3 Case Study: UATBC case structure

Sources: eNikshay App Design and Feedback - Case Structure and case_utils.py. These sources contain conflicting information. For example:

- case_utils.py references *prescription* and *voucher* while the sheet does not.
- case_utils.py has *referral* related to *episode*, not *person* as in the sheet.



With the current sharding (by case ID) configuration, the maximum number of queries needed to get all *live* cases for a device is 5 because there are 5 levels in the case hierarchy. Update: this is wrong; it could be more than 5. Example: if a case retrieved in the 5th query has unvisited children, then at least one more query is necessary. Because any given case may have multiple parents, the maximum number of queries is unbounded.

17.4 Algorithm to minimize queries while sharding on case ID

The algorithm (Python):

```
next_ids = get_cases_owned_by_device(owner_ids)
live_ids = set(next_ids)
while next_ids:
    related_ids = set(get_related_cases(next_ids))
    if not related_ids:
        break
    next_ids = related_ids - live_ids
    live_ids.update(related_ids)
```

All queries below are simplified for the purposes of demonstration. They use the simplified table definitions from the *Data Structure* section in this document, and they only return case IDs. If this algorithm is implemented it will likely make sense to expand the queries to retrieve all case data, including case relationship data, and to query directly from `CommCareCaseIndex` and `CommCareCase`.

The term "child" is a general term used to refer to a case that is related to another case by retaining a reference to the other case in its set of parent indices. It does not refer to the more restrictive "child" relationship type.

Definitions:

- `OWNER_DOMAIN` - the domain for which the query is being executed.
- `OWNER_IDS` - a set of user and group IDs for the device being restored.
- `NEXT_IDS` - a set of *live* case IDs.

`get_cases_owned_by_device()` retrieves all open cases that are not extension cases given a set of owner IDs for a device. That is, it retrieves all *live* cases that are directly owned by a device (user and groups). The result of this function can be retrieved with a single query:

```
select cx.case_id
from cases cx
  left outer join case_index ci
    on ci.domain = cx.domain and ci.child_id = cx.case_id
where
  cx.domain = OWNER_DOMAIN and
  cx.owner_id in OWNER_IDS and
  (ci.child_id is null or ci.child_type != EXTENSION) and
  cx.is_open = true
```

`get_related_cases()` retrieves all *live* cases related to the given set of *live* case IDs. The result of this function can be retrieved with a single query:

```
-- parent cases (outgoing)
select parent_id, child_id, child_type
from case_index
where domain = OWNER_DOMAIN
  and child_id in NEXT_IDS
union
-- child cases (incoming)
select parent_id, child_id, child_type
from case_index
where domain = OWNER_DOMAIN
  and parent_id in NEXT_IDS
  and child_type = EXTENSION
```

The `IN` operator used to filter on case ID sets should be optimized since case ID sets may be large.

Each of the above queries is executed on all shards and the results from each shard are merged into the final result set.

17.5 One query to rule them all.

Objective: retrieve all *live* cases for a device with a single query. This query answers the question Which cases end up on a user's phone? The sharding structure will need to be changed if we want to use something like this.

```
with owned_case_ids as (
  select case_id
  from cases
  where
    domain = OWNER_DOMAIN and
    owner_id in OWNER_IDS and
    is_open = true
), recursive parent_tree as (
  -- parent cases (outgoing)
```

(continues on next page)

(continued from previous page)

```
  select parent_id, child_id, child_type, array[child_id] as path
  from case_index
  where domain = OWNER_DOMAIN
    and child_id in owned_case_ids
  union
  -- parents of parents (recursive)
  select ci.parent_id, ci.child_id, ci.child_type, path || ci.child_id
  from case_index ci
    inner join parent_tree as refs on ci.child_id = refs.parent_id
  where ci.domain = OWNER_DOMAIN
    and not (ci.child_id = any(refs.path)) -- stop infinite recursion
), recursive child_tree as (
  -- child cases (incoming)
  select parent_id, child_id, child_type, array[parent_id] as path
  from case_index
  where domain = OWNER_DOMAIN
    and (parent_id in owned_case_ids or parent_id in parent_tree)
    and child_type = EXTENSION
  union
  -- children of children (recursive)
  select
    ci.parent_id,
    ci.child_id,
    ci.child_type,
    path || ci.parent_id
  from case_index ci
    inner join child_tree as refs on ci.parent_id = refs.child_id
  where ci.domain = OWNER_DOMAIN
    and not (ci.parent_id = any(refs.path)) -- stop infinite recursion
    and child_type = EXTENSION
)
select
  case_id as parent_id,
  null as child_id,
  null as child_type,
  null as path
from owned_case_ids
union
select * from parent_tree
union
select * from child_tree
```

17.6 Q & A

- Do we have documentation on existing restore logic?
  - Yes: new-idea-for-extension-cases.md
  - See also child/extension test cases
- new-idea-for-extension-cases.md: "[an extension case has] the ability (like a child case) to go out in the world and live its own life."

  What does it mean for an extension case to "live its own life"? Is it meaningful to have an extension case apart from the parent of which it is an extension? How are the attributes of an extension case "living its own life" different from one that is not living it's own life (I'm assuming *not living its own life* means it has the same lifecycle as its parent).
  - Danny Roberts:

    > haha i mean that may have been a pretty loosely picked phrase
    >
    > I think I specifically just meant you can assign it an owner separate from its parent's
- Is there an ERD or something similar for UATBC cases and their relationships?
  - Case structure diagram (outdated)
  - SDD _EY Comments_v5_eq.docx (page 24, outdated)
  - eNikshay App Design and Feedback - Case Structure - Kriti
  - case_utils.py - Farid

CHAPTER

EIGHTEEN

LOCATIONS

18.1 Location Permissions

18.1.1 Normal Access

Location Types - Users who can edit apps on the domain can edit location types. Locations - There is an "edit_locations" and a "view_locations" permission.

18.1.2 Restricted Access and Whitelist

Many large projects have mid-level users who should have access to a subset of the project based on the organization's hierarchy.

This is handled by a special permission called "Full Organization Access" which is enabled by default on all user roles. To restrict data access based on a user's location, projects may create a user role with this permission disabled.

This is checked like so:

```
user.has_permission(domain, 'access_all_locations')
```

We have whitelisted portions of HQ that have been made to correctly handle these restricted users. Anything not explicitly whitelisted is inaccessible to restricted users.

18.1.3 How data is associated with locations

Restricted users only have access to their section of the hierarchy. Here's a little about what that means conceptually, and how to implement these restrictions.

Locations: Restricted users should be able to see and edit their own locations and any descendants of those locations, as well as access data at those locations. See also `user_can_access_location_id`

Users: If a user is assigned to an accessible location, the user is also accessible. See also `user_can_access_other_user`

Groups: Groups are never accessible.

Forms: Forms are associated with a location via the submitting user, so if that user is currently accessible, so is the form. Note that this means that moving a user will affect forms even retroactively. See also `can_edit_form_location`

Cases: Case accessibility is determined by case owner. If the owner is a user, then the user must be accessible for the case to be accessible. If the owner is a location, then it must be accessible. If the owner is a case-sharing group, the case is not accessible to any restricted users. See also `user_can_access_case`

The SQLLocation queryset method accessible_to_user is helpful when implementing these restrictions. Also refer to the standard reports, which do this sort of filtering in bulk.

18.1.4 Whitelist Implementation

There is LocationAccessMiddleware which controls this whitelist. It intercepts every request, checks if the user has restricted access to the domain, and if so, only allows requests to whitelisted views. This middleware also guarantees that restricted users have a location assigned. That is, if a user should be restricted, but does not have an assigned location, they can't see anything. This is to prevent users from obtaining full access in the event that their location is deleted or improperly assigned.

The other component of this is uitabs. The menu bar and the sidebar on HQ are composed of a bunch of links and names, essentially. We run the url for each of these links against the same check that the middleware uses to see if it should be visible to the user. In this way, they only see menu and sidebar links that are accessible.

To mark a view as location safe, you apply the @location_safe decorator to it. This can be applied directly to view functions, view classes, HQ report classes, or tastypie resources (see implentation and existing usages for examples).

UCR and Report Builder reports will be automatically marked as location safe if the report contains a location choice provider. This is done using the conditionally_location_safe decorator, which is provided with a function that in this case checks that the report has at least one location choice provider.

When marking a view as location safe, you must also check for restricted users by using either request.can_access_all_locations or user.has_permission(domain, 'access_all_locations') and limit the data returned accordingly.

You should create a user who is restricted and click through the desired workflow to make sure it still makes sense, there could be for instance, ajax requests that must also be protected, or links to features the user shouldn't see.

CHAPTER

NINETEEN

REPORTING

A report is
a logical grouping of indicators with common config options (filters etc)

The way reports are produced in CommCare is still evolving so there are a number of different frameworks and methods for generating reports. Some of these are *legacy* frameworks and should not be used for any future reports.

19.1 Recommended approaches for building reports

Things to keep in mind:

- report API
- sqlagg
- couchdbkit-aggregate (legacy)

19.1.1 Example Custom Report Scaffolding

```
class MyBasicReport(GenericTabularReport, CustomProjectReport):
    name = "My Basic Report"
    slug = "my_basic_report"
    fields = ('corehq.apps.reports.filters.dates.DatespanFilter',)

    @property
    def headers(self):
        return DataTablesHeader(DataTablesColumn("Col A"),
                                DataTablesColumnGroup(
                                    "Group 1",
                                    DataTablesColumn("Col B"),
                                    DataTablesColumn("Col C")),
                                DataTablesColumn("Col D"))

    @property
    def rows(self):
        return [
            ['Row 1', 2, 3, 4],
            ['Row 2', 3, 2, 1]
        ]
```

19.2 Hooking up reports to CommCare HQ

Custom reports can be configured in code or in the database. To configure custom reports in code follow the following instructions.

First, you must add the app to *HQ_APPS* in *settings.py*. It must have an *__init__.py* and a *models.py* for django to recognize it as an app.

Next, add a mapping for your domain(s) to the custom reports module root to the *DOMAIN_MODULE_MAP* variable in *settings.py*.

Finally, add a mapping to your custom reports to *__init__.py* in your custom reports submodule:

```
from myproject import reports

CUSTOM_REPORTS = (
    ('Custom Reports', (
        reports.MyCustomReport,
        reports.AnotherCustomReport,
    )),
)
```

19.3 Reporting on data stored in SQL

As described above there are various ways of getting reporting data into and SQL database. From there we can query the data in a number of ways.

19.3.1 Extending the `SqlData` class

The `SqlData` class allows you to define how to query the data in a declarative manner by breaking down a query into a number of components.

class corehq.apps.reports.sqlreport.**SqlData**(*config=None*)

property columns

Returns a list of Column objects. These are used to make up the from portion of the SQL query.

property distinct_on

Returns a list of column names to create the DISTINCT ON portion of the SQL query

property filter_values

Return a dict mapping the filter keys to actual values e.g. {“enddate”: date(2013, 1, 1)}

property filters

Returns a list of filter statements. Filters are instances of sqlagg.filters.SqlFilter. See the sqlagg.filters module for a list of standard filters.

e.g. [EQ(‘date’, ‘enddate’)]

property group_by

Returns a list of ‘group by’ column names.

property keys

The list of report keys (e.g. users) or None to just display all the data returned from the query. Each value in this list should be a list of the same dimension as the 'group_by' list. If group_by is None then keys must also be None.

These allow you to specify which rows you expect in the output data. Its main use is to add rows for keys that don't exist in the data.

e.g.

group_by = ['region', 'sub_region'] keys = [['region1', 'sub1'], ['region1', 'sub2'] ...]

table_name = None

The name of the table to run the query against.

This approach means you don't write any raw SQL. It also allows you to easily include or exclude columns, format column values and combine values from different query columns into a single report column (e.g. calculate percentages).

In cases where some columns may have different filter values e.g. males vs females, **sqlagg** will handle executing the different queries and combining the results.

This class also implements the `corehq.apps.reports.api.ReportDataSource`.

See *Report API* and sqlagg for more info.

e.g.

```
class DemoReport(SqlTabularReport, CustomProjectReport):
    name = "SQL Demo"
    slug = "sql_demo"
    fields = ('corehq.apps.reports.filters.dates.DatespanFilter',)

    # The columns to include the the 'group by' clause
    group_by = ["user"]

    # The table to run the query against
    table_name = "user_report_data"

    @property
    def filters(self):
        return [
            BETWEEN('date', 'startdate', 'enddate'),
        ]

    @property
    def filter_values(self):
        return {
            "startdate": self.datespan.startdate_param_utc,
            "enddate": self.datespan.enddate_param_utc,
            "male": 'M',
            "female": 'F',
        }

    @property
    def keys(self):
        # would normally be loaded from couch
        return [["user1"], ["user2"], ['user3']]
```

(continues on next page)

(continued from previous page)

```
    @property
    def columns(self):
        return [
            DatabaseColumn("Location", SimpleColumn("user_id"), format_fn=self.username),
            DatabaseColumn("Males", CountColumn("gender"), filters=self.filters+[EQ(
↪'gender', 'male')]),
            DatabaseColumn("Females", CountColumn("gender"), filters=self.filters+[EQ(
↪'gender', 'female')]),
            AggregateColumn(
                "C as percent of D",
                self.calc_percentage,
                [SumColumn("indicator_c"), SumColumn("indicator_d")],
                format_fn=self.format_percent)
        ]

    _usernames = {"user1": "Location1", "user2": "Location2", 'user3': "Location3"}  #␣
↪normally loaded from couch
    def username(self, key):
        return self._usernames[key]

    def calc_percentage(num, denom):
        if isinstance(num, Number) and isinstance(denom, Number):
            if denom != 0:
                return num * 100 / denom
            else:
                return 0
        else:
            return None

    def format_percent(self, value):
        return format_datatables_data("%d%%" % value, value)
```

19.4 Report API

Part of the evolution of the reporting frameworks has been the development of a *report api*. This is essentially just a change in the architecture of reports to separate the data from the display. The data can be produced in various formats but the most common is an list of dicts.

e.g.

```
data = [
  {
    'slug1': 'abc',
    'slug2': 2
  },
  {
    'slug1': 'def',
    'slug2': 1
  }
  ...
]
```

This is implemented by creating a report data source class that extends `corehq.apps.reports.api.ReportDataSource` and overriding the `get_data()` function.

class corehq.apps.reports.api.**ReportDataSource**(*config=None*)

get_data(*start=None*, *limit=None*)

Intention: Override

Parameters
slugs – List of slugs to return for each row. Return all values if slugs = None or [].

Returns
A list of dictionaries mapping slugs to values.

e.g. [{'village': 'Mazu', 'births': 30, 'deaths': 28},{...}]

slugs()

Intention: Override

Returns
A list of available slugs.

These data sources can then be used independently or the CommCare reporting user interface and can also be reused for multiple use cases such as displaying the data in the CommCare UI as a table, displaying it in a map, making it available via HTTP etc.

An extension of this base data source class is the `corehq.apps.reports.sqlreport.SqlData` class which simplifies creating data sources that get data by running an SQL query. See section on *SQL reporting* for more info.

e.g.

```
class CustomReportDataSource(ReportDataSource):
    def get_data(self):
        startdate = self.config['start']
        enddate = self.config['end']

        ...

        return data

config = {'start': date(2013, 1, 1), 'end': date(2013, 5, 1)}
ds = CustomReportDataSource(config)
data = ds.get_data()
```

CHAPTER

TWENTY

REPORTING: MAPS IN HQ

20.1 What is the "Maps Report"?

We now have map-based reports in HQ. The "maps report" is not really a report, in the sense that it does not query or calculate any data on its own. Rather, it's a generic front-end visualization tool that consumes data from some other place… other places such as another (tabular) report, or case/form data (work in progress).

To create a map-based report, you must configure the map report template with specific parameters. These are:

- `data_source` – the backend data source which will power the report (required)
- `display_config` – customizations to the display/behavior of the map itself (optional, but suggested for anything other than quick prototyping)

This is how this configuration actually takes place:

- subclass the map report to provide/generate the config parameters. You should **not** need to subclass any code functionality. This is useful for making a more permanent map configuration, and when the configuration needs to be dynamically generated based on other data or domain config (e.g., for CommTrack)

20.2 Orientation

Abstractly, the map report consumes a table of data from some source. Each row of the table is a geographical feature (point or region). One column is identified as containing the geographical data for the feature. All other columns are arbitrary attributes of that feature that can be visualized on the map. Another column may indicate the name of the feature.

The map report contains, obviously, a map. Features are displayed on the map, and may be styled in a number of ways based on feature attributes. The map also contains a legend generated for the current styling. Below the map is a table showing the raw data. Clicking on a feature or its corresponding row in the table will open a detail popup. The columns shown in the table and the detail popup can be customized.

Attribute data is generally treated as either being numeric data or enumerated data (i.e., belonging to a number of discrete categories). Strings are inherently treated as enum data. Numeric data can be treated as enum data be specifying thresholds: numbers will be mapped to enum 'buckets' between consecutive thresholds (e.g, thresholds of `10`, `20` will create enum categories: `< 10`, `10-20`, `> 20`).

20.3 Styling

Different aspects of a feature's marker on the map can be styled based on its attributes. Currently supported visualizations (you may see these referred to in the code as "display axes" or "display dimensions") are:

- varying the size (numeric data only)
- varying the color/intensity (numeric data (color scale) or enum data (fixed color palette))
- selecting an icon (enum data only)

Size and color may be used concurrently, so one attribute could vary size while another varies the color... this is useful when the size represents an absolute magnitude (e.g., # of pregnancies) while the color represents a ratio (% with complications). Region features (as opposed to point features) only support varying color.

A particular configuration of visualizations (which attributes are mapped to which display axes, and associated styling like scaling, colors, icons, thresholds, etc.) is called a *metric*. A map report can be configured with many different metrics. The user selects one metric at a time for viewing. *Metrics may not correspond to table columns one-to-one*, as a single column may be visualized multiple ways, or in combination with other columns, or not at all (shown in detail popup only). If no metrics are specified, they will be auto-generated from best guesses based on the available columns and data feeding the report.

There are several sample reports that comprehensively demo the potential styling options:

- Demo 1
- Demo 2

See *Display Configuration*

20.4 Data Sources

Set this config on the `data_source` property. It should be a `dict` with the following properties:

- `geo_column` – the column in the returned data that contains the geo point (default: `"geo"`)
- `adapter` – which data adapter to use (one of the choices below)
- extra arguments specific to each data adapter

Note that any report filters in the map report are passed on verbatim to the backing data source.

One column of the data returned by the data source must be the geodata (in `geo_column`). For point features, this can be in the format of a geopoint xform question (e.g, `42.366 -71.104`). The geodata format for region features is outside the scope of the document.

20.4.1 `report`

Retrieve data from a `ReportDataSource` (the abstract data provider of Simon's new reporting framework – see *Report API*)

Parameters:

- `report` – fully qualified name of `ReportDataSource` class
- `report_params` – `dict` of static config parameters for the `ReportDataSource` (optional)

20.4.2 legacyreport

Retrieve data from a GenericTabularReport which has not yet been refactored to use Simon's new framework. *Not ideal* and should only be used for backwards compatibility. Tabular reports tend to return pre-formatted data, while the maps report works best with raw data (for example, it won't know 4% or 30 mg are numeric data, and will instead treat them as text enum values). *Read more*.

Parameters:

- report – fully qualified name of tabular report view class (descends from GenericTabularReport)
- report_params – dict of static config parameters for the ReportDataSource (optional)

20.4.3 case

Pull case data similar to the Case List.

(In the current implementation, you must use the same report filters as on the regular Case List report)

Parameters:

- geo_fetch – a mapping of case types to directives of how to pull geo data for a case of that type. Supported directives:
  - name of case property containing the geopoint data
  - "link:xxx" where xxx is the case type of a linked case; the adapter will then serach that linked case for geo-data based on the directive of the linked case type *(not supported yet)*

  In the absence of any directive, the adapter will first search any linked Location record *(not supported yet)*, then try the gps case property.

20.4.4 csv and geojson

Retrieve static data from a csv or geojson file on the server (only useful for testing/demo– this powers the demo reports, for example).

20.5 Display Configuration

Set this config on the display_config property. It should be a dict with the following properties:

(Whenever 'column' is mentioned, it refers to a column slug as returned by the data adapter)

All properties are optional. The map will attempt sensible defaults.

- name_column – column containing the name of the row; used as the header of the detail popup
- column_titles – a mapping of columns to display titles for each column
- detail_columns – a list of columns to display in the detail popup
- table_columns – a list of columns to display in the data table below the map
- enum_captions – display captions for enumerated values. A dict where each key is a column and each value is another dict mapping enum values to display captions. These enum values reflect the results of any transformations from metrics (including _other, _null, and -).

- `numeric_format` – a mapping of columns to functions that apply the appropriate numerical formatting for that column. Expressed as the body of a function that returns the formatted value (`return` statement required!). The unformatted value is passed to the function as the variable `x`.
- `detail_template` – an underscore.js template to format the content of the detail popup
- `metrics` – define visualization metrics (see *Styling*). An array of metrics, where each metric is a `dict` like so:
  - `auto` – column. Auto-generate a metric for this column with no additional manual input. Uses heuristics to determine best presentation format.

  OR

  - `title` – metric title in sidebar (optional)

  AND one of the following for each visualization property you want to control

  - `size` (static) – set the size of the marker (radius in pixels)
  - `size` (dynamic) – vary the size of the marker dynamically. A dict in the format:
    - `column` – column whose data to vary by
    - `baseline` – value that should correspond to a marker radius of 10px
    - `min` – min marker radius (optional)
    - `max` – max marker radius (optional)
  - `color` (static) – set the marker color (css color value)
  - `color` (dynamic) – vary the color of the marker dynamically. A dict in the format:
    - `column` – column whose data to vary by
    - `categories` – for enumerated data; a mapping of enum values to css color values. Mapping key may also be one of these magic values:
      - `_other`: a catch-all for any value not specified
      - `_null`: matches rows whose value is blank; if absent, such rows will be hidden
    - `colorstops` – for numeric data. Creates a sliding color scale. An array of colorstops, each of the format `[<value>, <css color>]`.
    - `thresholds` – (optional) a helper to convert numerical data into enum data via "buckets". Specify a list of thresholds. Each bucket comprises a range from one threshold up to but not including the next threshold. Values are mapped to the bucket whose range they lie in. The "name" (i.e., enum value) of a bucket is its lower threshold. Values below the lowest threshold are mapped to a special bucket called `"-"`.
  - `icon` (static) – set the marker icon (image url)
  - `icon` (dynamic) – vary the icon of the marker dynamically. A dict in the format:
    - `column` – column whose data to vary by
    - `categories` – as in `color`, a mapping of enum values to icon urls
    - `thresholds` – as in `color`

  `size` and `color` may be combined (such as one column controlling size while another controls the color). `icon` must be used on its own.

  For date columns, any relevant number in the above config (`thresholds`, `colorstops`, etc.) may be replaced with a date (in ISO format).

20.6 Raw vs. Formatted Data

Consider the difference between raw and formatted data. Numbers may be formatted for readability (`12,345,678`, `62.5%`, `27 units`); enums may be converted to human-friendly captions; null values may be represented as `--` or `n/a`. The maps report works best when it has the raw data and can perform these conversions itself. The main reason is so that it may generate useful legends, which requires the ability to appropriately format values that may never appear in the report data itself.

There are three scenarios of how a data source may provide data:

- *(worst)* only provide formatted data

  Maps report cannot distinguish numbers from strings from nulls. Data visualizations will not be useful.

- *(sub-optimal)* provide both raw and formatted data (most likely via the `legacyreport` adapter)

  Formatted data will be shown to the user, but maps report will not know how to format data for display in legends, nor will it know all possible values for an enum field – only those that appear in the data.

- *(best)* provide raw data, and explicitly define enum lists and formatting functions in the report config

CHAPTER

TWENTYONE

EXPORTS

Docs in corehq/apps/export/README.md

CHAPTER

TWENTYTWO

CHANGE FEEDS

The following describes our approach to change feeds on HQ. For related content see this presentation on the topic though be advised the presentation was last updated in 2015 and is somewhat out of date.

22.1 What they are

A change feed is modeled after the CouchDB `_changes` feed. It can be thought of as a real-time log of "changes" to our database. Anything that creates such a log is called a "(change) publisher".

Other processes can listen to a change feed and then do something with the results. Processes that listen to changes are called "subscribers". In the HQ codebase "subscribers" are referred to as "pillows" and most of the change feed functionality is provided via the pillowtop module. This document refers to pillows and subscribers interchangeably.

Common use cases for change subscribers:

- **ETL (our main use case)**
  - Saving docs to ElasticSearch
  - Custom report tables
  - UCR data sources
- Cache invalidation

22.2 Architecture

We use kafka as our primary back-end to facilitate change feeds. This allows us to decouple our subscribers from the underlying source of changes so that they can be database-agnostic. For legacy reasons there are still change feeds that run off of CouchDB's `_changes` feed however these are in the process of being phased out.

22.2.1 Topics

Topics are a kafka concept that are used to create logical groups (or "topics") of data. In the HQ codebase we use topics primarily as a 1:N mapping to HQ document classes (or `doc_type` s). Forms and cases currently have their own topics, while everything else is lumped in to a "meta" topic. This allows certain pillows to subscribe to the exact category of change/data they are interested in (e.g. a pillow that sends cases to elasticsearch would only subscribe to the "cases" topic).

22.2.2 Document Stores

Published changes are just "stubs" but do not contain the full data that was affected. Each change should be associated with a "document store" which is an abstraction that represents a way to retrieve the document from its original database. This allows the subscribers to retrieve the full document while not needing to have the underlying source hard-coded (so that it can be changed). To add a new document store, you can use one of the existing subclasses of `DocumentStore` or roll your own.

22.3 Publishing changes

Publishing changes is the act of putting them into kafka from somewhere else.

22.3.1 From Couch

Publishing changes from couch is easy since couch already has a great change feed implementation with the `_changes` API. For any database that you want to publish changes from the steps are very simple. Just create a `ConstructedPillow` with a `CouchChangeFeed` feed pointed at the database you wish to publish from and a `KafkaProcessor` to publish the changes. There is a utility function (`get_change_feed_pillow_for_db`) which creates this pillow object for you.

22.3.2 From SQL

Currently SQL-based change feeds are published from the app layer. Basically, you can just call a function that publishes the change in a `.save()` function (or a `post_save` signal). See the functions in form_processors.change_publishers and their usages for an example of how that's done.

It is planned (though unclear on what timeline) to find an option to publish changes directly from SQL to kafka to avoid race conditions and other issues with doing it at the app layer. However, this change can be rolled out independently at any time in the future with (hopefully) zero impact to change subscribers.

22.3.3 From anywhere else

There is not yet a need/precedent for publishing changes from anywhere else, but it can always be done at the app layer.

22.4 Subscribing to changes

It is recommended that all new change subscribers be instances (or subclasses) of `ConstructedPillow`. You can use the `KafkaChangeFeed` object as the change provider for that pillow, and configure it to subscribe to one or more topics. Look at usages of the `ConstructedPillow` class for examples on how this is done.

22.5 Porting a new pillow

Porting a new pillow to kafka will typically involve the following steps. Depending on the data being published, some of these may be able to be skipped (e.g. if there is already a publisher for the source data, then that can be skipped).

1. Setup a publisher, following the instructions above.
2. Setup a subscriber, following the instructions above.
3. For non-couch-based data sources, you must setup a `DocumentStore` class for the pillow, and include it in the published feed.
4. For any pillows that require additional bootstrap logic (e.g. setting up UCR data tables or bootstrapping elasticsearch indexes) this must be hooked up manually.

22.6 Mapping the above to CommCare-specific details

22.6.1 Topics

The list of topics used by CommCare can be found in corehq.apps.change_feed.topics.py. For most data models there is a 1:1 relationship between the data model and the model in CommCare HQ, with the exceptions of forms and cases, which each have two topics - one for the legacy CouchDB-based forms/cases, and one for the SQL-based models (suffixed by `-sql`).

22.6.2 Contents of the feed

Generally the contents of each change in the feed will documents that mirror the `ChangeMeta` class in pillowtop.feed.interface, in the form of a serialized JSON dictionary. An example once deserialized might look something like this:

```
{
  "document_id": "95dece4cd7c945ec83c6d2dd04d38673",
  "data_source_type": "sql",
  "data_source_name": "form-sql",
  "document_type": "XFormInstance",
  "document_subtype": "http://commcarehq.org/case",
  "domain": "dimagi",
  "is_deletion": false,
  "document_rev": null,
  "publish_timestamp": "2019-09-18T14:31:01.930921Z",
  "attempts": 0
}
```

Details on how to interpret these can be found in the comments of the linked class.

The *document_id*, along with the *document_type* and *data_source_type* should be sufficient to retrieve the underlying raw document out from the feed from the Document Store (see above).

CHAPTER

TWENTYTHREE

PILLOWS

23.1 Overview

23.1.1 What are pillows

Pillows are a component of the publisher/subscriber design pattern that is used for asynchronous communication.

A pillow subscribes to a change feed, and when changes are received, performs specific operations related to that change.

23.1.2 Why do we need pillows

In CommCare HQ, pillows are primarily used to update secondary databases like Elasticsearch and User Configurable Reports (UCRs). Examples of other use cases are invalidating cache or checking if alerts need to be sent.

23.1.3 How do pillows receive changes

We use Kafka as our message queue, which allows producers to publish changes to the queue, and consumers (i.e. pillows) to listen for and process those changes.

Kafka uses _topics_ to organize related changes, and pillows can listen for changes to one or more specific topics.

23.1.4 Why the name

Pillows, as part of the pillowtop framework, were created by us to consume and process changes from the CouchDB change feed. Our usage of pillows has since expanded beyond CouchDB.

23.2 Deconstructing a Pillow

All pillows inherit from the *ConstructedPillow* class. A pillow consists of a few parts:

1. Change Feed
2. Checkpoint
3. Processor(s)
4. Change Event Handler

23.2.1 Change Feed

The brief overview is that a change feed publishes changes which a pillow can subscribe to. When setting up a pillow, an instance of a *ChangeFeed* class is created and configured to only contain changes the pillow cares about.

For more information about change feeds, see *Change Feeds*.

23.2.2 Checkpoint

The checkpoint is a json field that tells processor where to start the change feed.

23.2.3 Processors

A processor is a method that operates on the incoming change. Historically, we had one processor per pillow, however we have since shifted to favor multiple processors for each pillow. This way, all processors can operate on the change which ensures all operations relevant for a change happen within relatively the same time window.

When creating a processor you should be aware of how much time it will take to process the record. A useful baseline is:

86400 seconds per day / # of expected changes per day = how long your processor should take

Note that it should be faster than this as most changes will come in at once instead of evenly distributed throughout the day.

23.2.4 Change Event Handler

This fires after each change has been processed. The main use case is to save the checkpoint to the database.

23.3 Error Handling

23.3.1 Errors

Pillows can fail to process a change for a number of reasons. The most common causes of pillow errors are a code bug, or a failure in a dependent service (e.g., attempting to save a change to Elasticsearch but it is unreachable).

Errors encountered in processors are handled by creating an instance of the *PillowError* database model.

23.3.2 Retries

The *run_pillow_retry_queue* command is configured to run continuously in a celery queue, and looks for new *PillowError* objects to retry. A pillow has the option to disable retrying errors via the *retry_errors* property.

If the related pillow reads from a Kafka change feed, the change associated with the error is re-published into Kafka. However if it reads from a Couch change feed, the pillow's processor is called directly with the change passed in. In both cases, the *PillowError* is deleted, a new one will be created if it fails again.

23.4 Monitoring

There are several datadog metrics with the prefix *commcare.change_feed* that can be helpful for monitoring pillows. Generally these metrics will have tags for pillow name, topic, and partition to filter on.

Metric (not including commcare.change_feed)	Description
change_lag	The current time - when the last change processed was put into the queue
changes.count	Number of changes processed
changes.success	Number of changes processed successfully
changes.exceptions	Number of changes processed with an exception
processor.timing	Time spent in processing a document. Different tags for extract/transform/load steps.
processed_offsets	Latest offset that has been processed by the pillow
current_offsets	The current offsets of each partition in kafka (useful for math in dashboards)
need_processing	current_offsets - processed_offsets

Generally when planning for pillows, you should:

- **Minimize change_lag**
  - ensures changes are processed in a reasonable time (e.g., up to date reports for users)
- **Minimize changes.exceptions**
  - ensures consistency across application (e.g., secondary databases contain accurate data)
  - more exceptions mean more load since they will be reprocessed at a later time
- **Minimize number of pillows running**
  - minimizes server resources required

The ideal setup would have 1 pillow with no exceptions and 0 second lag.

23.5 Troubleshooting

23.5.1 A pillow is falling behind

Otherwise known as "pillow lag", a pillow can fall behind for a few reasons:

1. The processor is too slow for the number of changes that are coming in.
2. There was an issue with the change feed that caused the checkpoint to be "rewound".
3. A processor continues to fail so changes are re-queued and processed again later.

Lag is inherent to asynchronous change processing, so the question is what amount of lag is acceptable for users.

Optimizing a processor

To solve #1 you should use any monitors that have been set up to attempt to pinpoint the issue. *commcare.change_feed.processor.timing* can help determine what processors/pillows are the root cause of slow processing.

If this is a UCR pillow use the *profile_data_source* management command to profile the expensive data sources.

Parallel Processors

To scale pillows horizontally do the following:

1. Look for what pillows are behind. This can be found in the change feed dashboard or the hq admin system info page.
2. Ensure you have enough resources on the pillow server to scale the pillows. This can be found through datadog.
3. Decide what topics need to have added partitions in kafka. There is no way to scale a couch pillow horizontally. Removing partitions isn't straightforward, so you should attempt scaling in small increments. Also make sure pillows are able to split partitions easily by using powers of 2.
4. Run *./manage.py add_kafka_partition <topic> <number partitions to have>*
5. In the commcare-cloud repo environments/<env>/app-processes.yml file change num_processes to the pillows you want to scale.
6. On the next deploy multiple processes will be used when starting pillows

Note that pillows will automatically divide up partitions based on the number of partitions and the number of processes for the pillow. It doesn't have to be one to one, and you don't have to specify the mapping manually. That means you can create more partitions than you need without changing the number of pillow processes and just restart pillows for the change to take effect. Later you can just change the number of processes without touching the number of partitions, and and just update the supervisor conf and restarting pillows for the change to take effect.

The UCR pillows also have options to split the pillow into multiple. They include *ucr_divsion*, *include_ucrs* and *exclude_ucrs*. Look to the pillow code for more information on these.

Rewound Checkpoint

Occasionally checkpoints will be "rewound" to a previous state causing pillows to process changes that have already been processed. This usually happens when a couch node fails over to another. If this occurs, stop the pillow, wait for confirmation that the couch nodes are up, and fix the checkpoint using: *./manage.py fix_checkpoint_after_rewind <pillow_name>*

Many pillow exceptions

commcare.change_feed.changes.exceptions has tag *exception_type* that reports the name and path of the exception encountered. These exceptions could be from coding errors or from infrastructure issues. If they are from infrastructure issues (e.g. ES timeouts) some solutions could be:

- Scale ES cluster (more nodes, shards, etc)
- Reduce number of pillow processes that are writing to ES
- Reduce other usages of ES if possible (e.g. if some custom code relies on ES, could it use UCRs, https://github.com/dimagi/commcare-hq/pull/26241)

23.5.2 Problem with checkpoint for pillow name: First available topic offset for topic is num1 but needed num2

This happens when the earliest checkpoint that kafka knows about for a topic is after the checkpoint the pillow wants to start at. This often happens if a pillow has been stopped for a month and has not been removed from the settings.

To fix this you should verify that the pillow is no longer needed in the environment. If it isn't, you can delete the checkpoint and re-deploy. This should eventually be followed up by removing the pillow from the settings.

If the pillow is needed and should be running you're in a bit of a pickle. This means that the pillow is not able to get the required document ids from kafka. It also won't be clear what documents the pillows has and has not processed. To fix this the safest thing will be to force the pillow to go through all relevant docs. Once this process is started you can move the checkpoint for that pillow to the most recent offset for its topic.

23.6 Pillows

corehq.pillows.case.**get_case_pillow**(*pillow_id='case-pillow'*, *ucr_division=None*, *include_ucrs=None*, *exclude_ucrs=None*, *num_processes=1*, *process_num=0*, *ucr_configs=None*, *skip_ucr=False*, *processor_chunk_size=10*, *topics=None*, *dedicated_migration_process=False*, ***kwargs*)

Return a pillow that processes cases. The processors include, UCR and elastic processors

Processors:

- *corehq.apps.userreports.pillow.ConfigurableReportPillowProcessor* (disabled when skip_ucr=True)
- *pillowtop.processors.elastic.BulkElasticProcessor*
- *corehq.pillows.case_search.get_case_search_processor()*
- *corehq.messaging.pillow.CaseMessagingSyncProcessor*

corehq.pillows.xform.**get_xform_pillow**(*pillow_id='xform-pillow'*, *ucr_division=None*, *include_ucrs=None*, *exclude_ucrs=None*, *num_processes=1*, *process_num=0*, *ucr_configs=None*, *skip_ucr=False*, *processor_chunk_size=10*, *topics=None*, *dedicated_migration_process=False*, ***kwargs*)

Generic XForm change processor

Processors:

- ***corehq.apps.userreports.pillow.ConfigurableReportPillowProcessor***
  - (disabled when skip_ucr=True)
- *pillowtop.processors.elastic.BulkElasticProcessor*
- ***corehq.pillows.user.UnknownUsersProcessor***
  - (disabled when RUN_UNKNOWN_USER_PILLOW=False)
- **pillowtop.form.FormSubmissionMetadataTrackerProcessor**
  - (disabled when RUN_FORM_META_PILLOW=False)
- corehq.apps.data_interfaces.pillow.CaseDeduplicationPillow`

corehq.pillows.case.**get_case_to_elasticsearch_pillow**(*pillow_id='CaseToElasticsearchPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Return a pillow that processes cases to Elasticsearch.

Processors:

- *pillowtop.processors.elastic.ElasticProcessor*

corehq.pillows.xform.**get_xform_to_elasticsearch_pillow**(*pillow_id='XFormToElasticsearchPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

XForm change processor that sends form data to Elasticsearch

Processors:

- *pillowtop.processors.elastic.ElasticProcessor*

corehq.pillows.user.**get_user_pillow**(*pillow_id='user-pillow'*, *num_processes=1*, *dedicated_migration_process=False*, *process_num=0*, *skip_ucr=False*, *processor_chunk_size=10*, ***kwargs*)

Processes users and sends them to ES and UCRs.

Processors:

- *pillowtop.processors.elastic.BulkElasticProcessor()*
- *corehq.apps.userreports.pillow.ConfigurableReportPillowProcessor()*

corehq.pillows.user.**get_user_pillow_old**(*pillow_id='UserPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Processes users and sends them to ES.

Processors:

- *pillowtop.processors.elastic.ElasticProcessor()*

corehq.apps.userreports.pillow.**get_location_pillow**(*pillow_id='location-ucr-pillow'*, *include_ucrs=None*, *num_processes=1*, *process_num=0*, *ucr_configs=None*, ***kwargs*)

Processes updates to locations for UCR

Note this is only applicable if a domain on the environment has *LOCATIONS_IN_UCR* flag enabled.

Processors:

- *corehq.apps.userreports.pillow.ConfigurableReportPillowProcessor()*

corehq.pillows.groups_to_user.**get_group_pillow**(*pillow_id='group-pillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Group pillow

Processors:

- *corehq.pillows.groups_to_user.GroupsToUsersProcessor*
- *corehq.pillows.group.get_group_to_elasticsearch_processor()*

corehq.pillows.group.**get_group_pillow_old**(*pillow_id='GroupPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Group pillow (old). Sends Group data to Elasticsearch

Processors:

- *corehq.pillows.group.get_group_to_elasticsearch_processor*

corehq.pillows.groups_to_user.**get_group_to_user_pillow**(*pillow_id='GroupToUserPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Group pillow that updates user data in Elasticsearch with group membership

Processors:

- *corehq.pillows.groups_to_user.GroupsToUsersProcessor*

corehq.pillows.ledger.**get_ledger_to_elasticsearch_pillow**(*pillow_id='LedgerToElasticsearchPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Ledger pillow

Note that this pillow's id references Elasticsearch, but it no longer saves to ES. It has been kept to keep the checkpoint consistent, and can be changed at any time.

Processors:

- *corehq.pillows.ledger.LedgerProcessor*

corehq.pillows.domain.**get_domain_kafka_to_elasticsearch_pillow**(*pillow_id='KafkaDomainPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Domain pillow to replicate documents to ES

Processors:

- *pillowtop.processors.elastic.ElasticProcessor*

corehq.pillows.sms.**get_sql_sms_pillow**(*pillow_id='SqlSMSPillow'*, *num_processes=1*, *process_num=0*, *processor_chunk_size=10*, ***kwargs*)

SMS Pillow

Processors:

- *pillowtop.processors.elastic.BulkElasticProcessor*

corehq.apps.userreports.pillow.**get_kafka_ucr_pillow**(*pillow_id='kafka-ucr-main'*, *ucr_division=None*, *include_ucrs=None*, *exclude_ucrs=None*, *topics=None*, *num_processes=1*, *process_num=0*, *dedicated_migration_process=False*, *processor_chunk_size=10*, ***kwargs*)

UCR pillow that reads from all Kafka topics and writes data into the UCR database tables.

Processors:

- *corehq.apps.userreports.pillow.ConfigurableReportPillowProcessor*

corehq.apps.userreports.pillow.**get_kafka_ucr_static_pillow**(*pillow_id='kafka-ucr-static'*, *ucr_division=None*, *include_ucrs=None*, *exclude_ucrs=None*, *topics=None*, *num_processes=1*, *process_num=0*, *dedicated_migration_process=False*, *processor_chunk_size=10*, ***kwargs*)

UCR pillow that reads from all Kafka topics and writes data into the UCR database tables.

Only processes *static* UCR datasources (configuration lives in the codebase instead of the database).

Processors:

- *corehq.apps.userreports.pillow.ConfigurableReportPillowProcessor*

corehq.pillows.synclog.**get_user_sync_history_pillow**(*pillow_id='UpdateUserSyncHistoryPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Synclog pillow

Processors:

- *corehq.pillows.synclog.UserSyncHistoryProcessor()*

corehq.pillows.application.**get_app_to_elasticsearch_pillow**(*pillow_id='ApplicationToElasticsearchPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

App pillow

Processors:

- *pillowtop.processors.elastic.BulkElasticProcessor*

corehq.pillows.app_submission_tracker.**get_form_submission_metadata_tracker_pillow**(*pillow_id='FormSubmissior num_processes=1*, *process_num=0*, ***kwargs*)

This gets a pillow which iterates through all forms and marks the corresponding app as having submissions.

Processors:

- *pillowtop.processors.form.FormSubmissionMetadataTrackerProcessor*

corehq.pillows.user.**get_unknown_users_pillow**(*pillow_id='unknown-users-pillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

This pillow adds users from xform submissions that come in to the User Index if they don't exist in HQ

Processors:

- *corehq.pillows.user.UnknownUsersProcessor*

corehq.messaging.pillow.**get_case_messaging_sync_pillow**(*pillow_id='case_messaging_sync_pillow'*, *topics=None*, *num_processes=1*, *process_num=0*, *processor_chunk_size=10*, ***kwargs*)

Pillow for synchronizing messaging data with case data.

Processors:

- *corehq.messaging.pillow.CaseMessagingSyncProcessor*

corehq.pillows.case_search.**get_case_search_to_elasticsearch_pillow**(*pillow_id='CaseSearchToElasticsearchPillow'*, *num_processes=1*, *process_num=0*, ***kwargs*)

Populates the *case search* Elasticsearch index.

Processors:

- corehq.pillows.case_search.CaseSearchPillowProcessor

corehq.pillows.cacheinvalidate.**_get_cache_invalidation_pillow**(*pillow_id*, *couch_db*, *couch_filter=None*)

Pillow that listens to changes and invalidates the cache whether it's a single doc being cached or a view.

Processors:

- corehq.pillows.cache_invalidate_pillow.CacheInvalidateProcessor

corehq.apps.change_feed.pillow.**get_change_feed_pillow_for_db**(*pillow_id*, *couch_db*, *default_topic=None*)

Generic pillow for inserting Couch documents into Kafka.

Reads from:

- CouchDB

Writes to:

- Kafka

23.7 Processors

class corehq.pillows.user.**UnknownUsersProcessor**

Monitors forms for user_ids we don't know about and creates an entry in ES for the user.

Reads from:

- Kafka topics: form-sql, form
- XForm data source

Writes to:

- UserES index

class corehq.apps.change_feed.pillow.**KafkaProcessor**(*data_source_type*, *data_source_name*, *default_topic*)

Generic processor for CouchDB changes to put those changes in a kafka topic

Reads from:

- CouchDB change feed

Writes to:

- Specified kafka topic
- DeletedCouchDoc SQL table

class corehq.pillows.groups_to_user.**GroupsToUsersProcessor**

When a group changes, this updates the user doc in UserES

Reads from:

- Kafka topics: group
- Group data source (CouchDB)

Writes to:

- UserES index

corehq.pillows.group.**get_group_to_elasticsearch_processor**()

Inserts group changes into ES

Reads from:

- Kafka topics: group

- Group data source (CouchDB)

Writes to:

- GroupES index

class corehq.pillows.ledger.**LedgerProcessor**

Updates ledger section and entry combinations (exports), daily consumption and case location ids

Reads from:

- Kafka topics: ledger
- Ledger data source

Writes to:

- LedgerSectionEntry postgres table
- Ledger data source

class corehq.pillows.cacheinvalidate.**CacheInvalidateProcessor**

Invalidates cached CouchDB documents

Reads from:

- CouchDB

Writes to:

- Redis

class corehq.pillows.synclog.**UserSyncHistoryProcessor**

Updates the user document with reporting metadata when a user syncs

Note when USER_REPORTING_METADATA_BATCH_ENABLED is True that this is written to a postgres table. Entries in that table are then batched and processed separately.

Reads from:

- CouchDB (user)
- SynclogSQL table

Writes to:

- CouchDB (user) (when batch processing disabled) (default)
- UserReportingMetadataStaging (SQL) (when batch processing enabled)

class pillowtop.processors.form.**FormSubmissionMetadataTrackerProcessor**

Updates the user document with reporting metadata when a user submits a form

Also marks the application as having submissions.

Note when USER_REPORTING_METADATA_BATCH_ENABLED is True that this is written to a postgres table. Entries in that table are then batched and processed separately

Reads from:

- CouchDB (user and app)
- XForm data source

Writes to:

- CouchDB (app)

- CouchDB (user) (when batch processing disabled) (default)
- UserReportingMetadataStaging (SQL) (when batch processing enabled)

class corehq.apps.userreports.pillow.**ConfigurableReportPillowProcessor**(*table_manager*)

Generic processor for UCR.

Reads from:

- SQLLocation
- Form data source
- Case data source

Writes to:

- UCR database

class pillowtop.processors.elastic.**ElasticProcessor**(*adapter*, *doc_filter_fn=None*, *change_filter_fn=None*)

Generic processor to transform documents and insert into ES.

Processes one document at a time.

Reads from:

- Usually Couch
- Sometimes SQL

Writes to:

- ES

class pillowtop.processors.elastic.**BulkElasticProcessor**(*adapter*, *doc_filter_fn=None*, *change_filter_fn=None*)

Generic processor to transform documents and insert into ES.

Processes one "chunk" of changes at a time (chunk size specified by pillow).

Reads from:

- Usually Couch
- Sometimes SQL

Writes to:

- ES

corehq.pillows.case_search.**get_case_search_processor**()

Case Search

Reads from:

- Case data source

Writes to:

- Case Search ES index

class corehq.messaging.pillow.**CaseMessagingSyncProcessor**

Reads from:

- Case data source

- Update Rules

Writes to:

- PhoneNumber
- Runs rules for SMS (can be many different things)

CHAPTER

TWENTYFOUR

MONITORING EMAIL EVENTS WITH AMAZON SES

If you are using Amazon SES as your email provider (through SMTP), you can monitor what happens to emails sent through commcare's messaging features (broadcasts, reminders, etc).

We use Amazon's Simple Notification System to send callbacks to the */log_email_event* endpoint.

1. Add *SES_CONFIGURATION_SET* to localsettings. Call this something memorable e.g. *production-email-events*. You'll use this name later. Deploy localsettings, and restart services (this needs to be done before the next steps). Also add a *SNS_EMAIL_EVENT_SECRET*, which should be treated like a password, and should be environment specific.
2. Create an SNS Topic here https://console.aws.amazon.com/sns/v3/home?region=us-east-1#/topics .
3. Add a subscription which points to *https://{HQ_ADDRESS}/log_email_event/{SNS_EMAIL_EVENT_SECRET}*. Where the secret you created in step 1 should be added at the end of the address. This should automatically get confirmed. If it doesn't, ensure there is no firewall or something else blocking access to this endpoint.
4. Create an SES Configuration Set, with the name you created in step 1.
5. Add the SNS topic you created in step 2 as the desination for this configuration step. Select the event types you want - we currently support *Send*, *Delivery*, and *Bounce*.
6. Messages you send with the X-COMMCAREHQ-MESSAGE-ID and X-SES-CONFIGURATION-SET headers should now receive notification updates. The X-COMMCAREHQ-MESSAGE-ID headers should include the ID of a MessagingSubEvent.

CHAPTER

TWENTYFIVE

USER CONFIGURABLE REPORTING

An overview of the design, API and data structures used here.

The docs on reporting, pillows, and change feeds, are useful background.

25.1 Data Flow

Reporting is handled in multiple stages. Here is the basic workflow.

Raw data (form or case) → [Data source config] → Row in database table → [Report config] → Report in HQ

Both the data source config and report config are JSON documents that live in the database. The data source config determines how raw data (forms and cases) gets mapped to rows in an intermediary table, while the report config(s) determine how that report table gets turned into an interactive report in the UI.

A UCR table is created when a new data source is created. The table's structure is updated whenever the UCR is "rebuilt", which happens when the data source config is edited. Rebuilds can also be kicked off manually via either `rebuild_indicator_table` or the UI. Rebuilding happens asynchronously. Data in the table is refreshed continuously by pillows.

25.2 Data Sources

Each data source configuration maps a filtered set of the raw data to indicators. A data source configuration consists of two primary sections:

1. A filter that determines whether the data is relevant for the data source
2. A list of indicators in that data source

In addition to these properties there are a number of relatively self-explanatory fields on a data source such as the `table_id` and `display_name`, and a few more nuanced ones. The full list of available fields is summarized in the following table:

Field	Description
filter	Determines whether the data is relevant for the data source
indicators	List of indicators to save
table_id	A unique ID for the table
display_name	A display name for the table that shows up in UIs
base_item_expression	Used for making tables off of repeat or list data
named_expressions	A list of named expressions that can be referenced in other filters and indicators
named_filters	A list of named filters that can be referenced in other filters and indicators

25.2.1 Data Source Filtering

When setting up a data source configuration, filtering defines what data applies to a given set of indicators. Some example uses of filtering on a data source include:

- Restricting the data by document type (e.g. cases or forms). This is a built-in filter.
- Limiting the data to a particular case or form type
- Excluding demo user data
- Excluding closed cases
- Only showing data that meets a domain-specific condition (e.g. pregnancy cases opened for women over 30 years of age)

Filter type overview

There are currently four supported filter types. However, these can be used together to produce arbitrarily complicated expressions.

Filter Type	Description
boolean_expression	A expression / logic statement (more below)
and	An "and" of other filters - true if all are true
or	An "or" of other filters - true if any are true
not	A "not" or inverse expression on a filter

To understand the `boolean_expression` type, we must first explain expressions.

Expressions

An *expression* is a way of representing a set of operations that should return a single value. Expressions can basically be thought of as functions that take in a document and return a value:

Expression: `function(document)` → `value`

In normal math/python notation, the following are all valid expressions on a `doc` (which is presumed to be a `dict` object:

- `"hello"`
- `7`
- `doc["name"]`
- `doc["child"]["age"]`
- `doc["age"] < 21`
- `"legal" if doc["age"] > 21 else "underage"`

In user configurable reports the following expression types are currently supported (note that this can and likely will be extended in the future):

Expression Type	Description	Example
identity	Just returns whatever is passed in	`doc`
constant	A constant	`"hello"`, `4`, `2014-12-20`
property_name	A reference to the property in a document	`doc["name"]`
property_path	A nested reference to a property in a document	`doc["child"]["age"]`
conditional	An if/else expression	`"legal" if doc["age"] > 21 else "minor"`
switch	A switch statement	`if doc["age"] == 21: "legal" elif doc["age"] == 60: "senior" else: ...`
array_index	An index into an array	`doc[1]`
split_string	Splitting a string and grabbing a specific element from it by index	`doc["foobar"].split(' ')[0]`
iterator	Combine multiple expressions into a list	`[doc.name, doc.age, doc.gender]`
related_doc	A way to reference something in another document	`form.case.owner_id`
root_doc	A way to reference the root document explicitly (only needed when making a data source from repeat/child data)	`repeat.parent.name`
ancestor_locatio	A way to retrieve the ancestor of a particular type from a location	
nested	A way to chain any two expressions together	`f1(f2(doc))`
dict	A way to emit a dictionary of key/value pairs	`{"name": "test", "value": f(doc)}`
add_days	A way to add days to a date	`my_date + timedelta(days=15)`
add_month	A way to add months to a date	`my_date + relativedelta(months=15)`
month_star	First day in the month of a date	`2015-01-20` -> `2015-01-01`
month_end	Last day in the month of a date	`2015-01-20` -> `2015-01-31`
diff_days	A way to get duration in days between two dates	`(to_date - from-date).days`
evaluator	A way to do arithmetic operations	`a + b*c / d`
base_iterat	Used with `base_item_expression` ` <#saving-multiple-ro ws-per-caseform>`__ - a way to get the current iteration number (starting from 0).	`loop.index`

Following expressions act on a list of objects or a list of lists (for e.g. on a repeat list) and return another list or value. These expressions can be combined to do complex aggregations on list data.

Expression Type	Description	Example
filter_items	Filter a list of items to make a new list	`[1, 2, 3, -1, -2, -3]` -> `[1, 2, 3]` (filter numbers greater than zero)
map_items	Map one list to another list	`[{'name': 'a', sex: 'f'}, {'name': 'b', gender: 'm'}]` -> `['a', 'b']` (list of names from list of child data)
sort_items	Sort a list based on an expression	`[{'name': 'a', age: 5}, {'name': 'b', age: 3}]` -> `[{'name': 'b', age: 3}, {'name': 'a', age: 5}]` (sort child data by age)
reduce_item	Aggregate a list of items into one value	sum on `[1, 2, 3]` -> `6`
flatten	Flatten multiple lists of items into one list	`[[1, 2], [4, 5]]` -> `[1, 2, 4, 5]`

JSON snippets for expressions

Here are JSON snippets for the various expression types. Hopefully they are self-explanatory.

Constant Expression

class corehq.apps.userreports.expressions.specs.**ConstantGetterSpec**(*_obj=None*, ***kwargs*)

There are two formats for constant expressions. The simplified format is simply the constant itself. For example `"hello"`, or `5`.

The complete format is as follows. This expression returns the constant `"hello"`:

```
{
    "type": "constant",
    "constant": "hello"
}
```

Property Name Expression

class corehq.apps.userreports.expressions.specs.**PropertyNameGetterSpec**(*_obj=None*, ***kwargs*)

This expression returns `doc["age"]`:

```
{
    "type": "property_name",
    "property_name": "age"
}
```

An optional `"datatype"` attribute may be specified, which will attempt to cast the property to the given data type. The options are “date”, “datetime”, “string”, “integer”, and “decimal”. If no datatype is specified, “string” will be used.

Property Path Expression

class corehq.apps.userreports.expressions.specs.**PropertyPathGetterSpec**(*_obj=None*, ***kwargs*)

This expression returns `doc["child"]["age"]`:

```
{
    "type": "property_path",
    "property_path": ["child", "age"]
}
```

An optional `"datatype"` attribute may be specified, which will attempt to cast the property to the given data type. The options are "date", "datetime", "string", "integer", and "decimal". If no datatype is specified, "string" will be used.

Jsonpath Expression

class corehq.apps.userreports.expressions.specs.**JsonpathExpressionSpec**(*_obj=None*, ***kwargs*)

This will execute the jsonpath expression against the current doc and emit the result.

```
{
    "type": "jsonpath",
    "jsonpath": "form..case.name",
}
```

Given the following doc:

```
{
    "form": {
        "case": {"name": "a"},
        "nested": {
            "case": {"name": "b"},
        },
        "list": [
            {"case": {"name": "c"}},
            {
                "nested": {
                    "case": {"name": "d"}
                }
            }
        ]
    }
}
```

This above expression will evaluate to `["a", "b", "c", "d"]`. Another example is `form.list[0].case.name` which will evaluate to `"c"`.

See also the *jsonpath* evaluator function.

For more information consult the following resources:

- Article by Stefan Goessner
- JSONPath expression syntax
- JSONPath Online Evaluator

Conditional Expression

class corehq.apps.userreports.expressions.specs.**ConditionalExpressionSpec**(*_obj=None*, ***kwargs*)

This expression returns `"legal" if doc["age"] > 21 else "underage"`:

```
{
    "type": "conditional",
    "test": {
        "operator": "gt",
        "expression": {
            "type": "property_name",
            "property_name": "age",
            "datatype": "integer"
        },
        "type": "boolean_expression",
        "property_value": 21
    },
    "expression_if_true": {
        "type": "constant",
        "constant": "legal"
    },
    "expression_if_false": {
        "type": "constant",
        "constant": "underage"
    }
}
```

Note that this expression contains other expressions inside it! This is why expressions are powerful. (It also contains a filter, but we haven't covered those yet - if you find the `"test"` section confusing, keep reading...)

Note also that it's important to make sure that you are comparing values of the same type. In this example, the expression that retrieves the age property from the document also casts the value to an integer. If this datatype is not specified, the expression will compare a string to the `21` value, which will not produce the expected results!

Switch Expression

class corehq.apps.userreports.expressions.specs.**SwitchExpressionSpec**(*_obj=None*, ***kwargs*)

This expression returns the value of the expression for the case that matches the switch on expression. Note that case values may only be strings at this time.

```
{
    "type": "switch",
    "switch_on": {
        "type": "property_name",
        "property_name": "district"
    },
    "cases": {
        "north": {
            "type": "constant",
            "constant": 4000
        },
```

(continues on next page)

(continued from previous page)

```
        "south": {
            "type": "constant",
            "constant": 2500
        },
        "east": {
            "type": "constant",
            "constant": 3300
        },
        "west": {
            "type": "constant",
            "constant": 65
        },
    },
    "default": {
        "type": "constant",
        "constant": 0
    }
}
```

Coalesce Expression

class corehq.apps.userreports.expressions.specs.**CoalesceExpressionSpec**(*_obj=None*, ***kwargs*)

This expression returns the value of the expression provided, or the value of the default_expression if the expression provided evaluates to a null or blank string.

```
{
    "type": "coalesce",
    "expression": {
        "type": "property_name",
        "property_name": "district"
    },
    "default_expression": {
        "type": "constant",
        "constant": "default_district"
    }
}
```

Array Index Expression

class corehq.apps.userreports.expressions.specs.**ArrayIndexExpressionSpec**(*_obj=None*, ***kwargs*)

This expression returns `doc["siblings"][0]`:

```
{
    "type": "array_index",
    "array_expression": {
        "type": "property_name",
        "property_name": "siblings"
    },
```

(continues on next page)

(continued from previous page)

```
    "index_expression": {
        "type": "constant",
        "constant": 0
    }
}
```

It will return nothing if the siblings property is not a list, the index isn't a number, or the indexed item doesn't exist.

Split String Expression

class corehq.apps.userreports.expressions.specs.**SplitStringExpressionSpec**(*_obj=None*, ***kwargs*)

This expression returns `(doc["foo bar"]).split(' ')[0]`:

```
{
    "type": "split_string",
    "string_expression": {
        "type": "property_name",
        "property_name": "multiple_value_string"
    },
    "index_expression": {
        "type": "constant",
        "constant": 0
    },
    "delimiter": ","
}
```

The delimiter is optional and is defaulted to a space. It will return nothing if the string_expression is not a string, or if the index isn't a number or the indexed item doesn't exist. The index_expression is also optional. Without it, the expression will return the list of elements.

Iterator Expression

class corehq.apps.userreports.expressions.specs.**IteratorExpressionSpec**(*_obj=None*, ***kwargs*)

```
{
    "type": "iterator",
    "expressions": [
        {
            "type": "property_name",
            "property_name": "p1"
        },
        {
            "type": "property_name",
            "property_name": "p2"
        },
        {
            "type": "property_name",
            "property_name": "p3"
```

(continues on next page)

(continued from previous page)

```
        },
    ],
    "test": {}
}
```

This will emit `[doc.p1, doc.p2, doc.p3]`. You can add a `test` attribute to filter rows from what is emitted - if you don't specify this then the iterator will include one row per expression it contains regardless of what is passed in. This can be used/combined with the `base_item_expression` to emit multiple rows per document.

Base iteration number expressions

class corehq.apps.userreports.expressions.specs.**IterationNumberExpressionSpec**(*_obj=None*, ***kwargs*)

These are very simple expressions with no config. They return the index of the repeat item starting from 0 when used with a `base_item_expression`.

```
{
    "type": "base_iteration_number"
}
```

Related document expressions

class corehq.apps.userreports.expressions.specs.**RelatedDocExpressionSpec**(*_obj=None*, ***kwargs*)

This can be used to lookup a property in another document. Here's an example that lets you look up `form.case.owner_id` from a form.

```
{
    "type": "related_doc",
    "related_doc_type": "CommCareCase",
    "doc_id_expression": {
        "type": "property_path",
        "property_path": ["form", "case", "@case_id"]
    },
    "value_expression": {
        "type": "property_name",
        "property_name": "owner_id"
    }
}
```

Ancestor location expression

class corehq.apps.locations.ucr_expressions.**AncestorLocationExpression**(*_obj=None*, ***kwargs*)

This is used to return a json object representing the ancestor of the given type of the given location. For instance, if we had locations configured with a hierarchy like `country -> state -> county -> city`, we could pass the location id of Cambridge and a location type of state to this expression to get the Massachusetts location.

```
{
    "type": "ancestor_location",
    "location_id": {
        "type": "property_name",
        "name": "owner_id"
    },
    "location_type": {
        "type": "constant",
        "constant": "state"
    }
}
```

If no such location exists, returns null.

Optionally you can specifiy `location_property` to return a single property of the location.

```
{
    "type": "ancestor_location",
    "location_id": {
        "type": "property_name",
        "name": "owner_id"
    },
    "location_type": {
        "type": "constant",
        "constant": "state"
    },
    "location_property": "site_code"
}
```

Nested expressions

class corehq.apps.userreports.expressions.specs.**NestedExpressionSpec**(*_obj=None*, ***kwargs*)

These can be used to nest expressions. This can be used, e.g. to pull a specific property out of an item in a list of objects.

The following nested expression is the equivalent of a `property_path` expression to `["outer", "inner"]` and demonstrates the functionality. More examples can be found in the practical examples.

```
{
    "type": "nested",
    "argument_expression": {
        "type": "property_name",
        "property_name": "outer"
    },
    "value_expression": {
        "type": "property_name",
```

(continues on next page)

(continued from previous page)

```
        "property_name": "inner"
    }
}
```

Dict expressions

class corehq.apps.userreports.expressions.specs.**DictExpressionSpec**(*_obj=None*, ***kwargs*)

These can be used to create dictionaries of key/value pairs. This is only useful as an intermediate structure in another expression since the result of the expression is a dictionary that cannot be saved to the database.

See the practical examples for a way this can be used in a `base_item_expression` to emit multiple rows for a single form/case based on different properties.

Here is a simple example that demonstrates the structure. The keys of `properties` must be text, and the values must be valid expressions (or constants):

```
{
    "type": "dict",
    "properties": {
        "name": "a constant name",
        "value": {
            "type": "property_name",
            "property_name": "prop"
        },
        "value2": {
            "type": "property_name",
            "property_name": "prop2"
        }
    }
}
```

"Add Days" expressions

class corehq.apps.userreports.expressions.date_specs.**AddDaysExpressionSpec**(*_obj=None*, ***kwargs*)

Below is a simple example that demonstrates the structure. The expression below will add 28 days to a property called "dob". The date_expression and count_expression can be any valid expressions, or simply constants.

```
{
    "type": "add_days",
    "date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
    "count_expression": 28
}
```

"Add Hours" expressions

class corehq.apps.userreports.expressions.date_specs.**AddHoursExpressionSpec**(*_obj=None*, ***kwargs*)

Below is a simple example that demonstrates the structure. The expression below will add 12 hours to a property called "visit_date". The date_expression and count_expression can be any valid expressions, or simply constants.

```
{
    "type": "add_hours",
    "date_expression": {
        "type": "property_name",
        "property_name": "visit_date",
    },
    "count_expression": 12
}
```

"Add Months" expressions

class corehq.apps.userreports.expressions.date_specs.**AddMonthsExpressionSpec**(*_obj=None*, ***kwargs*)

`add_months` offsets given date by given number of calendar months. If offset results in an invalid day (for e.g. Feb 30, April 31), the day of resulting date will be adjusted to last day of the resulting calendar month.

The date_expression and months_expression can be any valid expressions, or simply constants, including negative numbers.

```
{
    "type": "add_months",
    "date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
    "months_expression": 28
}
```

"Diff Days" expressions

class corehq.apps.userreports.expressions.date_specs.**DiffDaysExpressionSpec**(*_obj=None*, ***kwargs*)

`diff_days` returns number of days between dates specified by `from_date_expression` and `to_date_expression`. The from_date_expression and to_date_expression can be any valid expressions, or simply constants.

```
{
    "type": "diff_days",
    "from_date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
    "to_date_expression": "2016-02-01"
}
```

"Month Start Date" and "Month End Date" expressions

class corehq.apps.userreports.expressions.date_specs.**MonthStartDateExpressionSpec**(*_obj=None*, ***kwargs*)

`month_start_date` returns date of first day in the month of given date and `month_end_date` returns date of last day in the month of given date.

The `date_expression` can be any valid expression, or simply constant

```
{
    "type": "month_start_date",
    "date_expression": {
        "type": "property_name",
        "property_name": "dob",
    },
}
```

"Evaluator" expression

class corehq.apps.userreports.expressions.specs.**EvalExpressionSpec**(*_obj=None*, ***kwargs*)

`evaluator` expression can be used to evaluate statements that contain arithmetic (and simple python like statements). It evaluates the statement specified by `statement` which can contain variables as defined in `context_variables`.

```
{
    "type": "evaluator",
    "statement": "a + b - c + 6",
    "context_variables": {
        "a": 1,
        "b": 20,
        "c": {
            "type": "property_name",
            "property_name": "the_number_two"
        }
    }
}
```

This returns **25** (1 + 20 - 2 + 6).

`statement`

The expression statement to be evaluated.

`context_variables`

A dictionary of Expressions where keys are names of variables used in the `statement` and values are expressions to generate those variables.

Variable types must be one of:

- `str`
- `int`
- `float`
- `bool`

- `date`
- `datetime`

If `context_variables` is omitted, the current context of the expression will be used.

Expression limitations

Only a single expression is permitted.

Available operators:

- math operators (except the power operator)
- modulus
- negation
- comparison operators
- logical operators

In addition, expressions can perform the following operations:

- index: *case['name']*
- slice: *cases[0:2]*
- if statements: *1 if case.name == 'bob' else 0*
- list comprehension: *[i for i in range(3)]*
- dict, list, set construction: *{"a": 1, "b": set(cases), "c": list(range(4))}*

Available Functions

Only the following functions are available in the evaluation context:

`context`()

Get the current evaluation context. See also *root_context()*.

`date`(*value*, *fmt=None*)

Parse a string value as a date or timestamp. If `fmt` is not supplied the string is assumed to be in ISO 8601 format.

Parameters

fmt – If supplied, use this format specification to parse the date. See the Python documentation for Format Codes.

`float`(*value*)

Convert `value` to a floating point number.

`int`(*value*)

Convert `value` to an int. Value can be a number or a string representation of a number.

`jsonpath`(*expr*, *as_list=False*, *context=None*)

Evaluate a jsonpath expression.

See also *Jsonpath Expression*.

```
jsonpath("form.case.name")
jsonpath("name", context=jsonpath("form.case"))
jsonpath("form..case", as_list=True)
```

Parameters

- **expr** – The jsonpath expression.
- **as_list** – When set to True, always return the full list of matches, even if it is emtpy. If set to False then the return value will be *None* if no matches are found. If a single match is found the matched value will be returned. If more than one match is found, they will all be returned as a list.
- **context** – Optional context for evaluation. If not supplied the full context of the evaluator will be used.

Returns

See *as_list*.

named(*name*, *context=None*)

Call a named expression. See also *Named Expressions*.

```
named("my-named-expression")
named("my-named-expression", context=form.case)
```

rand()

Generate a random number between 0 and 1

randint(*max*)

Generate a random integer between 0 and `max`

range(*start*[, *stop*][, *skip*])

Produces a sequence of integers from start (inclusive) to stop (exclusive) by step. Note that for performance reasons this is limited to 100 items or less. See range.

root_context()

Get the root context of the evaluation. Similar to the `root_doc` expression.

See also *context()*.

round(*value*, *ndigits=None*)

Round a number to the nearest integer or `ndigits` after the decimal point. See round.

str(*value*)

Convert `value` to a string.

timedelta_to_seconds(*delta*)

Convert a TimeDelta object into seconds. This is useful for getting the number of seconds between two dates.

```
timedelta_to_seconds(time_end - time_start)
```

today()

Return the current UTC date.

See also *Evaluator Examples*.

'Get Case Sharing Groups' expression

class corehq.apps.userreports.expressions.specs.**CaseSharingGroupsExpressionSpec**(*_obj=None*, ***kwargs*)

`get_case_sharing_groups` will return an array of the case sharing groups that are assigned to a provided user ID. The array will contain one document per case sharing group.

```
{
    "type": "get_case_sharing_groups",
    "user_id_expression": {
        "type": "property_path",
        "property_path": ["form", "meta", "userID"]
    }
}
```

'Get Reporting Groups' expression

class corehq.apps.userreports.expressions.specs.**ReportingGroupsExpressionSpec**(*_obj=None*, ***kwargs*)

`get_reporting_groups` will return an array of the reporting groups that are assigned to a provided user ID. The array will contain one document per reporting group.

```
{
    "type": "get_reporting_groups",
    "user_id_expression": {
        "type": "property_path",
        "property_path": ["form", "meta", "userID"]
    }
}
```

Filter, Sort, Map and Reduce Expressions

We have following expressions that act on a list of objects or list of lists. The list to operate on is specified by `items_expression`. This can be any valid expression that returns a list. If the `items_expression` doesn't return a valid list, these might either fail or return one of empty list or `None` value.

map_items Expression

class corehq.apps.userreports.expressions.list_specs.**MapItemsExpressionSpec**(*_obj=None*, ***kwargs*)

`map_items` performs a calculation specified by `map_expression` on each item of the list specified by `items_expression` and returns a list of the calculation results. The `map_expression` is evaluated relative to each item in the list and not relative to the parent document from which the list is specified. For e.g. if `items_expression` is a path to repeat-list of children in a form document, `map_expression` is a path relative to the repeat item.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`map_expression` can be any valid expression relative to the items in above list.

```
{
    "type": "map_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_path",
        "property_path": ["form", "child_repeat"]
    },
    "map_expression": {
        "type": "property_path",
        "property_path": ["age"]
    }
}
```

Above returns list of ages. Note that the `property_path` in `map_expression` is relative to the repeat item rather than to the form.

filter_items Expression

class corehq.apps.userreports.expressions.list_specs.**FilterItemsExpressionSpec**(*_obj=None*, ***kwargs*)

`filter_items` performs filtering on given list and returns a new list. If the boolean expression specified by `filter_expression` evaluates to a `True` value, the item is included in the new list and if not, is not included in the new list.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`filter_expression` can be any valid boolean expression relative to the items in above list.

```
{
    "type": "filter_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_name",
        "property_name": "family_repeat"
    },
    "filter_expression": {
       "type": "boolean_expression",
        "expression": {
            "type": "property_name",
            "property_name": "gender"
        },
        "operator": "eq",
        "property_value": "female"
    }
}
```

sort_items Expression

class corehq.apps.userreports.expressions.list_specs.**SortItemsExpressionSpec**(*_obj=None*, ***kwargs*)

`sort_items` returns a sorted list of items based on sort value of each item.The sort value of an item is specified by `sort_expression`. By default, list will be in ascending order. Order can be changed by adding optional `order` expression with one of `DESC` (for descending) or `ASC` (for ascending) If a sort-value of an item is `None`, the item will appear in the start of list. If sort-values of any two items can't be compared, an empty list is returned.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`sort_expression` can be any valid expression relative to the items in above list, that returns a value to be used as sort value.

```
{
    "type": "sort_items",
    "items_expression": {
        "datatype": "array",
        "type": "property_path",
        "property_path": ["form", "child_repeat"]
    },
    "sort_expression": {
        "type": "property_path",
        "property_path": ["age"]
    }
}
```

reduce_items Expression

class corehq.apps.userreports.expressions.list_specs.**ReduceItemsExpressionSpec**(*_obj=None*, ***kwargs*)

`reduce_items` returns aggregate value of the list specified by `aggregation_fn`.

`items_expression` can be any valid expression that returns a list. If this doesn't evaluate to a list, `aggregation_fn` will be applied on an empty list. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

`aggregation_fn` is one of following supported functions names.

Function Name	Example
`count`	`['a', 'b']` -> 2
`sum`	`[1, 2, 4]` -> 7
`min`	`[2, 5, 1]` -> 1
`max`	`[2, 5, 1]` -> 5
`first_item`	`['a', 'b']` -> 'a'
`last_item`	`['a', 'b']` -> 'b'
`join`	`['a', 'b']` -> 'ab'

```
{
    "type": "reduce_items",
```

(continues on next page)

(continued from previous page)

```
    "items_expression": {
        "datatype": "array",
        "type": "property_name",
        "property_name": "family_repeat"
    },
    "aggregation_fn": "count"
}
```

This returns number of family members

flatten expression

class corehq.apps.userreports.expressions.list_specs.**FlattenExpressionSpec**(*_obj=None*, ***kwargs*)

`flatten` takes list of list of objects specified by `items_expression` and returns one list of all objects.

`items_expression` is any valid expression that returns a list of lists. It this doesn't evaluate to a list of lists an empty list is returned. It may be necessary to specify a `datatype` of `array` if the expression could return a single element.

```
{
    "type": "flatten",
    "items_expression": {},
}
```

Named Expressions

class corehq.apps.userreports.expressions.specs.**NamedExpressionSpec**(*_obj=None*, ***kwargs*)

Last, but certainly not least, are named expressions. These are special expressions that can be defined once in a data source and then used throughout other filters and indicators in that data source. This allows you to write out a very complicated expression a single time, but still use it in multiple places with a simple syntax.

Named expressions are defined in a special section of the data source. To reference a named expression, you just specify the type of `"named"` and the name as follows:

```
{
    "type": "named",
    "name": "my_expression"
}
```

This assumes that your named expression section of your data source includes a snippet like the following:

```
{
    "my_expression": {
        "type": "property_name",
        "property_name": "test"
    }
}
```

This is just a simple example - the value that `"my_expression"` takes on can be as complicated as you want and it can also reference other named expressions as long as it doesn't reference itself of create a recursive cycle.

See also the *named* evaluator function.

Boolean Expression Filters

A `boolean_expression` filter combines an *expression*, an *operator*, and a *property value* (a constant), to produce a statement that is either `True` or `False`. *Note: in the future the constant value may be replaced with a second expression to be more general, however currently only constant property values are supported.*

Here is a sample JSON format for simple `boolean_expression` filter:

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "age",
        "datatype": "integer"
    },
    "operator": "gt",
    "property_value": 21
}
```

This is equivalent to the python statement: `doc["age"] > 21`

Operators

The following operators are currently supported:

Operator	Description	Value type	Example
`eq`	is equal	constant	`doc["age"] == 21`
`not_eq`	is not equal	constant	`doc["age"] != 21`
`in`	single value is in a list	list	`doc["color"] in ["red", "blue"]`
`in_multi`	a value is in a multi select	list	`selected(doc["color"], "red")`
`any_in_mult`	one of a list of values in in a multiselect	list	`selected(doc["color"], ["red", "blue"])`
`lt`	is less than	number	`doc["age"] < 21`
`lte`	is less than or equal	number	`doc["age"] <= 21`
`gt`	is greater than	number	`doc["age"] > 21`
`gte`	is greater than or equal	number	`doc["age"] >= 21`
`regex`	matches regular expression	string	`re.search("^([Rr]ed|[Bb]lue)$", doc["color"])`

Compound filters

Compound filters build on top of `boolean_expression` filters to create boolean logic. These can be combined to support arbitrarily complicated boolean logic on data. There are three types of filters, *and*, *or*, and *not* filters. The JSON representation of these is below. Hopefully these are self explanatory.

"And" Filters

The following filter represents the statement: `doc["age"] < 21 and doc["nationality"] == "american"`:

```
{
    "type": "and",
    "filters": [
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "age",
                "datatype": "integer"
            },
            "operator": "lt",
            "property_value": 21
        },
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "nationality",
            },
            "operator": "eq",
            "property_value": "american"
        }
    ]
}
```

"Or" Filters

The following filter represents the statement: `doc["age"] > 21 or doc["nationality"] == "european"`:

```
{
    "type": "or",
    "filters": [
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "age",
                "datatype": "integer",
            },
            "operator": "gt",
            "property_value": 21
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "nationality",
            },
            "operator": "eq",
            "property_value": "european"
        }
    ]
}
```

"Not" Filters

The following filter represents the statement: `!(doc["nationality"] == "european")`:

```
{
    "type": "not",
    "filter": [
        {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "nationality",
            },
            "operator": "eq",
            "property_value": "european"
        }
    ]
}
```

Note that this could be represented more simply using a single filter with the ``not_eq`` operator, but "not" filters can represent more complex logic than operators generally, since the filter itself can be another compound filter.

Practical Examples

See practical examples for some practical examples showing various filter types.

25.2.2 Indicators

Now that we know how to filter the data in our data source, we are still left with a very important problem: *how do we know what data to save*? This is where indicators come in. Indicators are the data outputs - what gets computed and put in a column in the database.

A typical data source will include many indicators (data that will later be included in the report). This section will focus on defining a single indicator. Single indicators can then be combined in a list to fully define a data source.

The overall set of possible indicators is theoretically any function that can take in a single document (form or case) and output a value. However the set of indicators that are configurable is more limited than that.

Indicator Properties

All indicator definitions have the following properties:

Property	Description
type	A specified type for the indicator. It must be one of the types listed below.
column_id	The database column where the indicator will be saved.
display_name	A display name for the indicator (not widely used, currently).
comment	A string describing the indicator

Additionally, specific indicator types have other type-specific properties. These are covered below.

Indicator types

The following primary indicator types are supported:

Indicator Type	Description
boolean	Save 1 if a filter is true, otherwise 0.
expression	Save the output of an expression.
choice_list	Save multiple columns, one for each of a predefined set of choices
ledger_balances	Save a column for each product specified, containing ledger data

Note/todo: there are also other supported formats, but they are just shortcuts around the functionality of these ones they are left out of the current docs.

Boolean indicators

Now we see again the power of our filter framework defined above! Boolean indicators take any arbitrarily complicated filter expression and save a 1 to the database if the expression is true, otherwise a 0. Here is an example boolean indicator which will save 1 if a form has a question with ID `is_pregnant` with a value of `"yes"`:

```
{
    "type": "boolean",
    "column_id": "col",
    "filter": {
        "type": "boolean_expression",
        "expression": {
            "type": "property_path",
            "property_path": ["form", "is_pregnant"],
        },
        "operator": "eq",
        "property_value": "yes"
    }
}
```

Expression indicators

Similar to the boolean indicators - expression indicators leverage the expression structure defined above to create arbitrarily complex indicators. Expressions can store arbitrary values from documents (as opposed to boolean indicators which just store `0`'s and `1`'s). Because of this they require a few additional properties in the definition:

Property	Description
datatype	The datatype of the indicator. Current valid choices are: "date", "datetime", "string", "decimal", "integer", and "small_integer".
is_nullable	Whether the database column should allow null values.
is_primary_k	Whether the database column should be (part of?) the primary key. (TODO: this needs to be confirmed)
create_index	Creates an index on this column. Only applicable if using the SQL backend
expression	Any expression.
transform	(optional) transform to be applied to the result of the expression. (see "Report Columns > Transforms" section below)

Here is a sample expression indicator that just saves the "age" property to an integer column in the database:

```
{
    "type": "expression",
    "expression": {
        "type": "property_name",
        "property_name": "age"
    },
    "column_id": "age",
    "datatype": "integer",
    "display_name": "age of patient"
}
```

Choice list indicators

Choice list indicators take a single choice column (select or multiselect) and expand it into multiple columns where each column represents a different choice. These can support both single-select and multi-select quesitons.

A sample spec is below:

```
{
    "type": "choice_list",
    "column_id": "col",
    "display_name": "the category",
    "property_name": "category",
    "choices": [
        "bug",
        "feature",
        "app",
        "schedule"
    ],
    "select_style": "single"
}
```

Ledger Balance Indicators

Ledger Balance indicators take a list of product codes and a ledger section, and produce a column for each product code, saving the value found in the corresponding ledger.

Property	Description
ledger_section	The ledger section to use for this indicator, for example, "stock"
product_codes	A list of the products to include in the indicator. This will be used in conjunction with the `column_id` to produce each column name.
case_id_expres	An expression used to get the case where each ledger is found. If not specified, it will use the row's doc id.

```
{
    "type": "ledger_balances",
    "column_id": "soh",
    "display_name": "Stock On Hand",
    "ledger_section": "stock",
    "product_codes": ["aspirin", "bandaids", "gauze"],
    "case_id_expression": {
        "type": "property_name",
        "property_name": "_id"
    }
}
```

This spec would produce the following columns in the data source:

soh_aspirin	soh_bandaids	soh_gauze
20	11	5
67	32	9

If the ledger you're using is a due list and you wish to save the dates instead of integers, you can change the "type" from "ledger_balances" to "due_list_date".

Practical notes for creating indicators

These are some practical notes for how to choose what indicators to create.

Fractions

All indicators output single values. Though fractional indicators are common, these should be modeled as two separate indicators (for numerator and denominator) and the relationship should be handled in the report UI config layer.

25.2.3 Saving Multiple Rows per Case/Form

You can save multiple rows per case/form by specifying a root level `base_item_expression` that describes how to get the repeat data from the main document. You can also use the `root_doc` expression type to reference parent properties and the `base_iteration_number` expression type to reference the current index of the item. This can be combined with the `iterator` expression type to do complex data source transforms. This is not described in detail, but the following sample (which creates a table off of a repeat element called "time_logs" can be used as a guide). There are also additional examples in the practical examples:

```
{
    "domain": "user-reports",
    "doc_type": "DataSourceConfiguration",
    "referenced_doc_type": "XFormInstance",
    "table_id": "sample-repeat",
    "display_name": "Time Logged",
    "base_item_expression": {
        "type": "property_path",
        "property_path": ["form", "time_logs"]
    },
    "configured_filter": {
    },
    "configured_indicators": [
        {
            "type": "expression",
            "expression": {
                "type": "property_name",
                "property_name": "start_time"
            },
            "column_id": "start_time",
            "datatype": "datetime",
            "display_name": "start time"
        },
        {
            "type": "expression",
            "expression": {
                "type": "property_name",
                "property_name": "end_time"
            },
            "column_id": "end_time",
            "datatype": "datetime",
            "display_name": "end time"
        },
        {
            "type": "expression",
            "expression": {
                "type": "property_name",
                "property_name": "person"
            },
            "column_id": "person",
            "datatype": "string",
            "display_name": "person"
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "type": "expression",
            "expression": {
                "type": "root_doc",
                "expression": {
                    "type": "property_name",
                    "property_name": "name"
                }
            },
            "column_id": "name",
            "datatype": "string",
            "display_name": "name of ticket"
        }
    ]
}
```

25.2.4 Data Cleaning and Validation

Note this is only available for "static" data sources that are created in the HQ repository.

When creating a data source it can be valuable to have strict validation on the type of data that can be inserted. The attribute `validations` at the top level of the configuration can use UCR expressions to determine if the data is invalid. If an expression is deemed invalid, then the relevant error is stored in the `InvalidUCRData` model.

```
{
    "domain": "user-reports",
    "doc_type": "DataSourceConfiguration",
    "referenced_doc_type": "XFormInstance",
    "table_id": "sample-repeat",
    "base_item_expression": {},
    "validations": [{
         "name": "is_starred_valid",
         "error_message": "is_starred has unexpected value",
         "expression": {
             "type": "boolean_expression",
             "expression": {
                 "type": "property_name",
                 "property_name": "is_starred"
             },
             "operator": "in",
             "property_value": ["yes", "no"]
         }
    }],
    "configured_filter": { },
    "configured_indicators": [ ]
}
```

25.3 Report Configurations

A report configuration takes data from a data source and renders it in the UI. A report configuration consists of a few different sections:

1. *Report Filters* - These map to filters that show up in the UI, and should translate to queries that can be made to limit the returned data.
2. *Aggregation* - This defines what each row of the report will be. It is a list of columns forming the *primary key* of each row.
3. *Report Columns* - Columns define the report columns that show up from the data source, as well as any aggregation information needed.
4. *Charts* - Definition of charts to display on the report.
5. *Sort Expression* - How the rows in the report are ordered.
6. *Distinct On* - Pick distinct rows from result based on columns.

25.3.1 Samples

Here are some sample configurations that can be used as a reference until we have better documentation.

- Dimagi chart report
- GSID form report

25.3.2 Report Filters

The documentation for report filters is still in progress. Apologies for brevity below.

A note about report filters versus data source filters

Report filters are *completely* different from data source filters. Data source filters limit the global set of data that ends up in the table, whereas report filters allow you to select values to limit the data returned by a query.

Numeric Filters

Numeric filters allow users to filter the rows in the report by comparing a column to some constant that the user specifies when viewing the report. Numeric filters are only intended to be used with numeric (integer or decimal type) columns. Supported operators are =, , <, , >, and .

ex:

```
{
  "type": "numeric",
  "slug": "number_of_children_slug",
  "field": "number_of_children",
  "display": "Number of Children"
}
```

Date filters

Date filters allow you filter on a date. They will show a datepicker in the UI.

```
{
  "type": "date",
  "slug": "modified_on",
  "field": "modified_on",
  "display": "Modified on",
  "required": false
}
```

Date filters have an optional `compare_as_string` option that allows the date filter to be compared against an indicator of data type `string`. You shouldn't ever need to use this option (make your column a `date` or `datetime` type instead), but it exists because the report builder needs it.

Quarter filters

Quarter filters are similar to date filters, but a choice is restricted only to the particular quarter of the year. They will show inputs for year and quarter in the UI.

```
{
  "type": "quarter",
  "slug": "modified_on",
  "field": "modified_on",
  "display": "Modified on",
  "required": false
}
```

Pre-Filters

Pre-filters offer the kind of functionality you get from *data source filters*. This makes it easier to use one data source for many reports, especially if some of those reports just need the data source to be filtered slightly differently. Pre-filters do not need to be configured by app builders in report modules; fields with pre-filters will not be listed in the report module among the other fields that can be filtered.

A pre-filter's `type` is set to "pre":

```
{
  "type": "pre",
  "field": "at_risk_field",
  "slug": "at_risk_slug",
  "datatype": "string",
  "pre_value": "yes"
}
```

If `pre_value` is scalar (i.e. `datatype` is "string", "integer", etc.), the filter will use the "equals" operator. If `pre_value` is null, the filter will use "is null". If `pre_value` is an array, the filter will use the "in" operator. e.g.

```
{
  "type": "pre",
  "field": "at_risk_field",
  "slug": "at_risk_slug",
```

(continues on next page)

(continued from previous page)

```
  "datatype": "array",
  "pre_value": ["yes", "maybe"]
}
```

(If `pre_value` is an array and `datatype` is not "array", it is assumed that `datatype` refers to the data type of the items in the array.)

You can optionally specify the operator that the prevalue filter uses by adding a pre_operator argument. e.g.

```
{
  "type": "pre",
  "field": "at_risk_field",
  "slug": "at_risk_slug",
  "datatype": "array",
  "pre_value": ["maybe", "yes"],
  "pre_operator": "between"
}
```

Note that instead of using `eq`, `gt`, etc, you will need to use =, >, etc.

Dynamic choice lists

Dynamic choice lists provide a select widget that will generate a list of options dynamically.

The default behavior is simply to show all possible values for a column, however you can also specify a `choice_provider` to customize this behavior (see below).

Simple example assuming "village" is a name:

```
{
  "type": "dynamic_choice_list",
  "slug": "village",
  "field": "village",
  "display": "Village",
  "datatype": "string"
}
```

Choice providers

Currently the supported `choice_providers` are supported:

Field	Description
location	Select a location by name
user	Select a user
owner	Select a possible case owner owner (user, group, or location)

Location choice providers also support three additional configuration options:

- "include_descendants" - Include descendants of the selected locations in the results. Defaults to `false`.
- "show_full_path" - Display the full path to the location in the filter. Defaults to `false`. The default behavior shows all locations as a flat alphabetical list.

- "location_type" - Includes locations of this type only. Default is to not filter on location type.

Example assuming "village" is a location ID, which is converted to names using the location `choice_provider`:

```
{
  "type": "dynamic_choice_list",
  "slug": "village",
  "field": "location_id",
  "display": "Village",
  "datatype": "string",
  "choice_provider": {
      "type": "location",
      "include_descendants": true,
      "show_full_path": true,
      "location_type": "district"
  }
}
```

Choice lists

Choice lists allow manual configuration of a fixed, specified number of choices and let you change what they look like in the UI.

```
{
  "type": "choice_list",
  "slug": "role",
  "field": "role",
  "choices": [
    {"value": "doctor", "display": "Doctor"},
    {"value": "nurse"}
  ]
}
```

Drilldown by Location

This filter allows selection of a location for filtering by drilling down from top level.

```
{
  "type": "location_drilldown",
  "slug": "by_location",
  "field": "district_id",
  "include_descendants": true,
  "max_drilldown_levels": 3
}
```

- "include_descendants" - Include descendant locations in the results. Defaults to `false`.
- "max_drilldown_levels" - Maximum allowed drilldown levels. Defaults to 99

Internationalization

Report builders may specify translations for the filter display value. Also see the sections on internationalization in the Report Column and the *translations transform*.

```
{
    "type": "choice_list",
    "slug": "state",
    "display": {"en": "State", "fr": "État"},
}
```

25.3.3 Report Columns

Reports are made up of columns. The currently supported column types ares:

- *field* which represents a single value
- *percent* which combines two values in to a percent
- *aggregate_date* which aggregates data by month
- *expanded* which expands a select question into multiple columns
- *expression* which can do calculations on data in other columns

Field columns

Field columns have a type of `"field"`. Here's an example field column that shows the owner name from an associated `owner_id`:

```
{
    "type": "field",
    "field": "owner_id",
    "column_id": "owner_id",
    "display": "Owner Name",
    "format": "default",
    "transform": {
        "type": "custom",
        "custom_type": "owner_display"
    },
    "aggregation": "simple"
}
```

Percent columns

Percent columns have a type of `"percent"`. They must specify a `numerator` and `denominator` as separate field columns. Here's an example percent column that shows the percentage of pregnant women who had danger signs.

```
{
  "type": "percent",
  "column_id": "pct_danger_signs",
  "display": "Percent with Danger Signs",
  "format": "both",
```

(continues on next page)

(continued from previous page)

```
  "denominator": {
    "type": "field",
    "aggregation": "sum",
    "field": "is_pregnant",
    "column_id": "is_pregnant"
  },
  "numerator": {
    "type": "field",
    "aggregation": "sum",
    "field": "has_danger_signs",
    "column_id": "has_danger_signs"
  }
}
```

Formats

The following percentage formats are supported.

Format	Description	example
percent	A whole number percentage (the default format)	33%
fraction	A fraction	1/3
both	Percentage and fraction	33% (1/3)
numeric_percent	Percentage as a number	33
decimal	Fraction as a decimal number	.333

AggregateDateColumn

AggregateDate columns allow for aggregating data by month over a given date field. They have a type of `"aggregate_date"`. Unlike regular fields, you do not specify how aggregation happens, it is automatically grouped by month.

Here's an example of an aggregate date column that aggregates the `received_on` property for each month (allowing you to count/sum things that happened in that month).

```
{
   "column_id": "received_on",
   "field": "received_on",
   "type": "aggregate_date",
   "display": "Month"
 }
```

AggregateDate supports an optional "format" parameter, which accepts the same format string as *Date formatting*. If you don't specify a format, the default will be "%Y-%m", which will show as, for example, "2008-09".

Keep in mind that the only variables available for formatting are `year` and `month`, but that still gives you a fair range, e.g.

format	Example result
"%Y-%m"	"2008-09"
"%B, %Y"	"September, 2008"
"%b (%y)"	"Sep (08)"

IntegerBucketsColumn

Bucket columns allow you to define a series of ranges with corresponding names, then group together rows where a specific field's value falls within those ranges. These ranges are inclusive, since they are implemented using the `between` operator. It is the user's responsibility to make sure the ranges do not overlap; if a value falls into multiple ranges, it is undefined behavior which bucket it will be assigned to.

Here's an example that groups children based on their age at the time of registration:

```
{
    "display": "age_range",
    "column_id": "age_range",
    "type": "integer_buckets",
    "field": "age_at_registration",
    "ranges": {
        "infant": [0, 11],
        "toddler": [12, 35],
        "preschooler": [36, 60]
    },
    "else_": "older"
}
```

The `"ranges"` attribute maps conditional expressions to labels. If the field's value does not fall into any of these ranges, the row will receive the `"else_"` value, if provided.

SumWhenColumn and SumWhenTemplateColumn

Note: `SumWhenColumn` usage is limited to static reports, and `SumWhenTemplateColumn` usage is behind a feature flag.

Sum When columns allow you to aggregate data based on arbitrary conditions.

The `SumWhenColumn` allows any expression.

The `SumWhenTemplateColumn` is used in conjunction with a subclass of `SumWhenTemplateSpec`. The template defines an expression and typically accepts binds. An example:

Example using `sum_when`:

```
{
    "display": "under_six_month_olds",
    "column_id": "under_six_month_olds",
    "type": "sum_when",
    "field": "age_at_registration",
    "whens": [
        ["age_at_registration < 6", 1],
    ],
    "else_": 0
}
```

Equivalent example using `sum_when_template`:

```
{
    "display": "under_x_month_olds",
    "column_id": "under_x_month_olds",
    "type": "sum_when_template",
    "field": "age_at_registration",
    "whens": [
         {
             "type": "under_x_months",
             "binds": [6],
             "then": 1
         }
    ],
    "else_": 0
}
```

Expanded Columns

Expanded columns have a type of `"expanded"`. Expanded columns will be "expanded" into a new column for each distinct value in this column of the data source. For example:

If you have a data source like this:

```
+---------|----------|-------------+
| Patient | district | test_result |
+---------|----------|-------------+
| Joe     | North    | positive    |
| Bob     | North    | positive    |
| Fred    | South    | negative    |
+---------|----------|-------------+
```

and a report configuration like this:

```
aggregation columns:
["district"]

columns:
[
  {
    "type": "field",
    "field": "district",
    "column_id": "district",
    "format": "default",
    "aggregation": "simple"
  },
  {
    "type": "expanded",
    "field": "test_result",
    "column_id": "test_result",
    "format": "default"
  }
]
```

Then you will get a report like this:

```
+----------|----------------------|----------------------+
| district | test_result-positive | test_result-negative |
+----------|----------------------|----------------------+
| North    | 2                    | 0                    |
| South    | 0                    | 1                    |
+----------|----------------------|----------------------+
```

Expanded columns have an optional parameter `"max_expansion"` (defaults to 10) which limits the number of columns that can be created. WARNING: Only override the default if you are confident that there will be no adverse performance implications for the server.

Expression columns

Expression columns can be used to do just-in-time calculations on the data coming out of reports. They allow you to use any UCR expression on the data in the report row. These can be referenced according to the `column_id`s from the other defined column. They can support advanced use cases like doing math on two different report columns, or doing conditional logic based on the contents of another column.

A simple example is below, which assumes another called "number" in the report and shows how you could make a column that is 10 times that column.

```
{
    "type": "expression",
    "column_id": "by_tens",
    "display": "Counting by tens",
    "expression": {
        "type": "evaluator",
        "statement": "a * b",
        "context_variables": {
            "a": {
                "type": "property_name",
                "property_name": "number"
            },
            "b": 10
        }
    }
}
```

Expression columns cannot be used in aggregations or filters. If you need to group by a derived value, then you must add that directly to your data source.

The "aggregation" column property

The aggregation column property defines how the column should be aggregated. If the report is not doing any aggregation, or if the column is one of the aggregation columns this should always be `"simple"` (see *Aggregation* below for more information on aggregation).

The following table documents the other aggregation options, which can be used in aggregate reports.

Format	Description
simple	No aggregation
avg	Average (statistical mean) of the values
count_unique	Count the unique values found
count	Count all rows
min	Choose the minimum value
max	Choose the maximum value
sum	Sum the values

Column IDs

Column IDs in percentage fields *must be unique for the whole report*. If you use a field in a normal column and in a percent column you must assign unique `column_id` values to it in order for the report to process both.

Calculating Column Totals

To sum a column and include the result in a totals row at the bottom of the report, set the `calculate_total` value in the column configuration to `true`.

Not supported for the following column types: - expression

Internationalization

Report columns can be translated into multiple languages. To translate values in a given column check out the *translations transform* below. To specify translations for a column header, use an object as the `display` value in the configuration instead of a string. For example:

```
{
    "type": "field",
    "field": "owner_id",
    "column_id": "owner_id",
    "display": {
        "en": "Owner Name",
        "he": ""
    },
    "format": "default",
    "transform": {
        "type": "custom",
        "custom_type": "owner_display"
    },
    "aggregation": "simple"
}
```

The value displayed to the user is determined as follows: - If a display value is specified for the users language, that value will appear in the report. - If the users language is not present, display the `"en"` value. - If `"en"` is not present, show an arbitrary translation from the `display` object. - If `display` is a string, and not an object, the report shows the string.

Valid `display` languages are any of the two or three letter language codes available on the user settings page.

25.3.4 Aggregation

Aggregation in reports is done using a list of columns to aggregate on. This defines how indicator data will be aggregated into rows in the report. The columns represent what will be grouped in the report, and should be the `column_id`s of valid report columns. In most simple reports you will only have one level of aggregation. See examples below.

No aggregation

Note that if you use `is_primary_key` in any of your columns, you must include all primary key columns here.

```
["doc_id"]
```

Aggregate by 'username' column

```
["username"]
```

Aggregate by two columns

```
["column1", "column2"]
```

25.3.5 Transforms

Transforms can be used in two places - either to manipulate the value of a column just before it gets saved to a data source, or to transform the value returned by a column just before it reaches the user in a report. Here's an example of a transform used in a report config 'field' column:

```
{
    "type": "field",
    "field": "owner_id",
    "column_id": "owner_id",
    "display": "Owner Name",
    "format": "default",
    "transform": {
        "type": "custom",
        "custom_type": "owner_display"
    },
    "aggregation": "simple"
}
```

The currently supported transform types are shown below:

Translations and arbitrary mappings

The translations transform can be used to give human readable strings:

```
{
    "type": "translation",
    "translations": {
        "lmp": "Last Menstrual Period",
        "edd": "Estimated Date of Delivery"
    }
}
```

And for translations:

```
{
    "type": "translation",
    "translations": {
        "lmp": {
            "en": "Last Menstrual Period",
            "es": "Fecha Última Menstruación",
        },
        "edd": {
            "en": "Estimated Date of Delivery",
            "es": "Fecha Estimada de Parto",
        }
    }
}
```

To use this in a mobile ucr, set the `'mobile_or_web'` property to `'mobile'`

```
{
    "type": "translation",
    "mobile_or_web": "mobile",
    "translations": {
        "lmp": "Last Menstrual Period",
        "edd": "Estimated Date of Delivery"
    }
}
```

Displaying Readable User Name (instead of user ID)

This takes a *user_id* value and changes it to HQ's best guess at the user's display name, using their first and last name, if available, then falling back to their username.

```
{
    "type": "custom",
    "custom_type": "user_display_including_name"
}
```

Displaying username instead of user ID

```
{
    "type": "custom",
    "custom_type": "user_display"
}
```

Displaying username minus @domain.commcarehq.org instead of user ID

```
{
    "type": "custom",
    "custom_type": "user_without_domain_display"
}
```

Displaying owner name instead of owner ID

```
{
    "type": "custom",
    "custom_type": "owner_display"
}
```

Displaying month name instead of month index

```
{
    "type": "custom",
    "custom_type": "month_display"
}
```

Rounding decimals

Rounds decimal and floating point numbers to two decimal places.

```
{
    "type": "custom",
    "custom_type": "short_decimal_display"
}
```

Generic number formatting

Rounds numbers using Python's built in formatting.

See below for a few simple examples. Read the docs for complex ones. The input to the format string will be a *number* not a string.

If the format string is not valid or the input is not a number then the original input will be returned.

Round to the nearest whole number

```
{
    "type": "number_format",
    "format_string": "{0:.0f}"
}
```

Rich text formatting with Markdown

This can be used to do some rich text formatting, using [Markdown](https://www.markdownguide.org/).

There is no configuration required, it will assume the input is valid, markdown-ready text.

```
{
    "type": "markdown"
}
```

This transform works for report columns only. Using it in a data source will add HTML markup, but it will not be displayed properly in HQ.

Always round to 3 decimal places

```
{
    "type": "number_format",
    "format_string": "{0:.3f}"
}
```

Date formatting

Formats dates with the given format string. See here for an explanation of format string behavior. If there is an error formatting the date, the transform is not applied to that value.

```
{
   "type": "date_format",
   "format": "%Y-%m-%d %H:%M"
}
```

Converting an ethiopian date string to a gregorian date

Converts a string in the YYYY-MM-DD format to a gregorian date. For example, 2009-09-11 is converted to date(2017, 5, 19). If it is unable to convert the date, it will return an empty string.

```
{
   "type": "custom",
   "custom_type": "ethiopian_date_to_gregorian_date"
}
```

Converting a gregorian date string to an ethiopian date

Converts a string in the YYYY-MM-DD format to an ethiopian date. For example, 2017-05-19 is converted to date(2009, 09, 11). If it is unable to convert the date, it will return an empty string.

```
{
   "type": "custom",
   "custom_type": "gregorian_date_to_ethiopian_date"
}
```

25.3.6 Charts

There are currently three types of charts supported. Pie charts, and two types of bar charts.

Pie charts

A pie chart takes two inputs and makes a pie chart. Here are the inputs:

Field	Description
aggregation_colu mn	The column you want to group - typically a column from a select question
value_column	The column you want to sum - often just a count

Here's a sample spec:

```
{
    "type": "pie",
    "title": "Remote status",
    "aggregation_column": "remote",
    "value_column": "count"
}
```

Aggregate multibar charts

An aggregate multibar chart is used to aggregate across two columns (typically both of which are select questions). It takes three inputs:

Field	Description
pri- mary_aggregation	The primary aggregation. These will be the x-axis on the chart.
secondary_aggregati on	The secondary aggregation. These will be the slices of the bar (or individual bars in "grouped" format)
value_column	The column you want to sum - often just a count

Here's a sample spec:

```
{
    "type": "multibar-aggregate",
    "title": "Applicants by type and location",
```

(continues on next page)

(continued from previous page)

```
    "primary_aggregation": "remote",
    "secondary_aggregation": "applicant_type",
    "value_column": "count"
}
```

Multibar charts

A multibar chart takes a single x-axis column (typically a user, date, or select question) and any number of y-axis columns (typically indicators or counts) and makes a bar chart from them.

Field	Description
x_axis_colum	This will be the x-axis on the chart.
y_axis_colum	These are the columns to use for the secondary axis. These will be the slices of the bar (or individual bars in "grouped" format).

Here's a sample spec:

```
{
    "type": "multibar",
    "title": "HIV Mismatch by Clinic",
    "x_axis_column": "clinic",
    "y_axis_columns": [
        {
            "column_id": "diagnoses_match_no",
            "display": "No match"
        },
        {
            "column_id": "diagnoses_match_yes",
            "display": "Match"
        }
    ]
}
```

25.3.7 Sort Expression

A sort order for the report rows can be specified. Multiple fields, in either ascending or descending order, may be specified. Example:

Field should refer to report column IDs, not database fields.

```
[
  {
    "field": "district",
    "order": "DESC"
  },
  {
    "field": "date_of_data_collection",
    "order": "ASC"
```

(continues on next page)

(continued from previous page)

```
  }
]
```

25.3.8 Distinct On

Can be used to limit the rows in a report based on a single column or set of columns. The top most row is picked in case of duplicates.

This is different from aggregation in sense that this is done after fetching the rows, whereas aggregation is done before selecting the rows.

This is used in combination with a sort expression to have predictable results.

Please note that the columns used in distinct on clause should also be present in the sort expression as the first set of columns in the same order.

Pick distinct by a single column

Sort expression should have column1 and then other columns if needed

```
[
  {
    "field": "column1",
    "order": "DESC"
  },
  {
    "field": "column2",
    "order": "ASC"
  }
]
```

and distinct on would be

```
["column1"]
```

Pick distinct result based on two columns

Sort expression should have column1 and column2 in same order, More columns can be added after these if needed

```
[
  {
    "field": "column1",
    "order": "DESC"
  },
  {
    "field": "column2",
    "order": "ASC"
  }
]
```

and distinct on would be

```
["column1", "column2"]
```

25.4 Mobile UCR

Mobile UCR is a beta feature that enables you to make application modules and charts linked to UCRs on mobile. It also allows you to send down UCR data from a report as a fixture which can be used in standard case lists and forms throughout the mobile application.

The documentation for Mobile UCR is very sparse right now.

25.4.1 Filters

On mobile UCR, filters can be automatically applied to the mobile reports based on hardcoded or user-specific data, or can be displayed to the user.

The documentation of mobile UCR filters is incomplete. However some are documented below.

Custom Calendar Month

When configuring a report within a module, you can filter a date field by the 'CustomMonthFilter'. The choice includes the following options: - Start of Month (a number between 1 and 28) - Period (a number between 0 and n with 0 representing the current month).

Each custom calendar month will be "Start of the Month" to ("Start of the Month" - 1). For example, if the start of the month is set to 21, then the period will be the 21th of the month -> 20th of the next month.

Examples: Assume it was May 15: Period 0, day 21, you would sync April 21-May 15th Period 1, day 21, you would sync March 21-April 20th Period 2, day 21, you would sync February 21 -March 20th

Assume it was May 20: Period 0, day 21, you would sync April 21-May 20th Period 1, day 21, you would sync March 21-April 20th Period 2, day 21, you would sync February 21-March 20th

Assume it was May 21: Period 0, day 21, you would sync May 21-May 21th Period 1, day 21, you would sync April 21-May 20th Period 2, day 21, you would sync March 21-April 20th

25.5 Export

A UCR data source can be exported, to back an excel dashboard, for instance. The URL for exporting data takes the form *https://www.commcarehq.org/a/[domain]/configurable_reports/data_sources/export/[data source id]/*

The export supports a "$format" parameter which can be any of the following options: html, csv, xlsx, xls. The default format is csv.

This export can also be filtered to restrict the results returned. The filtering options are all based on the field names:

URL parameter	Value	Description
{field_name}	{exact value}	require an exact match
{field_name}-range	{start}..{end}	return results in range
{field_name}-lastndays	{number}	restrict to the last n days

This is configured in `export_data_source` and tested in `test_export`. It should be pretty straightforward to add support for additional filter types.

25.5.1 Export example

Let's say you want to restrict the results to only cases owned by a particular user, opened in the last 90 days, and with a child between 12 and 24 months old as an xlsx file. The querystring might look like this:

```
?$format=xlsx&owner_id=48l069n24myxk08hl563&opened_on-lastndays=90&child_age-range=12..24
```

25.6 Practical Notes

Some rough notes for working with user configurable reports.

25.6.1 Getting Started

The easiest way to get started is to start with sample data and reports.

Create a simple app and submit a few forms. You can then use report builder to create a report. Start at `a/DOMAIN/reports/builder/select_source/` and create a report based on your form, either a form list or form summary.

When your report is created, clicking "Edit" will bring you to the report builder editor. An individual report can be viewed in the UCR editor by changing the report builder URL, `/a/DOMAIN/reports/builder/edit/REPORT_ID/` to the UCR URL, `/a/DOMAIN/configurable_reports/reports/edit/REPORT_ID/`. In this view, you can examine the columns, filters, and aggregation columns that report builder created.

The UCR config UI also includes pages to add new data sources, imports reports, etc., all based at `/a/DOMAIN/configurable_reports/`. If you add a new report via the UCR UI and copy in the columns, filters, etc. from a report builder report, that new report will then automatically open in the UCR UI when you edit it. You can also take an existing report builder report and set `my_report.report_meta.created_by_builder` to false to force it to open in the UCR UI in the future.

Two example UCRs, a case-based UCR for the `dimagi` domain and a form-based UCR for the `gsid` domain, are checked into source code. Their data source specs and report specs are in `corehq/apps/userreports/examples/`.

The tests are also a good source of documentation for the various filter and indicator formats that are supported.

When editing data sources, you can check the progress of rebuilding using `my_datasource.meta.build.finished`

25.6.2 Static data sources

As well as being able to define data sources via the UI which are stored in the database you can also define static data sources which live as JSON documents in the source repository.

These are mainly useful for custom reports.

They conform to a slightly different style:

```
{
    "domains": ["live-domain", "test-domain"],
    "config": {
        ... put the normal data source configuration here
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Having defined the data source you need to use the `static_ucr_data_source_paths` extension point to make CommCare aware of your data source. Now when the static data source pillow is run it will pick up the data source and rebuild it.

Alternatively, the legacy method is to add the path to the data source file to the `STATIC_DATA_SOURCES` setting in `settings.py`.

Changes to the data source require restarting the pillow which will rebuild the SQL table. Alternately you can use the UI to rebuild the data source (requires Celery to be running).

25.6.3 Static configurable reports

Configurable reports can also be defined in the source repository. Static configurable reports have the following style:

```
{
    "domains": ["my-domain"],
    "data_source_table": "my_table",
    "report_id": "my-report",
    "config": {
        ... put the normal report configuration here
    }
}
```

Having defined the report you need to use the `static_ucr_report_paths` extension point to make CommCare aware of your report.

Alternatively, the legacy method is to add the path to the data source file to the `STATIC_UCR_REPORTS` setting in `settings.py`.

25.6.4 Custom configurable reports

Sometimes a client's needs for a rendered report are outside of the scope of the framework. To render the report using a custom Django template or with custom Excel formatting, define a subclass of `ConfigurableReportView` and override the necessary functions. Then include the python path to the class in the field `custom_configurable_report` of the static report and don't forget to include the static report in `STATIC_DATA_SOURCES` in `settings.py`.

25.6.5 Extending User Configurable Reports

When building a custom report for a client, you may find that you want to extend UCR with custom functionality. The UCR framework allows developers to write custom expressions, and register them with the framework. To do so:

1. Define a function that returns an expression object

```
def custom_expression(spec, evaluation_context):
    ...
```

2. Extend the `custom_ucr_expressions` extension point:

```
from corehq.apps.userreports.extension_points import custom_ucr_expressions

@custom_ucr_expressions.extend()
def ucr_expressions():
    return [
        ('expression_name', 'path.to.custom_expression'),
    ]
```

See also:

- CommCare Extension documentation for more details on using extensions.
- `custom_ucr_expressions` docstring for full extension point details.
- `location_type_name`: A way to get location type from a location document id.
- `location_parent_id`: A shortcut to get a location's parent ID a location id.
- `get_case_forms`: A way to get a list of forms submitted for a case.
- `get_subcases`: A way to get a list of subcases (child cases) for a case.
- `indexed_case`: A way to get an indexed case from another case.

You can find examples of these in practical examples.

25.6.6 Scaling UCR

Profiling data sources

You can use `./manage.py profile_data_source <domain> <data source id> <doc id>` to profile a data-source on a particular doc. It will give you information such as functions that take the longest and number of database queries it initiates.

Faster Reporting

If reports are slow, then you can add `create_index` to the data source to any columns that have filters applied to them.

Asynchronous Indicators

If you have an expensive data source and the changes come in faster than the pillow can process them, you can specify `asynchronous: true` in the data source. This flag puts the document id in an intermediary table when a change happens which is later processed by a celery queue. If multiple changes are submitted before this can be processed, a new entry is not created, so it will be processed once. This moves the bottle neck from kafka/pillows to celery.

The main benefit of this is that documents will be processed only once even if many changes come in at a time. This makes this approach ideal datasources that don't require 'live' data or where the source documents change very frequently.

It is also possible achieve greater parallelization than is currently available via pillows since multiple Celery workers can process the changes.

A diagram of this workflow can be found here

25.6.7 Inspecting database tables

The easiest way to inspect the database tables is to use the sql command line utility.

This can be done by runnning `./manage.py dbshell` or using `psql`.

The naming convention for tables is: `config_report_[domain name]_[table id]_[hash]`.

In postgres, you can see all tables by typing `\dt` and use sql commands to inspect the appropriate tables.

CHAPTER

TWENTYSIX

UCR EXAMPLES

This page lists some common examples/design patterns for user configurable reports and CommCare HQ data models.

CHAPTER

TWENTYSEVEN

DATA SOURCE FILTERS

The following are example filter expressions that are common in data sources.

27.1 Filters on forms

The following filters apply to data sources built on top of forms.

27.1.1 Filter by a specific form type using the XMLNS

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "xmlns"
    },
    "operator": "eq",
    "property_value": "http://openrosa.org/formdesigner/my-registration-form"
}
```

27.1.2 Filter by a set of form types using the XMLNS

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "xmlns"
    },
    "operator": "in",
    "property_value": [
        "http://openrosa.org/formdesigner/my-registration-form",
        "http://openrosa.org/formdesigner/my-follow-up-form",
        "http://openrosa.org/formdesigner/my-close-form"
    ]
}
```

27.2 Filters on cases

The following filters apply to data sources built on top of cases.

27.2.1 Filter by a specific case type

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "type"
    },
    "operator": "eq",
    "property_value": "child"
}
```

27.2.2 Filter by multiple case types

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "type"
    },
    "operator": "in",
    "property_value": ["child", "mother"]
}
```

27.2.3 Filter by only open cases

NOTE: this should be changed to use boolean datatypes once those exist.

```
{
    "type": "boolean_expression",
    "expression": {
        "type": "property_name",
        "property_name": "closed",
        "datatype": null

    },
    "operator": "eq",
    "property_value": false
}
```

CHAPTER

TWENTYEIGHT

DATA SOURCE INDICATORS

28.1 Count every contributing row (form or case)

```
{
    "type": "expression",
    "expression": {
        "type": "constant",
        "constant": 1
    },
    "column_id": "count",
    "datatype": "integer",
    "display_name": "count of forms"
}
```

28.2 Save a form property directly to a table

The following indicator stubs show how to save various properties to a data source. These can be copied directly into data sources or modified to suit specific apps/forms.

28.2.1 Submission date (received on)

This saves the submission date as a `date` object. If you want to include the time change the datatypes to `"datetime"`.

```
{
    "type": "expression",
    "expression": {
        "type": "property_name",
        "property_name": "received_on",
        "datatype": "date"
    },
    "display_name": "Submission date",
    "datatype": "date",
    "column_id": "received_on"
}
```

28.2.2 User ID

```
{
    "display_name": "User ID",
    "datatype": "string",
    "expression": {
        "type": "property_path",
        "property_path": [
            "form",
            "meta",
            "userID"
        ]
    },
    "is_primary_key": false,
    "transform": {},
    "is_nullable": true,
    "type": "expression",
    "column_id": "user_id"
}
```

28.2.3 A text or choice property

This is the same type of indicator that should be used for typical Impact 123 indicators. In the example below, the indicator is inside a form group question called "impact123".

```
{
    "type": "expression",
    "expression": {
        "type": "property_path",
        "property_path": ["form", "impact123", "cc_impact_1"]
    },
    "column_id": "impact1",
    "display_name": "Impact 1",
    "datatype": "string"
}
```

28.3 Related doc lookups

28.3.1 Get an owner name - whether it's a user, group or location

```
{
    "datatype":"string",
    "type":"expression",
    "column_id":"owner_name",
    "expression":{
        "test":{
            "operator":"eq",
            "expression":{
                "value_expression":{
```

(continues on next page)

(continued from previous page)

```
                    "type":"property_name",
                    "property_name":"doc_type"
                },
                "type":"related_doc",
                "related_doc_type":"Group",
                "doc_id_expression":{
                    "type":"property_name",
                    "property_name":"owner_id"
                }
            },
            "type":"boolean_expression",
            "property_value":"Group"
        },
        "expression_if_true":{
            "value_expression":{
                "type":"property_name",
                "property_name":"name"
            },
            "type":"related_doc",
            "related_doc_type":"Group",
            "doc_id_expression":{
                "type":"property_name",
                "property_name":"owner_id"
            }
        },
        "type":"conditional",
        "expression_if_false":{
            "type":"conditional",
            "test":{
                "operator":"eq",
                "expression":{
                    "value_expression":{
                        "type":"property_name",
                        "property_name":"doc_type"
                    },
                    "type":"related_doc",
                    "related_doc_type":"CommCareUser",
                    "doc_id_expression":{
                        "type":"property_name",
                        "property_name":"owner_id"
                    }
                },
                "type":"boolean_expression",
                "property_value":"CommCareUser"
            },
            "expression_if_true":{
                "value_expression":{
                    "type":"property_name",
                    "property_name":"username"
                },
                "type":"related_doc",
                "related_doc_type":"CommCareUser",
```

(continues on next page)

(continued from previous page)

```
                "doc_id_expression":{
                    "type":"property_name",
                    "property_name":"owner_id"
                }
            },
            "expression_if_false":{
                "value_expression":{
                    "type":"property_name",
                    "property_name":"name"
                },
                "type":"related_doc",
                "related_doc_type":"Location",
                "doc_id_expression":{
                    "type":"property_name",
                    "property_name":"owner_id"
                }
            }
        }
    }
}
```

28.3.2 Get a case property from a form that modifies the case

The following expression looks up a case name from a form that references that case.

To lookup a different property, or for more complex form/case relationships and advanced modules just adjust the property paths.

```
{
    "type":"related_doc",
    "related_doc_type":"CommCareCase",
    "doc_id_expression":{
        "type": "property_path",
        "property_path": [
            "form",
            "case",
            "@case_id"
        ]
    },
    "value_expression":{
        "type":"property_path",
        "property_path": [
            "name"
        ]
    }
}
```

Note: this is an example *expression*. To use it in a data source just wrap it in a column.

28.3.3 Get a custom user data property from a form submission

```
{
    "datatype":"string",
    "type":"expression",
    "column_id":"confirmed_referral_target",
    "expression":{
        "type":"related_doc",
        "related_doc_type":"CommCareUser",
        "doc_id_expression":{
            "type": "property_path",
            "property_path": [
                "form",
                "meta",
                "userID"
            ]
        },
        "value_expression":{
            "type":"property_path",
            "property_path": [
                "user_data",
                "confirmed_referral_target"
            ]
        }
    }
}
```

28.4 Getting the parent case ID from a case

```
{
    "type": "nested",
    "argument_expression": {
        "type": "array_index",
        "array_expression": {
            "type": "property_name",
            "property_name": "indices"
        },
        "index_expression": {
            "type": "constant",
            "constant": 0
        }
    },
    "value_expression": {
        "type": "property_name",
        "property_name": "referenced_id"
    }
}
```

28.5 Getting the location type from a location doc id

`location_id_expression` can be any expression that evaluates to a valid location id.

```
{
    "datatype":"string",
    "type":"expression",
    "expression": {
        "type": "location_type_name",
        "location_id_expression": {
            "type": "property_name",
            "property_name": "_id"
        }
    },
    "column_id": "district"
}
```

28.6 Getting a location's parent ID

`location_id_expression` can be any expression that evaluates to a valid location id.

```
{
    "type":"expression",
    "expression": {
        "type": "location_parent_id",
        "location_id_expression": {
            "type": "property_name",
            "property_name": "location_id"
        }
    },
    "column_id": "parent_location"
}
```

CHAPTER

TWENTYNINE

BASE ITEM EXPRESSIONS

29.1 Emit multiple rows (one per non-empty case property)

In this example we take 3 case properties and save one row per property if it exists.

```
{
    "type": "iterator",
    "expressions": [
        {
            "type": "property_name",
            "property_name": "p1"
        },
        {
            "type": "property_name",
            "property_name": "p2"
        },
        {
            "type": "property_name",
            "property_name": "p3"
        },
    ],
    "test": {
        "type": "not",
        "filter": {
            "type": "boolean_expression",
            "expression": {
                "type": "identity",
            },
            "operator": "in",
            "property_value": ["", null]
        }
    }
}
```

29.2 Emit multiple rows of complex data

In this example we take 3 case properties and emit the property name along with the value (only if non-empty). Note that the test must also change in this scenario.

```
{
    "type": "iterator",
    "expressions": [
        {
            "type": "dict",
            "properties": {
                "name": "p1",
                "value": {
                    "type": "property_name",
                    "property_name": "p1"
                }
            }
        },
        {
            "type": "dict",
            "properties": {
                "name": "p2",
                "value": {
                    "type": "property_name",
                    "property_name": "p2"
                }
            }
        },
        {
            "type": "dict",
            "properties": {
                "name": "p3",
                "value": {
                    "type": "property_name",
                    "property_name": "p3"
                }
            }
        }
    ],
    "test": {
        "type": "not",
        "filter": {
            "type": "boolean_expression",
            "expression": {
                "type": "property_name",
                "property_name": "value"
            },
            "operator": "in",
            "property_value": ["", null],
        }
    }
}
```

29.3 Evaluator Examples

29.3.1 Age in years to age in months

In the above example, `age_in_years` can be replaces with another expression to get the property from the doc

```
{
    "type": "evaluator",
    "statement": "30.4 * age_in_years",
    "context_variables": {
        "age_in_years": {
            "type": "property_name",
            "property_name": "age"
        }
    }
}
```

This will lookup the property `age` and substituite its value in the `statement`

29.3.2 weight_gain example

```
{
    "type": "evaluator",
    "statement": "weight_2 - weight_1",
    "context_variables": {
        "weight_1": {
            "type": "property_name",
            "property_name": "weight_at_birth"
        },
        "weight_2": {
            "type": "property_name",
            "property_name": "weight_at_1_year"
        }
    }
}
```

This will return value of `weight_at_1_year - weight_at_birth`

29.3.3 diff_seconds example

```
"expression": {
    "type": "evaluator",
    "statement": "timedelta_to_seconds(time_end - time_start)",
    "context_variables": {
        "time_start": {
            "datatype": "datetime",
            "type": "property_path",
            "property_path": [
                "form",
                "meta",
```

(continues on next page)

(continued from previous page)

```
                "timeStart"
            ]
        },
        "time_end": {
            "datatype": "datetime",
            "type": "property_path",
            "property_path": [
                "form",
                "meta",
                "timeEnd"
            ]
        }
    }
}
```

This will return the difference in seconds between two times (i.e. start and end of form)

29.3.4 Date format

These examples using Python f-strings to format the dates.

Convert a datetime to a formatted string

2022-01-01T15:32:54.109971Z 15:32 on 01 Jan 2022

```
{
    "type": "evaluator",
    "statement": "f'{date:%H:%M on %d %b %Y}'",
    "context_variables": {
        "date": {
            "type": "property_name",
            "property_name": "my_datetime",
            "datatype": "datetime",
        }
    }
}
```

Convert a datetime to show only 3 digits in the microseconds field

2022-01-01T15:32:54.109971Z 2022-01-01T15:32:54.110Z

```
{
    "type": "evaluator",
    "statement": "f'{date:%Y-%m-%dT%H:%M:%S}.' + '%03d' % round(int(f'{date:%f}')/1000)␣
↪+ 'Z'",
    "context_variables": {
        "date": {
            "type": "property_name",
            "property_name": "my_datetime",
            "datatype": "datetime",
        }
    }
}
```

29.4 Getting forms submitted for a case

```
{
    "type": "get_case_forms",
    "case_id_expression": {
        "type": "property_name",
        "property_name": "case_id"
    }
}
```

29.5 Getting forms submitted from specific forms for a case

```
{
    "type": "get_case_forms",
    "case_id_expression": {
        "type": "property_name",
        "property_name": "case_id"
    },
    "xmlns": [
        "http://openrosa.org/formdesigner/D8EED5E3-88CD-430E-984F-45F14E76A551",
        "http://openrosa.org/formdesigner/F1B73934-8B70-4CEE-B462-3E4C81F80E4A"
    ]
}
```

29.6 Getting the related case from a case

```
{
    "type": "indexed_case",
    "case_expression": {
        "type": "identity",
        "comment": "This just means the current document for a case based datasource"
    },
    "index": "parent"
}
```

To access a specific property from the related case, you can do something like:

```
{
    "type": "nested",
    "argument_expression": {
        "type": "indexed_case",
        "case_expression": {
            "type": "identity",
            "comment": "This just means the current document for a case based UCR"
        },
        "index": "parent"
    },
    "value_expression": {
```

(continues on next page)

(continued from previous page)

```
        "type": "property_name",
        "property_name": "some_case_property"
    }
}
```

29.7 Filter, Map, Reduce, Flatten and Sort expressions

29.7.1 Getting number of forms of a particular type

```
{
    "type": "reduce_items",
    "items_expression": {
        "type": "filter_items",
        "items_expression": {
         "type": "get_case_forms",
         "case_id_expression": {"type": "property_name", "property_name": "case_id"}
        },
        "filter_expression": {
                    "type": "boolean_expression",
                    "operator": "eq",
                    "expression": {"type": "property_name", "property_name": "xmls"},
                    "property_value": "gmp_xmlns"
        }
    },
    "aggregation_fn": "count"
}
```

It can be read as `reduce(filter(get_case_forms))`

29.7.2 Getting latest form property

```
{
    "type": "nested",
    "argument_expression": {
        "type": "reduce_items",
        "items_expression": {
            "type": "sort_items",
            "items_expression": {
                "type": "filter_items",
                "items_expression": {
                    "type": "get_case_forms",
                    "case_id_expression": {"type": "property_name", "property_name":
↪"case_id"}
                },
                "filter_expression": {
                    "type": "boolean_expression",
                    "operator": "eq",
                    "expression": {"type": "property_name", "property_name": "xmls"},
```

(continues on next page)

(continued from previous page)

```
                    "property_value": "gmp_xmlns"
                }
            },
            "sort_expression": {"type": "property_name", "property_name": "received_on"}
        },
        "aggregation_fn": "last_item"
    },
    "value_expression": {
        "type": "property_name",
        "property_name": "weight"
    }
}
```

This will return `weight` form-property on latest gmp form (xmlns is gmp_xmlns).

CHAPTER

THIRTY

REPORT EXAMPLES

30.1 Report filters

30.1.1 Date filter for submission date

This assumes that you have saved a "`received_on`" column from the form into the data source.

```
{
  "type": "date",
  "slug": "received_on",
  "field": "received_on",
  "display": "Submission date",
  "required": false
}
```

30.2 Report columns

30.2.1 Creating a date column for months

The following makes a column for a "`received_on`" data source column that will aggregate by the month received.

```
{
    "type": "aggregate_date",
    "column_id": "received_on",
    "field": "received_on",
    "display": "Month"
}
```

30.2.2 Expanded columns

The following snippet creates an expanded column based on a column that contains a fixed number of choices. This is the default column setup used in Impact 123 reports.

```
{
    "type": "expanded",
    "field": "impact1",
    "column_id": "impact1",
    "display": "impact1"
}
```

CHAPTER

THIRTYONE

CHARTS

31.1 Impact 123 grouped by date

This assumes a month-based date column and an expanded impact indicator column, as described above.

```
{
    "y_axis_columns": [
        "impact1-positive",
        "impact1-negative",
        "impact1-unknown"
    ],
    "x_axis_column": "received_on",
    "title": "Impact1 by Submission Date",
    "display_params": {},
    "aggregation_column": null,
    "type": "multibar"
}
```

CHAPTER

THIRTYTWO

UCR FAQ

32.1 What is UCR?

UCR stands for 'User Configurable Report'. They are user-generated reports, created within HQ via Reports > Create New Report

32.2 Report Errors

`The database table backing your report does not exist yet. Please wait while the report is populated.`

This problem is probably occurring for you locally. On staging and production environments, report tables are generated upon save by an asynchronous Celery task. Even with `CELERY_TASK_ALWAYS_EAGER=True` in `settings.py`, the code currently will not generate these synchronously. You can manually generate them via the following management command:

```
./manage.py rebuild_tables_by_domain <domain-name> --initiated_by <HQ_user_id>
```

CHAPTER

THIRTYTHREE

MESSAGING IN COMMCARE HQ

The term "messaging" in CommCare HQ commonly refers to the set of frameworks that allow the following types of use cases:

- sending SMS to contacts
- receiving SMS from contacts and performing pre-configured actions based on the content
- time-based and rule-based schedules to send messages to contacts
- creating alerts based on configurable criteria
- sending outbound calls to contacts and initiating an Interactive Voice Response (IVR) session
- collecting data via SMS surveys
- sending email alerts to contacts

The purpose of this documentation is to show how all of those use cases are performed technically by CommCare HQ. The topics below cover this material and should be followed in the order presented below if you have no prior knowledge of the messaging frameworks used in CommCare HQ.

33.1 Messaging Definitions

33.1.1 General Messaging Terms

SMS Gateway
: a third party service that provides an API for sending and receiving SMS

Outbound SMS
: an SMS that is sent from the SMS Gateway to a contact

Inbound SMS
: an SMS that is sent from a contact to the SMS Gateway

Mobile Terminating (MT) SMS
: an outbound SMS

Mobile Originating (MO) SMS
: an inbound SMS

Dual Tone Multiple Frequencies (DTMF) tones:
: the tones made by a telephone when pressing a button such as number 1, number 2, etc.

Interactive Voice Response (IVR) Session:
: a phone call in which the user is prompted to make choices using DTMF tones and the flow of the call can change based on those choices

IVR Gateway
a third party service that provides an API for handling IVR sessions

International Format (also referred to as E.164 Format) for a Phone Number:
a format for a phone number which makes it so that it can be reached from any other country; the format typically starts with +, then the country code, then the number, though there may be some subtle operations to perform on the number before putting into international format, such as removing a leading zero

SMS Survey
a way of collecting data over SMS that involves asking questions one SMS at a time and waiting for a contact's response before sending the next SMS

Structured SMS
a way for collecting data over SMS that involves collecting all data points in one SMS rather than asking one question at a time as in an SMS Survey; for example: "REGISTER Joe 25" could be one way to define a Structured SMS that registers a contact named Joe whose age is 25.

33.1.2 Messaging Terms Commonly Used in CommCare HQ

SMS Backend
the code which implements the API of a specific SMS Gateway

IVR Backend
the code which implements the API of a specific IVR Gateway

Two-way Phone Number
a phone number that the system has tied to a single contact in a single domain, so that the system can not only send oubound SMS to the contact, but the contact can also send inbound SMS and have the system process it accordingly; the system currently only considers a number to be two-way if there is a corehq.apps.sms.models.PhoneNumber entry for it that has verified = True

One-way Phone Number
a phone number that has not been tied to a single contact, so that the system can only send outbound SMS to the number; one-way phone numbers can be shared across many contacts in many domains, but only one of those numbers can be a two-way phone number

33.2 Contacts

A contact is a single person that we want to interact with through messaging. In CommCare HQ, at the time of writing, contacts can either be users (CommCareUser, WebUser) or cases (CommCareCase).

In order for the messaging frameworks to interact with a contact, the contact must implement the corehq.apps.sms.mixin.CommCareMobileContactMixin.

Contacts have phone numbers which allows CommCare HQ to interact with them. All phone numbers for contacts must be stored in International Format, and the frameworks always assume a phone number is given in International Format.

Regarding the + sign before the phone number, the rule of thumb is to never store the + when storing phone numbers, and to always display it when displaying phone numbers.

33.2.1 Users

A user's phone numbers are stored as the phone_numbers attribute on the CouchUser class, which is just a list of strings.

At the time of writing, WebUsers are only allowed to have one-way phone numbers.

CommCareUsers are allowed to have two-way phone numbers, but in order to have a phone number be considered to be a two-way phone number, it must first be verified. The verification process is initiated on the edit mobile worker page and involves sending an outbound SMS to the phone number and having it be acknowledged by receiving a validated response from it.

33.2.2 Cases

At the time of writing, cases are allowed to have only one phone number. The following case properties are used to define a case's phone number:

contact_phone_number
: the phone number, in International Format

contact_phone_number_is_verified
: must be set to 1 in order to consider the phone number a two-way phone number; the point here is that the health worker registering the case should verify the phone number and the form should set this case property to 1 if the health worker has identified the phone number as verified

If two cases are registered with the same phone number and both set the verified flag to 1, it will only be granted two-way phone number status to the case who registers it first.

If a two-way phone number can be granted for the case, a corehq.apps.sms.models.PhoneNumber entry with verified set to True is created for it. This happens automatically by running celery task run_case_update_rules_on_save for a case each time a case is saved.

33.2.3 Future State

Forcing the verification workflows before granting a phone number two-way phone number status has proven to be challenging for our users. In a (hopefully soon) future state, we will be doing away with all verification workflows and automatically consider a phone number to be a two-way phone number for the contact who registers it first.

33.3 Outbound SMS

The SMS framework uses a queuing architecture to make it easier to scale SMS processing power horizontally.

The process to send an SMS from within the code is as follows. The only step you need to do is the first, and the rest happen automatically.

1. **Invoke one of the send_sms* functions found in corehq.apps.sms.api:**

   send_sms
   : used to send SMS to a one-way phone number represented as a string

   send_sms_to_verified_number
   : use to send SMS to a two-way phone number represented as a PhoneNumber object

send_sms_with_backend
used to send SMS with a specific SMS backend

send_sms_with_backend_name
used to send SMS with the given SMS backend name which will be resolved to an SMS backend

2. The framework creates a corehq.apps.sms.models.QueuedSMS object representing the SMS to be sent.
3. The SMS Queue polling process (python manage.py run_sms_queue), which runs as a supervisor process on one of the celery machines, picks up the QueuedSMS object and passes it to corehq.apps.sms.tasks.process_sms.
4. process_sms attempts to send the SMS. If an error happens, it is retried up to 2 more times on 5 minute intervals. After 3 total attempts, any failure causes the SMS to be marked with error = True.
5. Whether the SMS was processed successfully or not, the QueuedSMS object is deleted and replaced by an identical looking corehq.apps.sms.models.SMS object for reporting.

At a deeper level, process_sms performs the following important functions for outbound SMS. To find out other more detailed functionality provided by process_sms, see the code.

1. If the domain has restricted the times at which SMS can be sent, check those and requeue the SMS if it is not currently an allowed time.
2. **Select an SMS backend by looking in the following order:**
   - If using a two-way phone number, look up the SMS backend with the name given in the backend_id property
   - If the domain has a default SMS backend specified, use it
   - Look up an appropriate global SMS backend by checking the phone number's prefix against the global SQLMobileBackendMapping entries
   - Use the catch-all global backend (found from the global SQLMobileBackendMapping entry with prefix = '*')
3. If the SMS backend has configured rate limiting or load balancing across multiple numbers, enforce those constraints.
4. Pass the SMS to the send() method of the SMS Backend, which is an instance of corehq.apps.sms.models.SQLSMSBackend.

33.4 Inbound SMS

Inbound SMS uses the same queueing architecture as outbound SMS does.

The entry point to processing an inbound SMS is the corehq.apps.sms.api.incoming function. All SMS backends which accept inbound SMS call the incoming function.

From there, the following functions are performed at a high level:

1. The framework creates a corehq.apps.sms.models.QueuedSMS object representing the SMS to be processed.
2. The SMS Queue polling process (python manage.py run_sms_queue), which runs as a supervisor process on one of the celery machines, picks up the QueuedSMS object and passes it to corehq.apps.sms.tasks.process_sms.

3. process_sms attempts to process the SMS. If an error happens, it is retried up to 2 more times on 5 minute intervals. After 3 total attempts, any failure causes the SMS to be marked with error = True.
4. Whether the SMS was processed successfully or not, the QueuedSMS object is deleted and replaced by an identical looking corehq.apps.sms.models.SMS object for reporting.

At a deeper level, process_sms performs the following important functions for inbound SMS. To find out other more detailed functionality provided by process_sms, see the code.

1. Look up a two-way phone number for the given phone number string.
2. If a two-way phone number is found, pass the SMS on to each inbound SMS handler (defined in settings.SMS_HANDLERS) until one of them returns True, at which point processing stops.
3. If a two-way phone number is not found, try to pass the SMS on to the SMS handlers that don't require two-way phone numbers (the phone verification workflow, self-registration over SMS workflows)

33.5 SMS Backends

We have one SMS Backend class per SMS Gateway that we make available.

SMS Backends are defined by creating a new directory under corehq.messaging.smsbackends, and the code for each backend has two main parts:

- The outbound part of the backend which is represented by a class that subclasses corehq.apps.sms.models.SQLSMSBackend
- The inbound part of the backend which is represented by a view that subclasses corehq.apps.sms.views.IncomingBackendView

33.5.1 Outbound

The outbound part of the backend code is responsible for interacting with the SMS Gateway's API to send an SMS.

All outbound SMS backends are subclasses of SQLSMSBackend, and you can't use a backend until you've created an instance of it and saved it in the database. You can have multiple instances of backends, if for example, you have multiple accounts with the same SMS gateway.

Backend instances can either be global, in which case they are shared by all projects in CommCare HQ, or they can belong to a specific project. If belonged to a specific project, a backend can optionally be shared with other projects as well.

To write the outbound backend code:

1. Create a subclass of corehq.apps.sms.models.SQLSMSBackend and implement the unimplemented methods:

   get_api_id
   : should return a string that uniquely identifies the backend type (but is shared across backend instances); we choose to not use the class name for this since class names can change but the api id should never change; the api id is only used for sms billing to look up sms rates for this backend type

   get_generic_name
   : a displayable name for the backend

get_available_extra_fields
each backend likely needs to store additional information, such as a username and password for authenticating with the SMS gateway; list those fields here and they will be accessible via the backend's config property

get_form_class
should return a subclass of corehq.apps.sms.forms.BackendForm, which should:

- have form fields for each of the fields in get_available_extra_fields, and
- implement the gateway_specific_fields property, which should return a crispy forms rendering of those fields

send
takes a corehq.apps.sms.models.QueuedSMS object as an argument and is responsible for interfacing with the SMS Gateway's API to send the SMS; if you want the framework to retry the SMS, raise an exception in this method, otherwise if no exception is raised the framework takes that to mean the process was successful. Unretryable error responses may be recorded on the message object with *msg.set_gateway_error(message)* where *message* is the error message or code returned by the gateway.

2. Add the backend to settings.HQ_APPS and settings.SMS_LOADED_SQL_BACKENDS
3. Run ./manage.py makemigrations sms; Django will just create a proxy model for the backend model, but no database changes will occur
4. Add an outbound test for the backend in corehq.apps.sms.tests.test_backends. This will test that the backend is reachable by the framework, but any testing of the direct API connection with the gateway must be tested manually.

Once that's done, you should be able to create instances of the backend by navigating to Messaging -> SMS Connectivity (for domain-level backend instances) or Admin -> SMS Connectivity and Billing (for global backend instances). To test it out, set it as the default backend for a project and try sending an SMS through the Compose SMS interface.

Things to look out for:

- Make sure you use the proper encoding of the message when you implement the send() method. Some gateways are picky about the encoding needed. For example, some require everything to be UTF-8. Others might make you choose between ASCII and Unicode. And for the ones that accept Unicode, you might need to sometimes convert it to a hex representation. And remember that get/post data will be automatically url-encoded when you use python requests. Consult the documentation for the gateway to see what is required.
- The message limit for a single SMS is 160 7-bit structures. That works out to 140 bytes, or 70 words. That means the limit for a single message is typically 160 GSM characters, or 70 Unicode characters. And it's actually a little more complicated than that since some simple ASCII characters (such as '{') take up two GSM characters, and each carrier uses the GSM alphabet according to language.

  So the bottom line is, it's difficult to know whether the given text will fit in one SMS message or not. As a result, you should find out if the gateway supports Concatenated SMS, a process which seamlessly splits up long messages into multiple SMS and stiches them back up without you having to do any additional work. You may need to have the gateway enable a setting to do this or include an additional parameter when sending SMS to make this work.
- If this gateway has a phone number that people can reply to (whether a long code or short code), you'll want to add an entry to the sms.Phoneblacklist model for the gateway's phone number so that the system won't allow sending SMS to this number as a precaution. You can do so in the Django admin, and you'll want to make sure that send_sms and can_opt_in are both False on the record.

33.5.2 Inbound

The inbound part of the backend code is responsible for exposing a view which implements the API that the SMS Gateway expects so that the gateway can connect to CommCare HQ and notify us of inbound SMS.

To write the inbound backend code:

1. Create a subclass of corehq.apps.sms.views.IncomingBackendView, and implement the unimplemented property:

   backend_class
   should return the subclass of SQLSMSBackend that was written above

2. Implement either the get() or post() method on the view based on the gateway's API. The only requirement of the framework is that this method call the corehq.apps.sms.api.incoming function, but you should also:

   - pass self.backend_couch_id as the backend_id kwarg to incoming()
   - if the gateway gives you a unique identifier for the SMS in their system, pass that identifier as the backend_message_id kwarg to incoming(); this can help later with debugging

3. Create a url for the view. The url pattern should accept an api key and look something like: r'^sms/(?P<api_key>[w-]+)/$' . The API key used will need to match the inbound_api_key of a backend instance in order to be processed.

4. Let the SMS Gateway know the url to connect to, including the API Key. To get the API Key, look at the value of the inbound_api_key property on the backend instance. This value is generated automatically when you first create a backend instance.

What happens behind the scenes is as follows:

1. A contact sends an inbound SMS to the SMS Gateway
2. The SMS Gateway connects to the URL configured above.
3. The view automatically looks up the backend instance by api key and rejects the request if one is not found.
4. Your get() or post() method is invoked which parses the parameters accordingly and passes the information to the inbound incoming() entry point.
5. The Inbound SMS framework takes it from there as described in the Inbound SMS section.

NOTE: The api key is part of the URL because it's not always easy to make the gateway send us an extra arbitrary parameter on each inbound SMS.

33.5.3 Rate Limiting

You may want (or need) to limit the rate at which SMS get sent from a given backend instance. To do so, just override the get_sms_rate_limit() method in your SQLSMSBackend, and have it return the maximum number of SMS that can be sent in a one minute period.

33.5.4 Load Balancing

If you want to load balance the Outbound SMS traffic automatically across multiple phone numbers, do the following:

1. Make your BackendForm subclass the corehq.apps.sms.forms.LoadBalancingBackendFormMixin
2. Make your SQLSMSBackend subclass the corehq.apps.sms.models.PhoneLoadBalancingMixin
3. Make your SQLSMSBackend's send method take a orig_phone_number kwarg. This will be the phone number to use when sending. This is always sent to the send() method, even if there is just one phone number to load balance over.

From there, the framework will automatically handle managing the phone numbers through the create/edit gateway UI and balancing the load across the numbers when sending. When choosing the originating phone number, the destination number is hashed and that hash is used to choose from the list of load balancing phone numbers, so that a recipient always receives messages from the same originating number.

If your backend uses load balancing and rate limiting, the framework applies the rate limit to each phone number separately as you would expect.

33.5.5 Backend Selection

There's also an **Automatic Choose** option, which selects a backend for each message based on the phone number's prefix. Domains can customize their prefix mappings, and there's a global mapping that HQ will fall back to if no domain-specific mapping is defined.

These prefix-backend mappings are stored in `SQLMobileBackend`. The global mappings can be accessed with `[(m.prefix, m.backend) for m in SQLMobileBackendMapping.objects.filter(is_global=True)]`

On production, this currently returns

```
('27', <SQLMobileBackend: Global Backend 'GRAPEVINE-ZA'>),
('999', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_TEST'>),
('1', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_TWILIO'>),
('258', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_MOZ'>),
('266', <SQLMobileBackend: Global Backend 'GRAPEVINE-ZA'>),
('265', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_TWILIO'>),
('91', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_UNICEL'>),
('268', <SQLMobileBackend: Global Backend 'GRAPEVINE-ZA'>),
('256', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_YO'>),
('*', <SQLMobileBackend: Global Backend 'MOBILE_BACKEND_MACH'>)
```

33.6 Scheduled Messages

The messaging framework supports scheduling messages to be sent on a one-time or recurring basis.

It uses a queuing architecture similar to the SMS framework, to make it easier to scale reminders processing power horizontally.

An earlier incarnation of this framework was called "reminders", so some code references to reminders remain, such as the `reminder_queue`.

33.6.1 Definitions

Scheduled messages are represented in the UI as "broadcasts" and "conditional alerts."

Broadcasts, represented by the subclasses of corehq.messaging.scheduling.models.abstract.Broadcast, allow configuring a recurring schedule to send a particular message type and content to a particular set of recipients.

Conditional alerts, represented by corehq.apps.data_interfaces.models.AutomaticUpdateRule, contain a similar recurring schedule but act on cases. They are configured to trigger on when cases meet a set of criteria, such as a case property changing to a specific value.

The two models share much of their code. This document primarily addresses conditional alerts and will refer to them as "rules," as most of the code does.

A rule definition, defines the rules for:

- what criteria cause a reminder to be triggered
- when the message should send once the criteria are fulfilled
- who the message should go to
- on what schedule and frequency the message should continue to be sent
- the content to send
- what causes the rule to stop

33.6.2 Conditional Alerts / Case Update Rules

A conditional alert, represented by corehq.apps.data_interfaces.models.AutomaticUpdateRule, defines an instance of a rule definition and keeps track of the state of the rule instance throughout its lifetime.

For example, a conditional alert definition may define a rule for sending an SMS to a case of type `patient`, and sending an SMS appointment reminder to the case 2 days before the case's `appointment_date` case property.

As soon as a case is created or updated in the given project to meet the criteria of having type `patient` and having an `appointment_date`, the framework will create a reminder instance to track it. After the message is sent 2 days before the `appointment_date`, the rule instance is deactivated to denote that it has completed the defined schedule and should not be sent again.

In order to keep messaging responsive to case changes, every time a case is saved, the corehq.messaging.tasks.sync_case_for_messaging function is called to handle any changes. This is controlled via *case-pillow*.

Similarly, any time a rule is updated, a corehq.messaging.tasks.run_messaging_rule task is spawned to rerun it against all cases in the project.

The aim of the framework is to always be completely responsive to all changes. So in the example above, if a case's `appointment_date` changes before the appointment reminder is actually sent, the framework will update the schedule instance (more on these below) automatically in order to reflect the new appointment date. And if the appointment reminder went out months ago but a new `appointment_date` value is given to the case for a new appointment, the same instance is updated again to reflect a new message that must go out.

Similarly, if the rule definition is updated to use a different case property other than `appointment_date`, all existing schedule instances are deleted and any new ones are created if they meet the criteria.

33.6.3 Lifecycle of a Rule

As mentioned above, whe a rule is changed, all cases of the relevant type in the domain are re-processed. The steps of this process are as follows:

1. When a conditional alert is created or activated, a corehq.messaging.tasks.initiate_messaging_rule_run task is spawned.
2. This locks the rule, so that it cannot be edited from the UI, and spawns a corehq.messaging.tasks.run_messaging_rule task.
3. This task spawns a corehq.messaging.tasks.sync_case_for_messaging_rule task for every case of the rule's case type. It also adds a corehq.messaging.tasks.set_rule_complete task to unlock the rule when all of the `sync_case` tasks are finished.
4. This task calls corehq.apps.data_interfaces.models.AutomaticUpdateRule.run_rule on its case.
5. `run_rule` checks whether or not the case meets the rule's criteria and acts accordingly. When the case matches, this calls `run_actions_when_case_matches` and then `when_case_matches`. Conditional alert actions use `CreateScheduleInstanceActionDefinition` which implements `when_case_matches` to call corehq.messaging.scheduling.tasks.refresh_case_alert_schedule_instances or corehq.messaging.scheduling.tasks.refresh_case_timed_schedule_instances depending on whether the rule is immediate or scheduled.
6. The refresh functions act on subclasses of corehq.messaging.scheduling.tasks.ScheduleInstanceRefresher, which create, update, and delete "schedule instance" objects, which are subclasses of corehq.messaging.scheduling.scheduling_partitioned.models.ScheduleInstance. These schedule instances track their schedule, recipients, and state relating to their next event. They are processed by a queue (see next section).

33.6.4 Queueing

All of the schedule instances in the database represent the queue of messages that should be sent. The way a schedule instance is processed is as follows:

1. The polling process (`python manage.py queue_schedule_instances`), which runs as a supervisor process on one of the celery machines, constantly polls for schedules that should be processed by querying for schedule instances that have a `next_event_due` property that is in the past.
2. Once a schedule instance that needs to be processed has been identified, the framework spawns one of several tasks from corehq.messaging.scheduling.tasks to handle it. These tasks include `handle_alert_schedule_instance`, `handle_timed_schedule_instance`, `handle_case_alert_schedule_instance`, and `handle_case_timed_schedule_instance`.
3. The handler looks at the schedule instances and instructs it to 1) take the appropriate action that has been configured (for example, send an sms), and 2) update the state of the instance so that it gets scheduled for the next action it must take based on the reminder definition. This is handled by corehq.messaging.scheduling.scheduling_partitioned.models.ScheduleInstance.handle_current_event

A second queue (`python manage.py run_sms_queue`), which is set up similarly on each celery machine that consumes from the `reminder_queue`,handles the sending of messages.

33.6.5 Event Handlers

A rule (or broadcast) sends content of one type. At the time of writing, the content a reminder definition can be configured to send includes:

- SMS
- SMS Survey
- Emails
- Push Notifications

In the case of SMS SurveysSessions, the survey content is defined using a form in an app which is then played to the recipients over SMS or Whatsapp.

33.7 Keywords

A Keyword (corehq.apps.sms.models.Keyword) defines an action or set of actions to be taken when an inbound SMS is received whose first word matches the keyword configuration.

Any number of actions can be taken, which include:

- Replying with an SMS or SMS Survey
- Sending an SMS or SMS Survey to another contact or group of contacts
- Processing the SMS as a Structured SMS

Keywords tie into the Inbound SMS framework through the keyword handler (corehq.apps.sms.handlers.keyword.sms_keyword_handler, see settings.SMS_HANDLERS), and use the Reminders framework to carry out their action(s).

Behind the scenes, all actions besides processing Structured SMS create a reminder definition to be sent immediately. So any functionality provided by a reminder definition can be added to be supported as a Keyword action.

CHAPTER

THIRTYFOUR

API

For user facing API documentation see https://help.commcarehq.org/display/commcarepublic/Data+APIs

34.1 Bulk User Resource

Resource name: `bulk_user`
First version available: `v0.5`

This resource is used to get basic user data in bulk, fast. This is especially useful if you need to get, say, the name and phone number of every user in your domain for a widget.

Currently the default fields returned are:

```
id
email
username
first_name
last_name
phone_numbers
```

34.1.1 Supported Parameters:

- `q` - query string
- `limit` - maximum number of results returned
- `offset` - Use with `limit` to paginate results
- `fields` - restrict the fields returned to a specified set

Example query string:

```
?q=foo&fields=username&fields=first_name&fields=last_name&limit=100&offset=200
```

This will return the first and last names and usernames for users matching the query "foo". This request is for the third page of results (200-300)

Additional notes:
It is simple to add more fields if there arises a significant use case.

Potential future plans: Support filtering in addition to querying. Support different types of querying. Add an order_by option

CHAPTER

THIRTYFIVE

COMMCARE FHIR INTEGRATION

CommCare HQ offers three ways of sharing data over FHIR:

1. Data forwarding allows CommCare cases to be sent to remote FHIR services.
2. The FHIR Importer fetches resources from a remote FHIR API and imports them as CommCare cases.
3. The FHIR API exposes CommCare cases as FHIR resources.

FHIR-related functionality is enabled using the "FHIR integration" feature flag.

35.1 Forwarding Cases to a FHIR API

35.1.1 Overview

CommCare can forward case data to a FHIR API. Patient data can be treated differently from other data, in order to follow a workflow to avoid duplicating patient records.

This documentation follows the process to set up this kind of integration.

The "FHIR Integration" feature flag will need to be enabled.

35.1.2 Data design

The first step is to determine what data to send.

A spreadsheet offers a good medium for documenting the FHIR resource types that are required, and their properties.

You will want columns for the CommCare case property name, the path to the corresponding value in the FHIR resource (values can be nested, e.g. "Encounter.subject.reference), and, if they aren't both strings/text, the data types of the CommCare and FHIR values.

It is useful to build the spreadsheet alongside the HL7 FHIR reference documentation so that it is easy to look up FHIR resource property data types and value sets where applicable.

This process will result in a good understanding of what the CommCare app needs to include, question types to use, and the values for multiple choice questions.

It can also be helpful to have an example of the FHIR resource you need to create. In addition to the case property values you need, this will also show you other values, like the code system that a code belongs to. e.g.

```
{
  /* case property values: */
  "code": "91300",
  "display": "Pfizer-BioNTech COVID-19 Vaccine",

  /* other values: */
  "system": "http://www.ama-assn.org/go/cpt",
  "version": "2021"
}
```

35.1.3 App building

The second step is to build the CommCare app for collecting the data to be sent.

Searching for patients by name

CommCare's Data Forwarding to FHIR can be configured to register Patients as their cases are registered in CommCare, and to search for patients, so as to avoid duplication.

You can find these options when setting up data forwarding: Go to "Project Settings" > "Data Forwarding", under "Forward Cases to a FHIR API", click "+ Add a service to forward to".

Check "Enable patient registration" to register patients. If this is not checked then patients registered in CommCare will not be created on the remote FHIR service.

Check "Enable patient search" checkbox to search the FHIR service for an existing patient that matches the CommCare patient. If this is not checked and patient registration is enabled, then CommCare will always create a new patient a patient is registered in CommCare.

If searching for patients is enabled, CommCare has some additional requirements of an app, regarding patients' names.

Patient search uses three patient name properties:

- `Patient.name[0].given`
- `Patient.name[0].family`
- `Patient.name[0].text`

It is worth stressing that CommCare only uses the first "name" instance of the Patient resource. If the resource has multiple values for name, patient search ignores the later values.

Patients are searched for using `Patient.name[0].text` and their CommCare case ID first. A good approach is for the patient registration form to join their given name with their family name to determine their full name, and map that to `Patient.name[0].text`. (It also makes a good case name.)

Patients are also searched for using "given" and "family" names together. If integration uses patient search, apps should ensure that at least one of those case properties has a value; ideally both.

Multiple values from a single question

A few data types in FHIR, like Coding for example, have more than one property that an integrator might want to set using a single multiple choice question in an app. For Coding, we might want to set the values of both the "code" and "display" properties.

The way an app can achieve this is to use a separator to split the question's chosen value. e.g. The "Johnson & Johnson" option of a "Vaccine type" multiple choice question could have a choice value of `91303|Janssen_COVID-19_Vaccine`. The form could have two hidden value questions:

- "vaccine_type_code", calculated as

```
substring-before(#form/vaccine_type, '|')
```

- "vaccine_type_name", calculated as

```
replace(substring-after(#form/vaccine_type, '|'), '_', ' ')
```

A note about using advanced modules

CommCare can send multiple FHIR resources in a single API call. It does this by wrapping them in a transaction bundle. If the remote FHIR API does not support this, it is possible to build an app that only sends one resource at a time. This is done by ensuring that each form submission touches no more than one case type that is configured for FHIR integration.

When a basic module creates a child case, the form submission will include both the existing parent case and the new child case. If both the parent and child case types are mapped to FHIR resource types, then CommCare will send both resources in a bundle.

We can use an advanced module for creating child cases. They allow us to limit form submissions to only include the new child case.

It is worth stressing that this is not normally necessary, but it may be useful to know that a CommCare app can be built in such a way that it sends only one FHIR resource at a time.

35.1.4 Mapping using the Data Dictionary

CommCare maps case types to FHIR resource types using the Data Dictionary. See *Mapping case properties using the Data Dictionary*.

35.1.5 Mapping using the Admin interface

More advanced mapping is done using the Admin interface. See *Advanced mapping using the Admin interface*.

35.1.6 Testing

App builders and integrators can check the integration as the app is being built, and the case properties are being mapped to FHIR resource properties. The following command starts a HAPI FHIR Docker container:

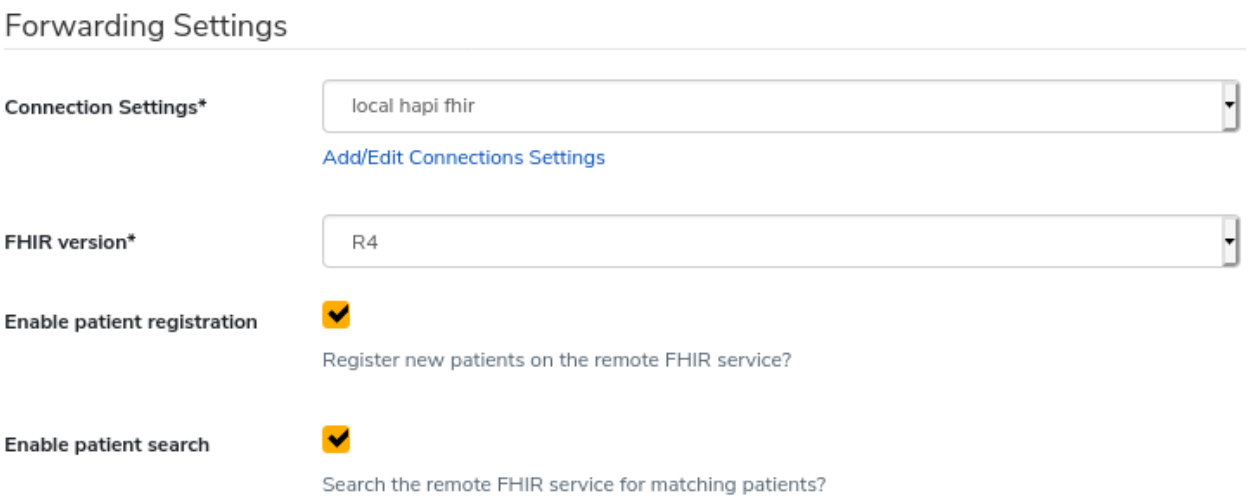
```
$ docker run -it -p 8425:8080 smartonfhir/hapi-5:r4-synthea
```

For a cloud-based environment, a public HAPI FHIR server is available at https://hapi.fhir.org/ for testing. (Do not sent PHI to a public server.)

The FHIR API base URL for the Docker container will be `http://localhost:8425/hapi-fhir-jpaserver/fhir/`. For the public HAPI FHIR server it is `http://hapi.fhir.org/baseR4`.

In CommCare HQ, navigate to "Project Settings" > "Connection Settings" > "Add Connection Settings" to add an entry for the HAPI FHIR instance.

Then under "Project Settings" > "Data Forwarding" > "Forward Cases to a FHIR API", add a service. Select the HAPI FHIR server. You can check "Enable patient search" to test this feature. If you leave it unchecked, CommCare will register a new FHIR Patient for every CommCare client case you create, without searching for an existing Patient.



With data forwarding set up, repeat the following steps to test the app and data mapping:

1. Complete a form using your app.
2. Check "Remote API Logs" to see what requests were made.
3. Select a request to see the request and response details.
4. Search for the corresponding resource in HAPI FHIR to confirm the result.

Testing as the app is built catches problems early, and increases confidence in the app and the integration.

35.2 Importing cases from a remote FHIR service

35.2.1 Overview

CommCare can poll a remote FHIR service, and import resources as new CommCare cases, or update existing ones.

There are three different strategies available to import resources of a particular resource type:

1. Import all of them.
2. Import some of them based on a search filter.
3. Import only the ones that are referred to by resources of a different resource type.

The first two strategies are simple enough. An example of the third strategy might be if we want CommCare to import ServiceRequests (i.e. referrals) from a remote FHIR service, and we want to import only the Patients that those referrals are for.

CommCare can import only those Patients, and also create parent-child case relationships linking a ServiceRequest as a child case of the Patient.

35.2.2 Configuring a FHIRImportConfig

Currently, all configuration is managed via Django Admin (except for adding Connection Settings).

> **Warning:** Django Admin cannot filter select box values by domain. Name your Connection Setting with the name of your domain so that typing the domain name in the select box will find it fast.

In Django Admin, navigate to FHIR > FHIR Import Configs. If you have any FHIRImportConfig instances, they will be listed there, and you can filter by domain. To add a new one, click "Add FHIR Import Config +".

The form is quite straight forward. You will need to provide the ID of a mobile worker in the "Owner ID" field. All cases that are imported will be assigned to this user.

This workflow will not scale for large projects. When such a project comes up, we have planned for two approaches, and will implement one or both based on the project's requirements:

1. Set the owner to a user, group or location.
2. Assign a FHIRImportConfig to a CommCare location, and set ownership to the mobile worker at that location.

35.2.3 Mapping imported FHIR resource properties

Resource properties are mapped via the Admin interface using ValueSource definitions, similar to *Advanced mapping using the Admin interface* for data forwarding and the FHIR API. But there are a few important differences:

The first difference is that FHIRRepeater and the FHIR API use FHIRResourceType instances (rendered as "FHIR Resource Types" in Django Admin) to configure mapping; FHIRImportConfig uses FHIRImportResourceType instances ("FHIR Import Resource Types").

To see what this looks like, navigate to FHIR > FHIR Importer Resource Types, and click "Add FHIR Importer Resource Type".

Select the FHIR Import Config, set the name of the FHIR resource type, and select the case type.

Note: The resource types you can import are not limited to the resource types that can be managed using the Data Dictionary. But if you want to send the same resources back to FHIR when they are modified in CommCare, then you will either need to stick to the Data Dictionary FHIR resource types limitation, or add the resource type you want to the list in corehq/motech/fhir/const.py.)

The "Import related only" checkbox controls that third import strategy mentioned earlier.

"Search params" is a dictionary of search parameters and their values to filter the resources to be imported. Reference documentation for the resource type will tell you what search parameters are available. (e.g. Patient search parameters)

"Import related only" and the "Search params" are applied together, to allow you to filter related resources.

There is a second important difference between FHIRImportResourceType and FHIRResourceType: With FHIRResourceType, the ValueSource configurations are used for *building* a FHIR resource. With FHIRImportResourceType they are used for *navigating* a FHIR resource.

So FHIRResourceType might include ValueSource configs for setting a Patient's phone number. They might look like this:

```
{
  "jsonpath":"$.telecom[0].system",
  "value": "phone"
}
```

```
{
  "jsonpath":"$.telecom[0].value",
  "case_property": "phone_number"
}
```

When we are navigating an imported resource to find the value of the Patient's phone number, we don't know whether it will be the first item in the "telecom" list. Instead, we search the "telecom" list for the item whose "system" is set to "phone". That is defined like this:

```
{
  "jsonpath":"$.telecom[?system='phone'].value",
  "case_property": "phone_number"
}
```

The third difference is that although the mappings will look the same for the most part, they may map to different case properties. This is because we have found that projects often want a mobile worker to check some of the imported values before overwriting existing values on the case. It is wise to confirm with the delivery team how to treat case properties that can be edited.

35.2.4 Configuring related resources

If a FHIR Importer resource type has "Import related only" checked, we need to configure how the resource type is related.

Navigate to FHIR > JSON Path to resource types, and click "Add JSON Path to resource type".

A ServiceRequest.subject is a reference to the Patient it is referring.

Set "Resource type" to "ServiceRequest".

Set "JSONPath" to "$.subject.reference".

Set "Related resource type" to "Patient".

If the "Related resource is parent" checkbox is not checked, then CommCare will just create a case for the Patient. If it is checked, then CommCare will also create an index on the case for the ServiceRequest as a child case, and link it to the case for the Patient as its parent case.

The child-to-parent relationship will follow the direction of the reference. So if a Foo resource has a reference to a Bar resource, then in CommCare the "foo" case will be the child of the "bar" case.

35.2.5 Testing FHIRImportConfig configuration

To make sure your configuration works as expected, add some test data to a FHIR server, and import it.

Here is a script I used for adding test data:

add_service_request.py:

```
#!/usr/bin/env python3
from datetime import date, timedelta
from random import choice
import requests
import string

BASE_URL = 'http://localhost:8425/hapi-fhir-jpaserver/fhir/'  # ends in '/'

GIVEN_NAMES = 'Alice Bethany Claire Deborah Eilidh Francesca'.split()
FAMILY_NAMES = 'Apple Barker Carter Davenport Erridge Franks'.split()
NOTE = 'Patient missed appt. Pls follow up.'


def add_patient():
    given_name = choice(GIVEN_NAMES)
    family_name = choice(FAMILY_NAMES)
    full_name = f'{given_name} {family_name}'
    patient = {
        'resourceType': 'Patient',
        'name': [{
            'given': [given_name],
            'family': family_name,
            'text': full_name,
        }],
        'telecom': [{
            'system': 'phone',
            'value': create_phone_number(),
        }],
    }
    response = requests.post(
        f'{BASE_URL}Patient/',
        json=patient,
        headers={'Accept': 'application/json'},
    )
    assert 200 <= response.status_code < 300, response.text
    return response.json()['id'], full_name
```

(continues on next page)

(continued from previous page)

```
def add_service_request(patient_id, patient_name):
    service_request = {
        'resourceType': 'ServiceRequest',
        'status': 'active',
        'intent': 'directive',
        'subject': {
            'reference': f'Patient/{patient_id}',
            'display': patient_name,
        },
        'note': [{
            'text': NOTE,
        }]
    }
    response = requests.post(
        f'{BASE_URL}ServiceRequest/',
        json=service_request,
        headers={'Accept': 'application/json'},
    )
    assert 200 <= response.status_code < 300, response.text


def create_phone_number():
    number = ''.join([choice(string.digits) for _ in range(9)])
    return f'0{number[0:2]} {number[2:5]} {number[5:]}'


if __name__ == '__main__':
    patient_id, patient_name = add_patient()
    add_service_request(patient_id, patient_name)
```

From a Python console, run your import with:

```
>>> from corehq.motech.fhir.tasks import run_daily_importers
>>> run_daily_importers()
```

35.3 The FHIR API

CommCare offers a FHIR R4 API. It returns responses in JSON.

The FHIR API is not yet targeted at external users. API users must be superusers.

The API focuses on the Patient resource. The endpoint for a Patient would be `https://www.commcarehq.org/a/<domain>/fhir/R4/Patient/<case-id>`

To search for the patient's Observations, the API accepts the "patient_id" search filter. For example, `https://www.commcarehq.org/a/<domain>/fhir/R4/Observation/?patient_id=<case-id>`

35.3.1 Using the FHIR API

Dimagi offers tools to help others use the CommCare HQ FHIR API:

A CommCare HQ Sandbox

The sandbox is a suite of Docker containers that launches a complete CommCare HQ instance and the services it needs:

1. Clone the CommCare HQ repository:

```
$ git clone https://github.com/dimagi/commcare-hq.git
```

2. Launch CommCare HQ using the script provided:

```
$ scripts/docker runserver
```

CommCare HQ is now accessible at `http://localhost:8000/`

A Reference API Client

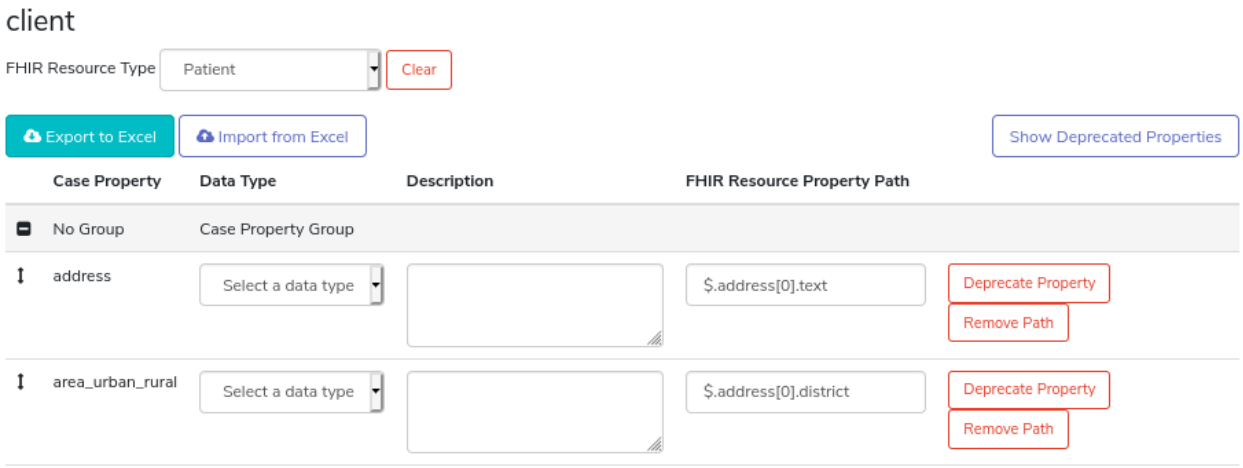
A simple example of a web service that calls the CommCare HQ FHIR API to retrieve patient data is available as a reference.

You can find it implemented using the Flask Python web framework, or FastAPI for async Python.

35.4 Mapping case properties using the Data Dictionary

The FHIR Resources to be sent by data forwarding, or shared by the FHIR API, are configured using the Data Dictionary. (Under the "Data" menu, choose "View All", and navigate to "Data Dictionary")

The Data Dictionary is enabled using the "Data Dictionary" feature flag.



For example, let us imagine mapping a "person" case type to the "Patient" FHIR resource type. You would select the "person" case type from the list of case types on the left.

Set the value of the "FHIR ResourceType" dropdown to "Patient".

The Data Dictionary supports simple mapping of case properties. You will see a table of case properties, and a column titled "FHIR Resource Property Path". This is where to enter the JSONPath to the resource property to set.

An example will help to illustrate this: Imagine the "person" case type has a "first_name" case property, and assume we want to map its value to the patient's given name.

1. Check the structure of a FHIR Patient on the HL7 website.
2. Note Patient.name has a cardinality of "0..*", so it is a list.
3. Check the HumanName datatype.
4. Note Patient.name.given also has a cardinality of "0..*".
5. Refer to JSONPath expression syntax to see how to refer to Patient's first given name. ... You will find it is `$.name[0].given[0]`. (To become more familiar with JSONPath, playing with the JSONPath Online Evaluator can be fun and useful.)
6. Fill the value "$.name[0].given[0]" into the "FHIR Resource Property Path" field for the "first_name" case property.
7. You can test this using a tool like the Postman REST Client or the RESTED Firefox add-on / Chrome extension, call the CommCare FHIR API endpoint for a patient. e.g. `https://www.commcarehq.org/a/<domain>/fhir/R4/Patient/<case-id>` (You will need to configure the REST client for API key authentication.) You will get a result similar to the following:

```
{
  "id": "<case-id>",
  "resourceType": "Patient",
  "name": [
    {
      "given": [
        "Jane"
      ]
    }
  ]
}
```

8. Use JSONPath to map the rest of the case properties you wish to represent in the Patient resource. For a simpler example, a "date_of_birth" case property would be mapped to "$.birthDate".

Playing with the JSONPath Online Evaluator can be fun and useful way to become more familiar with JSONPath.

35.5 Advanced mapping using the Admin interface

The Data Dictionary is meant to offer as simple an interface as possible for mapping case properties to FHIR resource properties. But what about FHIR resource properties whose values are not stored in case properties? Or FHIR resource properties whose data types are not the same as their corresponding case properties?

This can done using the Admin site, and is accessible to superusers.

Mappings are configured using ValueSource definitions. For more information about ValueSource, see the *Value Source* documentation.

Open the Admin site, and navigate to "FHIR" > "FHIR resource types".

There is a list of case types that have been mapped to resource types. Filter by domain if the list is long. Select the resource-type/case-type pair to configure.

Let us imagine we are configuring mappings from a "vaccine_dose" case type to the "Immunization" FHIR resource type.

If there are already mappings from case properties to resource properties, they are listed under "FHIR resource properties". They appear in the "Calculated value source" column, and shown as a JSON document. e.g.

```
{
  "case_property": "vaccine_type_code",
  "jsonpath": "$.vaccineCode.coding[0].code"
}
```

Continuing with the vaccineCode example, a remote service will need more context to make sense of a code. The Admin interface allows us to specify the coding system that the code applies to. The following two resource properties specify that the code is a CPT 2021 vaccine code.

```
{
  "jsonpath": "$.vaccineCode.coding[0].system",
  "value": "http://www.ama-assn.org/go/cpt"
}
```

```
{
  "jsonpath": "$.vaccineCode.coding[0].version",
  "value": "2021"
}
```

These set the "system" and "version" properties of the Coding instance to constant values.

Next, let us take a look at mapping a property from a parent case. The Immunization resource type has a "programEligibility" property. This is its coding system:

```
{
  "jsonpath": "$.programEligibility[0].coding[0].system",
  "value": "http://terminology.hl7.org/CodeSystem/immunization-program-eligibility"
}
```

If the value for programEligibility is stored on CommCare's "person" case type, the parent case of the "vaccine_dose" case, here is how to specify a value from the "person" case's "eligible" case property:

```
{
  "supercase_value_source": {
    "jsonpath": "$.programEligibility[0].coding[0].code",
    "case_property": "eligible"
  },
  "identifier": "parent",
  "referenced_type": "person",
  "relationship": "child"
}
```

Casting data types is another important use case for the Admin interface. Here is an example of how we ensure that an integer is sent in JSON format as an integer and not a string:

```
{
  "case_property": "dose_number",
  "jsonpath": "$.protocolApplied.doseNumberPositiveInt",
```

(continues on next page)

(continued from previous page)

```
  "external_data_type": "cc_integer"
}
```

We use the same approach to cast a string of space-separated values to a list of strings. This is particularly useful for the given names of a patient:

```
{
  "case_property": "given_names",
  "jsonpath": "$.name[0].given",
  "external_data_type": "fhir_list_of_string",
  "commcare_data_type": "cc_text"
}
```

For a complete list of the data types available, refer to corehq/motech/const.py and corehq/motech/fhir/const.py in the source code.

Note: Mappings are not designed for transforming values, just, well, mapping them. It is better to do more complex transformations inside a CommCare form, and store the result in a hidden value question. See the *Multiple values from a single question* section under *Forwarding Cases to a FHIR API* as an example.

CHAPTER

THIRTYSIX

THE MOTECH OPENMRS & BAHMNI MODULE

See the MOTECH README for a brief introduction to OpenMRS and Bahmni in the context of MOTECH.

36.1 OpenmrsRepeater

class corehq.motech.openmrs.repeaters.**OpenmrsRepeater**(**args*, ***kwargs*)

`OpenmrsRepeater` is responsible for updating OpenMRS patients with changes made to cases in CommCare. It is also responsible for creating OpenMRS "visits", "encounters" and "observations" when a corresponding visit form is submitted in CommCare.

The `OpenmrsRepeater` class is different from most repeater classes in three details:

1. It has a case type and it updates the OpenMRS equivalent of cases like the `CaseRepeater` class, but it reads forms like the `FormRepeater` class. So it subclasses `CaseRepeater` but its payload format is `form_json`.
2. It makes many API calls for each payload.
3. It can have a location.

Parameters

- **repeater_type** (*CharField*) – Repeater type
- **id** (*UUIDField*) – Primary key: Id
- **domain** (*CharIdField*) – Domain
- **name** (*CharField*) – Name
- **format** (*CharField*) – Format
- **request_method** (*CharField*) – Request method
- **is_paused** (*BooleanField*) – Is paused
- **next_attempt_at** (*DateTimeField*) – Next attempt at
- **last_attempt_at** (*DateTimeField*) – Last attempt at
- **options** (*JSONField*) – Options
- **connection_settings_id** (*IntegerField*) – Connection settings id
- **is_deleted** (*BooleanField*) – Is deleted
- **last_modified** (*DateTimeField*) – Last modified
- **date_created** (*DateTimeField*) – Date created

Reverse relationships:

Parameters

repeat_records (Reverse ForeignKey from SQLRepeatRecord) – All repeat records of this repeater (related name of repeater)

36.2 OpenMRS Repeater Location

Assigning an OpenMRS repeater to a location allows a project to integrate with multiple OpenMRS/Bahmni servers.

Imagine a location hierarchy like the following:

- (country) South Africa
  - (province) Gauteng
  - (province) Western Cape
    - (district) City of Cape Town
    - (district) Central Karoo
      - (municipality) Laingsburg

Imagine we had an OpenMRS server to store medical records for the city of Cape Town, and a second OpenMRS server to store medical records for the central Karoo.

When a mobile worker whose primary location is set to Laingsburg submits data, MOTECH will search their location and the locations above it until it finds an OpenMRS server. That will be the server that their data is forwarded to.

When patients are imported from OpenMRS, either using its Atom Feed API or its Reporting API, and new cases are created in CommCare, those new cases must be assigned an owner.

The owner will be the *first* mobile worker found in the OpenMRS server's location. If no mobile workers are found, the case's owner will be set to the location itself. A good way to manage new cases is to have just one mobile worker,

like a supervisor, assigned to the same location as the OpenMRS server. In the example above, in terms of organization levels, it would make sense to have a supervisor at the district level and other mobile workers at the municipality level.

See also: *PatientFinder*

36.3 OpenmrsConfig

class corehq.motech.openmrs.openmrs_config.**OpenmrsConfig**(*_obj=None*, ***kwargs*)

Configuration for an OpenMRS repeater is stored in an `OpenmrsConfig` document.

The `case_config` property maps CommCare case properties (mostly) to patient data, and uses the `OpenmrsCaseConfig` document schema.

The `form_configs` property maps CommCare form questions (mostly) to event, encounter and observation data, and uses the `OpenmrsFormConfig` document schema.

36.4 An OpenMRS Patient

The way we map case properties to an OpenMRS patient is based on how OpenMRS represents a patient. Here is an example of an OpenMRS patient (with some fields removed):

```
{
  "uuid": "d95bf6c9-d1c6-41dc-aecf-1c06bd71386c",
  "display": "GAN200000 - Test DrugDataOne",

  "identifiers": [
    {
      "uuid": "6c5ab204-a128-48f9-bfb2-3f65fd06785b",
      "identifier": "GAN200000",
      "identifierType": {
        "uuid": "81433852-3f10-11e4-adec-0800271c1b75",
      }
    }
  ],

  "person": {
    "uuid": "d95bf6c9-d1c6-41dc-aecf-1c06bd71386c",
    "display": "Test DrugDataOne",
    "gender": "M",
    "age": 3,
    "birthdate": "2014-01-01T00:00:00.000+0530",
    "birthdateEstimated": false,
    "dead": false,
    "deathDate": null,
    "causeOfDeath": null,
    "deathdateEstimated": false,
    "birthtime": null,

    "attributes": [
      {
        "display": "primaryContact = 1234",
```

(continues on next page)

(continued from previous page)


```
        "uuid": "2869508d-3484-4eb7-8cc0-ecaa33889cd2",
        "value": "1234",
        "attributeType": {
          "uuid": "c1f7fd17-3f10-11e4-adec-0800271c1b75",
          "display": "primaryContact"
        }
      },
      {
        "display": "caste = Tribal",
        "uuid": "06ab9ef7-300e-462f-8c1f-6b65edea2c80",
        "value": "Tribal",
        "attributeType": {
          "uuid": "c1f4239f-3f10-11e4-adec-0800271c1b75",
          "display": "caste"
        }
      },
      {
        "display": "General",
        "uuid": "b28e6bbc-91aa-4ba4-8714-cdde0653eb90",
        "value": {
          "uuid": "c1fc20ab-3f10-11e4-adec-0800271c1b75",
          "display": "General"
        },
        "attributeType": {
          "uuid": "c1f455e7-3f10-11e4-adec-0800271c1b75",
          "display": "class"
        }
      }
    ],

    "preferredName": {
      "display": "Test DrugDataOne",
      "uuid": "760f18ea-9321-4c31-9a43-338089fc5b4b",
      "givenName": "Test",
      "familyName": "DrugDataOne"
    },

    "preferredAddress": {
      "display": "123",
      "uuid": "c41f82e2-6af2-459c-96ff-26b66c8887ae",
      "address1": "123",
      "address2": "gp123",
      "address3": "Raigarh",
      "cityVillage": "RAIGARH",
      "countyDistrict": "Raigarh",
      "stateProvince": "Chattisgarh",
      "country": null,
      "postalCode": null
    },

    "names": [
      {
```


(continues on next page)

(continued from previous page)


```
        "display": "Test DrugDataOne",
        "uuid": "760f18ea-9321-4c31-9a43-338089fc5b4b",
        "givenName": "Test",
        "familyName": "DrugDataOne"
      }
    ],

    "addresses": [
      {
        "display": "123",
        "uuid": "c41f82e2-6af2-459c-96ff-26b66c8887ae",
        "address1": "123",
        "address2": "gp123",
        "address3": "Raigarh",
        "cityVillage": "RAIGARH",
        "countyDistrict": "Raigarh",
        "stateProvince": "Chattisgarh",
        "country": null,
        "postalCode": null
      }
    ]
  }
}
```


There are several things here to note:

- A patient has a UUID, identifiers, and a person.
- Other than "uuid", most of the fields that might correspond to case properties belong to "person".
- "person" has a set of top-level items like "gender", "age", "birthdate", etc. And then there are also "attributes". The top-level items are standard OpenMRS person properties. "attributes" are custom, and specific to this OpenMRS instance. Each attribute is identified by a UUID.
- There are two kinds of custom person attributes:
  1. Attributes that take any value (of its data type). Examples from above are "primaryContact = 1234" and "caste = Tribal".
  2. Attributes whose values are selected from a set. An example from above is "class", which is set to "General". OpenMRS calls these values "Concepts", and like everything else in OpenMRS each concept value has a UUID.
- A person has "names" and a "preferredName", and similarly "addresses" and "preferredAddress". Case properties are only mapped to preferredName and preferredAddress. We do not keep track of other names and addresses.

36.5 OpenmrsCaseConfig

Now that we know what a patient looks like, the `OpenmrsCaseConfig` schema will make more sense. It has the following fields that correspond to OpenMRS's fields:

- patient_identifiers
- person_properties
- person_attributes
- person_preferred_name
- person_preferred_address

Each of those assigns values to a patient one of three ways:

1. It can assign a constant. This uses the "value" key. e.g.

```
"person_properties": {
  "birthdate": {
    "value": "Oct 7, 3761 BCE"
  }
}
```

2. It can assign a case property value. Use "case_property" for this. e.g.

```
"person_properties": {
  "birthdate": {
    "case_property": "dob"
  }
}
```

3. It can map a case property value to a concept UUID. Use "case_property" with "value_map" to do this. e.g.

```
"person_attributes": {
  "c1f455e7-3f10-11e4-adec-0800271c1b75": {
    "case_property": "class",
    "value_map": {
      "sc": "c1fcd1c6-3f10-11e4-adec-0800271c1b75",
      "general": "c1fc20ab-3f10-11e4-adec-0800271c1b75",
      "obc": "c1fb51cc-3f10-11e4-adec-0800271c1b75",
      "other_caste": "c207073d-3f10-11e4-adec-0800271c1b75",
      "st": "c20478b6-3f10-11e4-adec-0800271c1b75"
    }
  }
}
```

Note: An easy mistake when configuring `person_attributes`: The OpenMRS UUID of a person attribute type is different from the UUID of its concept. For the person attribute type UUID, navigate to *Administration* > *Person* > **Manage PersonAttribute Types* and select the person attribute type you want. Note the greyed-out UUID. This is the UUID that you need. If the person attribute type is a concept, navigate to *Administration* > *Concepts* > *View Concept Dictionary* and search for the person attribute type by name. Select it from the search results. Note the UUID of the concept is different. Select each of its answers. Use their UUIDs in `value_map`.

There are two more `OpenmrsCaseConfig` fields:

- match_on_ids
- patient_finder

match_on_ids is a list of patient identifiers. They can be all or a subset of those given in OpenmrsCaseConfig.patient_identifiers. When a case is updated in CommCare, these are the IDs to be used to select the corresponding patient from OpenMRS. This is done by repeater_helpers.get_patient_by_id()

This is sufficient for projects that import their patient cases from OpenMRS, because each CommCare case will have a corresponding OpenMRS patient, and its ID, or IDs, will have been set by OpenMRS.

Note: MOTECH has the ability to create or update the values of patient identifiers. If an app offers this ability to users, then that identifier should not be included in match_on_ids. If the case was originally matched using only that identifier and its value changes, MOTECH may be unable to match that patient again.

For projects where patient cases can be registered in CommCare, there needs to be a way of finding a corresponding patient, if one exists.

If repeater_helpers.get_patient_by_id() does not return a patient, we need to search OpenMRS for a corresponding patient. For this we use PatientFinders. OpenmrsCaseConfig.patient_finder will determine which class of PatientFinder the OpenMRS repeater must use.

36.6 PatientFinder

class corehq.motech.openmrs.finders.**PatientFinder**(*_obj=None*, ***kwargs*)

The PatientFinder base class was developed as a way to handle situations where patient cases are created in CommCare instead of being imported from OpenMRS.

When patients are imported from OpenMRS, they will come with at least one identifier that MOTECH can use to match the case in CommCare with the corresponding patient in OpenMRS. But if the case is registered in CommCare then we may not have an ID, or the ID could be wrong. We need to search for a corresponding OpenMRS patient.

Different projects may focus on different kinds of case properties, so it was felt that a base class would allow some flexibility.

The PatientFinder.wrap() method allows you to wrap documents of subclasses.

The PatientFinder.find_patients() method must be implemented by subclasses. It returns a list of zero, one, or many patients. If it returns one patient, the OpenmrsRepeater.find_or_create_patient() will accept that patient as a true match.

Note: The consequences of a false positive (a Type II error) are severe: A real patient will have their valid values overwritten by those of someone else. So PatientFinder subclasses should be written and configured to skew towards false negatives (Type I errors). In other words, it is much better not to choose a patient than to choose the wrong patient.

36.6.1 Creating Missing Patients

If a corresponding OpenMRS patient is not found for a CommCare case, then `PatientFinder` has the option to create a patient in OpenMRS. This is managed with the optional `create_missing` property. Its value defaults to `false`. If it is set to `true`, then it will create a new patient if none are found.

For example:

```
"patient_finder": {
  "doc_type": "WeightedPropertyPatientFinder",
  "property_weights": [
    {"case_property": "given_name", "weight": 0.5},
    {"case_property": "family_name", "weight": 0.6}
  ],
  "searchable_properties": ["family_name"],
  "create_missing": true
}
```

If more than one matching patient is found, a new patient will not be created.

All required properties must be included in the payload. This is sure to include a name and a date of birth, possibly estimated. It may include an identifier. You can find this out from the OpenMRS Administration UI, or by testing the OpenMRS REST API.

36.6.2 WeightedPropertyPatientFinder

class corehq.motech.openmrs.finders.**WeightedPropertyPatientFinder**(**args*, ***kwargs*)

The `WeightedPropertyPatientFinder` class finds OpenMRS patients that match CommCare cases by assigning weights to case properties, and adding the weights of matching patient properties to calculate a confidence score.

36.7 OpenmrsFormConfig

MOTECH sends case updates as changes to patient properties and attributes. Form submissions can also create Visits, Encounters and Observations in OpenMRS.

Configure this in the "Encounters config" section of the OpenMRS Forwarder configuration.

An example value of "Encounters config" might look like this:

```
[
  {
    "doc_type": "OpenmrsFormConfig",
    "xmlns": "http://openrosa.org/formdesigner/9481169B-0381-4B27-BA37-A46AB7B4692D",
    "openmrs_start_datetime": {
      "form_question": "/metadata/timeStart",
      "external_data_type": "omrs_date"
    },
    "openmrs_visit_type": "c22a5000-3f10-11e4-adec-0800271c1b75",
    "openmrs_encounter_type": "81852aee-3f10-11e4-adec-0800271c1b75",
    "openmrs_observations": [
      {
        "doc_type": "ObservationMapping",
```

(continues on next page)

(continued from previous page)

```
        "concept": "5090AAAAAAAAAAAAAAAAAAAAAAAAAA",
        "value": {
          "form_question": "/data/height"
        }
      },
      {
        "doc_type": "ObservationMapping",
        "concept": "e1e055a2-1d5f-11e0-b929-000c29ad1d07",
        "value": {
          "form_question": "/data/lost_follow_up/visit_type",
          "value_map": {
            "Search": "e1e20e4c-1d5f-11e0-b929-000c29ad1d07",
            "Support": "e1e20f5a-1d5f-11e0-b929-000c29ad1d07"
          }
        },
        "case_property": "last_visit_type"
      }
    ]
  }
]
```

This example uses two form question values, "/data/height" and "/data/lost_follow_up/visit_type". They are sent as values of OpenMRS concepts "5090AAAAAAAAAAAAAAAAAAAAAAAAAA" and "e1e055a2-1d5f-11e0-b929-000c29ad1d07" respectively.

The OpenMRS concept that corresponds to the form question "/data/height" accepts a numeric value.

The concept for "/data/lost_follow_up/visit_type" accepts a discrete set of values. For this we use `FormQuestionMap` to map form question values, in this example "Search" and "Support", to their corresponding concept UUIDs in OpenMRS.

The `case_property` setting for `ObservationMapping` is optional. If it is set, when Observations are imported from OpenMRS (see *Atom Feed Integration* below) then the given case property will be updated with the value from OpenMRS. If the observation mapping is uses `FormQuestionMap` or `CasePropertyMap` with `value_map` (like the "last_visit_type" example above), then the CommCare case will be updated with the CommCare value that corresponds to the OpenMRS value's UUID.

Set the UUIDs of `openmrs_visit_type` and `openmrs_encounter_type` appropriately according to the context of the form in the CommCare app.

`openmrs_start_datetime` is an optional setting. By default, MOTECH will set the start of the visit and the encounter to the time when the form was completed on the mobile worker's device.

To change which timestamp is used, the following values for `form_question` are available:

- "/metadata/timeStart": The timestamp, according to the mobile worker's device, when the form was started
- "/metadata/timeEnd": The timestamp, according to the mobile worker's device, when the form was completed
- "/metadata/received_on": The timestamp when the form was submitted to HQ.

The value's default data type is datetime. But some organisations may need the value to be submitted to OpenMRS as just a date. To do this, set `external_data_type` to `omrs_date`, as shown in the example.

36.8 Provider

Every time a form is completed in OpenMRS, it creates a new Encounter.

Observations about a patient, like their height or their blood pressure, belong to an Encounter; just as a form submission in CommCare can have many form question values.

The OpenMRS Data Model documentation explains that an Encounter can be associated with health care providers.

It is useful to label data from CommCare by creating a Provider in OpenMRS for CommCare.

OpenMRS configuration has a field called "Provider UUID", and the value entered here is stored in `OpenmrsConfig.openmrs_provider`.

There are three different kinds of entities involved in setting up a provider in OpenMRS: A Person instance; a Provider instance; and a User instance.

Use the following steps to create a provider for CommCare:

From the OpenMRS Administration page, choose "Manage Persons" and click "Create Person". Name, date of birth, and gender are mandatory fields. "CommCare Provider" is probably a good name because OpenMRS will split it into a given name ("CommCare") and a family name ("Provider"). CommCare HQ's first Git commit is dated 2009-03-10, so that seems close enough to a date of birth. OpenMRS equates gender with sex, and is quite binary about it. You will have to decided whether CommCare is male or female. When you are done, click "Create Person". On the next page, "City/Village" is a required field. You can set "State/Province" to "Other" and set "City/Village" to "Cambridge". Then click "Save Person".

Go back to the OpenMRS Administration page, choose "Manage Providers" and click "Add Provider". In the "Person" field, type the name of the person you just created. You can also give it an Identifier, like "commcare". Then click Save.

You will need the UUID of the new Provider. Find the Provider by entering its name, and selecting it.

Make a note of the greyed UUID. This is the value you will need for "Provider UUID" in the configuration for the OpenMRS Repeater.

Next, go back to the OpenMRS Administration page, choose "Manage Users" and click "Add User". Under "Use a person who already exists" enter the name of your new person and click "Next". Give your user a username (like "commcare"), and a password. **Under "Roles" select "Provider"**. Click "Save User".

Now CommCare's "Provider UUID" will be recognised by OpenMRS as a provider. Copy the value of the Provider UUID you made a note of earlier into your OpenMRS configuration in CommCare HQ.

36.9 Atom Feed Integration

The OpenMRS Atom Feed Module allows MOTECH to poll feeds of updates to patients and encounters. The feed adheres to the Atom syndication format.

An example URL for the patient feed would be like "http://www.example.com/openmrs/ws/atomfeed/patient/recent".

Example content:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Patient AOP</title>
  <link rel="self" type="application/atom+xml" href="http://www.example.com/openmrs/ws/
↪atomfeed/patient/recent" />
  <link rel="via" type="application/atom+xml" href="http://www.example.com/openmrs/ws/
```

(continues on next page)

(continued from previous page)

```
↪atomfeed/patient/32" />
  <link rel="prev-archive" type="application/atom+xml" href="http://www.example.com/
↪openmrs/ws/atomfeed/patient/31" />
  <author>
    <name>OpenMRS</name>
  </author>
  <id>bec795b1-3d17-451d-b43e-a094019f6984+32</id>
  <generator uri="https://github.com/ICT4H/atomfeed">OpenMRS Feed Publisher</generator>
  <updated>2018-04-26T10:56:10Z</updated>
  <entry>
    <title>Patient</title>
    <category term="patient" />
    <id>tag:atomfeed.ict4h.org:6fdab6f5-2cd2-4207-b8bb-c2884d6179f6</id>
    <updated>2018-01-17T19:44:40Z</updated>
    <published>2018-01-17T19:44:40Z</published>
    <content type="application/vnd.atomfeed+xml"><![CDATA[/openmrs/ws/rest/v1/patient/
↪e8aa08f6-86cd-42f9-8924-1b3ea021aeb4?v=full]]></content>
  </entry>
  <entry>
    <title>Patient</title>
    <category term="patient" />
    <id>tag:atomfeed.ict4h.org:5c6b6913-94a0-4f08-96a2-6b84dbced26e</id>
    <updated>2018-01-17T19:46:14Z</updated>
    <published>2018-01-17T19:46:14Z</published>
    <content type="application/vnd.atomfeed+xml"><![CDATA[/openmrs/ws/rest/v1/patient/
↪e8aa08f6-86cd-42f9-8924-1b3ea021aeb4?v=full]]></content>
  </entry>
  <entry>
    <title>Patient</title>
    <category term="patient" />
    <id>tag:atomfeed.ict4h.org:299c435d-b3b4-4e89-8188-6d972169c13d</id>
    <updated>2018-01-17T19:57:09Z</updated>
    <published>2018-01-17T19:57:09Z</published>
    <content type="application/vnd.atomfeed+xml"><![CDATA[/openmrs/ws/rest/v1/patient/
↪e8aa08f6-86cd-42f9-8924-1b3ea021aeb4?v=full]]></content>
  </entry>
</feed>
```

Similarly, an encounter feed URL would be like "http://www.example.com/openmrs/ws/atomfeed/encounter/recent".

Example content:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Patient AOP</title>
  <link rel="self" type="application/atom+xml" href="https://13.232.58.186/openmrs/ws/
↪atomfeed/encounter/recent" />
  <link rel="via" type="application/atom+xml" href="https://13.232.58.186/openmrs/ws/
↪atomfeed/encounter/335" />
  <link rel="prev-archive" type="application/atom+xml" href="https://13.232.58.186/
↪openmrs/ws/atomfeed/encounter/334" />
  <author>
    <name>OpenMRS</name>
```

(continues on next page)

(continued from previous page)

```
  </author>
  <id>bec795b1-3d17-451d-b43e-a094019f6984+335</id>
  <generator uri="https://github.com/ICT4H/atomfeed">OpenMRS Feed Publisher</generator>
  <updated>2018-06-13T08:32:57Z</updated>
  <entry>
    <title>Encounter</title>
    <category term="Encounter" />
    <id>tag:atomfeed.ict4h.org:af713a2e-b961-4cb0-be59-d74e8b054415</id>
    <updated>2018-06-13T05:08:57Z</updated>
    <published>2018-06-13T05:08:57Z</published>
    <content type="application/vnd.atomfeed+xml"><![CDATA[/openmrs/ws/rest/v1/bahmnicore/
↪bahmniencounter/0f54fe40-89af-4412-8dd4-5eaebe8684dc?includeAll=true]]></content>
  </entry>
  <entry>
    <title>Encounter</title>
    <category term="Encounter" />
    <id>tag:atomfeed.ict4h.org:320834be-e9c8-4b09-a99e-691dff18b3e4</id>
    <updated>2018-06-13T05:08:57Z</updated>
    <published>2018-06-13T05:08:57Z</published>
    <content type="application/vnd.atomfeed+xml"><![CDATA[/openmrs/ws/rest/v1/bahmnicore/
↪bahmniencounter/0f54fe40-89af-4412-8dd4-5eaebe8684dc?includeAll=true]]></content>
  </entry>
  <entry>
    <title>Encounter</title>
    <category term="Encounter" />
    <id>tag:atomfeed.ict4h.org:fca253aa-b917-4166-946e-9da9baa901da</id>
    <updated>2018-06-13T05:09:12Z</updated>
    <published>2018-06-13T05:09:12Z</published>
    <content type="application/vnd.atomfeed+xml"><![CDATA[/openmrs/ws/rest/v1/bahmnicore/
↪bahmniencounter/c6d6c248-8cd4-4e96-a110-93668e48e4db?includeAll=true]]></content>
  </entry>
</feed>
```

At the time of writing, the Atom feeds do not use ETags or offer HEAD requests. MOTECH uses a GET request to fetch the document, and checks the timestamp in the `<updated>` tag to tell whether there is new content.

The feeds are paginated, and the page number is given at the end of the `href` attribute of the `<link rel="via" ...` tag, which is found at the start of the feed. A `<link rel="next-archive" ...` tag indicates that there is a next page.

MOTECH stores the last page number polled in the `OpenmrsRepeater.atom_feed_status["patient"].last_page` and `OpenmrsRepeater.atom_feed_status["encounter"]last_page` properties. When it polls again, it starts at this page, and iterates `next-archive` links until all have been fetched.

If this is the first time MOTECH is polling an Atom feed, it uses the `/recent` URL (as given in the example URL above) instead of starting from the very beginning. This is to allow Atom feed integration to be enabled for ongoing projects that may have a lot of established data. Administrators should be informed that enabling Atom feed integration will not import all OpenMRS patients into CommCare, but it will add CommCare cases for patients created in OpenMRS from the moment Atom feed integration is enabled.

36.9.1 Adding cases for OpenMRS patients

MOTECH needs three kinds of data in order to add a case for an OpenMRS patient:

1. The **case type**. This is set using the OpenMRS Repeater's "Case Type" field (i.e. OpenmrsRepeater.white_listed_case_types). It must have exactly one case type specified.
2. The **case owner**. This is determined using the OpenMRS Repeater's "Location" field (i.e. OpenmrsRepeater.location_id). The owner is set to the first mobile worker (specifically CommCareUser instance) found at that location.
3. The **case properties** to set. MOTECH uses the patient_identifiers, person_properties, person_preferred_name, person_preferred_address, and person_attributes given in "Patient config" (OpenmrsRepeater.openmrs_config.case_config) to map the values of an OpenMRS patient to case properties. All and only the properties in "Patient config" are mapped.

The **name of cases** updated from the Atom feed are set to the display name of the *person* (not the display name of patient because it often includes punctuation and an identifier).

When a new case is created, its case's owner is determined by the CommCare location of the OpenMRS repeater. (You can set the location when you create or edit the OpenMRS repeater in *Project Settings* > *Data Forwarding*.) The case will be assigned to the first mobile worker found at the repeater's location. The intention is that this mobile worker would be a supervisor who can pass the case to the appropriate person.

36.9.2 Importing OpenMRS Encounters

MOTECH can import both patient data and data about encounters using Atom feed integration. This can be used for updating case properties, associating clinical diagnoses with a patient, or managing referrals.

Bahmni includes diagnoses in the data of an encounter. The structure of a diagnosis is similar to that of an observation. Diagnoses can only be imported from Bahmni; Bahmni does not offer an API for adding or updating diagnoses in Bahmni. Configurations for observations and diagnoses are specified separately in the `OpenmrsFormConfig` definition to make the distinction obvious.

Here is an example `OpenmrsFormConfig`:

```
[
  {
    "doc_type": "OpenmrsFormConfig",
    "xmlns": "http://openrosa.org/formdesigner/9ECA0608-307A-4357-954D-5A79E45C3879",
    "openmrs_form": null,
    "openmrs_visit_type": "c23d6c9d-3f10-11e4-adec-0800271c1b75",

    "openmrs_start_datetime": {
      "direction": "in",
      "jsonpath": "encounterDateTime",
      "case_property": "last_clinic_visit_date",
      "external_data_type": "omrs_datetime",
      "commcare_data_type": "cc_date"
    },

    "openmrs_encounter_type": "81852aee-3f10-11e4-adec-0800271c1b75",
    "openmrs_observations": [
      {
        "doc_type": "ObservationMapping",
        "concept": "f8ca5471-4e76-4737-8ea4-7555f6d5af0f",
```

(continues on next page)

(continued from previous page)

```
        "value": {
          "case_property": "blood_group"
        },
        "case_property": "blood_group",
        "indexed_case_mapping": null
      },

      {
        "doc_type": "ObservationMapping",
        "concept": "397b9631-2911-435a-bf8a-ae4468b9c1d4",
        "value": {
          "direction": "in",
          "case_property": "[unused when direction = 'in']"
        },
        "case_property": null,
        "indexed_case_mapping": {
          "doc_type": "IndexedCaseMapping",
          "identifier": "parent",
          "case_type": "referral",
          "relationship": "extension",
          "case_properties": [
            {
              "jsonpath": "value",
              "case_property": "case_name",
              "value_map": {
                "Alice": "397b9631-2911-435a-bf8a-111111111111",
                "Bob": "397b9631-2911-435a-bf8a-222222222222",
                "Carol": "397b9631-2911-435a-bf8a-333333333333"
              }
            },
            {
              "jsonpath": "value",
              "case_property": "owner_id",
              "value_map": {
                "111111111111": "397b9631-2911-435a-bf8a-111111111111",
                "222222222222": "397b9631-2911-435a-bf8a-222222222222",
                "333333333333": "397b9631-2911-435a-bf8a-333333333333"
              }
            },
            {
              "jsonpath": "encounterDateTime",
              "case_property": "referral_date",
              "commcare_data_type": "date",
              "external_data_type": "posix_milliseconds"
            },
            {
              "jsonpath": "comment",
              "case_property": "referral_comment"
            }
          ]
        }
      }
```

(continues on next page)

(continued from previous page)

```
    ],

    "bahmni_diagnoses": [
      {
        "doc_type": "ObservationMapping",
        "concept": "all",
        "value": {
          "direction": "in",
          "case_property": "[unused when direction = 'in']"
        },
        "case_property": null,
        "indexed_case_mapping": {
          "doc_type": "IndexedCaseMapping",
          "identifier": "parent",
          "case_type": "diagnosis",
          "relationship": "extension",
          "case_properties": [
            {
              "jsonpath": "codedAnswer.name",
              "case_property": "case_name"
            },
            {
              "jsonpath": "certainty",
              "case_property": "certainty"
            },
            {
              "jsonpath": "order",
              "case_property": "is_primary",
              "value_map": {
                "yes": "PRIMARY",
                "no": "SECONDARY"
              }
            },
            {
              "jsonpath": "diagnosisDateTime",
              "case_property": "diagnosis_datetime"
            }
          ]
        }
      }
    ]
  }
]
```

There is a lot happening in that definition. Let us look at the different parts.

```
"xmlns": "http://openrosa.org/formdesigner/9ECA0608-307A-4357-954D-5A79E45C3879",
```

Atom feed integration uses the same configuration as data forwarding, because mapping case properties to observations normally applies to both exporting data to OpenMRS and importing data from OpenMRS.

For data forwarding, when the form specified by that XMLNS is submitted, MOTECH will export corresponding observations.

For Atom feed integration, when a new encounter appears in the encounters Atom feed, MOTECH will use the mappings specified for *any* form to determine what data to import. In other words, this XMLNS value is *not used* for Atom feed integration. It is only used for data forwarding.

```
"openmrs_start_datetime": {
  "direction": "in",
  "jsonpath": "encounterDateTime",
  "case_property": "last_clinic_visit_date",
  "external_data_type": "omrs_datetime",
  "commcare_data_type": "cc_date"
},
```

Data forwarding can be configured to set the date and time of the start of an encounter. Atom feed integration can be configured to import the start of the encounter. `"direction": "in"` tells MOTECH that these settings only apply to importing via the Atom feed. `"jsonpath": "encounterDateTime"` fetches the value from the "encounterDateTime" property in the document returned from OpenMRS. `"case_property": "last_clinic_visit_date"` saves that value to the "last_clinic_visit_date" case property. The data type settings convert the value from a datetime to a date.

```
{
  "doc_type": "ObservationMapping",
  "concept": "f8ca5471-4e76-4737-8ea4-7555f6d5af0f",
  "value": {
    "case_property": "blood_group"
  },
  "case_property": "blood_group",
  "indexed_case_mapping": null
},
```

The first observation mapping is configured for both importing and exporting. When data forwarding exports data, it uses `"value": {"case_property": "blood_group"}` to determine which value to send. When MOTECH imports via the Atom feed, it uses `"case_property": "blood_group", "indexed_case_mapping": null` to determine what to do with the imported value. These specific settings tell MOTECH to save the value to the "blood_group" case property, and not to create a subcase.

The next observation mapping gets more interesting:

```
{
  "doc_type": "ObservationMapping",
  "concept": "397b9631-2911-435a-bf8a-ae4468b9c1d4",
  "value": {
    "direction": "in",
    "case_property": "[unused when direction = 'in']"
  },
  "case_property": null,
  "indexed_case_mapping": {
    "doc_type": "IndexedCaseMapping",
    "identifier": "parent",
    "case_type": "referral",
    "relationship": "extension",
    "case_properties": [
      {
        "jsonpath": "value",
        "case_property": "case_name",
        "value_map": {
```

(continues on next page)

(continued from previous page)

```
          "Alice": "397b9631-2911-435a-bf8a-111111111111",
          "Bob": "397b9631-2911-435a-bf8a-222222222222",
          "Carol": "397b9631-2911-435a-bf8a-333333333333"
        }
      },
      {
        "jsonpath": "value",
        "case_property": "owner_id",
        "value_map": {
          "111111111111": "397b9631-2911-435a-bf8a-111111111111",
          "222222222222": "397b9631-2911-435a-bf8a-222222222222",
          "333333333333": "397b9631-2911-435a-bf8a-333333333333"
        }
      },
      {
        "jsonpath": "encounterDateTime",
        "case_property": "referral_date",
        "commcare_data_type": "date",
        "external_data_type": "posix_milliseconds"
      },
      {
        "jsonpath": "comment",
        "case_property": "referral_comment"
      }
    ]
  }
}
```

`"value": {"direction": "in"` … tells MOTECH only to use this observation mapping for importing via the Atom feed.

"indexed_case_mapping" is for creating a subcase. "identifier" is the name of the index that links the subcase to its parent, and the value "parent" is convention in CommCare; unless there are very good reasons to use a different value, "parent" should always be used.

`"case_type": "referral"` gives us a clue about what this configuration is for. The set of possible values of the OpenMRS concept will be IDs of people, who OpenMRS/Bahmni users can choose to refer patients to. Those people will have corresponding mobile workers in CommCare. This observation mapping will need to map the people in OpenMRS to the mobile workers in CommCare.

`"relationship": "extension"` sets what kind of subcase to create. CommCare uses two kinds of subcase relationships: "child"; and "extension". Extension cases are useful for referrals and diagnoses for two reasons: if the patient case is removed, CommCare will automatically remove its referrals and diagnoses; and mobile workers who have access to a patient case will also be able to see all their diagnoses and referrals.

The observation mapping sets four case properties:

1. case_name: This is set to the name of the person to whom the patient is being referred.
2. owner_id: This is the most important aspect of a referral system. "owner_id" is a special case property that sets the owner of the case. It must be set to a mobile worker's ID. When this is done, that mobile worker will get the patient case sent to their device on the next sync.
3. referral_date: The date on which the OpenMRS observation was made.
4. comment: The comment, if any, given with the observation.

The configuration for each case property has a "jsonpath" setting to specify where to get the value from the JSON data of the observation given by the OpenMRS API. See *How to Inspect an Observation or a Diagnosis* below.

Inspecting the observation also helps us with a subtle and confusing setting:

```
{
  "jsonpath": "encounterDateTime",
  "case_property": "referral_date",
  "commcare_data_type": "date",
  "external_data_type": "posix_milliseconds"
},
```

The value for the "referral_date" case property comes from the observation's "encounterDateTime" property. This property has the same name as the "encounterDateTime" property of the encounter. (We used it earlier under the "openmrs_start_datetime" setting to set the "last_clinic_visit_date" case property on the patient case.)

What is confusing is that "external_data_type" is set to "omrs_datetime" for encounter's "encounterDateTime" property. But here, for the observation, "external_data_type" is set to "posix_milliseconds". An "omrs_datetime" value looks like `"2018-01-18T01:15:09.000+0530"`. But a "posix_milliseconds" value looks like `1516218309000`

The only way to know that is to inspect the JSON data returned by the OpenMRS API.

The last part of the configuration deals with Bahmni diagnoses:

```
"bahmni_diagnoses": [
  {
    "doc_type": "ObservationMapping",
    "concept": "all",
    "value": {
      "direction": "in",
      "case_property": "[unused when direction = 'in']"
    },
    "case_property": null,
    "indexed_case_mapping": {
      "doc_type": "IndexedCaseMapping",
      "identifier": "parent",
      "case_type": "diagnosis",
      "relationship": "extension",
      "case_properties": [
        {
          "jsonpath": "codedAnswer.name",
          "case_property": "case_name"
        },
        {
          "jsonpath": "certainty",
          "case_property": "certainty"
        },
        {
          "jsonpath": "order",
          "case_property": "is_primary",
          "value_map": {
            "yes": "PRIMARY",
            "no": "SECONDARY"
          }
        },
        {
```

(continues on next page)

(continued from previous page)

```
          "jsonpath": "diagnosisDateTime",
          "case_property": "diagnosis_datetime"
        }
      ]
    }
  }
]
```

At a glance, it is clear that like the configuration for referrals, this configuration also uses extension cases. There are a few important differences.

`"concept": "all"` tells MOTECH to import all Bahmni diagnosis concepts, not just those that are explicitly configured.

`"value": {"direction": "in"` … The OpenMRS API does not offer the ability to add or modify a diagnosis. "direction" will always be set to "in".

The case type of the extension case is "diagnosis". This configuration sets four case properties. "case_name" should be considered a mandatory case property. It is set to the name of the diagnosis. The value of "jsonpath" is determined by inspecting the JSON data of an example diagnosis. The next section gives instructions for how to do that. Follow the instructions, and as a useful exercise, try to see how the JSON path "codedAnswer.name" was determined from the sample JSON data of a Bahmni diagnosis given by the OpenMRS API.

36.9.3 How to Inspect an Observation or a Diagnosis

To see what the JSON representation of an OpenMRS observation or Bahmni diagnosis is, you can use the official Bahmni demo server.

1. Log in as "superman" with the password "Admin123".
2. Click "Registration" and register a patient.
3. Click the "home" button to return to the dashboard, and click "Clinical".
4. Select your new patient, and create an observation or a diagnosis for them.
5. In a new browser tab or window, open the Encounter Atom feed.
6. Right-click and choose "View Page Source".
7. Find the URL of the latest encounter in the "CDATA" value in the "content" tag. It will look similar to this: "/openmrs/ws/rest/v1/bahmnicore/bahmniencounter/<UUID>?includeAll=true"
8. Construct the full URL, e.g. "https://demo.mybahmni.org/openmrs/ws/rest/v1/bahmnicore/bahmniencounter/<UUID>?includeAll=true" where "<UUID>" is the UUID of the encounter.
9. The OpenMRS REST Web Services API does not make it easy to get a JSON-formatted response using a browser. You can use a REST API Client like Postman, or you can use a command line tool like curl or Wget.

   Fetch the content with the "Accept" header set to "application/json".

   Using curl

   ```
   $ curl -u superman:Admin123 -H "Accept: application/json" \
       "https://demo.mybahmni.org/...?includeAll=true" > encounter.json
   ```

   Using wget

```
$ wget --user=superman --password=Admin123 \
    --header="Accept: application/json" \
    -O encounter.json \
    "https://demo.mybahmni.org/...?includeAll=true"
```

Open `encounter.json` in a text editor that can automatically format JSON for you. (Atom with the pretty-json package installed is not a bad choice.)

CHAPTER

THIRTYSEVEN

HOW DATA MAPPING WORKS

DHIS2-, OpenMRS- and FHIR Integration all use the ValueSource class to map CommCare data to API resources.

A ValueSource is given in JSON format. e.g.

```
{
  "case_property": "active",
  "jsonpath": "$.active"
}
```

This ValueSource maps the value from the case property named "active" to the "active" property of an API resource.

37.1 Different Sources of Values

The ValueSource class supports several different sources of values:

- `case_property`: As seen above, a ValueSource can be used for fetching a value from a case property, or setting a value on a case property.
- `form_question`: Fetches a value from a form question. e.g. "/data/foo/bar" will get the value of a form question named "bar" in the group "foo". Form metadata is also available, e.g. "/metadata/received_on" is the server time when the form submission was received. You can find more details in the source code at corehq.motech.value_source:FormQuestion
- `case_owner_ancestor_location_field`: Specifies a location metadata field name. The ValueSource will start at the location of the case owner, traverse up their location hierarchy, and return the first value it finds for a location with that field. This can be used for mapping CommCare locations to locations or organization units in a remote system.
- `form_user_ancestor_location_field`: Specifies a location metadata field name. Similar to *case_owner_ancestor_location_field* but for forms instead of cases. The ValueSource will start at the location of the user who submitted the form, traverse up their location hierarchy, and return the first value it finds for a location with that field. This can be used for mapping CommCare locations to locations or organization units in a remote system.
- `subcase_value_source`: Defines a ValueSource to be evaluated on the subcases of a case. e.g.

```
{
  "subcase_value_source": {"case_property": "name"}
  "case_type": "child",
  "is_closed": false,
  "jsonpath": "$.childrensNames"
}
```

- `supercase_value_source`: Defines a ValueSource to be evaluated on the parent/host case of a case. e.g.

```
{
  "supercase_value_source": {"case_property": "name"}
  "referenced_type": "mother",
  "jsonpath": "$.mothersName"
}
```

- `value`: A constant value. This can be used for exporting a constant, or it can be combined with *case_property* for importing a constant value to a case property. See corehq.motech.value_source:ConstantValue for more details.

37.2 Data Types

Integrating structured data with remote systems can involve converting data from one format or data type to another. Use data type declarations to cast the data type of a value.

For standard OpenMRS properties (person properties, name properties and address properties) MOTECH will set data types correctly, and integrators do not need to worry about them.

But administrators may want a value that is a date in CommCare to a datetime in a remote system, or vice-versa. To convert from one to the other, set data types for value sources.

The default is for both the CommCare data type and the external data type not to be set. e.g.

```
{
  "expectedDeliveryDate": {
    "case_property": "edd",
    "commcare_data_type": null,
    "external_data_type": null
  }
}
```

To set the CommCare data type to a date and the OpenMRS data type to a datetime for example, use the following:

```
{
  "expectedDeliveryDate": {
    "case_property": "edd",
    "commcare_data_type": "cc_date",
    "external_data_type": "omrs_datetime"
  }
}
```

For the complete list of CommCare data types, see MOTECH constants. For the complete list of DHIS2 data types, see DHIS2 constants. For the complete list of OpenMRS data types, see OpenMRS constants.

37.3 Import-Only and Export-Only Values

In configurations like OpenMRS Atom feed integration that involve both sending data to OpenMRS and importing data from OpenMRS, sometimes some values should only be imported, or only exported.

Use the `direction` property to determine whether a value should only be exported, only imported, or (the default behaviour) both.

For example, to import a patient value named "hivStatus" as a case property named "hiv_status" but not export it, use `"direction": "in"`:

```
{
  "hivStatus": {
    "case_property": "hiv_status",
    "direction": "in"
  }
}
```

To export a form question, for example, but not import it, use `"direction": "out"`:

```
{
  "hivStatus": {
    "case_property": "hiv_status",
    "direction": "out"
  }
}
```

Omit `direction`, or set it to `null`, for values that should be both imported and exported.

37.4 Getting Values From JSON Documents

JSONPath has emerged as a standard for navigating JSON documents. It is supported by PostgreSQL, SQL Server, and others. ValueSource uses it to read values from JSON API resources.

And, in the case of FHIR Integration, it also uses it to build FHIR resources.

See the article by Stefan Goessner, who created JSONPath, for more details.

OpenMRS observations and Bahmni diagnoses can be imported as extension cases of CommCare case. This is useful for integrating patient referrals, or managing diagnoses.

Values from the observation or diagnosis can be imported to properties of the extension case. MOTECH needs to traverse the JSON response from the remote system in order to get the right value. Value sources can use JSONPath to do this.

Here is a simplified example of a Bahmni diagnosis to get a feel for JSONPath:

```
{
  "certainty": "CONFIRMED",
  "codedAnswer": {
    "conceptClass": "Diagnosis",
    "mappings": [
      {
        "code": "T68",
        "name": "Hypothermia",
```

(continues on next page)

(continued from previous page)

```
        "source": "ICD 10 - WHO"
      }
    ],
    "shortName": "Hypothermia",
    "uuid": "f7e8da66-f9a7-4463-a8ca-99d8aeec17a0"
  },
  "creatorName": "Eric Idle",
  "diagnosisDateTime": "2019-10-18T16:04:04.000+0530",
  "order": "PRIMARY"
}
```

The JSONPath for "certainty" is simply "certainty".

The JSONPath for "shortName" is "codedAnswer.shortName".

The JSONPath for "code" is "codedAnswer.mappings[0].code".

For more details, see *How to Inspect an Observation or a Diagnosis* in the documentation for the MOTECH OpenMRS & Bahmni Module.

37.5 The value_source Module

class corehq.motech.value_source.**CaseOwnerAncestorLocationField**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *case_owner_ancestor_location_field: str*)

A reference to a location metadata value. The location may be the case owner, the case owner's location, or the first ancestor location of the case owner where the metadata value is set.

e.g.

```
{
  "doc_type": "CaseOwnerAncestorLocationField",
  "location_field": "openmrs_uuid"
}
```

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *case_owner_ancestor_location_field: str*) → None

Method generated by attrs for class CaseOwnerAncestorLocationField.

classmethod wrap(*data*)

Allows us to duck-type JsonObject, and useful for doing pre-instantiation transforms / dropping unwanted attributes.

class corehq.motech.value_source.**CaseProperty**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *case_property: str*)

A reference to a case property value.

e.g. Get the value of a case property named "dob":

```
{
  "case_property": "dob"
}
```

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *case_property: str*) → None

Method generated by attrs for class CaseProperty.

class corehq.motech.value_source.**CasePropertyConstantValue**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *value: str*, *value_data_type: str = 'cc_text'*, *case_property: str*)

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *value: str*, *value_data_type: str = 'cc_text'*, *case_property: str*) → None

Method generated by attrs for class CasePropertyConstantValue.

class corehq.motech.value_source.**ConstantValue**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *value: str*, *value_data_type: str = 'cc_text'*)

`ConstantValue` provides a `ValueSource` for constant values.

`value` must be cast as `value_data_type`.

`get_value()` returns the value for export. Use `external_data_type` to cast the export value.

`get_import_value()` and `deserialize()` return the value for import. Use `commcare_data_type` to cast the import value.

```
>>> one = ConstantValue.wrap({
...     "value": 1,
...     "value_data_type": COMMCARE_DATA_TYPE_INTEGER,
...     "commcare_data_type": COMMCARE_DATA_TYPE_DECIMAL,
...     "external_data_type": COMMCARE_DATA_TYPE_TEXT,
... })
>>> info = CaseTriggerInfo("test-domain", None)
>>> one.deserialize("foo")
1.0
>>> one.get_value(info)  # Returns '1.0', not '1'. See note below.
'1.0'
```

Note: `one.get_value(info)` returns `'1.0'`, not `'1'`, because `get_commcare_value()` casts `value` as `commcare_data_type` first. `serialize()` casts it from `commcare_data_type` to `external_data_type`.

This may seem counter-intuitive, but we do it to preserve the behaviour of `ValueSource.serialize()`.

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *value: str*, *value_data_type: str = 'cc_text'*) → None

Method generated by attrs for class ConstantValue.

deserialize(*external_value: Any*) → Any

Converts the value's external data type or format to its data type or format for CommCare, if necessary, otherwise returns the value unchanged.

class corehq.motech.value_source.**FormQuestion**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *form_question: str*)

A reference to a form question value.

e.g. Get the value of a form question named "bar" in the group "foo":

```
{
  "form_question": "/data/foo/bar"
}
```

Note: Normal form questions are prefixed with "/data". Form metadata, like "received_on" and "userID", are prefixed with "/metadata".

The following metadata is available:

Name	Description
deviceID	An integer that identifies the user's device
timeStart	The device time when the user opened the form
timeEnd	The device time when the user completed the form
received_on	The server time when the submission was received
username	The user's username without domain suffix
userID	A large unique number expressed in hexadecimal
instanceID	A UUID identifying this form submission

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *form_question: str*) → None

Method generated by attrs for class FormQuestion.

class corehq.motech.value_source.**FormUserAncestorLocationField**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *form_user_ancestor_location_field: str*)

A reference to a location metadata value. The location is the form user's location, or the first ancestor location of the form user where the metadata value is set.

e.g.

```
{
  "doc_type": "FormUserAncestorLocationField",
  "location_field": "dhis_id"
}
```

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *form_user_ancestor_location_field: str*) → None

Method generated by attrs for class FormUserAncestorLocationField.

classmethod wrap(*data*)

Allows us to duck-type JsonObject, and useful for doing pre-instantiation transforms / dropping unwanted attributes.

class corehq.motech.value_source.**SubcaseValueSource**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *subcase_value_source: dict*, *case_types: List[str] | None = None*, *is_closed: bool | None = None*)

A reference to a list of child/extension cases.

Evaluates nested ValueSource config, allowing for recursion.

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *subcase_value_source: dict*, *case_types: List[str] | None = None*, *is_closed: bool | None = None*) → None

Method generated by attrs for class SubcaseValueSource.

set_external_value(*external_data*, *info*)

Builds `external_data` by reference.

Currently implemented for dicts using JSONPath but could be implemented for other types as long as they are mutable.

class corehq.motech.value_source.**SupercaseValueSource**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *supercase_value_source: dict*, *identifier: str | None = None*, *referenced_type: str | None = None*, *relationship: str | None = None*)

A reference to a list of parent/host cases.

Evaluates nested ValueSource config, allowing for recursion.

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*, *supercase_value_source: dict*, *identifier: str | None = None*, *referenced_type: str | None = None*, *relationship: str | None = None*) → None

Method generated by attrs for class SupercaseValueSource.

set_external_value(*external_data*, *info*)

Builds `external_data` by reference.

Currently implemented for dicts using JSONPath but could be implemented for other types as long as they are mutable.

class corehq.motech.value_source.**ValueSource**(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*)

Subclasses model a reference to a value, like a case property or a form question.

Use the `get_value()` method to fetch the value using the reference, and serialize it, if necessary, for the external system that it is being sent to.

__init__(*, *external_data_type: str | None = None*, *commcare_data_type: str | None = None*, *direction: str | None = None*, *value_map: dict | None = None*, *jsonpath: str | None = None*) → None

Method generated by attrs for class ValueSource.

deserialize(*external_value: Any*) → Any

Converts the value's external data type or format to its data type or format for CommCare, if necessary, otherwise returns the value unchanged.

get_value(*case_trigger_info: CaseTriggerInfo*) → Any

Returns the value referred to by the ValueSource, serialized for the external system.

serialize(*value: Any*) → Any

Converts the value's CommCare data type or format to its data type or format for the external system, if necessary, otherwise returns the value unchanged.

set_external_value(*external_data: dict*, *info: CaseTriggerInfo*)

Builds `external_data` by reference.

Currently implemented for dicts using JSONPath but could be implemented for other types as long as they are mutable.

classmethod wrap(*data: dict*)

Allows us to duck-type JsonObject, and useful for doing pre-instantiation transforms / dropping unwanted attributes.

corehq.motech.value_source.**deserialize**(*value_source_config: JsonDict*, *external_value: Any*) → Any

Converts the value's external data type or format to its data type or format for CommCare, if necessary, otherwise returns the value unchanged.

corehq.motech.value_source.**get_case_location**(*case*)

If the owner of the case is a location, return it. Otherwise return the owner's primary location. If the case owner does not have a primary location, return None.

corehq.motech.value_source.**get_form_question_values**(*form_json*)

Given form JSON, returns question-value pairs, where questions are formatted "/data/foo/bar".

e.g. Question "bar" in group "foo" has value "baz":

```
>>> get_form_question_values({'form': {'foo': {'bar': 'baz'}}})
{'/data/foo/bar': 'baz'}
```

corehq.motech.value_source.**get_import_value**(*value_source_config: JsonDict*, *external_data: dict*) → Any

Returns the external value referred to by the value source definition, deserialized for CommCare.

corehq.motech.value_source.**get_value**(*value_source_config: JsonDict*, *case_trigger_info: CaseTriggerInfo*) → Any

Returns the value referred to by the value source definition, serialized for the external system.

CHAPTER

THIRTYEIGHT

GENERAL OVERVIEW

38.1 What is SSO?

Single sign-on (SSO) defines a scheme for authenticating a user with a single username and password across several related, but independent, applications.

Note: Pay attention the difference between **authentication** and **authorization**.

Authentication is the process used to identify whether the credentials a user provides are valid

Authorization is the process used to determine the set of permissions to grant a user that determines what they can view or edit on the system.

38.2 Types of Protocols

Two of the most common types of protocols for SSO in web applications are OIDC (OpenID Connect) and SAML (Security Assertion Markup Language).

CommCare HQ currently uses SAML 2.0 to handle SSO handshake procedures. It is a very mature and secure protocol and is the standard for most enterprise applications.

OIDC is an authentication layer built on top of OAuth 2.0. When compared to SAML it is considered a less mature protocol, but offers the pros of being lightweight and mobile + API friendly. However, not all Identity Providers support OIDC, while all of them likely support SAML. In the future we may consider OIDC support if necessary.

38.3 How Does SSO Work?

SSO is based on a trust relationship between an application, the Service Provider (SP) and an Identity Provider (IdP). CommCare HQ in this case acts as the Service Provider.

To create this trust, x509 certificates are exchanged between the IdP and the SP. The IdP uses the certificate to securely sign (and sometimes encrypt) identity information that it sends back to the SP in the form of a token. Because the SP knows the IdP's public certificate, it knows that the information it receives is coming from a trusted source and knows it's safe to login the user based on the username that it receives in the token.

CHAPTER

THIRTYNINE

ARCHITECTURE

Note: Everything related to SSO on CommCare HQ can be found in *corehq.apps.sso*

We have four primary models in our current SSO architecture, which is based on the SAML 2.0 protocol.

39.1 IdentityProvider

The *IdentityProvider* model is responsible for storing all the certificate information for a given Identity Provider (IdP), like Azure AD.

It is also linked to a *BillingAccount*. This link determines what project spaces fall under the jurisdiction of the *IdentityProvider*. Any project space that has a *Subscription* linked to the *BillingAccount owner* will automatically trust the authentication information provided when a user is logged in with SSO.

Note: While the project spaces subscribed under an *IdentityProvider*'s *BillingAccount* automatically trust the IdP's authentication of a user, a user who signs into CommCare HQ with SSO via that IdP will not automatically gain access to that project space.

Authorization is still managed within CommCare HQ, and a user has to be invited to a project in order to have access to that project.

However, SSO allows a user to login to HQ without the need for going through the usual sign-up process.

39.2 AuthenticatedEmailDomain

A user on CommCare HQ is tied to an *IdentityProvider* based on the email domain of their username. An **email domain** is the portion of an email address that follows the @ sign.

We tie email domains to *IdentityProviders* with the *AuthenticatedEmailDomain* model.

If a user's email domain matches an *AuthenticatedEmailDomain*, during the SSO login process they will be directed to the login workflow determined by the active *IdentityProvider* associated with the *AuthenticatedEmailDomain*.

Note: A user will only be forced to use SSO at login and sign up if *ENFORCE_SSO_LOGIN* in *localsettings* is set to *True*. Otherwise, they will be able to login with a username and password (if they created an account originally this way) or by visiting the *IdentityProvider*'s login URL (see the *get_login_url* method).

39.3 UserExemptFromSingleSignOn

Even if the *ENFORCE_SSO_LOGIN* flag in *localsettings* is set to *True*, we still need to allow certain users the ability to always login to CommCare HQ with their username and password. Generally, these users are one or two enterprise admins (or other special users).

We require at least one user to be exempt from SSO login in the event that the *IdentityProvider* is misconfigured or the certificate expires and a user needs to gain access to their enterprise console to fix the situation.

39.4 TrustedIdentityProvider

Project spaces that are not associated with the *BillingAccount* tied to a given *IdentityProvider* **do not** automatically trust users who have authenticated with that *IdentityProvider*.

In order for a user to access a project space that is outside of their *IdentityProvider*'s jurisdiction, an admin of that project space must first agree to trust the *IdentityProvider* associated with that user.

This trust can either be established from the Web Users list or when inviting a web user that is using SSO to the project.

Once a trust is established, any user authenticating with that *IdentityProvider* who is also a member of the project space can now access the project space as if they had logged in with a username and password.

CHAPTER

FORTY

LOCAL SETUP

Note: Before you begin, please understand that if you are trying to use SAML authentication from *localhost*, it will likely fail on the round-trip handshake, as it is not a public server.

40.1 Pre-Requisites

First, ensure that you have accounting admin privileges (see *add_operations_user* management command)

40.1.1 Create a Project

1. Navigate to + *Add Project*
2. Project Name: *sparrow*
3. (Click) **Create Project**

40.1.2 Create an Enterprise Software Plan

1. Navigate to *Accounting* –> *Software Plans*.
2. **(Click) + Add New Software Plan**
    - Name: *Sparrow Test Flight*
    - Edition: *Enterprise*
    - [x] Is customer software plan
    - (Click) **Create Software Plan**
3. **Navigate to *Roles and Features* tab.**
    - Role: *Enterprise Plan (enterprise_plan_v0)*
    - **Add Features**
        - Choose *User Advanced* –> (Click) **Add Feature**
        - Choose *SMS Advanced* –> (Click) **Add Feature**
    - **Add Products**
        - Choose *CommCare Advanced* –> (Click) **Add Product**

- (Click) **Update Plan Version**

4. **Navigate to *Version Summary* tab.**
    - **Observe**: *Sparrow Test Flight (v1)* details look ok.

40.1.3 Update the Billing Account for Initial Project

1. **Navigate to *Accounting –> Billing Accounts*.**
    - **Report Filters**
        - (Click) **Apply**
2. Navigate to account named *Account for Project sparrow*.
3. **On to *Account* tab.**
    - Name: *Sparrow Inc*
    - **Client Contact Emails:**
        - *client@example.edu*
        - *admin@example.edu*
    - Dimagi Contact Email: *user@example.org*
    - [x] Account is Active
    - [x] Is Customer Billing Account
    - **Enterprise Admin Emails:**
        - *admin@example.edu*
4. **Navigate to *Subscriptions* tab.**
    - (Click) **Edit** (should be only one existing subscription)
5. **Navigate to *Upgrade/Downgrade* tab.**
    - Edition: *Enterprise*
    - New Software Plan: *Sparrow Test Flight (v1)*
    - Note: *Upgrade. (or suitable upgrade note)*
    - (Click) **Change Subscription**

40.1.4 Add more projects to this subscription

1. **Add a new project.**
    1. Navigate to + *Add Project*
    2. Project Name: *song-sparrow*
    3. (Click) **Create Project**
2. **Navigate to *Accounting –> Subscriptions*.**
    - **Report Filters**
        - Project Space: *song-sparrow*

        - (Click) **Apply**
    - Locate subscription for the project (should be only one)
    - (Click) **Edit**
3. **Navigate to the *Upgrade/Downgrade* tab.**
    - Edition: *Enterprise*
    - New Software Plan: *Sparrow Test Flight (v1)*
    - Note: *Upgrade. (or suitable upgrade note)*
    - (Click) **Change Subscription**
4. **Navigate to the *Subscription* tab.**
    - Transfer Subscription To: *Sparrow Inc*
    - (Click) **Update Subscription**
5. Repeat…

40.2 Configure an Identity Provider

1. Navigate to *Accounting –> Identity Providers (SSO)*.
2. (Click) **+ Add New Identity Provider**
    - Billing Account Owner: *Sparrow Inc*
    - Public Name: *Azure AD for Sparrow Inc*
    - Slug for SP Endpoints: *sparrow*
    - (Click) **Create Identity Provider**
3. Navigate to *Authenticated Email Domains* tab.
    - @: *example.edu*
    - (Click) **Add Email Domain**
3. Navigate to *SSO Exempt Users* tab.
    - *admin@example.edu*
    - (Click) **Add User**
4. Navigate to *Identity Provider* tab.
    - [x] Allow Enterprise Admins to edit SSO Enterprise Settings
    - (Click) **Update Configuration**
    - (Click) *Edit Enterprise Settings* (below "Allow…" checkbox)
    - *Configure IdP settings…*

CHAPTER

FORTYONE

ADDING A NEW IDENTITY PROVIDER TYPE

Note: These instructions are for adding a new Identity Provider Type / Service (e.g. Okta, OneLogin, Azure AD, etc.). To add a new active Identity Provider, you can follow the steps in Local Setup or in our SSO Enterprise Guides on Confluence.

41.1 Before Beginning

41.1.1 What Protocol will be used?

As of the writing of this documentation, there are only two protocols used for SSO (*SAML* and *OIDC/OAuth2*). We support both. Of the two, *OIDC* is generally easier to implement than *SAML* but the Identity Provider you wish to add may have a preference for one over the other. For instance, Azure AD's workflows clearly prefer *SAML*.

Another thing to note about protocol choice is that *OIDC* is generally easier to test locally, while *SAML* requires testing on a publicly accessible machine like *staging*.

41.2 Steps for Adding the IdP Type

41.2.1 1. Make model changes and add migrations

You should add the new *IdentityProviderType* to *corehq.apps.sso.models* and create a migration for the *sso* app. Then you should add the *IdentityProviderType* to the appropriate protocol in *IdentityProviderProtocol*'s *get_supported_types()* method.

41.2.2 2. Test out the new provider type locally or on staging

Note: It is important to consider how we will support this new Identity Provider in the long term. Once a new IdP Type is added, we will need to ensure that we can properly do regression tests during QA for any changed SSO code. In order to do this, it's best to ensure that we are able to setup a developer/test account with the Identity Provider.

You can follow the steps in Local Setup of this section to add the new Identity Provider. The biggest challenge will likely be determining where to obtain all the requirements necessary to set up the connection in the Provider's UI. For instance, with *OIDC* you need to take note of where the *Issuer URL*, *Client ID*, and *Client Secret* are in the UI. Some Identity Providers are more challenging to find these than others!

Note: Pay attention to the language and field order of our forms in comparison with what a user might encounter in the Identity Provider's UI. It might be appropriate to change the order of certain fields, sections and/or language for the Enterprise Admin SSO forms to match what the user sees in their provider's UI.

Now you can activate the *IdentityProvider*. It's easiest to use *dimagi.org* as the email domain to map users to, as this is a domain alias for our email accounts (e.g. emails from foo@dimagi.com will also go to foo@dimagi.org). Please do NOT use *dimagi.com*!

If you are doing tests on staging, take note you will likely have to deactivate and remove the email domain from another Identity Provider (like Azure AD or One Login) that previously used this email domain for QA. Also, if you plan to test on staging, the *Dimagi SSO* enterprise account mapped to the *dimagi-sso-1*, *dimagi-sso-2*, and *dimagi-sso-3* domains will be ready to test this new IdP.

41.2.3 3. Log in as an SSO user

With the new *IdentityProvider* configured and active, you can now log in as an SSO user from the login screen. During this test you can identify any additional code changes that need to be made. For instance, a new *OIDC* IdentityProvider might not send the expected *user_data* through, so changes might need to be made where that data is accessed (see *corehq.apps.sso.views.oidc*). A new *SAML* provider might require changes to *get_saml2_config()* that are specific to its requirements (see *corehq.apps.sso.configuration*), but make sure that the existing *IdentityProvider*'s configurations remain unchanged.

41.2.4 4. Walk through workflows with our technical writer

Once you have verified that your changes are correctly logging in users and not throwing any errors, it's time to proceed to setting up a meeting with a technical writer so that the setup process with the new Identity Provider can be appropriately documented on Confluence for our Enterprise partners. Another goal of this meeting should be to document any steps for the QA team to follow when setting up the Identity Provider.

41.2.5 5. Initiate QA

Now it's time to initiate QA on *staging* for the new *IdentityProviderType*. In your QA request, be sure to include the new setup information as documented by our technical writer and any credentials or sign up steps for QA to obtain a developer account.

41.2.6 6. Determine whether a penetration test is required

If the only code changes required were language updates and adding a new *IdentityProviderType* this step can be skipped. However, if you needed to update something like the *SAML* configuration, code in the *SSOBackend*, or areas where authentication / user verification currently happens, then it might be a good idea to schedule a penetration test with our security firm.

41.2.7 7. Pilot test with the Enterprise Partner on Production

Once QA is complete and our security firm has verified that no vulnerabilities exist (if applicable), it is time to onboard our Enterprise Partner in the production environment. This will involve using the Login Enforcement option and the "SSO Test Users" feature that you will find in the *Edit Identity Provider* form. When Login Enforcement is set to *Test*, then only the *SSO Test Users* listed will be required to login with SSO from the homepage. This is a great way for the Enterprise partner to test their setup and processes without immediately forcing all users under their email domain to sign in with SSO.

CHAPTER

FORTYTWO

INTERNATIONALIZATION

This page contains the most common techniques needed for managing CommCare HQ localization strings. For more comprehensive information, consult the Django Docs translations page or this helpful blog post.

42.1 How translations work

There are three major components to the translations system - identifying strings that need to be translated, having human translators write translations for those strings, and then actually inserting those translations when appropriate.

1. `./manage.py makemessages` scans the source code and pulls out strings that have been tagged for translation. It does not actually execute code, and you should not put anything other than string literals in there. No string formatting, no variable references, nothing.
2. Transifex is a third party site we use for translating these strings. Human translators look at the strings generated by makemessages and type up appropriate translations. They have no context beyond what's in the string itself. Be nice to them and give template variables names, so instead of "{} version" they see "{build_date} version", which will be easier to understand.
3. Finally, strings are actually translated when the code is executed on the production servers. `gettext` is a simple function call that takes the string provided and looks for it in the translations for the active language. If present, it returns the corresponding translation. You can think of it like this

```
TRANSLATIONS = {
    "en": {"hello": None},
    "es": {"hello": "hola"},
    "fra": {"hello": "bonjour"},
}

def naive_gettext(str):
    return TRANSLATIONS[get_language()][str] or str
```

42.1.1 Concrete Examples

Table 1: Examples

What's in code	What gets stored	Comments
`_("Hello!")`	"Hello"	
`_("Hello, {}").format(name)`	"Hello, {}"	Hard for the translator to understand
`_("Hello, {name}").format(name=name)`	"Hello, {name}"	Much better
`_("I have a {} {}").format(color, animal)`	"I have a {} {}"	Inscrutable, and the translator can't reorder the args
`_("Hello, {name}".format(name=name))`	"Hello, {name}"	This is an error, it'll be interpolated before the lookup, and that string won't be present in the translations file
`_(f"Today is {day}")`	"Today is {day}"	Also an error, for the same reason.
`DAY = "Friday"; _(DAY)`		`DAY` isn't a string. This is an error, it won't even appear in the translations file.
`_("Hello, ") + name`	"Hello, "	Bad idea. The translator can't move `name` to the beginning or middle.
`_("Hello, ") + name + _(". How are you?")`	"Hello, " ". How are you?"	Even worse. This will result in two strings that will not be translated together.

42.2 Tagging strings in views

TL;DR: `gettext` should be used in code that will be run per-request. `gettext_lazy` should be used in code that is run at module import.

The management command `makemessages` pulls out strings marked for translation so they can be translated via transifex. All three gettext functions mark strings for translation. The actual translation is performed separately. This is where the gettext functions differ.

- `gettext`: The function immediately returns the translation for the currently selected language.
- `gettext_lazy`: The function converts the string to a translation "promise" object. This is later coerced to a string when rendering a template or otherwise forcing the promise.
- `gettext_noop`: This function only marks a string as translation string, it does not have any other effect; that is, it always returns the string itself. This should be considered an advanced tool and generally avoided. It could be useful if you need access to both the translated and untranslated strings.

The most common case is just wrapping text with gettext.

```
from django.utils.translation import gettext as _

def my_view(request):
    messages.success(request, _("Welcome!"))
```

Typically when code is run as a result of a module being imported, there is not yet a user whose locale can be used for translations, so it must be delayed. This is where *gettext_lazy* comes in. It will mark a string for translation, but delay the actual translation as long as possible.

```
class MyAccountSettingsView(BaseMyAccountView):
    urlname = 'my_account_settings'
    page_title = gettext_lazy("My Information")
    template_name = 'settings/edit_my_account.html'
```

When variables are needed in the middle of translated strings, interpolation can be used as normal. However, named variables should be used to ensure that the translator has enough context.

```
message = _("User '{user}' has successfully been {action}.").format(
    user=user.raw_username,
    action=_("Un-Archived") if user.is_active else _("Archived"),
)
```

This ends up in the translations file as:

```
msgid "User '{user}' has successfully been {action}."
```

42.2.1 Using `gettext_lazy`

The *gettext_lazy* method will work in the majority of translation situations. It flags the string for translation but does not translate it until it is rendered for display. If the string needs to be immediately used or manipulated by other methods, this might not work.

When using the value immediately, there is no reason to do lazy translation.

```
return HttpResponse(gettext("An error was encountered."))
```

It is easy to forget to translate form field names, as Django normally builds nice looking text for you. When writing forms, make sure to specify labels with a translation flagged value. These will need to be done with *gettext_lazy*.

```
class BaseUserInfoForm(forms.Form):
    first_name = forms.CharField(label=gettext_lazy('First Name'), max_length=50,␣
↪required=False)
    last_name = forms.CharField(label=gettext_lazy('Last Name'), max_length=50,␣
↪required=False)
```

`gettext_lazy`, a cautionary tale

`gettext_lazy` returns a proxy object, not a string, which can cause complications. These proxies will be coerced to a string when used as one, using the user's language if a request is active and available, and using the default language (English) otherwise.

```
>>> group_name = gettext_lazy("mobile workers")
>>> type(group_name)
django.utils.functional.lazy.<locals>.__proxy__
>>> group_name.upper()
'MOBILE WORKERS'
>>> type(group_name.upper())
str
```

Converting `gettext_lazy` proxy objects to json will crash. You should use `corehq.util.json.CommCareJSONEncoder` to properly coerce it to a string.

```
>>> import json
>>> from django.utils.translation import gettext_lazy
>>> json.dumps({"message": gettext_lazy("Hello!")})
TypeError: Object of type __proxy__ is not JSON serializable
>>> from corehq.util.json import CommCareJSONEncoder
>>> json.dumps({"message": gettext_lazy("Hello!")}, cls=CommCareJSONEncoder)
'{"message": "Hello!"}'
```

42.3 Tagging strings in template files

There are two ways translations get tagged in templates.

For simple and short plain text strings, use the *trans* template tag.

```
{% trans "Welcome to CommCare HQ" %}
```

More complex strings (requiring interpolation, variable usage or those that span multiple lines) can make use of the *blocktrans* tag.

If you need to access a variable from the page context:

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

If you need to make use of an expression in the translation:

```
{% blocktrans with amount=article.price %}
    That will cost $ {{ amount }}.
{% endblocktrans %}
```

This same syntax can also be used with template filters:

```
{% blocktrans with myvar=value|filter %}
    This will have {{ myvar }} inside.
{% endblocktrans %}
```

In general, you want to avoid including HTML in translations. This will make it easier for the translator to understand and manipulate the text. However, you can't always break up the string in a way that gives the translator enough context to accurately do the translation. In that case, HTML inside the translation tags will still be accepted.

```
{% blocktrans %}
    Manage Mobile Workers <small>for CommCare Mobile and
    CommCare HQ Reports</small>
{% endblocktrans %}
```

Text passed as constant strings to template block tag also needs to be translated. This is most often the case in CommCare with forms.

```
{% crispy form _("Specify New Password") %}
```

42.4 Tagging strings in JavaScript

Happily, Django also has support for translations in JavaScript.

JavaScript has a `gettext` function that works exactly the same as in python:

```
gettext("Welcome to CommCare HQ")
```

`gettext` is available globally in HQ, coming from django.js which is available via the base RequireJS setup, so it doesn't need to be added as a dependency to modules that use it.

For translations with interpolated variables, use Underscore's _.template function similarly to python's string formatting, calling `gettext` on the template and __then__ interpolating variables:

```
_.template(gettext("Hello, <%- name %>, it is <%- day %>."))({
    name: firstName,
    day: today,
})
```

42.5 Keeping translations up to date

Once a string has been added to the code, we can update the .po file by running *makemessages*.

To do this for all langauges:

```
$ django-admin makemessages --all
```

It will be quicker for testing during development to only build one language:

```
$ django-admin makemessages -l fra
```

After this command has run, your .po files will be up to date. To have content in this file show up on the website you still need to compile the strings.

```
$ django-admin compilemessages
```

You may notice at this point that not all tagged strings with an associated translation in the .po shows up translated. That could be because Django made a guess on the translated value and marked the string as fuzzy. Any string marked fuzzy will not be displayed and is an indication to the translator to double check this.

Example:

```
#: corehq/__init__.py:103
#, fuzzy
msgid "Export Data"
msgstr "Exporter des cas"
```

CHAPTER

FORTYTHREE

UI HELPERS

There are a few useful UI helpers in our codebase which you should be aware of. Save time and create consistency.

43.1 Paginated CRUD View

Use *corehq.apps.hqwebapp.views.CRUDPaginatedViewMixin* the with a *TemplateView* subclass (ideally one that also subclasses *corehq.apps.hqwebapp.views.BasePageView* or *BaseSectionPageView*) to have a paginated list of objects which you can create, update, or delete.

43.1.1 The Basic Paginated View

In its very basic form (a simple paginated view) it should look like:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedViewMixin):
    # your template should extend hqwebapp/base_paginated_crud.html
    template_name = 'puppyapp/paginated_puppies.html

    # all the user-visible text
    limit_text = "puppies per page"
    empty_notification = "you have no puppies"
    loading_message = "loading_puppies"

    # required properties you must implement:

    @property
    def total(self):
        # How many documents are you paginating through?
        return Puppy.get_total()

    @property
    def column_names(self):
        # What will your row be displaying?
        return [
            "Name",
            "Breed",
            "Age",
        ]

    @property
```

(continues on next page)

(continued from previous page)

```
    def page_context(self):
        # This should at least include the pagination_context that␣
↪CRUDPaginatedViewMixin provides
        return self.pagination_context

    @property
    def paginated_list(self):
        """
        This should return a list (or generator object) of data formatted as follows:
        [
            {
                'itemData': {
                    'id': <id of item>,
                    <json dict of item data for the knockout model to use>
                },
                'template': <knockout template id>
            }
        ]
        """
        for puppy in Puppy.get_all()[self.skip:self.skip + self.limit]:
            yield {
                'itemData': {
                    'id': puppy._id,
                    'name': puppy.name,
                    'breed': puppy.breed,
                    'age': puppy.age,
                },
                'template': 'base-puppy-template',
            }

    def post(self, *args, **kwargs):
        return self.paginate_crud_response
```

The template should use knockout templates to render the data you pass back to the view. Each template will have access to everything inside of *itemData*. Here's an example:

```
{% extends 'hqwebapp/base_paginated_crud.html' %}

{% block pagination_templates %}
<script type="text/html" id="base-puppy-template">
    <td data-bind="text: name"></td>
    <td data-bind="text: breed"></td>
    <td data-bind="text: age"></td>
</script>
{% endblock %}
```

43.1.2 Allowing Creation in your Paginated View

If you want to create data with your paginated view, you must implement the following:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...
    def get_create_form(self, is_blank=False):
        if self.request.method == 'POST' and not is_blank:
            return CreatePuppyForm(self.request.POST)
        return CreatePuppyForm()

    def get_create_item_data(self, create_form):
        new_puppy = create_form.get_new_puppy()
        return {
            'itemData': {
                'id': new_puppy._id,
                'name': new_puppy.name,
                'breed': new_puppy.breed,
                'age': new_puppy.age,
            },
            # you could use base-puppy-template here, but you might want to add an␣
↪update button to the
            # base template.
            'template': 'new-puppy-template',
        }
```

The form returned in *get_create_form()* should make use of crispy forms.

```
from django import forms
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Layout
from crispy_forms.bootstrap import StrictButton, InlineField

class CreatePuppyForm(forms.Form):
    name = forms.CharField()
    breed = forms.CharField()
    dob = forms.DateField()

    def __init__(self, *args, **kwargs):
        super(CreatePuppyForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.form_style = 'inline'
        self.helper.form_show_labels = False
        self.helper.layout = Layout(
            InlineField('name'),
            InlineField('breed'),
            InlineField('dob'),
            StrictButton(
                format_html('<i class="fa fa-plus"></i> {}', "Create Puppy"),
                css_class='btn-primary',
                type='submit'
            )
        )
```

(continues on next page)

(continued from previous page)

```
    def get_new_puppy(self):
        # return new Puppy
        return Puppy.create(self.cleaned_data)
```

43.1.3 Allowing Updating in your Paginated View

If you want to update data with your paginated view, you must implement the following:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...
    def get_update_form(self, initial_data=None):
        if self.request.method == 'POST' and self.action == 'update':
            return UpdatePuppyForm(self.request.POST)
        return UpdatePuppyForm(initial=initial_data)

    @property
    def paginated_list(self):
        for puppy in Puppy.get_all():
            yield {
                'itemData': {
                    'id': puppy._id,
                    ...
                    # make sure you add in this line, so you can use the form in your␣
↪template:
                    'updateForm': self.get_update_form_response(
                        self.get_update_form(puppy.inital_form_data)
                    ),
                },
                'template': 'base-puppy-template',
            }

    @property
    def column_names(self):
        return [
            ...
            # if you're adding another column to your template, be sure to give it a name␣
↪here...
            _('Action'),
        ]

    def get_updated_item_data(self, update_form):
        updated_puppy = update_form.update_puppy()
        return {
            'itemData': {
                'id': updated_puppy._id,
                'name': updated_puppy.name,
                'breed': updated_puppy.breed,
                'age': updated_puppy.age,
            },
            'template': 'base-puppy-template',
        }
```

The *UpdatePuppyForm* should look something like:

```
class UpdatePuppyForm(CreatePuppyForm):
    item_id = forms.CharField(widget=forms.HiddenInput())

    def __init__(self, *args, **kwargs):
        super(UpdatePuppyForm, self).__init__(*args, **kwargs)
        self.helper.form_style = 'default'
        self.helper.form_show_labels = True
        self.helper.layout = Layout(
            Div(
                Field('item_id'),
                Field('name'),
                Field('breed'),
                Field('dob'),
                css_class='modal-body'
            ),
            FormActions(
                StrictButton(
                    "Update Puppy",
                    css_class='btn btn-primary',
                    type='submit',
                ),
                HTML('<button type="button" class="btn btn-default" data-dismiss="modal">
↪Cancel</button>'),
                css_class="modal-footer"
            )
        )

    def update_puppy(self):
        return Puppy.update_puppy(self.cleaned_data)
```

You should add the following to your *base-puppy-template* knockout template:

```
<script type="text/html" id="base-puppy-template">
    ...
    <td> <!-- actions -->
        <button type="button"
                data-toggle="modal"
                data-bind="
                    attr: {
                        'data-target': '#update-puppy-' + id
                    }
                "
                class="btn btn-primary">
            Update Puppy
        </button>

        <div class="modal fade"
             data-bind="
                attr: {
                    id: 'update-puppy-' + id
                }
             ">
```

(continues on next page)

(continued from previous page)

```
            <div class="modal-dialog">
                <div class="modal-content">
                    <div class="modal-header">
                        <button type="button"
                                class="close"
                                data-dismiss="modal"
                                aria-hidden="true">&times;</button>
                        <h3>
                            Update puppy <strong data-bind="text: name"></strong>:
                        </h3>
                    </div>
                    <div class="modal-body">
                        <div data-bind="html: updateForm"></div>
                    </div>
                </div>
            </div>
        </div>
    </td>
</script>
```

43.1.4 Allowing Deleting in your Paginated View

If you want to delete data with your paginated view, you should implement something like the following:

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...

    def get_deleted_item_data(self, item_id):
        deleted_puppy = Puppy.get(item_id)
        deleted_puppy.delete()
        return {
            'itemData': {
                'id': deleted_puppy._id,
                ...
            },
            'template': 'deleted-puppy-template',  # don't forget to implement this!
        }
```

You should add the following to your *base-puppy-template* knockout template:

```
<script type="text/html" id="base-puppy-template">
    ...
    <td> <!-- actions -->
        ...
        <button type="button"
                data-toggle="modal"
                data-bind="
                    attr: {
                        'data-target': '#delete-puppy-' + id
                    }
                "
```

(continues on next page)

(continued from previous page)

```
                class="btn btn-danger">
            <i class="fa fa-remove"></i> Delete Puppy
        </button>

        <div class="modal fade"
             data-bind="
                attr: {
                    id: 'delete-puppy-' + id
                }
             ">
             <div class="modal-dialog">
                <div class="modal-content">
                    <div class="modal-header">
                        <button type="button" class="close" data-dismiss="modal" aria-
↪hidden="true">&times;</button>
                        <h3>
                           Delete puppy <strong data-bind="text: name"></strong>?
                        </h3>
                    </div>
                    <div class="modal-body">
                        <p class="lead">
                            Yes, delete the puppy named <strong data-bind="text: name"></
↪strong>.
                        </p>
                    </div>
                    <div class="modal-footer">
                        <button type="button"
                                class="btn btn-default"
                                data-dismiss="modal">
                            Cancel
                        </button>
                        <button type="button"
                                class="btn btn-danger delete-item-confirm"
                                data-loading-text="Deleting Puppy...">
                            <i class="fa fa-remove"></i> Delete Puppy
                        </button>
                    </div>
                </div>
            </div>
        </div>
    </td>
</script>
```

43.1.5 Refreshing The Whole List Base on Update

If you want to do something that affects an item's position in the list (generally, moving it to the top), this is the feature you want.

You implement the following method (note that a return is not expected):

```
class PuppiesCRUDView(BaseSectionView, CRUDPaginatedMixin):
    ...

    def refresh_item(self, item_id):
        # refresh the item here
        puppy = Puppy.get(item_id)
        puppy.make_default()
        puppy.save()
```

Add a button like this to your template:

```
<button type="button"
        class="btn refresh-list-confirm"
        data-loading-text="Making Default...">
    Make Default Puppy
</button>
```

Now go on and make some CRUD paginated views!

CHAPTER

FORTYFOUR

USING CLASS-BASED VIEWS IN COMMCARE HQ

We should move away from function-based views in django and use class-based views instead. The goal of this section is to point out the infrastructure we've already set up to keep the UI standardized.

44.1 The Base Classes

There are two styles of pages in CommCare HQ. One page is centered (e.g. registration, org settings or the list of projects). The other is a two column, with the left gray column acting as navigation and the right column displaying the primary content (pages under major sections like reports).

44.1.1 A Basic (Centered) Page

To get started, subclass *BasePageView* in *corehq.apps.hqwebapp.views*. *BasePageView* is a subclass of django's *TemplateView*.

```
class MyCenteredPage(BasePageView):
    urlname = 'my_centered_page'
    page_title = "My Centered Page"
    template_name = 'path/to/template.html'

    @property
    def page_url(self):
        # often this looks like:
        return reverse(self.urlname)

    @property
    def page_context(self):
        # You want to do as little logic here.
        # Better to divvy up logical parts of your view in other instance methods or␣
↪properties
        # to keep things clean.
        # You can also do stuff in the get() and post() methods.
        return {
            'some_property': self.compute_my_property(),
            'my_form': self.centered_form,
        }
```

urlname
: This is what django urls uses to identify your page

page_title
This text will show up in the *<title>* tag of your template. It will also show up in the primary heading of your template.

If you want to do use a property in that title that would only be available after your page is instantiated, you should override:

```
@property
def page_name(self):
    return format_html("This is a page for <strong>{}</strong>", self.kitten.name)
```

page_name will not show up in the *<title>* tags, as you can include html in this name.

template_name
Your template should extend *hqwebapp/base_page.html*

It might look something like:

```
{% extends 'hqwebapp/base_page.html' %}

{% block js %}{{ block.super }}
    {# some javascript imports #}
{% endblock %}

{% block js-inline %}{{ block.super }}
    {# some inline javascript #}
{% endblock %}

{% block page_content %}
    My page content! Woo!
{% endblock %}

{% block modals %}{{ block.super }}
    {# a great place to put modals #}
{% endblock %}
```

44.1.2 A Section (Two-Column) Page

To get started, subclass *BaseSectionPageView* in *corehq.apps.hqwebapp.views*. You should implement all the things described in the minimal setup for *A Basic (Centered) Page* in addition to:

```
class MySectionPage(BaseSectionPageView):
    ...  # everything from BasePageView

    section_name = "Data"
    template_name = 'my_app/path/to/template.html'

    @property
    def section_url(self):
        return reverse('my_section_default')
```

Note: Domain Views

If your view uses *domain*, you should subclass *BaseDomainView*. This inserts the domain name as into the *main_context* and adds the *login_and_domain_required* permission. It also implements *page_url* to assume the basic *reverse* for a page in a project: *reverse(self.urlname, args=[self.domain])*

section_name
This shows up as the root name on the section breadcrumbs.

template_name
Your template should extend *hqwebapp/bootstrap3/base_section.html*

It might look something like:

```
{% extends 'hqwebapp/bootstrap3/base_section.html' %}

{% block js %}{{ block.super }}
    {# some javascript imports #}
{% endblock %}

{% block js-inline %}{{ block.super }}
    {# some inline javascript #}
{% endblock %}

{% block main_column %}
    My page content! Woo!
{% endblock %}

{% block modals %}{{ block.super }}
    {# a great place to put modals #}
{% endblock %}
```

44.2 Adding to Urlpatterns

Your *urlpatterns* should look something like:

```
urlpatterns = patterns(
    'corehq.apps.my_app.views',
    ...,
    url(r'^my/page/path/$', MyCenteredPage.as_view(), name=MyCenteredPage.urlname),
)
```

44.3 Hierarchy

If you have a hierarchy of pages, you can implement the following in your class:

```
class MyCenteredPage(BasePageView):
    ...

    @property
    def parent_pages(self):
        # This will show up in breadcrumbs as MyParentPage > MyNextPage > MyCenteredPage
```

(continues on next page)

(continued from previous page)

```
        return [
            {
                'title': MyParentPage.page_title,
                'url': reverse(MyParentPage.urlname),
            },
            {
                'title': MyNextPage.page_title,
                'url': reverse(MyNextPage.urlname),
            },
        ]
```

If you have a hierarchy of pages, it might be wise to implement a *BaseParentPageView* or *Base<InsertSectionName>View* that extends the *main_context* property. That way all of the pages in that section have access to the section's context. All page-specific context should go in *page_context*.

```
class BaseKittenSectionView(BaseSectionPageView):

    @property
    def main_context(self):
        main_context = super(BaseParentView, self).main_context
        main_context.update({
            'kitten': self.kitten,
        })
        return main_context
```

44.4 Permissions

To add permissions decorators to a class-based view, you need to decorate the *dispatch* instance method.

```
class MySectionPage(BaseSectionPageView):
    ...

    @method_decorator(can_edit)
    def dispatch(self, request, *args, **kwargs)
        return super(MySectionPage, self).dispatch(request, *args, **kwargs)
```

44.5 GETs and POSTs (and other http methods)

Depending on the type of request, you might want to do different things.

```
class MySectionPage(BaseSectionPageView):
    ...

    def get(self, request, *args, **kwargs):
        # do stuff related to GET here...
        return super(MySectionPage, self).get(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        # do stuff related to post here...
        return self.get(request, *args, **kwargs)  # or any other HttpResponse object
```

44.5.1 Limiting HTTP Methods

If you want to limit the HTTP request types to just GET or POST, you just have to override the *http_method_names* class property:

```
class MySectionPage(BaseSectionPageView):
    ...
    http_method_names = ['post']
```

Note: Other Allowed Methods

put, *delete*, *head*, *options*, and *trace* are all allowed methods by default.

CHAPTER

FORTYFIVE

FORMS IN HQ

See the HQ Style Guide for guidance on form UI, whether you're creating a custom HTML form or using crispy forms.

45.1 Making forms CSRF safe

HQ is protected against cross site request forgery attacks i.e. if a *POST/PUT/DELETE* request doesn't pass csrf token to corresponding View, the View will reject those requests with a 403 response. All HTML forms and AJAX calls that make such requests should contain a csrf token to succeed. Making a form or AJAX code pass csrf token is easy and the Django docs give detailed instructions on how to do so. Here we list out examples of HQ code that does that

1. If crispy form is used to render HTML form, csrf token is included automagically
2. For raw HTML form, use *{% csrf_token %}* tag in the form HTML, see tag_csrf_example.
3. If request is made via AJAX, it will be automagically protected by *ajax_csrf_setup.js* (which is included in base bootstrap template) as long as your template is inherited from the base template. (*ajax_csrf_setup.js* overrides *$.ajaxSettings.beforeSend* to accomplish this)
4. If an AJAX call needs to override *beforeSend* itself, then the super *$.ajaxSettings.beforeSend* should be explicitly called to pass csrf token. See ajax_csrf_example
5. If HTML form is created in Javascript using raw nodes, csrf-token node should be added to that form. See js_csrf_example_1 and js_csrf_example_2
6. If an inline form is generated using outside of *RequestContext* using *render_to_string* or its cousins, use *csrf_inline* custom tag. See inline_csrf_example
7. If a View needs to be exempted from csrf check (for whatever reason, say for API), use *csrf_exampt* decorator to avoid csrf check. See csrf_exempt_example
8. For any other special unusual case refer to Django docs. Essentially, either the HTTP request needs to have a csrf-token or the corresponding View should be exempted from CSRF check.

CHAPTER

FORTYSIX

DIMAGI JAVASCRIPT GUIDE

Dimagi's internal JavaScript guide for use in the CommCare HQ project.

Javascript code should be functional in all current major browsers, following the ECMAScript 2015 (ES6) standards, and should follow the guidelines described in this document.

46.1 Table of contents

46.1.1 Static Files Organization

All* JavaScript code should be in a .js file and encapsulated as a module using `hqDefine`.

JavaScript files belong in the `static` directory of a Django app, which we structure as follows:

```
myapp/
  static/myapp/
    css/
    font/
    images/
    js/       <= JavaScript
    less/
    lib/      <= Third-party code: This should be rare, since most third-party code␣
↪should be coming from yarn
    spec/     <= JavaScript tests
    ...       <= May contain other directories for data files, i.e., `json/`
  templates/myapp/
    mytemplate.html
```

* There are a few places we do intentionally use script blocks, such as configuring less.js in CommCare HQ's main template, `hqwebapp/base.html`. These are places where there are just a few lines of code that are truly independent of the rest of the site's JavaScript. They are rare.

46.1.2 Managing Dependencies

HQ's JavaScript is being gradually migrated from a legacy, unstructured coding style the relies on the ordering of script tags to instead use RequireJS for dependency management. This means that dependencies are managed differently depending on which area of the code you're working in. This page is a developer's guide to understanding which area you're working in and what that means for your code.

How do I know whether or not I'm working with RequireJS?

You are likely working with RequireJS, as most of HQ has been migrated. However, several major areas have **not** been migrated: app manager, reports, and web apps. Test code also does not currently use RequireJS; see Testing for working with tests.

To tell for sure, look at your module's `hqDefine` call, at the top of the file.

RequireJS modules look like this, with all dependencies loaded as part of `hqDefine`:

```
hqDefine("my_app/js/my_file", [
    "knockout",
    "hqwebapp/js/initial_page_data"
], function (
    ko,
    initialPageData
) {
    var myObservable = ko.observable(initialPageData.get("thing"));
    ...
});
```

Non-RequireJS modules look like this, with no list and no function parameters. HQ modules are loaded using `hqImport` in the body, and third party libraries aren't declared at all, instead relying on globals:

```
hqDefine("my_app/js/my_file", function () {
    var myObservable = ko.observable(hqImport("hqwebapp/js/initial_page_data").get("thing
↪"));
    ...
});
```

How do I write a new page?

New code should be written in RequireJS, which is oriented around a single "entry point" into the page.

Most pages have some amount of logic only relevant to that page, so they have a file that includes that logic and then depends on other modules for shared logic.

data_dictionary.js fits this common pattern:

```
hqDefine("data_dictionary/js/data_dictionary", [    // Module name must match filename
    "jquery",                                       // Common third-party dependencies
    "knockout",
    "underscore",
    "hqwebapp/js/initial_page_data",                // Dependencies on HQ files always␣
↪match the file's path
    "hqwebapp/js/main",
    "analytix/js/google",
```

(continues on next page)

(continued from previous page)

```
    "hqwebapp/js/knockout_bindings.ko",             // This one doesn't need a named␣
↪parameter because it only adds
                                                    // knockout bindings and is not␣
↪referenced in this file
], function (
    $,                                              // These common dependencies use␣
↪these names for compatibility
    ko,                                             // with non-requirejs pages, which␣
↪rely on globals
    _,
    initialPageData,                                // Any dependency that will be␣
↪referenced in this file needs a name.
    hqMain,
    googleAnalytics
) {
    /* Function definitions, knockout model definitions, etc. */

    var dataUrl = initialPageData.reverse('data_dictionary_json');  // Refer to␣
↪dependencies by their named parameter
    ...

    $(function () {
        /* Logic to run on documentready */
    });

    // Other code isn't going to depend on this module, so it doesn't return anything or␣
↪returns 1
});
```

To register your module as the RequireJS entry point, add the `requirejs_main` template tag to your HTML page, near the top but outside of any other block:

```
{% requirejs_main 'data_dictionary/js/data_dictionary' %}
```

Some pages don't have any unique logic but do rely on other modules. These are usually pages that use some common widgets but don't have custom UI interactions.

If your page only relies on a single js module, you can use that as the module's entry point:

```
{% requirejs_main 'locations/js/widgets' %}
```

If your page relies on multiple modules, it still needs one entry point. You can handle this by making a module that has no body, just a set of dependencies, like in gateway_list.js:

```
hqDefine("sms/js/gateway_list", [
    "hqwebapp/js/crud_paginated_list_init",
    "hqwebapp/js/bootstrap3/widgets",
], function () {
    // No page-specific logic, just need to collect the dependencies above
});
```

Then in your HTML page:

```
{% requirejs_main 'sms/js/gateway_list' %}
```

The exception to the above is if your page inherits from a page that doesn't use RequireJS. This is rare, but one example would be adding a new page to app manager that inherits from `managed_app.html`.

How do I add a new dependency to an existing page?

RequireJS

Add the new module to your module's `hqDefine` list of dependencies. If the new dependency will be directly referenced in the body of the module, also add a parameter to the `hqDefine` callback:

```
hqDefine("my_app/js/my_module", [
    ...
    "hqwebapp/js/my_new_dependency",
], function (
    ...,
    myDependency
) {
    ...
    myDependency.myFunction();
});
```

Non-RequireJS

In your HTML template, add a script tag to your new dependency. Your template likely already has scripts included in a `js` block:

```
{% block js %}{{ block.super }}
  ...
  <script src="{% static 'hqwebapp/js/my_new_dependency.js' %}"></script>
{% endblock js %}
```

In your JavaScript file, use `hqImport` to get access to your new dependency:

```
hqDefine("my_app/js/my_module", function () {
    ...
    var myDependency = hqImport("hqwebapp/js/my_new_dependency");
    myDependency.myFunction();
});
```

Do **not** add the RequireJS-style dependency list and parameters. It's easy to introduce bugs that won't be visible until the module is actually migrated, and migrations are harder when they have pre-existing bugs. See the troubleshooting section of the RequireJS Migration Guide if you're curious about the kinds of issues that crop up.

How close are we to a world where we'll just have one set of conventions?

As above, most code is migrated, but most of the remaining areas have significant complexity.

hqDefine.sh generates metrics for the current status of the migration and locates umigrated files. At the time of writing:

```
$ ./scripts/codechecks/hqDefine.sh

98%     (825/843) of HTML files are free of inline scripts
88%     (375/427) of JS files use hqDefine
59%     (249/427) of JS files specify their dependencies
91%     (765/843) of HTML files are free of script tags
```

Why aren't we using something more fully-featured, more modern, or cooler than RequireJS?

This migration began quite a while ago. At the time, the team discussed options and selected RequireJS. The majority of the work done to move to RequireJS has been around reorganizing code into modules and explicitly declaring dependencies, which would be necessary for any kind of modern dependency management. We are not permanently wedded to RequireJS, although it is unlikely that we will migrate to another tool while a significant amount of code is still in the legacy state.

46.1.3 Historical Background on Module Patterns

This page discusses the evolution of HQ's javascript module usage. For practical documentation on writing modules, see Managing Dependencies.

We talk about JavaScript modules, but (at least pre-ES6) JavaScript has no built in support for modules. It's easy to say that, but think about how crazy that is. If this were Python, it would mean your program's main file has to directly list all of the files that will be needed, in the correct order, and then the files share state through global variables. That's insane.

And it's also JavaScript. Fortunately, there are things you can do to enforce and respect the boundaries that keep us sane by following one of a number of patterns.

We're in the process of migrating to RequireJS. Part of this process has included developing a lighter-weight alternative module system called `hqDefine`.

`hqDefine` serves as a stepping stone between legacy code and requirejs modules: it adds encapsulation but not full-blown dependency management. New code is written in RequireJS, but `hqDefine` exists to support legacy code that does not yet use RequireJS.

Before diving into `hqDefine`, I want to talk first about the status quo convention for sanity with no module system. As we'll describe, it's a step down from our current preferred choice, but it's still miles ahead of having no convention at all.

The Crockford Pattern

The Crockford module pattern was popularized in Douglas Crockford's classic 2008 book *JavaScript: The Good Parts*. (At least that's how we heard about it here at Dimagi.) It essentially has two parts.

1. The first and more important of the two parts is to *limit the namespace footprint of each file to a single variable* using a closure (`((function () { /* your code here */ }());`).
2. The second is to pick a single global namespace that you "own" (at Yahoo where he worked, theirs was `YAHOO`; ours is `COMMCAREHQ`) and assign all your modules to properties (or properties of properties, etc.) of that one global namespace.

Putting those together, it looks something like this:

```
MYNAMESPACE.myModule = function () {
    // things inside here are private
    var myPrivateGreeting = "Hello";
    // unless you put them in the return object
    var sayHi = function (name) {
        console.log(myPrivateGreeting + " from my module, " + name);
    };
    return {
        sayHi: sayHi,
        favoriteColor: "blue",
    };
}();
```

This uses a pattern so common in JavaScript that it has its own acronym "IIFE" for "Immediately Invoked Function Expression". By wrapping the contents of the module in a function expression, you can use variables and functions local to your module and inaccessible from outside it.

I should also note that within our code, we've largely only adopted the first of the two steps; i.e. we do not usually expose our modules under `COMMCAREHQ`, but rather as a single module `MYMODULE` or `MyModule`. Often we even slip into exposing these "public" values (`sayHi` and `favoriteColor` in the example above) directly as globals, and you can see how looseness in the application of this pattern can ultimately degenerate into having barely any system at all. Notably, however, exposing modules as globals or even individual functions as globals—but while wrapping their contents in a closure— is still enormously preferable to being unaware of the convention entirely. For example, if you remove the closure from the example above (**don't do this**), you get:

```
/* This is a toxic example, do not follow */

// actually a global
var myPrivateGreeting = "Hello";
// also a global
var sayHi = function (name) {
    console.log(myPrivateGreeting + " from my module, " + name);
};
// also a global
myModule = {
    sayHi: sayHi,
    favoriteColor: "blue",
};
```

In this case, `myPrivateGreeting` (now poorly named), `sayHi`, and `myModule` would now be in the global namespace and thus can be directly referenced *or overwritten*, possibly unintentionally, by any other JavaScript run on the same page.

Despite being a great step ahead from nothing, this module pattern falls short in a number of ways.

1. It relies too heavily on programmer discipline, and has too many ways in which it is easy to cut corners, or even apply incorrectly with good intentions
2. If you use the `COMMCAREHQ.myJsModule` approach, it's easy to end up with unpredictable naming.
3. If you nest properties like `COMMCAREHQ.myApp.myJsModule`, you need boilerplate to make sure `COMMCAREHQ.myApp` isn't `undefined`. We never solved this properly and everyone just ended up avoiding it by not using the `COMMCAREHQ` namespace.
4. From the calling code, especially without using the `COMMCAREHQ` namespace, there's little to cue a reader as to where a function or module is coming from; it's just getting plucked out of thin (and global) air

This is why we are now using our own lightweight module system, described in the next sesion.

hqDefine

There are many great module systems out there, so why did we write our own? The answer's pretty simple: while it's great to start with require.js or system.js, with a code base HQ's size, getting from here to there is nearly impossible without an intermediate step.

Using the above example again, using `hqDefine`, you'd write your file like this:

```
// file commcare-hq/corehq/apps/myapp/static/myapp/js/myModule.js
hqDefine('myapp/js/myModule', function () {
    // things inside here are private
    var myPrivateGreeting = "Hello";
    // unless you put them in the return object
    var sayHi = function (name) {
        console.log(myPrivateGreeting + " from my module, " + name);
    };
    return {
        sayHi: sayHi,
        favoriteColor: "blue",
    };
});
```

and when you need it in another file

```
// some other file
function () {
    var sayHi = hqImport('myapp/js/myModule').sayHi;
    // ... use sayHi ...
}
```

If you compare it to the above example, you'll notice that the closure function itself is exactly the same. It's just being passed to `hqDefine` instead of being called directly.

`hqDefine` is an intermediate step on the way to full support for AMD modules, which in HQ is implemented using RequireJS. `hqDefine` checks whether or not it is on a page that uses AMD modules and then behaves in one of two ways: * If the page has been migrated, meaning it uses AMD modules, `hqDefine` just delegates to `define`. * If the page has not been migrated, `hqDefine` acts as a thin wrapper around the Crockford module pattern. `hqDefine` takes a function, calls it immediately, and puts it in a namespaced global; `hqImport` then looks up the module in that global.

In the first case, by handing control over to RequireJS, `hqDefine/hqImport` also act as a module *loader*. But in the second case, they work only as a module *dereferencer*, so in order to use a module, it still needs to be included as a `<script>` on your html page:

```
<script src="{% static 'myapp/js/myModule.js' %}"></script>
```

Note that in the example above, the module name matches the end of the filename, the same name used to identify the file when using the `static` tag, but without the `js` extension. This is necessary for RequireJS to work properly. For consistency, all modules, regardless of whether or not they are yet compatible with RequireJS, should be named to match their filename.

`hqDefine` and `hqImport` provide a consistent interface for both migrated and unmigrated pages, and that interface is also consistent with RequireJS, making it easy to eventually "flip the switch" and remove them altogether once all code is compatible with RequireJS.

46.1.4 RequireJS Migration Guide

This page is a guide to upgrading legacy code in HQ to use RequireJS. For information on how to work within existing code, see Managing Dependencies. Both that page and Historical Background on Module Patterns are useful background for this guide.

Background: modules and pages

The RequireJS migration deals with both **pages** (HTML) and **modules** (JavaScript). Any individual page is either migrated or not. Individual modules are also migrated or not, but a "migrated" module may be used on both RequireJS and non-RequireJS pages.

Logic in `hqModules.js` determines whether or not we're in a RequireJS environment and changes the behavior of `hqDefine` accordingly. In a RequireJS environment, `hqDefine` just passes through to RequireJS's `define`. Once all pages have been migrated, we'll be able to delete `hqModules.js` altogether and switch all of the `hqDefine` calls to `define`.

These docs walk through the process of migrating a single page to RequireJS.

Basic Migration Process

Prerequisites: Before a page can be migrated, **all** of its dependencies must already be in external JavaScript files and must be using `hqDefine`. This is already true for the vast majority of code in HQ. Pages that are not descendants of hqwebapp/base.html, which are rare, cannot yet be migrated.

Once these conditions are met, migrating to RequireJS is essentially the process of explicitly adding each module's dependencies to the module's definition, and also updating each HTML page to reference a single "main" module rather than including a bunch of `<script>` tags: 1. Add `requirejs_main` tag and remove `<script>` tags 1. Add dependencies 1. Test

Sample PRs: - RequireJS migration: dashboard is an example of an easy migration, where all dependencies are already migrated - RequireJS proof of concept migrates a few pages (lookup tables, data dictionary) and many of our commonly-used modules (analytics, `hq.helpers.js`, etc.). This also contains the changes to `hqModules.js` that make `hqDefine` support both migrated and unmigrated pages.

Add `requirejs_main` tag and remove `<script>` tags

The `requirejs_main` tag is what indicates that a page should use RequireJS. The page should have one "main" module. Most of our pages are already set up like this: they might include a bunch of scripts, but there's one in particular that handles the event handlers, bindings, etc. that are specific to that page.

Considerations when choosing or creating a main module

- Most often, there's already a single script that's only included on the page you're migrating, which you can use as the main module.
- It's fine for multiple pages to use the same main module - this may make sense for closely related pages.
- Sometimes a page will have some dependencies but no page-specific logic, so you can make a main module with an empty body, as in invoice_main.js.
- Sometimes you can add a dependency or two to an existing module and then use it as your main module. This can work fine, but be cautious of adding bloat or creating dependencies between django apps. There's a loose hierarchy:
  - Major third-party libraries: jQuery, knockout, underscore
  - hqwebapp
  - analytics
  - app-specific reusable modules like accounting/js/widgets, which are also sometimes used as main modules
  - page-specific modules like accounting/js/subscriptions_main
- There's a growing convention of using the suffix `_main` for main modules - more specifically, for any module that runs logic in a document ready handler.
- HTML files that are only used as the base for other templates don't need to have a main module or a `requirejs_main` tag.

Add `{% requirejs_main 'myApp/js/myModule' %}` near the top of the template: it can go after `load` and `extends` but should appear before content blocks. Note that it's a module name, not a file name, so it doesn't include `.js`.

Remove other `<script>` tags from the file. You'll be adding these as dependencies to the main module.

Add dependencies

In your main module, add any dependent modules. Pre-RequireJS, a module definition looks like this:

```
hqDefine("app/js/utils", function() {
   var $this = $("#thing");
   hqImport("otherApp/js/utils").doSomething($thing);
   ...
});
```

The migrated module will have its dependencies passed as an array to `hqDefine`, and those dependencies will become parameters to the module's encompassing function:

```
hqDefine("app/js/utils", [
   "jquery",
   "otherApp/js/utils"
], function(
```

(continues on next page)

(continued from previous page)

```
    $,
    otherUtils
) {
    var $this = $("#thing");
    otherUtils.doSomething($thing);
    ...
});
```

To declare dependencies:

- Check if the module uses jQuery, underscore, or knockout, and if so add them (their module names are all lowercase: 'jquery', 'knockout', 'underscore').
- Search the module for `hqImport` calls. Add any imported modules do the dependency list and parameter list, and replace calls to `hqImport(...)` with the new parameter name.
- If you removed any `<script>` tags from the template and haven't yet added them to the dependency list, do that.
- **Check the template's parent template**
  - **If the parent has a `requirejs_main` module, the template you're migrating should include a dependency on that module.**
    - If the parent still has `<script>` tags, the template you're migrating should include those as dependencies. It's usually convenient to migrate the parent and any "sibling" templates at the same time so you can remove the `<script>` tags altogether. If that isn't possible, make the parent check before including script tags: `{% if requirejs_main %}<script ...></script>{% endif %}`
    - Also check the parent's parent template, etc. Stop once you get to `hqwebapp/base.html`, `hqwebapp/bootstrap3/two_column.html`, or `hqwebapp/bootstrap3/base_section.html`, which already support requirejs.
- Check the view for any hqwebapp decorators like `use_jquery_ui` which are used to include many common yet not global third-party libraries. Note that you typically should **not** remove the decorator, because these decorators often control both css and js, but you **do** need to add any js scripts controlled by the decorator to your js module.
- If the module uses any globals from third parties, add the script as a dependency and also add the global to `thirdPartyGlobals` in hqModules.js which prevents errors on pages that use your module but are not yet migrated to requirejs.

Dependencies that aren't directly referenced as modules **don't** need to be added as function parameters, but they **do** need to be in the dependency list, so just put them at the end of the list. This tends to happen for custom knockout bindings, which are referenced only in the HTML, or jQuery plugins, which are referenced via the jQuery object rather than by the module's name.

Test

It's often prohibitively time-consuming to test every JavaScript interaction on a page. However, it's always important to at least load the page to check for major errors. Beyond that, test for weak spots based on the changes you made:

- If you replaced any `hqImport` calls that were inside of event handlers or other callbacks, verify that those areas still work correctly. When a migrated module is used on an unmigrated page, its dependencies need to be available at the time the module is defined. This is a change from previous behavior, where the dependencies didn't need to be defined until `hqImport` first called them. We do not currently have a construct to require dependencies after a module is defined.

- The most likely missing dependencies are the invisible ones: knockout bindings and jquery plugins like select2. These often don't error but will look substantially different on the page if they haven't been initialized.
- If your page depends on any third-party modules that might not yet be used on any RequireJS pages, test them. Third-party modules sometimes need to be upgraded to be compatible with RequireJS.
- If your page touched any javascript modules that are used by pages that haven't yet been migrated, test at least one of those non-migrated pages.
- Check if your base template has any descendants that should also be migrated.

Troubleshooting

Troubleshooting migration issues

When debugging RequireJS issues, the first question is whether or not the page you're on has been migrated. You can find out by checking the value of `window.USE_REQUIREJS` in the browser console.

Common issues on RequireJS pages:

- JS error like `$(...).something is not a function`: this indicates there's a missing dependency. Typically "something" is either `select2` or a jQuery UI widget like `datepicker`. To fix, add the missing dependency to the module that's erroring.
- Missing functionality, but no error: this usually indicates a missing knockout binding. To fix, add the file containing the binding to the module that applies that binding, which usually means adding `hqwebapp/js/knockout_bindings.ko` to the page's main module.
- JS error like `something is not defined` where `something` is one of the parameters in the module's main function: this can indicate a circular dependency. This is rare in HQ. Track down the circular dependency and see if it makes sense to eliminate it by reorganizing code. If it doesn't, you can use hqRequire to require the necessary module at the point where it's used rather than at the top of the module using it.
- JS error like `x is not defined` where `x` is a third-party module, which is the dependency of another third party module `y` and both of them are non RequireJs modules. You may get this intermittent error when you want to use `y` in the migrated module and `x` and `y` does not support AMD. You can fix this using shim or hqRequire. Example of this could be `d3` and `nvd3`

Common issues on non-RequireJS pages:

- JS error like `something is not defined` where `something` is a third-party module: this can happen if a non-RequireJS page uses a RequireJS module which uses a third party module based on a global variable. There's some code that mimicks RequireJS in this situation, but it needs to know about all of the third party libraries. To fix, add the third party module's global to thirdPartyMap in hqModules.js.
- JS error like `something is not defined` where `something` is an HQ module: this can happen when script tags are ordered so that a module appears before one of its dependencies. This can happen to migrated modules because one of the effects of the migration is to typically import all of a module's dependencies at the time the module is defined, which in a non-RequireJS context means all of the dependencies' script tags must appear before the script tags that depend on them. Previously, dependencies were not imported until `hqImport` was called, which could be later on, possibly in an event handler or some other code that would never execute until the entire page was loaded. To fix, try reordering the script tags. If you find there's a circular dependency, use `hqRequire` as described above.

Troubleshooting the RequireJS build process

Tactics that can help track down problems with the RequireJS build process, which usually manifest as errors that happen on staging but not locally:

- To turn off minification, you can run `build_requirejs` with the `--no_optimize` option. This also makes the script run much faster.
- To stop using the CDN, comment out resource_versions.js in hqwebapp/base.html. Note that this will still fetch a few files, such as `hqModules.js` and `{bootstrap_version}/requirejs_config.js`, from the CDN. To turn off the CDN entirely, comment out all of the code that manipulates `resource_versions` in build_requirejs.
- To mimic the entire build process locally:
  - Collect static files: `manage.py collectstatic --noinput` This is necessary if you've made any changes to `{bootstrap_version}/requirejs.yml` or `{bootstrap_version}/requirejs_config.js`, since the build script pulls these files from `staticfiles`, not `corehq`.
  - Compile translation files: `manage.py compilejsi18n`
  - Run the build script: `manage.py build_requirejs --local`
    - This will **overwrite** your local versions of `{bootstrap_version}/requirejs_config.js` and `resource_versions.js`, so be cautious running it if you have uncommitted changes.
    - This will also copy the generated bundle files from `staticfiles` back into `corehq`.
    - If you don't need to test locally but just want to see the results of dependency tracing, leave off the `--local`. A list of each bundle's contents will be written to `staticfiles/build.txt`, but no files will be added to or overwritten in `corehq`.

46.1.5 Third-Party Libraries

This page discusses when to use the major, UI-framework-level, libraries we depend on, along with a few common code conventions for these libraries.

jQuery

jQuery is available throughout HQ. We use jQuery 3.

Prefix jQuery variables with a `$`:

```
var $rows = $("#myTable tr"),
    firstRow = $rows[0];
```

Underscore

Underscore is available throughout HQ for utilities.

Knockout

Knockout is also available throughout HQ and should be used for new code. We use Knockout 3.0.

Prefix knockout observables with an `o`:

```
var myModel = function (options) {
    var self = this;
    self.type = options.type;
    self.oTotal = ko.observable(0);
};
```

...so that in HTML it's apparent what you're dealing with:

```
<input data-bind="visible: type === 'large' && oTotal() > 10, value: oTotal" />

Current total: <span data-bind="text: oTotal"></div>
```

Backbone and Marionette

Backbone is used in Web Apps. It **should not** be used outside of Web Apps. Within Web Apps, we use Marionette for most UI management.

Yarn

We use yarn for package management, so new libraries should be added to package.json.

46.1.6 External Packages

This page discusses how to add new dependencies with yarn. Be cautious of adding new dependencies, which introduce an indefinite maintenance burden.

Yarn

Yarn can manage components that contain HTML, CSS, JavaScript, fonts or even image files. Yarn doesn't concatenate or minify code or do anything else - it just installs the right versions of the packages you need and their dependencies.

Yarn packages

Yarn packages can be installed from a variety of sources, including a registered yarn package (a repo that has a `package.json` file defined), a Github shorthand (`<user or org>/<repo_name>`), a Github URL, or just a plain URL that points to a javascript file.

When you install a package, it will be installed in a directory called `node_modules`. For example if you were to run `yarn add jquery`, you would find a directory `node_modules/jquery`.

Specifying packages in `package.json`

To ensure a package gets installed for a project, you must specify it in the `package.json` file. This is equivalent to the `requirements.txt` file for `pip`. Similar to `pip install` for python, for yarn, use `yarn upgrade` When specifying a yarn package you can use many techniques. Here are a few examples:

```
// Installs the jquery package at version 1.11.1 to `node_modules/jquery`
"jquery": "1.11.1"

// Because `jquery-other` does not refer to a yarn package we must specify it in the
// versioning. Yarn will install this package to `node_modules/jquery-other`.
"jquery-other": "npm:jquery#1.2.0"

// This will install jquery from a github hash
"jquery-github": "jquery/jquery#44cb97e0cfc8d3e62bef7c621bfeba6fe4f65d7c"
```

To generalize, an install declaration looks like this:

```
<name>:<package>#<version>
```

Where <package> is optional if <name> == <package>. A package can be any of these things:

Type	Example
Registered package name	jquery
Git endpoint	https://github.com/user/package.git
Git shorthand	user/repo
URL	http://example.com/script.js

There are more, but those are the important ones. Find the others here

A version can be any of these things:

Type	Example
semver	`#1.2.3`
version range	`#~1.2.3`
Git tag	`#<git tag>`
Git commit	`#<commit sha>`
Git branch	`#<branch>`

Using Yarn packages in HQ

To use these packages in HQ you need to find where the js file you are looking for. In the case of jquery, it stores the minified jquery version in `jquery/dist/jquery.min.js`:

```
<script src="{% static 'jquery/dist/jquery.min.js' %}"></script>
```

Note: The `node_modules` bit is intentionally left off the path. Django already knows to look in that folder.

46.1.7 Integration Patterns

Sometimes you want to have at your fingertips in client-side code things that live primarily live on the server. This interface between JavaScript code and the data and systems we take for granted on the server can get messy and ugly.

This section lays out some conventions for getting the data you need to your JavaScript code and points you to some frameworks we've set up for making particularly common things really easy.

JavaScript in Django Templates

The `initial_page_data` template tag and `initial_page_data.js` library are for passing generic data from python to JavaScript.

In a Django template, use `initial_page_data` to register a variable. The data can be a template variable or a constant.

```
{% initial_page_data 'renderReportTables' True %}
{% initial_page_data 'defaultRows' report_table.default_rows|default:10 %}
{% initial_page_data 'tableOptions' table_options %}
```

Your JavaScript can then include `<script src="{% static 'hqwebapp/js/initial_page_data.js' %}"></script>` and access this data using the same names as in the Django template:

```
var get = hqImport('hqwebapp/js/initial_page_data').get,
    renderReportTables = get('renderReportTables'),
    defaultRows = get('defaultRows'),
    tableOptions = get('tableOptions');
```

When your JavaScript data is a complex object, it's generally cleaner to build it in your view than to pass a lot of variables through the Django template and then build it in JavaScript. So instead of a template with

```
{% initial_page_data 'width' 50 %}
{% initial_page_data 'height' 100 %}
{% initial_page_data 'thingType' type %}
{% if type == 'a' %}
    {% initial_page_data 'aProperty' 'yes' %}
{% else %}
    {% initial_page_data 'bProperty' 'yes' %}
{% endif %}
```

that then builds an object in JavaScript, when building your view context

```
options = {
    'width': 50,
    'height': 100,
    'thingType': type,
})
if type == 'a':
    options.update({'aProperty': 'yes'})
else:
    options.update({'bProperty': 'yes'})
context.update({'options': options})
```

and then use a single `{% initial_page_data 'thingOptions' %}` in your Django template.

Note that the `initial_page_data` approach uses a global namespace (as does the inline JavaScript approach). That is a problem for another day. An error will be thrown if you accidentally register two variables with the same name with `initial_page_data`.

Initial Page Data in Tests

Since initial page data contains server-provided data, JavaScript tests relying on it may need to fake it. The `register` method allows setting initial page data in JavaScript instead of in a Django template:

```
hqDefine("my_app/js/spec/my_test", ["hqwebapp/js/initial_page_data"], function␣
↪(initialPageData) {
    initialPageData.register("apps", [{
        "_id": "my-app_id",
    }])
    ...
});
```

Partials

The initial page data pattern can get messy when working with partials: the `initial_page_data` tag generally needs to go into a base template (a descendant of hqwebapp/base.html), not the partial template, so you can end up with tags in a template - or multiple templates - not obviously related to the partial.

An alternative approach to passing server data to partials is to encode is as `data-` attributes. This can get out of hand if there's a lot of complex data to pass, but it often works well for partials that define a widget that just needs a couple of server-provided options. Report filters typically use this approach.

I18n

Just like Django lets you use `ugettext('...')` in python and `{% trans '...' %}`, you can also use `gettext('...')` in any JavaScript.

For any page extending our main template, there's nothing further you need to do to get this to work. If you're interested in how it works, any page with `<script src="{% statici18n LANGUAGE_CODE %}"></script>` in the template will have access to the global `django` module and its methods.

If `djangojs.js` is missing, you can run `./manage.py compilejsi18n` to regenerate it.

For more on Django JS I18n, check out https://docs.djangoproject.com/en/1.7/topics/i18n/translation/.

Django URLs

Just like you might use `{% url ... %}` to resolve a URL in a template

```
<a href="{% url 'all_widget_info' domain %}">Widget Info</a>
```

(or `reverse(...)` to resolve a URL in python), you can use `{% registerurl %}` to make a URL available in javascript, through the initial_page_data.reverse utility (modeled after Django's python `reverse` function).

in template

```
{% registerurl 'all_widget_info' domain %}
```

in js

```
var initial_page_data = hqImport('hqwebapp/js/initial_page_data');

$.get(initial_page_data.reverse('all_widget_info')).done(function () {...});
```

As in this example, prefer inlining the call to `initial_page_data.reverse` over assigning its return value to a variable if there's no specific motivation for doing so.

In addition, you may keep positional arguments of the url unfilled by passing the special string `'---'` to `{% registerurl %}` and passing the argument value to `initial_page_data.reverse` instead.

in template

```
{% registerurl 'more_widget_info' domain '---' %}
```

in js

```
var initial_page_data = hqImport('hqwebapp/js/initial_page_data');
var widgetId = 'xxxx';
$.get(initial_page_data.reverse('more_widget_info', widgetId)).done(function () {...});
```

`registerurl` is essentially a special case of initial page data, and it gets messy when used in partials in the same way as initial page data. Encoding a url in a DOM element, in an attribute like `data-url`, is sometimes cleaner than using the `registerurl` template tag. See partials above for more detail.

Like initial page data, `registerurl` can be used in JavaScript tests directly:

```
hqDefine("my_app/js/spec/my_test", ["hqwebapp/js/initial_page_data"], function␣
↪(initialPageData) {
    initialPageData.registerUrl("apps", [{
        "build_schema": "/a/---/data/export/build_full_schema/",
    }])
    ...
});
```

Toggles and Feature Previews

In python you generally have the ability to check at any point whether a toggle or feature preview is enabled for a particular user on a particular domain.

In JavaScript it's even easier, because the user and domain are preset for you. To check, for example, whether the `IS_DEVELOPER` toggle is enabled, use

```
COMMCAREHQ.toggleEnabled('IS_DEVELOPER')
```

and to check whether the `'ENUM_IMAGE'` feature preview is enabled, use

```
COMMCAREHQ.previewEnabled('ENUM_IMAGE')
```

and that's pretty much it.

On a page that doesn't inherit from our main templates, you'll also have to include

```
<script src="{% static 'hqwebapp/js/hqModules.js' %}"></script>
<script src="{% static 'hqwebapp/js/toggles.js' %}"></script>
<script src="{% static 'style/js/bootstrap3/main.js' %}"></script>
```

Domain Privileges

In python you generally have the ability to check at any point whether a domain has a particular privilege.

In JavaScript, all privileges for the current domain are available and easy to check. For example, you can check whether the domain has the `export_ownership` privilege by including the *privileges* JS module

```
hqDefine('your/js/module', [
    ...
    'hqwebapp/js/privileges'
], function (
    ...
    privileges
) {...};
```

and then checking for the privilege using

```
var hasPrivilege = privileges.hasPrivilege('export_ownership')
```

On a page that doesn't inherit from our main templates, you'll also have to include

```
<script src="{% static 'hqwebapp/js/privileges.js' %}"></script>
```

Remote Method Invocation

We use our own `dimagi/jquery.rmi` library to post ajax calls to methods in Django Views that have been tagged to allow remote method invocation. Each RMI request creates a Promise for handling the server response.

`dimagi/jquery.rmi` was modeled after Djangular's RMI). Since that project is now dead we have internalized the relevant parts of it as `corehq.util.jquerymi`.

The README for dimagi/jquery.rmi has excellent instructions for usage.

The `notifications` app is a good example resource to study how to use this library:

- `NotificationsServiceRMIView` is an example of the type of view that can accept RMI posts.
- `NotificationsService.ko.js` is an example of the client-side invocation and handling.
- `style/bootstrap3/base.html` has a good example for usage of `NotificationsService`.

```
<script type="text/javascript" src="{% static '/notifications/js/NotificationsService.ko.
↪js' %}"></script>
<script type="text/javascript">
    $(function () {
        $('#js-settingsmenu-notifications').startNotificationsService('{% url
↪'notifications_service' %}');
    });
</script>
```

NOTE: It is not always the case that the RMI view is a separate view from the one hosting the client-side requests and responses. More often it's the same view, but the current examples are using Angular.js as of this writing.

46.1.8 Security

JavaScript and HTML code is subject to XSS attacks if user input is not correctly sanitized.

Python

Read the Django docs on XSS

We occasionally use the `safe` filter within templates and the `mark_safe` function in views.

Read the docs on Django's html and safestring utils.

JavaScript templates

HQ uses Underscore templates templates in some areas. Default to using `<%- ... %>` syntax to interpolate values, which properly escapes.

Any value interpolated with `<%= ... %>` must be previously escaped.

JavaScript code

In Knockout, be sure to escape any value passed to an html binding.

The DOMPurify library is available to sanitize user input. DOMPurify works by stripping potentially malicious markup. It does not escape input.

46.1.9 Static Files

Static files include any `css`, `js`, and image files that are not dynamically generated during runtime. `css` is typically compiled from `less` and minified prior to server runtime. `js` files are collected, combined, and minified prior to server runtime. As of this writing we don't compile our JavaScript from a higher level scripting language. Image files generally stay as-is. The only 'dynamic' images come from file attachments in our database.

Due to their static natures, the primary objective for static files is to make as few requests to them during page load, meaning that our goal is to combine static files into one larger, minified file when possible. An additional goal is to ensure that the browser caches the static files it is served to make subsequent page loads faster.

Collectstatic

Collectstatic is a Django management command that combs through each app's `static` directory and pulls all the static files together under one location, typically a folder at the root of the project.

During deploy, `manage.py collectstatic --noinput -v 0` is executed during the `__do_collecstatic` phase. The exact static files directory is defined by `settings.STATIC_ROOT`, and the default is named `staticfiles`.

Since Django Compressor is run after `collectstatic`, this movement of `less` files poses an issue for files that reference relative imports outside of the app's `static` directory. For instance, `style`'s `variables.less` references `bootstrap/variables.less`, which is in the `node_modules` directory.

In order to fix the moved references, it is required that `manage.py fix_less_imports_collectstatic` is run after `collectstatic`.

Once you run this, it's a good idea to regenerate static file translations with `manage.py compilejsi18n`.

In short, before testing anything that intends to mimic production static files. First run:

```
manage.py collectstatic
manage.py fix_less_imports_collectstatic
manage.py compilejsi18n
```

Compression

Django Compressor is the library we use to handle compilation of `less` files and the minification of `js` and compiled `css` files.

Compressor blocks are defined inside the `{% compress css %}{% endcompress %}` or `{% compress js %}{% endcompress %}` blocks in Django templates. Each block will be processed as one unit during the different steps of compression.

Best practice is to wrap all script tags and stylesheet links in compress blocks, in order to reduce file size and number of server requests made per page.

There are three ways of utilizing Django Compressor's features:

1. Dev Setup: Server-side on the fly `less` compilation

This does not combine any files in compress blocks, and as no effect on `js` blocks. This is the default dev configuration.

How is this enabled?

Make sure your `localsettings.py` file has the following set:

```
COMPRESS_ENABLED = False
COMPRESS_OFFLINE = False
```

2. Production-like Setup: Compress Offline

Pros:

- Closest mirror to production's setup.
- Easy to flip between Option 2 and Option 3

Cons:

- If you're doing a lot of front end changes, you have to re-run `collectstatic`, `fix_less_imports_collectstatic`, and `compress` management commands and restart the server AFTER each change. This will be a pain!

NOTE: If you are debugging `OfflineCompressionError`s from production or staging, you should be compressing offline locally to figure out the issue.

How to enable?

Do everything from Option 2 for LESS compilers setup.

Have the following set in `localsettings.py`:

```
COMPRESS_ENABLED = True
COMPRESS_OFFLINE = True
```

Notice that `COMPRESS_MINT_DELAY`, `COMPRESS_MTIME_DELAY`, and `COMPRESS_REBUILD_TIMEOUT` are not set.

Map Files

`#todo`

CDN

A content delivery network or content distribution network (CDN) is a globally distributed network of proxy servers deployed in multiple data centers. The goal of a CDN is to serve content to end-users with high availability and high performance. CDNs serve a large fraction of the Internet content today, including web objects (text, graphics and scripts), downloadable objects (media files, software, documents), applications (e-commerce, portals).

CDN for HQ

CommCare HQ uses a CloudFront as CDN to deliver its staticfiles. CloudFront is configured in the Amazon Console. You can find credentials in the dimagi shared keypass under AWS Dev Account. CloudFront provides us with two URLs. A CDN URL for staging and one for production. On compilation of the static files, we prefix the static file with the CloudFront URL. For example:

```
# Path to static file
<script src="/static/js/awesome.js"/>
# This gets converted to
<script src="<some hash>.cloudfront.net/static/js/awesome.js"/>
```

When a request gets made to the cloudfront URL, amazon serves the page from the nearest edge node if it has the file cached. If it doesn't have the file, it will go to our server and fetch the file. By default the file will live on the server for 24 hours.

Cache Busting

In order to ensure that the CDN has the most up to date version, we append a version number to the end of the javascript file that is a sha of the file. This infrastructure was already in place for cache busting. This means that awesome.js will actually be rendered as `awesome.js?version=123`. The CDN recognizes this as a different static file and then goes to our nginx server to fetch the file.

This cache busting is primarily handled by the `resource_static` management command, which runs during deploy. This command hashes the contents of every static file in HQ and stores the resulting hash codes in a YAML file, `resource_versions.yml`. This file is also updated by the `build_requirejs` command during deploy, adding versions for RequireJS bundle files - these files are auto-generated by `build_requirejs`, so they don't exist yet when `resource_static` runs. The `static` template tag in `hq_shared_tags` then handles appending the version number to the script tag's `src`.

Note that this cache busting is irrelevant to files that are contained within a `compress` block. Each compressed block generated a file that contains a hash in the filename, so there's no need for the URL parameter.

46.1.10 Inheritance

We use a functional approach to inheritance, in this style:

```
var animal = function(options) {
  var self = {},
      location = 0,
      speed = 5;
  self.name = options.name;

  self.run = function(time) {
    location += time * speed;
  };

  self.getLocation = function() {
    return location;
  }

  return self;
};

var bear = animal({ name: 'Oso' });
bear.run(1);
// bear.name => "Oso"
// bear.getLocation() => 5
// bear.location => undefined

var bird = function(options) {
  var self = animal(options);

  self.fly = function(time) {
    // Flying is fast
    self.run(time);
    self.run(time);
  };

  return self;
};

var duck = bird({ name: 'Pato' });
duck.run(1);
duck.fly(1);
// duck.name => "Pato"
// duck.getLocation => 15
```

Note that: - A class-like object is defined as a function that returns an instance. - The instance is initialized to an empty object, or to an instance of the parent class if there is one. - Create a private member by adding a local variable. - Create a public member by attaching a variable to the instance that will be returned. - Class name are `lowerFirstCamelCase`, distinct from `UpperFirstCamelCase` which is used for built-in objects like `Date` that require the `new` operator.

Avoid prototypical inheritance, which does not support information hiding as well.

Avoid classical-style inheritance (the `new` operator) because it also isn't great for information hiding and because forgetting to use `new` when creating an object can lead to nasty bugs.

Our approach to inheritance is heavily influenced by Crockford's *Javascript: The Good Parts*, which is good background reading.

Moving from classical inheritance to functional

Most of our code uses functional inheritance, while some of it uses classical inheritance. We don't have an active plan to replace all classical inheritance with functional, but if you have a reason to chang a particular classical class, it can often be converted to a functional style fairly mechanically:

- In the class definition, make sure the instance is initialized to an empty object instead of `this`. There's usually a `var self = this;` line that should be switched to `var self = {};`
- Throughout the class definition, make sure the code is consistently using `self` instead of `this`
- Make sure the class definition returns `self` at the end (typically it won't return anything)
- Update class name from `UpperCamelCase` to `lowerCamelCase`
- Remove `new` operator from anywhere the class is instantiated

Sample pull request

Code that actually manipulates the prototype needs more thought.

46.1.11 Code Review

All of the general standards of code review apply equally to JavaScript. See Code Contributions and Review for general guidance on code review. This document summarizes points to keep in mind specifically when reviewing JavaScript in CommCare HQ.

Language

Make any user-facing language as clear as possible.

- Proofread it.
- Limit jargon and overly technical language (using pre-existing HQ terms is okay)
- Don't let internal names bleed into user-facing content
  - "Lookup tables" not "fixtures"
  - "Web apps" not "cloudcare"
  - "Mobile worker" not "mobile user" or "CommCare User"
  - etc.

Translations

- All user-facing text should be translated with `gettext`, which is globally available in HQ JavaScript.
- Strings that contain variables should use `_.template` as described in the translations docs.

Time Zones

- All user-facing dates and times should be displayed in a time zone that will make sense for the user. Look at usage of `UserTime` and more generally at `corehq.util.timezones`.

Security

- Use `<%- ... %>` in Underscore templates to HTML escape values.
- Use `DomPurify` to HTML escape user input that will be displayed, but not in a template.

Delays and Errors

- Any potentially long-running requests, including all AJAX requests, should use a spinner or similar indicator.
  - jQuery: Use `disableButton` to disable & add a spinner, then `enableButton` when the request succeeds or fails.
  - Knockout: These usally need custom-but-usually-short disable/spinner code, probably using a boolean observable and a `disable` binding in the HTML.
  - There may not be spinner/disable code if there's an HTML form and it uses the `disable-on-submit` class.
- Any AJAX requests should have an `error` callback.
  - This usually doesn't need to be fancy, just to display a generic "Try again"-type error near the action that was taken. Most requests aren't error-prone, this is typically just to defend against generic platform errors like the user getting logged out.

Coding Standards

Again, standards in JavaScript are largely the same as in Python. However, there are a few issues that are either specific to JavaScript or more frequently arise in it:

- Naming. JavaScript is often messy because it sometimes uses server naming conventions, which are different, for server data. Push the author to leave the code better than they found it. Don't allow the same identifier to be used with different capitalizations, e.g., `firstName` and `first_name` in the same file. Find a synonym for one of them.
- JavaScript should be enclosed in modules and those modules should explicitly declare dependencies, as in the first code block here. Exceptions are app manager, reports, and web apps.
- Avoid long lists of params. Prefer kwargs-style objects and use `assert_properties` to verify they contain the expected options.
- Make sure any js access of initial page data is guaranteed not to happen until the page is fully loaded. Not doing so risks a race condition that will break the page. Keep an eye out that any new initial page data accessed in js is made available in HTML (usually not an issue unless the author didn't test at all).
- Prefer knockout to jQuery. Avoid mixing knockout and jQuery. Recall that you don't have to solve the author's problems. It's enough to say, "This is a lot of jQuery, have you considered making a knockout model?"

46.1.12 Testing

Best Practices

Writing good tests in javascript is similar to writing good tests in any other language. There are a few best practices that are more pertinent to javascript testing.

Mocking

When mocks are needed, use the `sinon.js` framework.

Setup

In order to run the javascript tests you'll need to install the required npm packages:

```
$ yarn install --frozen-lockfile
```

It's recommended to install grunt globally in order to use grunt from the command line:

```
$ npm install -g grunt
$ npm install -g grunt-cli
```

In order for the tests to run the **development server needs to be running on port 8000**.

Test Organization

HQ's JavaScript tests are organized around django apps. Test files are stored in the django app they test. Tests infrastructure is stored in its own django app, `mocha`.

Most django apps with JavaScript tests have a single set of tests. These will have an HTML template in `corehq/apps/<app_name>/templates/<app_name>/spec/mocha.html`, which inherits from the mocha app's base template. Test cases are stored in `corehq/apps/<app_name>/static/<app_name>/js/spec/<test_suite_name>_spec.js`

A few django apps have multiple test "configs" that correspond to different templates. Each config template will be in `corehq/apps/<app>/templates/<app>/spec/<config>/mocha.html` and its tests will be in `corehq/apps/<app_name>/static/<app_name>/<config>/spec/`. These are defined in `Gruntfile.js` as `<app_name>/<config_name>`, e.g., `cloudcare/form_entry`.

Dependency Management

Tests are one of the few areas of HQ's JavaScript that do not use RequireJS. Instead, dependencies are included in the relevant HTML template as script tags.

Running tests from the command line

To run the javascript tests for a particular app run:

```
$ grunt test:<app_name> // (e.g. grunt test:app_manager)
```

To list all the apps available to run:

```
$ grunt list
```

Running tests from the browser

To run a django app's tests from the browser, visit this url:

```
http://localhost:8000/mocha/<app_name>
```

To run a specific config:

```
http://localhost:8000/mocha/<app_name>/<config>  // (e.g. http://localhost:8000/mocha/
↪cloudcare/form_entry)
```

Adding a new app or config

There are three steps to adding a new app:

1. Add the django app name to the `Gruntfile.js` file.
2. Create a mocha template in `corehq/apps/<app>/templates/<app>/spec/mocha.html` to run tests. See an example on here.
3. Create tests that are included in the template in `corehq/apps/<app>/static/<app>/spec/`

To add an additional config to an existing app, specify the app in the `Gruntfile.js` like this:

```
<app_name>/<config>  // (e.g. cloudcare/form_entry)
```

The template and tests then also being in config-specific directories, as described above.

46.1.13 Linting

Our recommended linter is ESLint. This is what our Stickler configuration uses.

Running ESLint locally

The best time to find out about a code error (or stylistic faux pas) is when you type it out. For this reason, you should run a linter locally.

```
# Install it through npm:
$ npm install -g eslint

# Try it out locally if you like
$ eslint path/to/file.js
```

PyCharm

PyCharm has different ways of setting this up depending on the version.

- Instructions for 2016.1
- Instructions for 2017.3

If you get errors you may need to downgrade ESLint to version 5. This appears to be an issue on all versions of PyCharm prior to 2019.1.3.

Vim

NeoMake

Install NeoMake if you haven't already.

```
let g:neomake_javascript_enabled_makers = ['eslint']
```

Syntastic

Install syntastic if you haven't already.

```
let g:syntastic_javascript_checkers = ['eslint']
```

Configuring our lint rules

The .eslintrc.js file in the root of the commcare-hq repository defines the rules to check.

While this configuration is fairly stable, see the docs for help should you need to update it.

Looking up rules

Let's say you ran eslint on this code

```
var obj = {
    foo: 3,
    foo: 5
};
```

You'd probably get an error like: > Duplicate key 'foo'. (no-dupe-keys)

The rule then is `no-dupe-keys`. You can look it up on the rules page for a description.

Adding an exception

A foolish consistency is the hobgoblin of simple minds. Sometimes it IS okay to use `console.log`. Here are a couple ways to say "yes, this IS okay".

```
console.log('foo'); // eslint-disable-line no-console

// eslint-disable-next-line no-console
console.log('foo');
```

See the docs for more options to disable rules for on a case by case basis.

CHAPTER

FORTYSEVEN

TESTING INFRASTRUCTURE

Tests are run with nose. Unlike many projects that use nose, tests cannot normally be invoked with the `nosetests` command because it does not perform necessary Django setup. Instead, tests are invoked using the standard Django convention: `./manage.py test`.

47.1 Nose plugins

Nose plugins are used for various purposes, some of which are optional and can be enabled with command line parameters or environment variables. Others are required by the test environment and are always enabled. Custom plugins are registered with django-nose via the `NOSE_PLUGINS` setting in testsettings.

One very important always-enabled plugin applies patches before tests are run. The patches remain in effect for the duration of the test run unless utilities are provided to temporarily disable them. For example, sync_users_to_es is a decorator/context manager that enables syncing of users to ElasticSearch when a user is saved. Since this syncing involves custom test setup not done by most tests it is disabled by default, but it can be temporarily enabled using `sync_users_to_es` in tests that need it.

CHAPTER

FORTYEIGHT

TESTING BEST PRACTICES

48.1 Test set up

Doing a lot of work in the `setUp` call of a test class means that it will be run on every test. This quickly adds a lot of run time to the tests. Some things that can be easily moved to `setUpClass` are domain creation, user creation, or any other static models needed for the test.

Sometimes classes share the same base class and inherit the `setUpClass` function. Below is an example:

```
# BAD EXAMPLE

class MyBaseTestClass(TestCase):

    @classmethod
    def setUpClass(cls):
        ...


class MyTestClass(MyBaseTestClass):

    def test1(self):
        ...

class MyTestClassTwo(MyBaseTestClass):

    def test2(self):
        ...
```

In the above example the `setUpClass` is run twice, once for `MyTestClass` and once for `MyTestClassTwo`. If `setUpClass` has expensive operations, then it's best for all the tests to be combined under one test class.

```
# GOOD EXAMPLE

class MyBigTestClass(TestCase):

    @classmethod
    def setUpClass(cls):
        ...

    def test1(self):
        ...
```

(continues on next page)

(continued from previous page)

```
    def test2(self):
        ...
```

However this can lead to giant Test classes. If you find that all the tests in a package or module are sharing the same set up, you can write a setup method for the entire package or module. More information on that can be found here.

48.2 Test tear down

It is important to ensure that all objects you have created in the test database are deleted when the test class finishes running. This often happens in the `tearDown` method or the `tearDownClass` method. However, unneccessary cleanup "just to be safe" can add a large amount of time onto your tests.

48.3 Using SimpleTestCase

The SimpleTestCase runs tests without a database. Many times this can be achieved through the use of the mock library. A good rule of thumb is to have 80% of your tests be unit tests that utilize `SimpleTestCase`, and then 20% of your tests be integration tests that utilize the database and `TestCase`.

CommCare HQ also has some custom in mocking tools.

- Fake Couch - Fake implementation of CouchDBKit api for testing purposes.
- ESQueryFake - For faking ES queries.

48.4 Squashing Migrations

There is overhead to running many migrations at once. Django allows you to squash migrations which will help speed up the migrations when running tests.

CHAPTER

FORTYNINE

ANALYZING TEST COVERAGE

Test coverage is computed on travis using *coverage.py* and sent to codecov for aggregation and analysis.

This page goes over some basic ways to analyze code coverage locally.

49.1 Using coverage.py

First thing is to install the coverage.py library:

```
$ pip install coverage
```

Now you can run your tests through the coverage.py program:

```
$ coverage run manage.py test commtrack
```

This will create a binary *commcare-hq/.coverage* file (that is already ignored by our *.gitignore*) which contains all the magic bits about what happened during the test run.

You can be as specific or generic as you'd like with what selection of tests you run through this. This tool will track which lines of code in the app have been hit during execution of the tests you run. If you're only looking to analyze (and hopefully increase) coverage in a specific model or utils file, it might be helpful to cut down on how many tests you're running.

49.1.1 Make an HTML view of the data

The simplest (and probably fastest) way to view this data is to build an HTML view of the code base with the coverage data:

```
$ coverage html
```

This will build a *commcare-hq/coverage-report/* directory with a ton of HTML files in it. The important one is *commcare-hq/coverage-report/index.html*.

49.1.2 View the result in Vim

Install coveragepy.vim (https://github.com/alfredodeza/coveragepy.vim) however you personally like to install plugins. This plugin is old and out of date (but seems to be the only reasonable option) so because of this I personally think the HTML version is better.

Then run *:Coveragepy report* in Vim to build the report (this is kind of slow).

You can then use *:Coveragepy hide* and *:Coveragepy show* to add/remove the view from your current buffer.

CHAPTER

FIFTY

MOCHA TESTS

50.1 Adding a new app to test

There are three steps to adding a new app to test:

1. Add the app name to the `Gruntfile.js` file. Note: the app has to correspond to an actual Django app.
2. Create a mocha template in `corehq/apps/<app>/templates/<app>/spec/mocha.html` to run tests. See an example on here.
3. Create tests that are included in the template in `corehq/apps/<app>/static/<app>/spec/`

50.2 Creating an alternative configuration for an app

Occasionally there's a need to use a different mocha template to run tests for the same app. In order to create multiple configurations, specify the app in the *Gruntfile.js* like this: `<app>#<config>`

Now mocha will look for that template in `corehq/apps/<app>/templates/<app>/spec/<config>/mocha.html`

The url to visit that test suite is `http://localhost:8000/mocha/<app>/<config>`

WRITING TESTS BY USING ES FAKES

In order to be able to use these ES fakes. All calls to ES in the code you want to test
must go through one of the ESQuery subclasses, such as UserES or GroupES.
In testing, a **fake** is a component that provides an actual implementation of an API,
but which is incomplete or otherwise unsuitable for production.
(See http://stackoverflow.com/a/346440/240553 for the difference between fakes, mocks, and stubs.)
`ESQueryFake` and its subclasses (`UserESFake`, etc.) do just this for the `ESQuery`
classes. Whereas the real classes hand off the work to an Elasticsearch cluster,
the fakes do the filtering, sorting, and slicing in python memory, which is lightweight
and adequate for tests. Beware that this method is, of course,
inadequate for assuring that the `ESQuery` classes themselves are producing
the correct Elasticsearch queries, and also introduces the potential for bugs to go
unnoticed because of bugs in `ESQueryFake` classes themselves. But assuming correct
implementations of the fakes, it does an good job of testing the calling code,
which is usually the primary subject of a test.
The anatomy of a fake is something like this:

- For each real `ESQuery` subclass (I'll use `UserES` as the example), there is a corresponding fake (`UserESFake`). In cases where such a fake does not exist when you need it, follow the instructions below for getting started on a new fake.
- For each filter method or public method used on the `ESQuery` base class a method should exist on `ESQueryFake` that has the same behavior
- For each filter method on `UserES`, a method should exist on `UserESFake` that has the same behavior.

New fakes and methods are implemented only as actually needed by tests
(otherwise it's very difficult to be confident the implementations are correct),
so until some mythical future day in which all code that relies on ES goes through
an `ESQuery` subclass and is thoroughly tested, the fake implementations are
intentionally incomplete. As such, an important part of their design is that they alert
their caller (the person using them to write a test) if the code being tested calls a
method on the fake that is not yet implemented. Since more often than not a number of

methods will need to be added for the same test, the fakes currently are designed to have each call to an unimplemented filter result in a no-op, and will output a logging statement telling the caller how to add the missing method. This lets the caller run the test once to see a print out of every missing function, which they can then add in one go and re-run the tests. (The danger is that they will miss the logging output; however in cases where a filter method is missing, it is pretty likely that the test will fail which will prod them to look further and find the logging statement.)

51.1 How to set up your test to use ES fakes

Patch your test to use `UserESFake` (assuming you want to patch `UserES`), making sure to patch `UserES` in the *files in which it is used*, not the file in which it is declared

```
@mock.patch('corehq.apps.users.analytics.UserES', UserESFake)

@mock.patch('corehq.apps.userreports.reports.filters.choice_providers.UserES',␣
↪UserESFake)

  class MyTest(SimpleTestCase):

      def setUp(self):

...

         UserESFake.save*doc(user.*doc)

...

      def tearDown(self):

          UserESFake.reset_docs()
```

51.2 How to set up a new ES fake

Adding a new fake is very easy. See `corehq.apps.es.fake.users_fake` for a simple example.

CHAPTER

FIFTYTWO

PROFILING

52.1 Practical guide to profiling a slow view or function

This will walkthrough one way to profile slow code using the @profile decorator.

At a high level this is the process:

1. Find the function that is slow
2. Add a decorator to save a raw profile file that will collect information about function calls and timing
3. Use libraries to analyze the raw profile file and spit out more useful information
4. Inspect the output of that information and look for anomalies
5. Make a change, observe the updated load times and repeat the process as necessary

52.1.1 Finding the slow function

This is usually pretty straightforward. The easiest thing to do is typically use the top-level entry point for a view call. In this example we are investigating the performance of commtrack location download, so the relevant function would be commtrack.views.location_export.

```
@login_and_domain_required
def location_export(request, domain):
    response = HttpResponse(mimetype=Format.from_format('xlsx').mimetype)
    response['Content-Disposition'] = 'attachment; filename="locations.xlsx"'
    dump_locations(response, domain)
    return response
```

52.1.2 Getting profile output on stderr

Use the *profile* decorator to get profile output printed to stderr.

```
from dimagi.utils import profile
@login_and_domain_required
@profile
def location_export(request, domain):
    ...
```

profile may also be used as a context manager. See the docstring for more details.

52.1.3 Getting a profile dump

To get a profile dump, simply add the following decoration to the function.

```
from dimagi.utils.decorators.profile import profile_dump
@login_and_domain_required
@profile_dump('locations_download.prof')
def location_export(request, domain):
    response = HttpResponse(mimetype=Format.from_format('xlsx').mimetype)
    response['Content-Disposition'] = 'attachment; filename="locations.xlsx"'
    dump_locations(response, domain)
    return response
```

Now each time you load the page a raw dump file will be created with a timestamp of when it was run. These are created in /tmp/ by default, however you can change it by adding a value to your settings.py like so:

```
PROFILE_LOG_BASE = "/home/czue/profiling/"
```

Note that the files created are huge; this code should only be run locally.

52.1.4 Profiling in production

The same method can be used to profile functions in production. Obviously we want to be able to turn this on and off and possibly only profile a limited number of function calls.

This can be accomplished by using an environment variable to set the probability of profiling a function. Here's an example:

```
@profile_dump('locations_download.prof', probability=float(os.getenv('PROFILE_LOCATIONS_
↪EXPORT', 0))
def location_export(request, domain):
    ....
```

By default this wil not do any profiling but if the *PROFILE_LOCATIONS_EXPORT* environment variable is set to a value between 0 and 1 and the Django process is restarted then the function will get profiled. The number of profiles that are done will depend on the value of the environment variable. Values closer to 1 will get more profiling.

You can also limit the total number of profiles to be recorded using the *limit* keyword argument. You could also expose this via an environment variable or some other method to make it configurable:

```
@profile_dump('locations_download.prof', 1, limit=10)
def location_export(request, domain):
    ....
```

Warning: In a production environment the *limit* may not apply absolutely since there are likely multiple processes running in which case the limit will get applied to each one. Also, the limit will be reset if the processes are restarted.

Any profiling in production should be closely monitored to ensure that it does not adversely affect performance or fill up available disk space.

52.1.5 Creating a more useful output from the dump file

The raw profile files are not human readable, and you need to use something like cProfile to make them useful.

SnakeViz is a great option for viewing .prof files:

```
$ pip install snakeviz
$ snakeviz /path/to/profile_dump.prof
```

Alternately you can use a script that will output a readable version of the profile data to the console. You can find such a script in the commcarehq-scripts repository. You can read the source of that script to generate your own analysis, or just use it directly as follows:

```
$ ./reusable/convert_profile.py /path/to/profile_dump.prof
```

52.1.6 Reading the output of the analysis file

The analysis file is broken into two sections. The first section is an ordered breakdown of calls by the **cumulative** time spent in those functions. It also shows the number of calls and average time per call.

The second section is harder to read, and shows the callers to each function.

This analysis will focus on the first section. The second section is useful when you determine a huge amount of time is being spent in a function but it's not clear where that function is getting called.

Here is a sample start to that file:

```
loading profile stats for locations_download/commtrack-location-20140822T205905.prof
         361742 function calls (355960 primitive calls) in 8.838 seconds

   Ordered by: cumulative time, call count
   List reduced from 840 to 200 due to restriction <200>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    8.838    8.838 /home/czue/src/commcare-hq/corehq/apps/
↪locations/views.py:336(location_export)
        1    0.011    0.011    8.838    8.838 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:248(dump_locations)
      194    0.001    0.000    8.128    0.042 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:136(parent)
      190    0.002    0.000    8.121    0.043 /home/czue/src/commcare-hq/corehq/apps/
↪cachehq/mixins.py:35(get)
      190    0.003    0.000    8.021    0.042 submodules/dimagi-utils-src/dimagi/utils/
↪couch/cache/cache_core/api.py:65(cached_open_doc)
      190    0.013    0.000    7.882    0.041 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/client.py:362(open_doc)
      396    0.003    0.000    7.762    0.020 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/_socketio.py:56(readinto)
      396    7.757    0.020    7.757    0.020 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/_socketio.py:24(<lambda>)
      196    0.001    0.000    7.414    0.038 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/resource.py:40(json_body)
      196    0.011    0.000    7.402    0.038 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/wrappers.py:270(body_string)
```

(continues on next page)

(continued from previous page)

```
      590    0.019    0.000    7.356    0.012 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/reader.py:19(readinto)
      198    0.002    0.000    0.618    0.003 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/resource.py:69(request)
      196    0.001    0.000    0.616    0.003 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/resource.py:105(get)
      198    0.004    0.000    0.615    0.003 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/resource.py:164(request)
      198    0.002    0.000    0.605    0.003 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/client.py:415(request)
      198    0.003    0.000    0.596    0.003 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/client.py:293(perform)
      198    0.005    0.000    0.537    0.003 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/client.py:456(get_response)
      396    0.001    0.000    0.492    0.001 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/http.py:135(headers)
      790    0.002    0.000    0.452    0.001 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/http.py:50(_check_headers_complete)
      198    0.015    0.000    0.450    0.002 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/http.py:191(__next__)
1159/1117    0.043    0.000    0.396    0.000 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/jsonobject/base.py:559(__init__)
    13691    0.041    0.000    0.227    0.000 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/jsonobject/base.py:660(__setitem__)
      103    0.005    0.000    0.219    0.002 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:65(location_custom_properties)
      103    0.000    0.000    0.201    0.002 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:70(<genexpr>)
  333/303    0.001    0.000    0.190    0.001 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/jsonobject/base.py:615(wrap)
      289    0.002    0.000    0.185    0.001 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:31(__init__)
        6    0.000    0.000    0.176    0.029 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/client.py:1024(_fetch_if_needed)
```

See also:

Description of columns

The most important thing to look at is the cumtime (cumulative time) column. In this example we can see that the vast majority of the time (over 8 of the 8.9 total seconds) is spent in the cached_open_doc function (and likely the library calls below are called by that function). This would be the first place to start when looking at improving profile performance. The first few questions that would be useful to ask include:

- Can we optimize the function?
- Can we reduce calls to that function?
- In the case where that function is hitting a database or a disk, can the code be rewritten to load things in bulk?

In this practical example, the function is clearly meant to already be caching (based on the name alone) so it's possible that the results would be different if caching was enabled and the cache was hot. It would be good to make sure we test with those two parameters true as well. This can be done by changing your localsettings file and setting the following two variables:

```
COUCH_CACHE_DOCS = True
COUCH_CACHE_VIEWS = True
```

Reloading the page twice (the first time to prime the cache and the second time to profile with a hot cache) will then produce a vastly different output:

```
loading profile stats for locations_download/commtrack-location-20140822T211654.prof
         303361 function calls (297602 primitive calls) in 0.484 seconds

   Ordered by: cumulative time, call count
   List reduced from 741 to 200 due to restriction <200>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.484    0.484 /home/czue/src/commcare-hq/corehq/apps/
↪locations/views.py:336(location_export)
        1    0.004    0.004    0.484    0.484 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:248(dump_locations)
1159/1117    0.017    0.000    0.160    0.000 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/jsonobject/base.py:559(__init__)
        4    0.000    0.000    0.128    0.032 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:62(filter_by_type)
        4    0.000    0.000    0.128    0.032 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/client.py:986(all)
      103    0.000    0.000    0.128    0.001 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/client.py:946(iterator)
        4    0.000    0.000    0.128    0.032 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/client.py:1024(_fetch_if_needed)
        4    0.000    0.000    0.128    0.032 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/client.py:995(fetch)
        9    0.000    0.000    0.124    0.014 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/_socketio.py:56(readinto)
        9    0.124    0.014    0.124    0.014 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/_socketio.py:24(<lambda>)
        4    0.000    0.000    0.114    0.029 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/couchdbkit/resource.py:40(json_body)
        4    0.000    0.000    0.114    0.029 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/restkit/wrappers.py:270(body_string)
       13    0.000    0.000    0.114    0.009 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/http_parser/reader.py:19(readinto)
      103    0.000    0.000    0.112    0.001 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:70(<genexpr>)
    13691    0.018    0.000    0.094    0.000 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/jsonobject/base.py:660(__setitem__)
      103    0.002    0.000    0.091    0.001 /home/czue/src/commcare-hq/corehq/apps/
↪locations/util.py:65(location_custom_properties)
      194    0.000    0.000    0.078    0.000 /home/czue/src/commcare-hq/corehq/apps/
↪locations/models.py:136(parent)
      190    0.000    0.000    0.076    0.000 /home/czue/src/commcare-hq/corehq/apps/
↪cachehq/mixins.py:35(get)
      103    0.000    0.000    0.075    0.001 submodules/dimagi-utils-src/dimagi/utils/
↪couch/database.py:50(iter_docs)
        4    0.000    0.000    0.075    0.019 submodules/dimagi-utils-src/dimagi/utils/
↪couch/bulk.py:81(get_docs)
```

(continues on next page)

(continued from previous page)

```
        4    0.000    0.000    0.073    0.018 /home/czue/.virtualenvs/commcare-hq/local/
↪lib/python2.7/site-packages/requests/api.py:80(post)
```

Yikes! It looks like this is already quite fast with a hot cache! And there don't appear to be any obvious candidates for further optimization. If it is still a problem it may be an indication that we need to prime the cache better, or increase the amount of data we are testing with locally to see more interesting results.

52.1.7 Aggregating data from multiple runs

In some cases it is useful to run a function a number of times and aggregate the profile data. To do this follow the steps above to create a set of '.prof' files (one for each run of the function) then use the gather_profile_stats.py script to aggregate the data.

This will produce a file which can be analysed with the convert_profile.py script.

52.1.8 Additional references

- http://django-extensions.readthedocs.org/en/latest/runprofileserver.html

52.2 Memory profiling

Refer to these resources which provide good information on memory profiling:

- Diagnosing memory leaks
- Using heapy
- Diving into python memory
- **Memory usage graphs with ps**
  - *while true; do ps -C python -o etimes=,pid=,%mem=,vsz= >> mem.txt; sleep 1; done*
- You can also use the "resident_set_size" decorator and context manager to print the amount of memory allocated to python before and after the method you think is causing memory leaks:

```
from dimagi.utils.decorators.profile import resident_set_size

@resident_set_size()
def function_that_uses_a_lot_of_memory:
    [u'{}'.format(x) for x in range(1,100000)]

def somewhere_else():
    with resident_set_size(enter_debugger=True):
        # the enter_debugger param will enter a pdb session after your method has run so␣
↪you can do more exploration
        # do memory intensive things
```

CHAPTER

FIFTYTHREE

CACHING AND MEMOIZATION

There are two primary ways of caching in CommCare HQ - using the decorators `@quickcache` and `@memoized`. At their core, these both do the same sort of thing - they store the results of function, like this simplified version:

```
cache = {}

def get_object(obj_id):
    if obj_id not in cache:
        obj = expensive_query_for_obj(obj_id)
        cache[obj_id] = obj
    return cache[obj_id]
```

In either case, it is important to remember that the body of the function being cached is not evaluated at all when the cache is hit. This results in two primary concerns - what to cache and how to identify it. You should cache only functions which are referentially transparent, that is, "pure" functions which return the same result when called multiple times with the same set of parameters.

This document describes the use of these two utilities.

53.1 Memoized

Memoized is an in-memory cache. At its simplest, it's a replacement for the two common patterns used in this example class:

```
class MyClass(object):

    def __init__(self):
        self._all_objects = None
        self._objects_by_key = {}

    @property
    def all_objects(self):
        if self._all_objects is None:
            result = do_a_bunch_of_stuff()
            self._all_objects = result
        return self._all_objects

    def get_object_by_key(self, key):
        if key not in self._objects_by_key:
            result = do_a_bunch_of_stuff(key)
```

(continues on next page)

(continued from previous page)

```
            self._objects_by_key[key] = result
        return self._objects_by_key[key]
```

With the memoized decorator, this becomes:

```
from memoized import memoized

class MyClass(object):

    @property
    @memoized
    def all_objects(self):
        return do_a_bunch_of_stuff()

    @memoized
    def get_object_by_key(self, key):
        return do_a_bunch_of_stuff(key)
```

When decorating a class method, `@memoized` stores the results of calls to those methods on the class instance. It stores a result for every unique set of arguments passed to the decorated function. This persists as long as the class does (or until you manually invalidate), and will be garbage collected along with the instance.

You can decorate any callable with `@memoized` and the cache will persist for the life of the callable. That is, if it isn't an instance method, the cache will probably be stored in memory for the life of the process. This should be used sparingly, as it can lead to memory leaks. However, this can be useful for lazily initializing singleton objects. Rather than computing at module load time:

```
def get_classes_by_doc_type():
    # Look up all subclasses of Document
    return result

classes_by_doc_type = get_classes_by_doc_type()
```

You can memoize it, and only compute if and when it's needed. Subsequent calls will hit the cache.

```
@memoized
def get_classes_by_doc_type():
    # Look up all subclasses of Document
    return result
```

53.2 Quickcache

`@quickcache` behaves much more like a normal cache. It stores results in a caching backend (Redis, in CCHQ) for a specified timeout (5 minutes, by default). This also means they can be shared across worker machines. Quickcache also caches objects in local memory (10 seconds, by default). This is faster to access than Redis, but its not shared across machines.

Quickcache requires you to specify which arguments to "vary on", that is, which arguments uniquely identify a cache

For examples of how it's used, check out the repo. For background, check out Why we made quickcache

53.3 The Differences

Memoized returns the same actual python object that was originally returned by the function. That is, `id(obj1) == id(obj2)` and `obj1 is obj2`. Quickcache, on the other hand, saves a copy (however, if you're within the `memoized_timeout`, you'll get the original object, but don't write code which depends on it.).

Memoized is a python-only library with no other dependencies; quickcache is configured on a per-project basis to use whatever cache backend is being used, in our case django-cache backends.

Incidentally, quickcache also uses some inspection magic that makes it not work in a REPL context (i.e. from running *python* interactively or *./manage.py shell*)

53.4 Lifecycle

Memoized on instance method:
The cache lives on the instance itself, so it gets garbage collected along with the instance

Memoized on any other function/callable:
The cache lives on the callable, so if it's globally scoped and never gets garbage collected, neither does the cache

Quickcache:
Garbage collection happens based on the timeouts specified: memoize_timeout for the local cache and timeout for redis

53.5 Scope

In-memory caching (memoized or quickcache) is scoped to a single process on a single machine. Different machines or different processes on the same machine do not share these caches between them.

For this reason, memoized is usually used when you want to cache things only for duration of a request, or for globally scoped objects that need to be always available for very fast retrieval from memory.

Redis caching (quickcache only) is globally shared between processes on all machines in an environment. This is used to share a cache across multiple requests and webworkers (although quickcache also provides a short-duration, lightning quick, in-memory cache like @memoized, so you should never need to use both).

53.6 Decorating various things

Memoized is more flexible here - it can be used to decorate any callable, including a class. In practice, it's much more common and practical to limit ourselves to normal functions, class methods, and instance methods. Technically, if you do use it on a class, it has the effect of caching the result of calling the class to create an instance, so instead of creating a new instance, if you call the class twice with the same arguments, you'll get the same (*obj1 is obj2*) python object back.

Quickcache must go on a function—whether standalone or within a class—and does not work on other callables like a class or other custom callable. In practice this is not much of a limitation.

53.7 Identifying cached values

Cached functions usually have a set of parameters passed in, and will return different results for different sets of parameters.

Best practice here is to use as small a set of parameters as possible, and to use simple objects as parameters when possible (strings, booleans, integers, that sort of thing).

```
@quickcache(['domain_obj.name', 'user._id'], timeout=10)
def count_users_forms_by_device(domain_obj, user):
    return {
        XFormInstance.objects.count_forms_by_device(domain_obj.name, device.device_id)
        for device in user.devices
    }
```

The first argument to `@quickcache` is an argument called `vary_on` which is a list of the parameters used to identify each result stored in the cache. Taken together, the variables specified in vary_on should constitute all inputs that would change the value of the output. You may be thinking "Well, isn't that just all of the arguments?" Often, yes. However, also very frequently, a function depends not on the exact object being passed in, but merely on one or a few properties of that object. In the example above, we want the function to return the same result when called with the same domain name and user ID, not necessarily the same exact objects. Quickcache handles this by allowing you to pass in strings like `parameter.attribute`. Additionally, instead of a list of parameters, you may pass in a function, which will be called with the arguments of the cached function to return a cache key.

Memoized does not provide these capabilities, and instead always uses all of the arguments passed in. For this reason, you should only memoize functions with simple arguments. At a minimum, all arguments to memoized must be hashable. You'll notice that the above function doesn't actually use anything on the `domain_obj` other than name, so you could just refactor it to accept `domain` instead (this also means code calling this function won't need to fetch the domain object to pass to this function, only to discard everything except the name anyways).

You don't need to let this consideration muck up your function's interface. A common practice is to make a helper function with simple arguments, and decorate that. You can then still use the top-level function as you see fit. For example, let's pretend the above function is an instance method and you want to use memoize rather than quickcache. You could split it apart like this:

```
@memoized
def _count_users_forms_by_device(self, domain, device_id):
    return XFormInstance.objects.count_forms_by_device(domain, device_id)

def count_users_forms_by_device(self, domain_obj, user):
    return {
        self._count_users_forms_by_device(domain_obj.name, device.device_id)
        for device in user.devices
    }
```

53.8 What can be cached

Memoized:
All arguments must be hashable; notably, lists and dicts are not hashable, but tuples are.

Return values can be anything.

Quickcache:
All vary_on values must be "basic" types (all the way down, if they are collections): string types, bool, number, list/tuple (treated as interchangeable), dict, set, None. Arbitrary objects are not allowed, nor are lists/tuples/dicts/sets containing objects, etc.

Return values can be anything that's pickleable. More generally, quickcache dictates what values you can vary_on, but leaves what values you can return up to your caching backend; since we use django cache, which uses pickle, our return values have to be pickleable.

53.9 Invalidation

> "There are only two hard problems in computer science - cache invalidation and naming things" (and off-by-one errors)

Memoized doesn't allow invalidation except by blowing away the whole cache for all parameters. Use `<function>.reset_cache()`

If you are trying to clear the cache of a memoized *@property*, you will need to invalidate the cache manually with `self._<function_name>_cache.clear()`

One of quickcache's killer features is the ability to invalidate the cache for a specific function call. To invalidate the cache for `<function>(*args, **kwargs)`, use `<function>.clear(*args, **kwargs)`. Appropriately selecting your args makes this easier.

To sneakily prime the cache of a particular call with a preset value, you can use `<function>.set_cached_value(*args, **kwargs).to(value)`. This is useful when you are already holding the answer to an expensive computation in your hands and want to do the next caller the favor of not making them do it. It's also useful for when you're dealing with a backend that has delayed refresh as is the case with Elasticsearch (when configured a certain way).

53.10 Other ways of caching

Redis is sometimes accessed manually or through other wrappers for special purposes like locking. Some of those are:

RedisLockableMixIn
Provides `get_locked_obj`, useful for making sure only one instance of an object is accessible at a time.

CriticalSection
Similar to the above, but used in a `with` construct - makes sure a block of code is never run in parallel with the same identifier.

QuickCachedDocumentMixin
Intended for couch models - quickcaches the `get` method and provides automatic invalidation on save or delete.

CachedCouchDocumentMixin
Subclass of QuickCachedDocumentMixin which also caches some couch views

CHAPTER

FIFTYFOUR

PLUGINS

There are a number of plugins which sit on top of the core CommCare functionality that enable a specific set of functionality. For safety these plugins aren't available to end-users when the platform is hosted for external signups in a multi-tenant configuration, rather these plugins are enabled by system administrators.

When hosting the CommCare HQ, be aware that plugins aren't fully supported by the core committers and generally have a higher support burden. They may require directly reading the code to provide support or understand in full. A smaller percentage of CommCare's open source developer community typically has knowledge on any given plugin. If you are enabling plugins in your local environment, please make sure you have sufficient engineering expertise to be able to read direct code-level documentation. Plugins can be managed through the admin UI, available at `https://<hq.server.url>/hq/flags/`.

The CommCare Community of Practice urges all plugin maintainers to follow our best practices for documentation. Each commit should include a description of the functionality and links to relevant tickets.

Plugins allow limiting access to a set of functionality.

They are implemented as a couchdb-backed django app, designed to be *simple* and *fast* (automatically cached).

Most plugins are configured by manually adding individual users or domains in the plugins admin UI. These are defined by adding a new `StaticToggle` in this file. See `PredictablyRandomToggle` and `DynamicallyPredictablyRandomToggle` if you need a plugin to be defined for a random subset of users.

Namespaces define the type of access granted. NAMESPACE_DOMAIN allows the plugin to be enabled for individual project spaces. NAMESPACE_USER allows the plugin to be enabled for individual users, with the functionality visible to only that user but on any project space they visit.

NAMESPACE_DOMAIN is preferred for most flags, because it can be confusing for different users to experience different behavior. Domain-based flags are like a lightweight privilege that's independent of a software plan. User-based flags are more like a lightweight permission that's independent of user roles (and therefore also independent of domain).

Tags document the feature's expected audience, particularly services projects versus SaaS projects.

See descriptions below. Tags have no technical effect. When in doubt, use TAG_CUSTOM to limit your plugin's support burden.

When adding a new plugin, define it near related plugins - this file is frequently edited, so appending it to the end of the file invites merge conflicts.

To access your plugin:

- In python, StaticToggle has `enabled_for_request`, which takes care of detecting which namespace(s) to check, and `enabled`, which requires the caller to specify the namespace.
- For python views, the `required_decorator` is useful.
- For python tests, the `flag_enabled` decorator is useful.

- In HTML, there's a `toggle_enabled` template tag.
- In JavaScript, the `hqwebapp/js/toggles` modules provides as `toggleEnabled` method.

(Note: Plugins were historically called Feature Flags and Toggles)

CHAPTER

FIFTYFIVE

COMMTRACK

55.1 What happens during a CommTrack submission?

This is the life-cycle of an incoming stock report via sms.

1. SMS is received and relevant info therein is parsed out
2. The parsed sms is converted to an HQ-compatible xform submission. This includes:
   - stock info (i.e., just the data provided in the sms)
   - location to which this message applies (provided in message or associated with sending user)
   - standard HQ submission meta-data (submit time, user, etc.)

   Notably missing: anything that updates cases
3. The submission is *not* submitted yet, but rather processed further on the server. This includes:
   - looking up the product sub-cases that actually store stock/consumption values. (step (2) looked up the location ID; each supply point is a case associated with that location, and actual stock data is stored in a sub-case – one for each product – of the supply point case)
   - applying the stock actions for each product in the correct order (a stock report can include multiple actions; these must be applied in a consistent order or else unpredictable stock levels may result)
   - computing updated stock levels and consumption (using somewhat complex business and reconciliation logic)
   - dumping the result in case blocks (added to the submission) that will update the new values in HQ's database
   - **post-processing also makes some changes elsewhere in the instance, namely:**
     - also added are 'inferred' transactions (if my stock was 20, is now 10, and i had receipts of 15, my inferred consumption was 25). This is needed to compute consumption rate later. Conversely, if a deployment tracks consumption instead of receipts, receipts are inferred this way.
     - transactions are annotated with the order in which they were processed

   Note that normally CommCare generates its own case blocks in the forms it submits.
4. The updated submission is submitted to HQ like a normal form

55.2 Submitting a stock report via CommCare

CommTrack-enabled CommCare submits xforms, but those xforms **do not** go through the post-processing step in (3) above. Therefore these forms must generate their own case blocks and mimic the end result that commtrack expects. This is severely lacking as we have not replicated the full logic from the server in these xforms (unsure if that's even possible, nor do we like the prospect of maintaining the same logic in two places), nor can these forms generate the inferred transactions. As such, the capabilities of the mobile app are greatly restricted and cannot support features like computing consumption.

This must be fixed and it's really not worth even discussing much else about using a mobile app until it is.

CHAPTER

FIFTYSIX

ELASTICSEARCH

56.1 Overview

56.1.1 Indexes

We have indexes for each of the following doc types:

- Applications - `hqapps`
- Cases - `hqcases`
- Domains - `hqdomains`
- Forms - `xforms`
- Groups - `hqgroups`
- Users - `hqusers`
- SMS logs - `smslogs`
- Case Search - `case_search`

Each index has a corresponding mapping file in `corehq/pillows/mappings/`. Each mapping has a hash that reflects the current state of the mapping. This can just be a random alphanumeric string. The hash is appended to the index name so the index is called something like `xforms_1cce1f049a1b4d864c9c25dc42648a45`. Each type of index has an alias with the short name, so you should normally be querying just `xforms`, not the fully specified index+hash. All of HQ code except the index maintenance code uses aliases to read and write data to indices.

Whenever the mapping is changed, this hash should be updated. That will trigger the creation of a new index on deploy (by the `$ ./manage.py ptop_preindex` command). Once the new index is finished, the alias is *flipped* (`$ ./manage.py ptop_es_manage --flip_all_aliases`) to point to the new index, allowing for a relatively seamless transition.

56.1.2 Keeping indexes up-to-date

Pillowtop looks at the changes feed from couch and listens for any relevant new/changed docs. In order to have your changes appear in elasticsearch, pillowtop must be running:

```
$ ./manage.py run_ptop --all
```

You can also run a once-off reindex for a specific index:

```
$ ./manage.py ptop_reindexer_v2 user
```

56.1.3 Changing a mapping or adding data

If you're adding additional data to elasticsearch, you'll need modify that index's mapping file in order to be able to query on that new data.

Adding data to an index

Each pillow has a function or class that takes in the raw document dictionary and transforms it into the document that get's sent to ES. If for example, you wanted to store username in addition to user_id on cases in elastic, you'd add `username` to `corehq.pillows.mappings.case_mapping`, then modify `transform_case_for_elasticsearch` function to do the appropriate lookup. It accepts a `doc_dict` for the case doc and is expected to return a `doc_dict`, so just add the `username` to that.

Building the new index

Once you've made the change, you'll need to build a new index which uses that new mapping. Updating index name in the mapping file triggers HQ to create the new index with new mapping and reindex all data, so you'll have to update the index hash and alias at the top of the mapping file. The hash suffix to the index can just be a random alphanumeric string and is usually the date of the edit by convention. The alias should also be updated to a new one of format `xforms_<date-modified>` (the date is just by convention), so that production operations continue to use the old alias pointing to existing index. This will trigger a preindex as outlined in the *Indexes* section. In subsequent commits alias can be flipped back to what it was, for example `xforms`. Changing the alias name doesn't trigger a reindex.

Updating indexes in a production environment

Updates in a production environment should be done in two steps, so to not show incomplete data.

1. Setup a release of your branch using cchq <env> setup_limited_release:keep_days=n_days
2. In your release directory, kick off a index using `./mange.py ptop_preindex`
3. Verify that the reindex has completed successfully - This is a weak point in our current migration process - This can be done by using ES head or the ES APIs to compare document counts to the previous index. - You should also actively look for errors in the ptop_preindex command that was ran
4. Merge your PR and deploy your latest master branch.

56.1.4 How to un-bork your broken indexes

Sometimes things get in a weird state and (locally!) it's easiest to just blow away the index and start over.

1. Delete the affected index. The easiest way to do this is with elasticsearch-head. You can delete multiple affected indices with `curl -X DELETE http://localhost:9200/*`. * can be replaced with any regex to delete matched indices, similar to bash regex.
2. Run `$ ./manage.py ptop_preindex && ./manage.py ptop_es_manage --flip_all_aliases`.
3. Try again

56.1.5 Querying Elasticsearch - Best Practices

Here are the most basic things to know if you want to write readable and reasonably performant code for accessing Elasticsearch.

Use ESQuery when possible

Check out /es_query

- Prefer the cleaner `.count()`, `.values()`, `.values_list()`, etc. execution methods to the more low level `.run().hits`, `.run().total`, etc. With the latter easier to make mistakes and fall into anti-patterns and it's harder to read.
- Prefer adding filter methods to using `set_query()` unless you really know what you're doing and are willing to make your code more error prone and difficult to read.

Prefer "get" to "search"

Don't use search to fetch a doc or doc fields by doc id; use "get" instead. Searching by id can be easily an order of magnitude (10x) slower. If done in a loop, this can effectively grind the ES cluster to a halt.

Bad::

```
POST /hqcases_2016-03-04/case/_search
{
  "query": {
    "filtered": {
      "filter": {
        "and": [{"terms": {"_id": [case_id]}}, {"match_all": {}}]
      },
      "query": {"match_all":{}}
    }
  },
  "_source": ["name"],
  "size":1000000
}
```

Good::

```
GET /hqcases_2016-03-04/case/<case_id>?_source_include=name
```

Prefer scroll queries

Use a scroll query when fetching lots of records.

Prefer filter to query

Don't use `query` when you could use `filter` if you don't need rank.

Use size(0) with aggregations

Use `size(0)` when you're only doing aggregations thing—otherwise you'll get back doc bodies as well! Sometimes that's just abstractly wasteful, but often it can be a serious performance hit for the operation as well as the cluster.

The best way to do this is by using helpers like ESQuery's `.count()` that know to do this for you—your code will look better and you won't have to remember to check for that every time. (If you ever find *helpers* not doing this correctly, then it's definitely worth fixing.)

56.2 Elasticsearch App

56.2.1 Elasticsearch Index Management

CommCare HQ data in Elasticsearch is integral to core application functionality. The level that the application relies on Elasticsearch data varies from index to index. Currently, Elasticsearch contains both authoritative data (for example `@indexed_on` case property and `UnknownUser` user records) and data used for real-time application logic (the `users` index, for example).

In order to guarantee stability (or "manageability", if you will) of this core data, it is important that Elasticsearch indexes are maintained in a consistent state across all environments as a concrete design feature of CommCare HQ. This design constraint is accomplished by managing Elasticsearch index modifications (for example: creating indexes, updating index mappings, etc) exclusively through Django's migration framework. This ensures that all Elasticsearch index modifications will be part of standard CommCare HQ code deployment procedures, thereby preventing Elasticsearch index state drift between maintained CommCare HQ deployments.

One or more migrations are required any time the following Elasticsearch state configurations are changed in code:

- index names
- index aliases
- analyzers
- mappings
- tuning parameters

Elasticsearch allows changing an index's `number_of_replicas` tuning parameter on a live index. In the future, the configuration settings (i.e. "live state") of that value should be removed from the CommCare HQ codebase entirely in order to decouple it from application logic.

Creating Elasticsearch Index Migrations

Like Django Model migrations, Elasticsearch index migrations can be quite verbose. To aid in creating these migrations, there is a Django manage command that can generate migration files for Elasticsearch index operations. Since the Elasticsearch index state is not a Django model, Django's model migration framework cannot automatically determine what operations need to be included in a migration, or even when a new migration is required. This is why creating these migrations is a separate command and not integrated into the default `makemigrations` command.

To create a new Elasticsearch index migration, use the `make_elastic_migration` management command and provide details for the required migration operations via any combination of the `-c/--create`, `-u/--update` and/or `-d/--delete` command line options.

Similar to Django model migrations, this management command uses the index metadata (mappings, analysis, etc) from the existing Elasticsearch code, so it is important that this command is executed *after* making changes to index metadata. To provide an example, consider a hypothetical scenario where the following index changes are needed:

- create a new `users` index
- update the mapping on the existing `groups` index to add a new property named `pending_users`
- delete the existing index named `groups-sandbox`

After the new property has been added to the `groups` index mapping in code, the following management command would create a migration file (e.g. `corehq/apps/es/migrations/0003_groups_pending_users.py`) for the necessary operations:

```
./manage.py make_elastic_migration --name groups_pending_users -c users -u␣
↪groups:pending_users -d groups-sandbox
```

Updating Elastic Index Mappings

Prior to the `UpdateIndexMapping` migration operation implementation, Elastic mappings were always applied "in full" any time a mapping change was needed. That is: the entire mapping (from code) was applied to the existing index via the Put Mapping API. This technique had some pros and cons:

- **Pro**: the mapping update logic in code was simple because it did not have to worry about which *existing* mapping properties are persistent (persist on the index even if omitted in a PUT request payload) and which ones are volatile (effectively "unset" if omitted in a PUT request payload).
- **Con**: it requires that *all* mapping properties are explicitly set on every mapping update, making mapping updates impossible if the existing index mapping in Elasticsearch has diverged from the mapping in code.

Because CommCare HQ Elastic mappings have been able to drift between environments, it is no longer possible to update some index mappings using the historical technique. On some indexes, the live index mappings have sufficiently diverged that there is no common, "full mapping definition" that can be applied on all environments. This means that in order to push mapping changes to all environments, new mapping update logic is needed which is capable of updating individual properties on an Elastic index mapping while leaving other (existing) properties unchanged.

The `UpdateIndexMapping` migration operation adds this capability. Due to the complex behavior of the Elasticsearch "Put Mapping" API, this implementation is limited to only support changing the mapping `_meta` and `properties` items. Changing other mapping properties (e.g. `date_detection`, `dynamic`, etc) is not yet implemented. However, the current implementation does ensure that the existing values are retained (unchanged). Historically, these values are rarely changed, so this limitation does not hinder any kind of routine maintenance operations. Implementing the ability to change the other properties will be a simple task when there is a clear definition of how that functionality needs to work, for example: when a future feature/change requires changing these properties for a specific reason.

Comparing Mappings In Code Against Live Indexes

When modifying mappings for an existing index, it can be useful to compare the new mapping (as defined in code) to the live index mappings in Elasticsearch on a CommCare HQ deployment. This is possible by dumping the mappings of interest into local files and comparing them with a diff utility. The `print_elastic_mappings` Django manage command makes this process relatively easy. Minimally, this can be accomplished in as few as three steps:

1. Export the local code mapping into a new file.
2. Export the mappings from a deployed environment into a local file.
3. Compare the two files.

In practice, this might look like the following example:

```
./manage.py print_elastic_mappings sms --no-names > ./sms-in-code.py
cchq <env> django-manage print_elastic_mappings smslogs_2020-01-28:sms --no-names > ./
↪sms-live.py
diff -u ./sms-live.py ./sms-in-code.py
```

Elastic Index Tuning Configurations

CommCare HQ provides a mechanism for individual deployments (environments) to tune the performance characteristics of their Elasticsearch indexes via Django settings. This mechanism can be used by defining an `ES_SETTINGS` dictionary in `localsettings.py` (or by configuring the requisite Elasticsearch parameters in a CommCare Cloud environment). Tuning parameters can be specified in one of two ways:

1. **"default"**: configures the tuning settings for *all* indexes in the environment.
2. **index identifier**: configures the tuning settings for *a specific* index in the environment – these settings take precedence over "default" settings.

For example, if an environment wishes to explicitly configure the "case_search" index with six shards, and all others with only three, the configuration could be specified in `localsettings.py` as:

```
ES_SETTINGS = {
    "default": {"number_of_shards": 3},
    "case_search": {"number_of_shards": 6},
}
```

Configuring a tuning setting with the special value `None` will result in that configuration item being reset to the Elasticsearch cluster default (unless superseded by another setting with higher precedence). Refer to corehq/app/es/index/settings.py file for the full details regarding what items (index and tunning settings values) are configurable, as well as what default tuning settings will be used when not customized by the environment.

Important note: These Elasticsearch index tuning settings are not "live". That is: changing their values on a deployed environment will not have any immediate affect on live indexes in Elasticsearch. Instead, these values are only ever used when an index is created (for example, during a fresh CommCare HQ installation or when an existing index is reindexed into a new one). This means that making new values become "live" involves an index migration and reindex, which requires changes in the CommCare HQ codebase.

56.2.2 Adapter Design

The HQ Elastic adapter design came about due to the need for reindexing Elasticsearch indexes in a way that is transparent to parts of HQ that write to Elasticsearch (e.g. pillowtop). Reindexing is necessary for making changes to index mappings, is a prerequisite to upgrading an Elasticsearch cluster, and is also needed for changing low-level index configurations (e.g. sharding).

There is an existing procedure draft that documents the steps that were used on one occasion to reindex the `case_search` index. This procedure leveraged a custom pillow to "backfill" the cloned index (i.e. initially populated using Elasticsearch Reindex API). That procedure only works for a subset of HQ Elasticsearch indexes, and is too risky to be considered as an ongoing Elastic maintenance strategy. There are several key constraints that an HQ reindexing procedure should meet which the existing procedure does not:

- simple and robust
- performed with standard maintenance practices
- provides the ability to test and verify the integrity of a new index before it is too late to be rejected
- allows HQ Elasticsearch index state to remain decoupled from the commcare-cloud codebase
- is not disruptive – does not prohibit any other kind of standard maintenance that might come up while the operation is underway
- is "fire and forget" – does not require active polling of intermediate state in order to progress the overall operation
- is practical for third party HQ hosters to use

One way to accomplish these constraints is to implement an "index multiplexing" feature in HQ, where Elasticsearch write operations are duplicated across two indexes. This design facilitates maintaining two up-to-date versions of any index (a primary read/write index and a secondary write-only index), allowing HQ to run in a "normal" state (i.e. not a custom "maintenance" state) while providing the ability to switch back and forth (swapping primary and secondary) before fully committing to abandoning one of them. Creating a copy of an index is the unavoidable nature of a reindex operation, and multiplexing allows safe switching from one to the other without causing disruptions or outages while keeping both up-to-date.

The least disruptive way to accomplish a multiplexing design is with an adapter layer that operates between the low-level third party Elasticsearch Python client library and high-level HQ components which need to read/write data in an Elasticsearch index. HQ already has the initial framework for this layer (the `ElasticsearchInterface` class), so the adapter layer is not a new concept. The reason that the `ElasticsearchInterface` implementation cannot be modified in-place to accommodate multiplexing is because it is the wrong level of abstraction. The `ElasticsearchInterface` abstraction layer was designed as an Elasticsearch **version** abstraction. It provides a common set of functions and methods so that the high-level HQ "consumer" that uses it can interact with Elasticsearch documents without knowing which Elasticsearch *version* is on the backend. It is below "index-level" logic, and does not implement index-specific functionality needed in order for some indexes to be handled differently than others (e.g. some indexes are indexed individually while others are multiplexed). The document adapter implementation is a **document** abstraction layer. It provides a common set of functions and methods to allow high-level HQ code to perform Elasticsearch operations at the document level, allowing unique adapters to handle their document operations differently from index to index.

With a multiplexing adapter layer, reindexing an Elasticsearch index can be as few as four concise steps, none of which are time-critical in respect to each other:

1. Merge and deploy a PR that configures multiplexing on an index.
2. Execute an idempotent management command that updates the secondary index from its primary counterpart.
3. Merge and deploy a PR that disables multiplexing for the index, (now using only the new index).
4. Execute a management command to delete the old index.

Note: the above steps are not limited to a single index at a time. That is, the implementation does not prohibit configuring multiplexing and reindexing multiple indexes at once.

This reindex procedure is inherently safe because:

- At any point in the process, the rollback procedure is a simple code change (i.e. revert PR, deploy).
- The operation responsible for populating the secondary index is idempotent *and* decoupled from the index configuration, allowing it to undergo change iterations without aborting the entire process (thereby losing reindex progress).
- Instructions for third party hosters can follow the same process that Dimagi uses, which guarantees that any possible problems encountered by a third party hoster are not outside the Dimagi main track.

Design Details

Reindex Procedure Details

1. Configure multiplexing on an index by passing in `secondary` index name to `create_document_adapter`.
   - Ensure that there is a migration in place for creating the index (see *Creating Elasticsearch Index Migrations* above).
   - *(Optional)* If the reindex involves other meta-index changes (shards, mappings, etc), also update those configurations at this time.

     Note Currently the Adapter will not support reindexing on specific environments but it would be compatible to accommodate it in future. This support will be added once we get to V5 of ES.
   - Configure `create_document_adapter` to return an instance of `ElasticMultiplexAdapter` by passing in `secondary` index name.

     ```
     case_adapter = create_document_adapter(
         ElasticCase,
         "hqcases_2016-03-04",
         "case",
         secondary="hqcase_2022-10-20"
     )
     ```
   - Add a migration which performs all cluster-level operations required for the new (secondary) index. For example:
     - creates the new index
     - configures shards, replicas, etc for the index
     - sets the index mapping
   - Review, merge and deploy this change. At Django startup, the new (secondary) index will automatically and immediately begin receiving document writes. Document reads will always come from the primary index.
2. Execute a management command to sync and verify the secondary index from the primary.

   Note: This command is not yet implemented.

   This management command is idempotent and performs four operations in serial. If any of the operations complete with unexpected results, the command will abort with an error.

   1. Executes a Elastic `reindex` request with parameters to populate the secondary index from the primary, configured to not overwrite existing documents in the target (secondary) index.
   2. Polls the reindex task progress, blocking until complete.

Note: the reindex API also supports a "blocking" mode which may be advantageous due to limitations in Elasticsearch 2.4's Task API. As such, this step **2.** might be removed in favor of a blocking reindex during the 2.4 –> 5.x upgrade.

3. Performs a cleanup operation on the secondary index to remove tombstone documents.
4. Performs a verification pass to check integrity of the secondary index.

   Note: An exact verification algorithm has not been designed, and complex verification operations may be left out of the first implementation. The reason it is outlined *in* this design is to identify that verification is supported and would happen *at this point* in the process. The initial implementation will at least implement feature-equivalency with the previous process (i.e. ensure document counts are equal between the two indexes), and tentatively an "equivalency check" of document `_id`'s (tentative because checking equality while the multiplexer is running is a race condition).

Example command (not yet implemented):

```
./manage.py elastic_sync_multiplexed ElasticBook
```

3. Perform a primary/secondary "swap" operation one or more times as desired to run a "live test" on the new (secondary) index while keeping the old (primary) index up-to-date.
   - Reconfigure the adapter by swapping the "primary" and "secondary" index names.
   - Add a migration that cleans up tombstone documents on the "new primary" index prior to startup.

   Note: In theory, this step can be optional (e.g. if the sync procedure becomes sufficiently trusted in the future, or for "goldilox" indexes where rebuilding from source is feasible but advantageous to avoid, etc).
4. Disable multiplexing for the index.
   - Reconfigure the document adapter for the index by changing the "primary index name" to the value of the "secondary index name" and remove the secondary configuration (thus reverting the adapter back to a single-index adapter).
   - Add a migration that cleans up tombstone documents on the index.
   - Review, merge and deploy this change.
5. Execute a management command to delete the old index. Example:

```
./manage.py prune_elastic_index ElasticBook
```

56.2.3 Elastic Client Adapters

The `corehq.apps.es.client` module encapsulates the CommCare HQ Elasticsearch client adapters. It implements a high-level Elasticsearch client protocol necessary to accomplish all interactions with the backend Elasticsearch cluster. Client adapters are split into two usage patterns, the "Management Adapter" and "Document Adapters". Client adapters are instantiated at import time in order to perform index verification when Django starts. Downstream code needing an adapter import and use the adapter instance.

Management Adapter

There is only one management adapter, `corehq.apps.es.client.manager`. This adapter is used for performing all cluster management tasks such as creating and updating indices and their mappings, changing index settings, changing cluster settings, etc. This functionality is split into a separate class for a few reasons:

1. The management adapter is responsible for low-level Elastic operations which document adapters should never be performing because the scope of a document adapter does not extend beyond a single index.
2. Elasticsearch 5+ implements security features which limit the kinds of operations a connection can be used for. The separation in these client adapter classes is designed to fit into that model.

```
from corehq.apps.es.client import manager

manager.index_create("books")
mapping = {"properties": {
    "author": {"type": "text"},
    "title": {"type": "text"},
    "published": {"type": "date"},
}}
manager.index_put_mapping("books", "book", mapping)
manager.index_refresh("books")
manager.index_delete("books")
```

Document Adapters

Document adapter classes are defined on a per-index basis and include specific properties and functionality necessary for maintaining a single type of "model" document in a single index. Each index in Elasticsearch needs to have a cooresponding `ElasticDocumentAdapter` subclass which defines how the Python model is applied to that specific index. At the very least, a document adapter subclass must define the following:

- A `mapping` which defines the structure and properties for documents managed by the adapter.
- A `from_python()` classmethod which can convert a Python model object into the JSON-serializable format for writing into the adapter's index.

The combination of `(index_name, type)` constrains the document adapter to a specific HQ document mapping. Comparing an Elastic cluster to a Postgres database (for the sake of analogy), the Elastic **index** is analogous to a Postgres **schema** object (e.g. `public`), and the `_type` property is analogous to a Postgres **table** object. The combination of both index name *and* `_type` fully constrains the properties that make up a specific Elastic document.

Document adapters are instantiated once at runtime, via the `create_document_adapter()` function. The purpose of this function is to act as a shim, returning an `ElasticDocumentAdapter` instance *or* an `ElasticMultiplexAdapter` instance (see *Multiplexing Document Adapters* below); depending on whether or not a secondary index is defined by the `secondary` keyword argument.

A simple example of a document model and its corresponding adapter:

```
class Book:

    def __init__(self, isbn, author, title, published):
        self.isbn = isbn
        self.author = author
        self.title = title
        self.published = published
```

(continues on next page)

(continued from previous page)

```
class ElasticBook(ElasticDocumentAdapter):

    mapping = {"properties": {
        "author": {"type": "text"},
        "title": {"type": "text"},
        "published": {"type": "date"},
    }}

    @classmethod
    def from_python(cls, book):
        source = {
            "author": book.author,
            "title": book.title,
            "published": book.published,
        }
        return book.isbn, source


books_adapter = create_document_adapter(
    ElasticBook,
    index_name="books",
    type_="book",
)
```

Using this adapter in practice might look as follows:

```
# index new
new_book = Book(
    "978-1491946008",
    "Luciano Ramalho",
    "Fluent Python: Clear, Concise, and Effective Programming",
    datetime.date(2015, 2, 10),
)
books_adapter.index(new_book)
# fetch existing
classic_book = books_adapter.get("978-0345391803")
```

Multiplexing Document Adapters

The `ElasticMultiplexAdapter` is a wrapper around two `ElasticDocumentAdapter` instances: a primary and a secondary. The multiplexing adapter provides the same public methods as a standard document adapter, but it performs Elasticsearch write operations against both indexes in order to keep them in step with document changes. The multiplexing adapter provides the following functionality:

- All read operations (`exists()`, `get()`, `search()`, etc) are always performed against the *primary* adapter only. Read requests are never performed against the secondary adapter.
- The `update()` write method always results in two sequential requests against the underlying indexes:
    1. An update request against the primary adapter that simultaneously fetches the full, post-update document body.

2. An upsert update request against the secondary adapter with the document returned in the primary update response.

- All other write operations (`index()`, `delete()`, `bulk()`, etc) leverage the Elasticsearch Bulk API to perform the required operations against both indexes simultaneously in as few requests against the backend as possible (a single request in some cases).
  - The `index()` method always achieves the index into both indexes with a single request.
  - The `delete()` method attempts to perform the delete against both indexes in a single request, and will only perform a second request in order to index a tombstone on the secondary (if the primary delete succeeded and the secondary delete failed with a 404 status).
  - The `bulk()` method (the underlying method for all bulk operations) performs actions against both indexes simultaneously by chunking the actions prior to calling `elasticsearch.helpers.bulk()` (as opposed to relying on that function to perform the chunking). This allows all bulk actions to be applied against both the primary and secondary indexes in parallel, thereby keeping both indexes synchronized throughout the duration of potentially large (multi-request) bulk operations.

56.2.4 Tombstone

The concept of Tombstone in the ES mulitplexer is there to be placeholder for the docs that get deleted on the primary index prior to that document being indexed on the secondary index. It means that whenever an adapter is multiplexed and a document is deleted, then the secondary index will receive a tombstone entry for that document *if and only if* the primary index delete succeeds and the secondary index delete fails due to a not found condition (404). The python class defined to represent these tombstones is `corehq.apps.es.client.Tombstone`.

Scenario without tombstones: If a multiplexing adapter deletes a document in the secondary index (which turns out to be a no-op because the document does not exist there yet), and then that same document is copied to the secondary index by the reindexer, then it will exist indefinitely in the secondary even though it has been deleted in the primary.

Put another way:

- Reindexer: gets batch of objects from primary index to copy to secondary.
- Multiplexer: deletes a document in that batch (in both primary and secondary indexes).
- Reindexer: writes deleted (now stale) document into secondary index.
- Result: secondary index contains a document that has been deleted.

With tombstones: this will not happen because the reindexer uses a "ignore existing documents" copy mode, so it will never overwrite a tombstone with a stale (deleted) document.

Tombstones will only exist in the secondary index and will be deleted as a final step following a successful sync (reindex) operation. Since tombstones can only be created while the primary and secondary indexes are out of sync (secondary index does not yet contain all primary documents), then once the sync is complete, the multiplexer will no longer create new tombstones.

A sample tombstone document would look like

```
{
  "__is_tombstone__" : True
}
```

56.2.5 Code Documentation

HQ Elasticsearch client logic (adapters).

class corehq.apps.es.client.**BaseAdapter**

Base adapter that includes methods common to all adapters.

__init__()

info()

Return the Elasticsearch server info.

ping()

Ping the Elasticsearch service.

class corehq.apps.es.client.**BulkActionItem**(*op_type*, *doc=None*, *doc_id=None*)

A wrapper for documents to be processed via Elasticsearch's Bulk API. Collections of these objects can be passed to an ElasticDocumentAdapter's `.bulk()` method for processing.

Instances of this class are meant to be acquired via one of the factory methods rather than instantiating directly (via `__init__()`).

class OpType(*value*)

An enumeration.

__init__(*op_type*, *doc=None*, *doc_id=None*)

classmethod delete(*doc*)

Factory method for a document delete action

classmethod delete_id(*doc_id*)

Factory method for a document delete action providing only the ID

classmethod index(*doc*)

Factory method for a document index action

property is_delete

`True` if this is a delete action, otherwise `False`.

property is_index

`True` if this is an index action, otherwise `False`.

class corehq.apps.es.client.**ElasticDocumentAdapter**(*index_name*, *type_*)

Base for subclassing document-specific adapters.

Subclasses must define the following:

- `mapping`: attribute (`dict`)
- `from_python(...)`: classmethod for converting models into Elastic format

__init__(*index_name*, *type_*)

A document adapter for a single index.

Parameters

- **index_name** – the name of the index that this adapter interacts with
- **type** – the index `_type` for the mapping

bulk(*actions*, *refresh=False*, *raise_errors=True*)

Use the Elasticsearch library's `bulk()` helper function to process documents en masse.

Equivalent to the legacy `ElasticsearchInterface.bulk_ops(...)` method.

Parameters

- **actions** – iterable of `BulkActionItem` instances
- **refresh** – `bool` refresh the effected shards to make this operation visible to search
- **raise_errors** – whether or not exceptions should be raised if bulk actions fail. The default (`True`) matches that of the elasticsearch-py library's `bulk()` helper function (i.e. raise).

bulk_delete(*doc_ids*, ***bulk_kw*)

Convenience method for bulk deleting many documents by ID without the BulkActionItem boilerplate.

Parameters

- **doc_ids** – iterable of document IDs to be deleted
- **bulk_kw** – extra parameters passed verbatim to the `ElasticDocumentAdapter.bulk()` method.

bulk_index(*docs*, ***bulk_kw*)

Convenience method for bulk indexing many documents without the BulkActionItem boilerplate.

Parameters

- **docs** – iterable of (Python model) documents to be indexed
- **bulk_kw** – extra parameters passed verbatim to the `ElasticDocumentAdapter.bulk()` method.

count(*query*)

Return the number of documents matched by the `query`

Parameters
query – `dict` query body

Returns
`int`

delete(*doc_id*, *refresh=False*)

Delete an existing document from Elasticsearch

Equivalent to the legacy `ElasticsearchInterface.delete_doc(...)` method.

Parameters

- **doc_id** – `str` ID of the document to delete
- **refresh** – `bool` refresh the effected shards to make this operation visible to search

delete_tombstones()

Deletes all tombstones documents present in the index

TODO: This should be replaced by delete_by_query https://www.elastic.co/guide/en/elasticsearch/reference/5.1/docs-delete-by-query.html when on ES version >= 5

exists(*doc_id*)

Check if a document exists for the provided `doc_id`

Equivalent to the legacy `ElasticsearchInterface.doc_exists(...)` method.

Parameters
doc_id – str ID of the document to be checked

Returns
bool

export_adapter()

Get an instance of this document adapter configured for "export" queries (i.e. the low-level Elasticsearch client object is configured with longer request timeouts, etc).

from_python(*doc*)

Transform a Python model object or model dict into the json-serializable (dict) format suitable for indexing in Elasticsearch.

Parameters
doc – document (instance of a Python model) or a dict representation of that model

Returns
tuple of (doc_id, source_dict) suitable for being indexed/updated/deleted in Elasticsearch

get(*doc_id*, *source_includes=None*)

Return the document for the provided doc_id

Equivalent to the legacy ElasticsearchInterface.get_doc(...) method.

Parameters

- **doc_id** – str ID of the document to be fetched
- **source_includes** – a list of fields to extract and return. If None (the default), the entire document is returned.

Returns
dict

get_docs(*doc_ids*)

Return multiple docs for the provided doc_ids

Equivalent to the legacy ElasticsearchInterface.get_bulk_docs(...) method.

Parameters
doc_ids – iterable of document IDs (str's)

Returns
dict

index(*doc*, *refresh=False*)

Index (send) a new document in (to) Elasticsearch

Equivalent to the legacy ElasticsearchInterface.index_doc(...) method.

Parameters

- **doc** – the (Python model) document to index
- **refresh** – bool refresh the effected shards to make this operation visible to search

iter_docs(*doc_ids*, *chunk_size=100*)

Return a generator which fetches documents in chunks.

Parameters

- **doc_ids** – iterable of document IDs (str s)

- **chunk_size** – int number of documents to fetch per query

Yields
dict documents

scroll(*query*, *scroll='5m'*, *size=None*)

Perfrom a scrolling search, yielding each doc until the entire context is exhausted.

Parameters

- **query** – dict raw search query.
- **scroll** – str time value specifying how long the Elastic cluster should keep the search context alive.
- **size** – int scroll size (number of documents per "scroll" page) When set to None (the default), the default scroll size is used.

Yields
dict documents

search(*query*, ***kw*)

Perform a query (search) and return the result.

Parameters

- **query** – dict search query to execute
- **kw** – extra parameters passed directly to the underlying elasticsearch.Elasticsearch.search() method.

Returns
dict

to_json(*doc*)

Convenience method that returns the full "from python" document (including the _id key, if present) as it would be returned by an adapter search result.

This method is not used by the adapter itself, and is only present for other code which wishes to work with documents in a couch-like format.

Parameters
doc – document (instance of a Python model)

update(*doc_id*, *fields*, *return_doc=False*, *refresh=False*, *_upsert=False*, *retry_on_conflict=None*)

Update an existing document in Elasticsearch

Equivalent to the legacy ElasticsearchInterface.update_doc_fields(...) method.

Parameters

- **doc_id** – str ID of the document to update
- **fields** – dict of name/values to update on the existing Elastic doc
- **return_doc** – bool return the full updated doc. When False (the default), None is returned.
- **refresh** – bool refresh the effected shards to make this operation visible to search
- **_upsert** – bool. Only needed for multiplexing, use the *index()* method instead. Create a new document if one doesn't already exist. When False (the default), performing an update request for a missing document will raise an exception.

- **retry_on_conflict** – int number of times to retry the update if there is a conflict. Ignored if None (the default). Otherwise, the value it is passed directly to the low-level *update()* method.

Returns
dict or None

class corehq.apps.es.client.**ElasticManageAdapter**

cancel_task(*task_id*)

Cancells a running task in ES

Parameters
task_id – str ID of the task

Returns
dict of task details

Raises
TaskError or TaskMissing (subclass of TaskError)

cluster_health(*index=None*)

Return the Elasticsearch cluster health.

cluster_routing(*, *enabled*)

Enable or disable cluster routing.

Parameters
enabled – bool whether to enable or disable routing

get_aliases()

Return the cluster aliases information.

Returns
dict with format {<alias>: [<index>, ...], ...}

get_indices(*full_info=False*)

Return the cluster index information of active indices.

Parameters
full_info – bool whether to return the full index info (default False)

Returns
dict

get_node_info(*node_id*, *metric*)

Return a specific metric from the node info for an Elasticsearch node.

Parameters

- **node_id** – str ID of the node
- **metric** – str name of the metric to fetch

Returns
deserialized JSON (dict, list, str, etc)

get_task(*task_id*)

Return the details for an active task

Parameters
task_id – str ID of the task

Returns
dict of task details

Raises
TaskError or TaskMissing (subclass of TaskError)

index_close(*index*)

Close an index.

Parameters
index – str index name

index_configure_for_reindex(*index*)

Update an index with settings optimized for reindexing.

Parameters
index – str index for which to change the settings

index_configure_for_standard_ops(*index*)

Update an index with settings optimized standard HQ performance.

Parameters
index – str index for which to change the settings

index_create(*index*, *metadata=None*)

Create a new index.

Parameters

- **index** – str index name
- **metadata** – dict full index metadata (mappings, settings, etc)

index_delete(*index*)

Delete an existing index.

Parameters
index – str index name

index_exists(*index*)

Check if index refers to a valid index identifier (index name or alias).

Parameters
index – str index name or alias

Returns
bool

index_flush(*index*)

Flush an index.

Parameters
index – str index name

index_get_mapping(*index*, *type_*)

Returns the current mapping for a doc type on an index.

Parameters

- **index** – str index to fetch the mapping from
- **type** – str doc type to fetch the mapping for

Returns
mapping `dict` or `None` if index does not have a mapping

`index_get_settings`(*index*, *values=None*)

Returns the current settings for an index.

Parameters

- **index** – `str` index to fetch settings for
- **values** – Optional collection of explicit settings to provide in the return value. If `None` (the default) all settings are returned.

Returns
`dict`

Raises
`KeyError` (only if invalid `values` are provided)

`index_put_alias`(*index*, *name*)

Assign an alias to an existing index. This uses the `Elasticsearch.update_aliases()` method to perform both 'remove' and 'add' actions simultaneously, which is atomic on the server-side. This ensures that the alias is **only** assigned to one index at a time, and that (if present) an existing alias does not vanish momentarily.

See: https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-aliases.html

Parameters

- **index** – `str` name of the index to be aliased
- **name** – `str` name of the alias to assign to `index`

`index_put_mapping`(*index*, *type_*, *mapping*)

Update the mapping for a doc type on an index.

Parameters

- **index** – `str` index where the mapping should be updated
- **type** – `str` doc type to update on the index
- **mapping** – `dict` mapping for the provided doc type

`index_refresh`(*index*)

Convenience method for refreshing a single index.

`index_set_replicas`(*index*, *replicas*)

Set the number of replicas for an index.

Parameters

- **index** – `str` index for which to change the replicas
- **replicas** – `int` number of replicas

`index_validate_query`(*index*, *query*, *params={}*)

Returns True if passed *query* is valid else will return false

`indices_info`()

Retrieve meta information about all the indices in the cluster. This will also return closed indices :returns: `dict` A dict with index name in keys and index meta information.

indices_refresh(*indices*)

Refresh a list of indices.

Parameters
indices – iterable of index names or aliases

reindex(*source*, *dest*, *wait_for_completion=False*, *refresh=False*, *batch_size=1000*, *purge_ids=False*, *requests_per_second=None*)

Starts the reindex process in elastic search cluster

Parameters

- **source** – `str` name of the source index
- **dest** – `str` name of the destination index
- **wait_for_completion** – `bool` would block the request until reindex is complete
- **refresh** – `bool` refreshes index
- **requests_per_second** – `int` throttles rate at which reindex issues batches of index operations by padding each batch with a wait time.
- **batch_size** – `int` The size of the scroll batch used by the reindex process. larger batches may process more quickly but risk errors if the documents are too large. 1000 is the recommended maximum and elasticsearch default, and can be reduced if you encounter scroll timeouts.
- **purge_ids** – `bool` adds an inline script to remove the _id field from documents source. these cause errors on reindexing the doc, but the script slows down the reindex substantially, so it is only recommended to enable this if you have run into the specific error it is designed to resolve.

Returns
None if wait_for_completion is True else would return task_id of reindex task

class corehq.apps.es.client.**ElasticMultiplexAdapter**(*primary_adapter*, *secondary_adapter*)

__init__(*primary_adapter*, *secondary_adapter*)

bulk(*actions*, *refresh=False*, *raise_errors=True*)

Apply bulk actions on the primary and secondary.

Bulk actions are applied against the primary and secondary in chunks of 500 actions at a time (replicates the behavior of the the `bulk()` helper function). Chunks are applied against the primary and secondary simultaneously by chunking the original `actions` in blocks of (up to) 250 and performing a single block of (up to) 500 actions against both indexes in parallel. Tombstone documents are indexed on the secondary for any `delete` actions which succeed on the primary but fail on the secondary.

delete(*doc_id*, *refresh=False*)

Delete from both primary and secondary via the `bulk()` method in order to perform both actions in a single HTTP request (two, if a tombstone is required).

index(*doc*, *refresh=False*)

Index into both primary and secondary via the `bulk()` method in order to perform both actions in a single HTTP request.

update(*doc_id*, *fields*, *return_doc=False*, *refresh=False*, *_upsert=False*, ***kw*)

Update on the primary adapter, fetching the full doc; then upsert the secondary adapter.

class corehq.apps.es.client.**Tombstone**(*doc_id*)

Used to create Tombstone documents in the secondary index when the document from primary index is deleted.

This is required to avoid a potential race condition that might ocuur when we run reindex process along with the multiplexer

__init__(*doc_id*)

corehq.apps.es.client.**create_document_adapter**(*cls*, *index_name*, *type_*, *, *secondary=None*)

Creates and returns a document adapter instance for the parameters provided.

One thing to note here is that the behaviour of the function can be altered with django settings.

The function would return multiplexed adapter only if - ES_<app name>_INDEX_MULTIPLEXED is True - Secondary index is provided.

The indexes would be swapped only if - ES_<app_name>_INDEX_SWAPPED is set to True - secondary index is provided

If both ES_<app name>_INDEX_MULTIPLEXED and ES_<app_name>_INDEX_SWAPPED are set to True then primary index will act as secondary index and vice versa.

Parameters

- **cls** – an `ElasticDocumentAdapter` subclass
- **index_name** – the name of the index that the adapter interacts with
- **type** – the index `_type` for the adapter's mapping.
- **secondary** – the name of the secondary index in a multiplexing configuration. If an index name is provided and ES_<app name>_INDEX_MULTIPLEXED is set to True, then returned adapter will be an instance of `ElasticMultiplexAdapter`. If `None` (the default), the returned adapter will be an instance of `cls`. ES_<app name>_INDEX_MULTIPLEXED will be ignored if secondary is None.

Returns

a document adapter instance.

corehq.apps.es.client.**get_client**(*for_export=False*)

Get an elasticsearch client instance.

Parameters

for_export – (optional `bool`) specifies whether the returned client should be optimized for slow export queries.

Returns

elasticsearch.Elasticsearch instance.

56.3 Querying Elasticsearch

56.3.1 ESQuery

ESQuery is a library for building elasticsearch queries in a friendly, more readable manner.

Basic usage

There should be a file and subclass of ESQuery for each index we have.

Each method returns a new object, so you can chain calls together like SQLAlchemy. Here's an example usage:

```
q = (FormsES()
     .domain(self.domain)
     .xmlns(self.xmlns)
     .submitted(gte=self.datespan.startdate_param,
                lt=self.datespan.enddateparam)
     .source(['xmlns', 'domain', 'app_id'])
     .sort('received_on', desc=False)
     .size(self.pagination.count)
     .start(self.pagination.start)
     .terms_aggregation('babies.count', 'babies_saved'))
result = q.run()
total_docs = result.total
hits = result.hits
```

Generally useful filters and queries should be abstracted away for re-use, but you can always add your own like so:

```
q.filter({"some_arbitrary_filter": {...}})
q.set_query({"fancy_query": {...}})
```

For debugging or more helpful error messages, you can use `query.dumps()` and `query.pprint()`, both of which use `json.dumps()` and are suitable for pasting in to ES Head or Marvel or whatever

Filtering

Filters are implemented as standalone functions, so they can be composed and nested `q.OR(web_users(), mobile_users())`. Filters can be passed to the `query.filter` method: `q.filter(web_users())`

There is some syntactic sugar that lets you skip this boilerplate and just call the filter as if it were a method on the query class: `q.web_users()` In order to be available for this shorthand, filters are added to the `builtin_filters` property of the main query class. I know that's a bit confusing, but it seemed like the best way to make filters available in both contexts.

Generic filters applicable to all indices are available in `corehq.apps.es.filters`. (But most/all can also be accessed as a query method, if appropriate)

Filtering Specific Indices

There is a file for each elasticsearch index (if not, feel free to add one). This file provides filters specific to that index, as well as an appropriately-directed ESQuery subclass with references to these filters.

These index-specific query classes also have default filters to exclude things like inactive users or deleted docs. These things should nearly always be excluded, but if necessary, you can remove these with `remove_default_filters`.

Language

- es_query - the entire query, filters, query, pagination
- filters - a list of the individual filters
- query - the query, used for searching, not filtering
- field - a field on the document. User docs have a 'domain' field.
- lt/gt - less/greater than
- lte/gte - less/greater than or equal to

class corehq.apps.es.es_query.**ESQuery**(*index=None*, *for_export=False*)

This query builder only outputs the following query structure:

```
{
  "query": {
    "bool": {
      "filter": {
        "and": [
          <filters>
        ]
      },
      "query": <query>
    }
  },
  <size, sort, other params>
}
```

__init__(*index=None*, *for_export=False*)

add_query(*new_query*, *clause*)

Add a query to the current list of queries

aggregation(*aggregation*)

Add the passed-in aggregation to the query

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

dumps(*pretty=False*)

Returns the JSON query that will be sent to elasticsearch.

exclude_source()

Turn off _source retrieval. Mostly useful if you just want the doc_ids

fields(*fields*)

Restrict the fields returned from elasticsearch

Deprecated. Use *source* instead.

filter(*filter*)

Add the passed-in filter to the query. All filtering goes through this class.

property filters

Return a list of the filters used in this query, suitable if you want to reproduce a query with additional filtering.

get_ids()

Performs a minimal query to get the ids of the matching documents

For very large sets of IDs, use `scroll_ids` instead

nested_sort(*path*, *field_name*, *nested_filter*, *desc=False*, *reset_sort=True*, *sort_missing=None*)

Order results by the value of a nested field

pprint()

pretty prints the JSON query that will be sent to elasticsearch.

remove_default_filter(*default*)

Remove a specific default filter by passing in its name.

remove_default_filters()

Sensible defaults are provided. Use this if you don't want 'em

run()

Actually run the query. Returns an ESQuerySet object.

scroll()

Run the query against the scroll api. Returns an iterator yielding each document that matches the query.

scroll_ids()

Returns a generator of all matching ids

scroll_ids_to_disk_and_iter_docs()

Returns a `ScanResult` for all matched documents.

Used for iterating docs for a very large query where consuming the docs via `self.scroll()` may exceed the amount of time that the scroll context can remain open. This is achieved by:

1. Fetching the IDs for all matched documents (via `scroll_ids()`) and caching them in a temporary file on disk, then
2. fetching the documents by (chunked blocks of) IDs streamed from the temporary file.

Original design PR: https://github.com/dimagi/commcare-hq/pull/20282

Caveats: - There is no guarantee that the returned ScanResult's `count` property will match the number of yielded docs. - Documents that are present when `scroll_ids()` is called, but are deleted prior to being fetched in full will be missing from the results, and this scenario will *not* raise an exception. - If Elastic document ID values are ever reused (i.e. new documents are created with the same ID of a previously-deleted document) then this method would become unsafe because it could yield documents that were not matched by the query.

search_string_query(*search_string*, *default_fields*)

Accepts a user-defined search string

set_query(*query*)

Set the query. Most stuff we want is better done with filters, but if you actually want Levenshtein distance or prefix querying…

set_sorting_block(*sorting_block*)

To be used with *get_sorting_block*, which interprets datatables sorting

size(*size*)

Restrict number of results returned. Analagous to SQL limit, except when performing a scroll, in which case this value becomes the number of results to fetch per scroll request.

sort(*field*, *desc=False*, *reset_sort=True*)

Order the results by field.

source(*include*, *exclude=None*)

Restrict the output of _source in the queryset. This can be used to return an object in a queryset

start(*start*)

Pagination. Analagous to SQL offset.

values(**fields*)

modeled after django's QuerySet.values

class corehq.apps.es.es_query.**ESQuerySet**(*raw*, *query*)

The object returned from ESQuery.run

- ESQuerySet.raw is the raw response from elasticsearch
- ESQuerySet.query is the ESQuery object

__init__(*raw*, *query*)

property doc_ids

Return just the docs ids from the response.

property hits

Return the docs from the response.

static normalize_result(*query*, *result*)

Return the doc from an item in the query response.

property total

Return the total number of docs matching the query.

class corehq.apps.es.es_query.**HQESQuery**(*index=None*, *for_export=False*)

Query logic specific to CommCareHQ

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

exception corehq.apps.es.es_query.**InvalidQueryError**

Query parameters cannot be assembled into a valid search.

56.3.2 Available Filters

The following filters are available on any ESQuery instance - you can chain any of these on your query.

Note also that the `term` filter accepts either a list or a single element. Simple filters which match against a field are based on this filter, so those will also accept lists. That means you can do `form_query.xmlns(XMLNS1)` or `form_query.xmlns([XMLNS1, XMLNS2, ...])`.

Contributing: Additions to this file should be added to the `builtin_filters` method on either ESQuery or HQESQuery, as appropriate (is it an HQ thing?).

corehq.apps.es.filters.**AND**(**filters*)

Filter docs to match all of the filters passed in

corehq.apps.es.filters.**NOT**(*filter_*)

Exclude docs matching the filter passed in

corehq.apps.es.filters.**OR**(**filters*)

Filter docs to match any of the filters passed in

corehq.apps.es.filters.**date_range**(*field*, *gt=None*, *gte=None*, *lt=None*, *lte=None*)

Range filter that accepts date and datetime objects as arguments

corehq.apps.es.filters.**doc_id**(*doc_id*)

Filter by doc_id. Also accepts a list of doc ids

corehq.apps.es.filters.**doc_type**(*doc_type*)

Filter by doc_type. Also accepts a list

corehq.apps.es.filters.**domain**(*domain_name*)

Filter by domain.

corehq.apps.es.filters.**empty**(*field*)

Only return docs with a missing or null value for `field`

corehq.apps.es.filters.**exists**(*field*)

Only return docs which have a value for `field`

corehq.apps.es.filters.**geo_bounding_box**(*field*, *top_left*, *bottom_right*)

Only return geopoints stored in `field` that are located within the bounding box defined by `top_left` and `bottom_right`.

`top_left` and `bottom_right` accept a range of data types and formats.

More info: Geo Bounding Box Query

corehq.apps.es.filters.**geo_grid**(*field*, *geohash*)

Filters cases by the geohash grid cell in which they are located.

corehq.apps.es.filters.**geo_polygon**(*field*, *points*)

Filters `geo_point` values in `field` that fall within the polygon described by the list of `points`.

More info: Geo Polygon Query

Parameters

- **field** – A field with Elasticsearch data type `geo_point`.
- **points** – A list of points that describe a polygon. Elasticsearch supports a range of formats for list items.

Returns

A filter dict.

corehq.apps.es.filters.**geo_shape**(*field*, *shape*, *relation='intersects'*)

Filters cases by case properties indexed using the `geo_point` type.

More info: The Geoshape query reference

Parameters

- **field** – The field where geopoints are stored

- **shape** – A shape definition given in GeoJSON geometry format. More info: The GeoJSON specification (RFC 7946)
- **relation** – The relation between the shape and the case property values.

Returns
A filter definition

corehq.apps.es.filters.**missing**(*field*)

Only return docs missing a value for `field`

corehq.apps.es.filters.**nested**(*path*, *filter_*)

Query nested documents which normally can't be queried directly

corehq.apps.es.filters.**non_null**(*field*)

Only return docs with a real, non-null value for `field`

corehq.apps.es.filters.**range_filter**(*field*, *gt=None*, *gte=None*, *lt=None*, *lte=None*)

Filter `field` by a range. Pass in some sensible combination of `gt` (greater than), `gte` (greater than or equal to), `lt`, and `lte`.

corehq.apps.es.filters.**term**(*field*, *value*)

Filter docs by a field 'value' can be a singleton or a list.

56.3.3 Available Queries

Queries are used for actual searching - things like relevancy scores, Levenstein distance, and partial matches.

View the elasticsearch documentation to see what other options are available, and put 'em here if you end up using any of 'em.

corehq.apps.es.queries.**filtered**(*query*, *filter_*)

Filtered query for performing both filtering and querying at once

corehq.apps.es.queries.**geo_distance**(*field*, *geopoint*, ***kwargs*)

Filters cases to those within a certain distance of the provided geopoint

eg: geo_distance('gps_location', GeoPoint(-33.1, 151.8), kilometers=100)

corehq.apps.es.queries.**match_all**()

No-op query used because a default must be specified

corehq.apps.es.queries.**nested**(*path*, *query*, **args*, ***kwargs*)

Creates a nested query for use with nested documents

Keyword arguments such as score_mode and others can be added.

corehq.apps.es.queries.**nested_filter**(*path*, *filter_*, **args*, ***kwargs*)

Creates a nested query for use with nested documents

Keyword arguments such as score_mode and others can be added.

corehq.apps.es.queries.**search_string_query**(*search_string*, *default_fields*)

All input defaults to doing an infix search for each term. (This may later change to some kind of fuzzy matching).

This is also available via the main ESQuery class.

56.3.4 Aggregate Queries

Aggregations are a replacement for Facets

Here is an example used to calculate how many new pregnancy cases each user has opened in a certain date range.

```
res = (CaseES()
       .domain(self.domain)
       .case_type('pregnancy')
       .date_range('opened_on', gte=startdate, lte=enddate))
       .aggregation(TermsAggregation('by_user', 'opened_by')
       .size(0)

buckets = res.aggregations.by_user.buckets
buckets.user1.doc_count
```

There's a bit of magic happening here - you can access the raw json data from this aggregation via `res.aggregation('by_user')` if you'd prefer to skip it.

The `res` object has a `aggregations` property, which returns a namedtuple pointing to the wrapped aggregation results. The name provided at instantiation is used here (`by_user` in this example).

The wrapped `aggregation_result` object has a `result` property containing the aggregation data, as well as utilties for parsing that data into something more useful. For example, the `TermsAggregation` result also has a `counts_by_bucket` method that returns a `{bucket: count}` dictionary, which is normally what you want.

As of this writing, there's not much else developed, but it's pretty easy to add support for other aggregation types and more results processing

class corehq.apps.es.aggregations.**AggregationRange**(*start=None*, *end=None*, *key=None*)

Note that a range includes the "start" value and excludes the "end" value. i.e. start <= X < end

Parameters

- **start** – range start
- **end** – range end
- **key** – optional key name for the range

class corehq.apps.es.aggregations.**AggregationTerm**(*name*, *field*)

field

Alias for field number 1

name

Alias for field number 0

class corehq.apps.es.aggregations.**AvgAggregation**(*name*, *field*)

class corehq.apps.es.aggregations.**CardinalityAggregation**(*name*, *field*)

class corehq.apps.es.aggregations.**DateHistogram**(*name*, *datefield*, *interval*, *timezone=None*)

Aggregate by date range. This can answer questions like "how many forms were created each day?".

Parameters

- **name** – what do you want to call this aggregation
- **datefield** – the document's date field to look at
- **interval** – the date interval to use - from DateHistogram.Interval

- **timezone** – do bucketing using this time zone instead of UTC

__init__(*name*, *datefield*, *interval*, *timezone=None*)

class corehq.apps.es.aggregations.**ExtendedStatsAggregation**(*name*, *field*, *script=None*)

Extended stats aggregation that computes an extended stats aggregation by field

class corehq.apps.es.aggregations.**FilterAggregation**(*name*, *filter*)

Bucket aggregation that creates a single bucket for the specified filter

Parameters

- **name** – aggregation name
- **filter** – filter body

__init__(*name*, *filter*)

class corehq.apps.es.aggregations.**FiltersAggregation**(*name*, *filters=None*)

Bucket aggregation that creates a bucket for each filter specified using the filter name.

Parameters
name – aggregation name

__init__(*name*, *filters=None*)

add_filter(*name*, *filter*)

Parameters

- **name** – filter name
- **filter** – filter body

class corehq.apps.es.aggregations.**GeoBoundsAggregation**(*name*, *field*)

A metric aggregation that computes the bounding box containing all geo_point values for a field.

More info: Geo Bounds Aggregation

__init__(*name*, *field*)

class corehq.apps.es.aggregations.**GeohashGridAggregation**(*name*, *field*, *precision*)

A multi-bucket aggregation that groups `geo_point` and `geo_shape` values into buckets that represent a grid.

More info: Geohash grid aggregation

__init__(*name*, *field*, *precision*)

Initialize a GeohashGridAggregation

Parameters

- **name** – The name of this aggregation
- **field** – The case property that stores a geopoint
- **precision** – A value between 1 and 12

High precision geohashes have a long string length and represent cells that cover only a small area (similar to long-format ZIP codes like "02139-4075").

Low precision geohashes have a short string length and represent cells that each cover a large area (similar to short-format ZIP codes like "02139").

class corehq.apps.es.aggregations.**MaxAggregation**(*name*, *field*)

class corehq.apps.es.aggregations.**MinAggregation**(*name*, *field*)

Bucket aggregation that returns the minumum value of a field

Parameters

- **name** – aggregation name
- **field** – name of the field to min

class corehq.apps.es.aggregations.**MissingAggregation**(*name*, *field*)

A field data based single bucket aggregation, that creates a bucket of all documents in the current document set context that are missing a field value (effectively, missing a field or having the configured NULL value set).

Parameters

- **name** – aggregation name
- **field** – name of the field to bucket on

__init__(*name*, *field*)

class corehq.apps.es.aggregations.**NestedAggregation**(*name*, *path*)

A special single bucket aggregation that enables aggregating nested documents.

Parameters

path – Path to nested document

__init__(*name*, *path*)

class corehq.apps.es.aggregations.**NestedTermAggregationsHelper**(*base_query*, *terms*)

Helper to run nested term-based queries (equivalent to SQL group-by clauses). This is not at all related to the ES 'nested aggregation'. The final aggregation is a count of documents.

Example usage:

```
# counting all forms submitted in a domain grouped by app id and user id

NestedTermAggregationsHelper(
    base_query=FormES().domain(domain_name),
    terms=[
        AggregationTerm('app_id', 'app_id'),
        AggregationTerm('user_id', 'form.meta.userID'),
    ]
).get_data()
```

This works by bucketing docs first by one terms aggregation, then within that bucket, bucketing further by the next term, and so on. This is then flattened out to appear like a group-by-multiple.

__init__(*base_query*, *terms*)

class corehq.apps.es.aggregations.**RangeAggregation**(*name*, *field*, *ranges=None*, *keyed=True*)

Bucket aggregation that creates one bucket for each range :param name: the aggregation name :param field: the field to perform the range aggregations on :param ranges: list of AggregationRange objects :param keyed: set to True to have the results returned by key instead of as a list (see RangeResult.normalized_buckets)

__init__(*name*, *field*, *ranges=None*, *keyed=True*)

class corehq.apps.es.aggregations.**StatsAggregation**(*name*, *field*, *script=None*)

Stats aggregation that computes a stats aggregation by field

Parameters

- **name** – aggregation name
- **field** – name of the field to collect stats on
- **script** – an optional field to allow you to script the computed field

__init__(*name*, *field*, *script=None*)

class corehq.apps.es.aggregations.**SumAggregation**(*name*, *field*)

Bucket aggregation that sums a field

Parameters

- **name** – aggregation name
- **field** – name of the field to sum

__init__(*name*, *field*)

class corehq.apps.es.aggregations.**TermsAggregation**(*name*, *field*, *size=None*, *missing=None*)

Bucket aggregation that aggregates by field

Parameters

- **name** – aggregation name
- **field** – name of the field to bucket on
- **size** –
- **missing** – define how documents that are missing a value should be treated. By default, they will be ignored. If a value is supplied here it will be used where the value is missing.

__init__(*name*, *field*, *size=None*, *missing=None*)

class corehq.apps.es.aggregations.**TopHitsAggregation**(*name*, *field=None*, *is_ascending=True*, *size=1*, *include=None*)

A top_hits metric aggregator keeps track of the most relevant document being aggregated This aggregator is intended to be used as a sub aggregator, so that the top matching documents can be aggregated per bucket.

Parameters

- **name** – Aggregation name
- **field** – This is the field to sort the top hits by. If None, defaults to sorting by score.
- **is_ascending** – Whether to sort the hits in ascending or descending order.
- **size** – The number of hits to include. Defaults to 1.
- **include** – An array of fields to include in the hit. Defaults to returning the whole document.

__init__(*name*, *field=None*, *is_ascending=True*, *size=1*, *include=None*)

class corehq.apps.es.aggregations.**ValueCountAggregation**(*name*, *field*)

56.3.5 AppES

class corehq.apps.es.apps.**AppES**(*index=None*, *for_export=False*)

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

index = 'apps'

class corehq.apps.es.apps.**ElasticApp**(*index_name*, *type_*)

canonical_name = 'apps'

property mapping

property model_cls

settings_key = 'hqapps'

corehq.apps.es.apps.**app_id**(*app_id*)

corehq.apps.es.apps.**build_comment**(*comment*)

corehq.apps.es.apps.**cloudcare_enabled**(*cloudcare_enabled*)

corehq.apps.es.apps.**created_from_template**(*from_template=True*)

corehq.apps.es.apps.**is_build**(*build=True*)

corehq.apps.es.apps.**is_released**(*released=True*)

corehq.apps.es.apps.**uses_case_sharing**(*case_sharing=True*)

corehq.apps.es.apps.**version**(*version*)

56.3.6 UserES

Here's an example adapted from the case list report - it gets a list of the ids of all unknown users, web users, and demo users on a domain.

```
from corehq.apps.es import users as user_es

user_filters = [
    user_es.unknown_users(),
    user_es.web_users(),
    user_es.demo_users(),
]

query = (user_es.UserES()
         .domain(self.domain)
         .OR(*user_filters)
         .show_inactive())

owner_ids = query.get_ids()
```

class corehq.apps.es.users.**ElasticUser**(*index_name*, *type_*)

canonical_name = 'users'

property mapping

property model_cls

settings_key = 'hqusers'

class corehq.apps.es.users.**UserES**(*index=None*, *for_export=False*)

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

default_filters = {'active': {'term': {'is_active': True}}, 'not_deleted': {'term': {'base_doc': 'couchuser'}}}

index = 'users'

show_inactive()

Include inactive users, which would normally be filtered out.

show_only_inactive()

corehq.apps.es.users.**admin_users**()

Return only AdminUsers. Admin users are mock users created from xform submissions with unknown user ids whose username is "admin".

corehq.apps.es.users.**analytics_enabled**(*enabled=True*)

corehq.apps.es.users.**created**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.users.**demo_users**()

Matches users whose username is demo_user

corehq.apps.es.users.**domain**(*domain*, *allow_enterprise=False*)

corehq.apps.es.users.**domains**(*domains*)

corehq.apps.es.users.**is_active**(*active=True*)

corehq.apps.es.users.**is_practice_user**(*practice_mode=True*)

corehq.apps.es.users.**last_logged_in**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.users.**location**(*location_id*)

corehq.apps.es.users.**missing_or_empty_user_data_property**(*property_name*)

A user_data property doesn't exist, or does exist but has an empty string value.

corehq.apps.es.users.**mobile_users**()

corehq.apps.es.users.**role_id**(*role_id*)

corehq.apps.es.users.**unknown_users**()

Return only UnknownUsers. Unknown users are mock users created from xform submissions with unknown user ids.

corehq.apps.es.users.**user_data**(*key*, *value*)

corehq.apps.es.users.**user_ids**(*user_ids*)

corehq.apps.es.users.**username**(*username*)

corehq.apps.es.users.**web_users**()

56.3.7 CaseES

Here's an example getting pregnancy cases that are either still open or were closed after May 1st.

```
from corehq.apps.es import cases as case_es

q = (case_es.CaseES()
     .domain('testproject')
     .case_type('pregnancy')
     .OR(case_es.is_closed(False),
         case_es.closed_range(gte=datetime.date(2015, 05, 01))))
```

class corehq.apps.es.cases.**CaseES**(*index=None*, *for_export=False*)

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

index = 'cases'

class corehq.apps.es.cases.**ElasticCase**(*index_name*, *type_*)

canonical_name = 'cases'

property mapping

property model_cls

settings_key = 'hqcases'

corehq.apps.es.cases.**active_in_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

Restricts cases returned to those with actions during the range

corehq.apps.es.cases.**case_ids**(*case_ids*)

corehq.apps.es.cases.**case_name**(*name*)

corehq.apps.es.cases.**case_type**(*type_*)

corehq.apps.es.cases.**closed_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**is_closed**(*closed=True*)

corehq.apps.es.cases.**modified_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**open_case_aggregation**(*name='open_case'*, *gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**opened_by**(*user_id*)

corehq.apps.es.cases.**opened_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**owner**(*owner_id*)

corehq.apps.es.cases.**owner_type**(*owner_type*)

corehq.apps.es.cases.**server_modified_range**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**touched_total_aggregation**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.cases.**user**(*user_id*)

corehq.apps.es.cases.**user_ids_handle_unknown**(*user_ids*)

56.3.8 FormES

class corehq.apps.es.forms.**ElasticForm**(*index_name*, *type_*)

canonical_name = 'forms'

property mapping

property model_cls

settings_key = 'xforms'

class corehq.apps.es.forms.**FormES**(*index=None*, *for_export=False*)

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

default_filters = {'has_domain': {'exists': {'field': 'domain'}}, 'has_user': {'exists': {'field': 'form.meta.userID'}}, 'has_xmlns': {'exists': {'field': 'xmlns'}}, 'is_xform_instance': {'term': {'doc_type': 'xforminstance'}}}

domain_aggregation()

index = 'forms'

only_archived()

Include only archived forms, which are normally excluded

user_aggregation()

corehq.apps.es.forms.**app**(*app_ids*)

corehq.apps.es.forms.**completed**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.forms.**form_ids**(*form_ids*)

corehq.apps.es.forms.**submitted**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.forms.**updating_cases**(*case_ids*)

return only those forms that have case blocks that touch the cases listed in *case_ids*

corehq.apps.es.forms.**user_id**(*user_ids*)

corehq.apps.es.forms.**user_ids_handle_unknown**(*user_ids*)

corehq.apps.es.forms.**user_type**(*user_types*)

corehq.apps.es.forms.**xmlns**(*xmlnss*)

56.3.9 DomainES

```
from corehq.apps.es import DomainES

query = (DomainES()
         .in_domains(domains)
         .created(gte=datespan.startdate, lte=datespan.enddate)
         .size(0))
```

class corehq.apps.es.domains.**DomainES**(*index=None*, *for_export=False*)

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

default_filters = {'not_snapshot': {'bool': {'must_not': {'term': {'is_snapshot': True}}}}}

index = 'domains'

only_snapshots()

Normally snapshots are excluded, instead, return only snapshots

class corehq.apps.es.domains.**ElasticDomain**(*index_name*, *type_*)

analysis = {'analyzer': {'comma': {'pattern': '\\s*,\\s*', 'type': 'pattern'}, 'default': {'filter': ['lowercase'], 'tokenizer': 'whitespace', 'type': 'custom'}}}

canonical_name = 'domains'

property mapping

property model_cls

settings_key = 'hqdomains'

corehq.apps.es.domains.**created**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.domains.**created_by_user**(*creating_user*)

corehq.apps.es.domains.**in_domains**(*domains*)

corehq.apps.es.domains.**incomplete_domains**()

corehq.apps.es.domains.**is_active**(*is_active=True*)

corehq.apps.es.domains.**is_active_project**(*is_active=True*)

corehq.apps.es.domains.**last_modified**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.domains.**non_test_domains**()

corehq.apps.es.domains.**real_domains**()

corehq.apps.es.domains.**self_started**()

56.3.10 SMSES

class corehq.apps.es.sms.**ElasticSMS**(*index_name*, *type_*)

canonical_name = 'sms'

property mapping

property model_cls

settings_key = 'smslogs'

class corehq.apps.es.sms.**SMSES**(*index=None*, *for_export=False*)

property builtin_filters

A list of callables that return filters. These will all be available as instance methods, so you can do `self.term(field, value)` instead of `self.filter(filters.term(field, value))`

index = 'sms'

user_aggregation()

corehq.apps.es.sms.**direction**(*direction_*)

corehq.apps.es.sms.**incoming_messages**()

corehq.apps.es.sms.**outgoing_messages**()

corehq.apps.es.sms.**processed**(*processed=True*)

corehq.apps.es.sms.**processed_or_incoming_messages**()

corehq.apps.es.sms.**received**(*gt=None*, *gte=None*, *lt=None*, *lte=None*)

corehq.apps.es.sms.**to_commcare_case**()

corehq.apps.es.sms.**to_commcare_user**()

corehq.apps.es.sms.**to_commcare_user_or_case**()

corehq.apps.es.sms.**to_couch_user**()

corehq.apps.es.sms.**to_web_user**()

CHAPTER

FIFTYSEVEN

MIDDLEWARE

57.1 What is middleware?

HQ uses a number of types of middleware, defined in `settings.py`. For background on middleware, see the Django docs

57.2 TimeoutMiddleware

`TimeoutMiddleware` controls a session-based inactivity timeout, where the user is logged out after a period of inactivity.

The default timeout, defined in `settings.py` as `INACTIVITY_TIMEOUT`, is two weeks, long enough that regular users will not encounter it.

Most of `TimeoutMiddleware` deals with domains that enforce a shorter timeout for security purposes.

The shorter timeout is enabled using the "Shorten Inactivity Timeout" checkbox in Project Settings > Privacy, and stored as the `Domain` attribute `secure_sessions`. This document wil refer to domains using this feature as "secure" domains. By default, secure domains time their users out after 30 minutes of inactivity. This duration is controlled by `SECURE_TIMEOUT` in `settings.py`.

"Activity" refers to web requests to HQ. This includes formplayer requests, as formplayer issues a request to HQ for session details that extends the user's session; see `SessionDetailsView`. In secure domains, there is also javascript-based logic in `hqwebapp/js/inactivity` that periodically pings HQ for the purpose of extending the session, provided that the user has recently pressed a key or clicked their mouse.

When a user's session times out, they are logged out, so their next request will redirect them to the login page. This works acceptably for most of HQ but is a bad experience when in an area that relies heavily on ajax requests, which will all start to fail without indicating to the user why. To avoid this there is a logout UI, also controlled by `hqwebapp/js/inactivity`, which tracks when the user's session is scheduled to expire. This UI pops up a warning dialog when the session is close to expiring. When the session does expire, the UI pops up a dialog that allows the user to re-login without leaving the page. This UI is enabled whenever the user is on a domain-specific page and has a secure session. Note that the user's session may be secure even if the domain they are viewing is not secure; more on this below.

A user's session is marked secure is any of the following are true:

- The user is viewing a secure domain.
- The user is a member of **any** domain that uses secure sessions.
- The session has already been marked secure by a previous request.

This behavior makes secure sessions "sticky", following the user around after they visit a secure domain. Note that the session is cleared when the user logs out or is forced out due to inactivity.

The feature flag SECURE_SESSION_TIMEOUT allows domains to customize the length of their timeout. When this is on, the domain can specify a number of minutes, which will be used in place of the default 30-minute SECURE_TIMEOUT. When a user is affected by multiple domains, with different timeout durations, the minimum duration is used. As with the secure session flag itself, the relevant durations are the current domain, and other domains where the user is a member, and the duration value currently stored in the session. So a user who belongs to two secure domains, one with the standard 30-minute timeout and one with a 15-minute timeout, will always experience a 15-minute timeout. A user who belongs to no secure domains but who visits a domain with a 45-minute timeout will then experience a 45-minute timeout until the next time they log out and back in.

CHAPTER

FIFTYEIGHT

USING THE SHARED NFS DRIVE

On our production servers (and staging) we have an NFS drive set up that we can use for a number of things:

- store files that are generated asynchronously for retrieval in a later request * previously we needed to save these files to Redis so that they would be available to all the Django workers on the next request * doing this has the added benefit of allowing apache / nginx to handle the file transfer instead of Django
- store files uploaded by the user that require asynchronous processing

58.1 Using apache / nginx to handle downloads

```
import os
import tempfile
from wsgiref.util import FileWrapper
from django.conf import settings
from django.http import StreamingHttpResponse
from django_transfer import TransferHttpResponse

transfer_enabled = settings.SHARED_DRIVE_CONF.transfer_enabled
if transfer_enabled:
    path = os.path.join(settings.SHARED_DRIVE_CONF.transfer_dir, uuid.uuid4().hex)
else:
    fd, path = tempfile.mkstemp()
    os.close(fd)

make_file(path)

if transfer_enabled:
    response = TransferHttpResponse(path, content_type=self.zip_mimetype)
else:
    response = StreamingHttpResponse(FileWrapper(open(path)), content_type=self.zip_
↪mimetype)

response['Content-Length'] = os.path.getsize(fpath)
response["Content-Disposition"] = 'attachment; filename="%s"' % filename
return response
```

This also works for files that are generated asynchronously:

```
@task
def generate_download(download_id):
```

(continues on next page)

(continued from previous page)

```
    use_transfer = settings.SHARED_DRIVE_CONF.transfer_enabled
    if use_transfer:
        path = os.path.join(settings.SHARED_DRIVE_CONF.transfer_dir, uuid.uuid4().hex)
    else:
        fd, path = tempfile.mkstemp()
        os.close(fd)

    generate_file(path)

    common_kwargs = dict(
        mimetype='application/zip',
        content_disposition='attachment; filename="{fname}"'.format(fname=filename),
        download_id=download_id,
    )
    if use_transfer:
        expose_file_download(
            path,
            use_transfer=use_transfer,
            **common_kwargs
        )
    else:
        expose_cached_download(
            FileWrapper(open(path)),
            expiry=(1 * 60 * 60),
            **common_kwargs
        )
```

58.2 Saving uploads to the NFS drive

For files that are uploaded and require asynchronous processing e.g. imports, you can also use the NFS drive:

```
from soil.util import expose_file_download, expose_cached_download

uploaded_file = request.FILES.get('Filedata')
if hasattr(uploaded_file, 'temporary_file_path') and settings.SHARED_DRIVE_CONF.temp_dir:
    path = settings.SHARED_DRIVE_CONF.get_temp_file()
    shutil.move(uploaded_file.temporary_file_path(), path)
    saved_file = expose_file_download(path, expiry=60 * 60)
else:
    uploaded_file.file.seek(0)
    saved_file = expose_cached_download(uploaded_file.file.read(), expiry=(60 * 60))

process_uploaded_file.delay(saved_file.download_id)
```

CHAPTER

FIFTYNINE

HOW TO USE AND REFERENCE FORMS AND CASES PROGRAMATICALLY

With the introduction of the new architecture for form and case data it is now necessary to use generic functions and accessors to access and operate on the models.

This document provides a basic guide for how to do that.

Models
CommCareCase
CaseTransaction
CaseAttachment
CommCareCaseIndex
XFormInstance
XFormOperation

Form Instance API

Property / method	Description
form.form_id	The instance ID of the form
form.is_normal form.is_deleted form.is_archived form.is_error form.is_deprecated form.is_duplicate form.is_submission_error_log	Replacement for checking the doc_type of a form
form.attachments	The form attachment objects
form.get_attachment	Get an attachment by name
form.archive	Archive a form
form.unarchive	Unarchive a form
form.to_json	Get the JSON representation of a form
form.form_data	Get the XML form data

Case API

Property / method	Description
case.case_id	ID of the case
case.is_deleted	Replacement for doc_type check
case.case_name	Name of the case
case.get_attachment	Get attachment by name
case.dynamic_case_properties	Dictionary of dynamic case properties
case.get_subcases	Get subcase objects
case.get_index_map	Get dictionary of case indices

59.1 Model acessors

To access models from the database there are classes that abstract the actual DB operations. These classes are generally names `<type>Accessors` and must be instantiated with a domain name in order to know which DB needs to be queried.

Forms

- XFormInstance.objects.get_form(form_id, domain)
- XFormInstance.objects.get_forms(form_ids, domain)
- XFormInstance.objects.iter_forms(form_ids, domain)
- XFormInstance.objects.save_new_form(form)
  - only for new forms
- XFormInstance.objects.get_with_attachments(form, domain)
  - Preload attachments to avoid having to the the DB again

Cases

- CommCareCase.objects.get_case(case_id, domain)
- CommCareCase.objects.get_cases(case_ids, domain)
- CommCareCase.objects.iter_cases(case_ids, domain)
- CommCareCase.objects.get_case_ids_in_domain(domain, type='dog')

Ledgers

- LedgerAccessors(domain).get_ledger_values_for_case(case_id)

For more details see:

- `corehq.form_processor.interfaces.dbaccessors.LedgerAccessors`

59.2 Unit Tests

To create a form in unit tests use the following pattern:

```
from corehq.form_processor.utils import get_simple_wrapped_form, TestFormMetadata

def test_my_form_function(self):
    # This TestFormMetadata specifies properties about the form to be created
    metadata = TestFormMetadata(
```

(continues on next page)

(continued from previous page)

```
        domain=self.user.domain,
        user_id=self.user._id,
    )
    form = get_simple_wrapped_form(
        form_id,
        metadata=metadata
    )
```

Creating cases can be done with the `CaseFactory`:

```
from casexml.apps.case.mock import CaseFactory

def test_my_case_function(self):
    factory = CaseFactory(domain='foo')
    case = factory.create_case(
        case_type='my_case_type',
        owner_id='owner1',
        case_name='bar',
        update={'prop1': 'abc'}
    )
```

59.2.1 Cleaning up

Cleaning up in tests can be done using the `FormProcessorTestUtils1` class:

```
from corehq.form_processor.tests.utils import FormProcessorTestUtils

def tearDown(self):
    FormProcessorTestUtils.delete_all_cases()
    # OR
    FormProcessorTestUtils.delete_all_cases(domain=domain)

    FormProcessorTestUtils.delete_all_xforms()
    # OR
    FormProcessorTestUtils.delete_all_xforms(domain=domain)
```

CHAPTER

SIXTY

PLAYING NICE WITH CLOUDANT/COUCHDB

We have a lot of views:

```
$ find . -path *_design*/map.js | wc -l
     159
```

Things to know about views:

1. Every time you create or update a doc, each map function is run on it and the btree for the view is updated based on the change in what the maps emit for that doc. Deleting a doc causes the btree to be updated as well.
2. Every time you update a view, all views in the design doc need to be run, from scratch, in their entirety, on every single doc in the database, regardless of doc_type.

Things to know about our Cloudant cluster:

1. It's slow. You have to wait in line just to say "hi". Want to fetch a single doc? So does everyone else. Get in line, I'll be with you in just 1000ms.
2. That's pretty much it.

Takeaways:

1. Don't save docs! If nothing changed in the doc, just don't save it. Couchdb isn't smart enough to realize that nothing changed, so saving it incurs most of the overhead of saving a doc that actually changed.
2. Don't make http requests! If you need a bunch of docs by id, get them all in one request or a few large requests using `dimagi.utils.couch.database.iter_docs`.
3. Don't make http requests! If you want to save a bunch of docs, save them all at once (after excluding the ones that haven't changed and don't need to be saved!) using `MyClass.get_db().bulk_save(docs)`. Note that this isn't good for saving thousands of documents, because it doesn't do any chunking.
4. Don't save too many docs in too short a time! To give the views time to catch up, rate-limit your saves if going through hundreds of thousands of docs. One way to do this is to save N docs and then make a tiny request to the view you think will be slowest to update, and then repeat.
5. Use different databases! All forms and cases save to the main database, but there is a *_meta* database we have just added for new doc or migrated doc types. When you use a different database you create two advantages: a) Documents you save don't contribute to the view indexing load of all of the views in the main database. b) Views you add don't have to run on all forms and cases.
6. Split views! When a single view changes, the **entire design doc** has to reindex. If you make a new view, it's much better to make a new design doc for it than to put it in with some other big, possibly expensive views. We use the *couchapps* folder/app for this.

CHAPTER

SIXTYONE

CELERY

Official Celery documentation: http://docs.celeryproject.org/en/latest/ What is it ==========

Celery is a library we use to perform tasks outside the bounds of an HTTP request.

61.1 How to use celery

All celery tasks should go into a tasks.py file or tasks module in a django app. This ensures that `autodiscover_tasks` can find the task and register it with the celery workers.

These tasks should be decorated with one of the following:

1. `@task` defines a task that is called manually (with `task_function_name.delay` in code)
2. `@periodic_task` defines a task that is called at some interval (specified by `crontab` in the decorator)
3. `@serial_task` defines a task that should only ever have one job running at one time

61.2 Best practices

Do not pass objects to celery. Instead, IDs can be passed and the celery task can retrieve the object from the database using the ID. This keeps message lengths short and reduces burden on RabbitMQ as well as preventing tasks from operating on stale data.

Do not specify `serializer='pickle'` for new tasks. This is a deprecated message serializer and by default, we now use JSON.

61.3 Queues

Table 1: Queues

Queu	I/O Bour	Target max time-to-start	Target max time-to-start comments	Description of usage	How long does the typical task take to complete?	Best practices / Notes
send_		hours	30 minutes: reports should be sent as close to schedule as possible. EDIT: this queue only affects mvp-* and ews-ghana	This is used specifically for domains who are abusing Scheduled Reports and overwhelming the background queue. See settings.THROTTL		
submission_	no?	hours	1 hour: not critical if this gets behind as long as it can keep up within a few hours	Reprocess form submissions that errored in ways that can be handled by HQ. Triggered by 'submission_reprocessin process.	seconds	
sumologic_	yes	hours	1 hour: OK for this to get behind	Forward device logs to sumologic. Triggered by device log submission from mobile.	seconds	Non-essential queue
analytics_q	yes	minutes		Used to run tasks related to external analytics tools like HubSpot. Triggered by user actions on the site.	instantaneous (seconds)	
remind		minutes		Run reminder tasks related to case changes. Triggered by case change signal.	seconds	
remind	yes	minutes	15 minutes: since these are scheduled	Runs the reminder rule tasks for re-	seconds	

61.4 Soil

Soil is a Dimagi utility to provide downloads that are backed by celery.

To use soil:

```
from soil import DownloadBase
from soil.progress import update_task_state
from soil.util import expose_cached_download

@task
def my_cool_task():
    DownloadBase.set_progress(my_cool_task, 0, 100)

    # do some stuff

    DownloadBase.set_progress(my_cool_task, 50, 100)

    # do some more stuff

    DownloadBase.set_progress(my_cool_task, 100, 100)

    expose_cached_download(payload, expiry, file_extension)
```

For error handling update the task state to failure and provide errors, HQ currently supports two options:

61.4.1 Option 1

This option raises a celery exception which tells celery to ignore future state updates. The resulting task result will not be marked as "successful" so `task.successful()` will return `False` If calling with `CELERY_TASKS_ALWAYS_EAGER = True` (i.e. a dev environment), and with `.delay()`, the exception will be caught by celery and `task.result` will return the exception.

```
from celery.exceptions import Ignore
from soil import DownloadBase
from soil.progress import update_task_state
from soil.util import expose_cached_download

@task
def my_cool_task():
    try:
        # do some stuff
    except SomeError as err:
        errors = [err]
        update_task_state(my_cool_task, states.FAILURE, {'errors': errors})
        raise Ignore()
```

61.4.2 Option 2

This option raises an exception which celery does not catch. Soil will catch this and set the error to the error message in the exception. The resulting task will be marked as a failure meaning `task.failed()` will return `True` If calling with `CELERY_TASKS_ALWAYS_EAGER = True` (i.e. a dev environment), the exception will "bubble up" to the calling code.

```
from soil import DownloadBase
from soil.progress import update_task_state
from soil.util import expose_cached_download

@task
def my_cool_task():
    # do some stuff
    raise SomeError("my uncool error")
```

61.5 Testing

As noted in the [celery docs](http://docs.celeryproject.org/en/v4.2.1/userguide/testing.html) testing tasks in celery is not the same as in production. In order to test effectively, mocking is required.

An example of mocking with Option 1 from the soil documentation:

```
@patch('my_cool_test.update_state')
def my_cool_test(update_state):
    res = my_cool_task.delay()
    self.assertIsInstance(res.result, Ignore)
    update_state.assert_called_with(
        state=states.FAILURE,
        meta={'errors': ['my uncool errors']}
    )
```

61.6 Other references

https://docs.google.com/presentation/d/1iiiVZDiOGXoLeTvEIgM_rGgw6Me5_wM_Cyc64bl7zns/edit#slide=id.g1d621cb6fc_0_372

https://docs.google.com/spreadsheets/d/10uv0YBVTGi88d6mz6xzwXRLY5OZLW1FJ0iarHI6Orck/edit?ouid=112475836275787837666&usp=sheets_home&ths=true

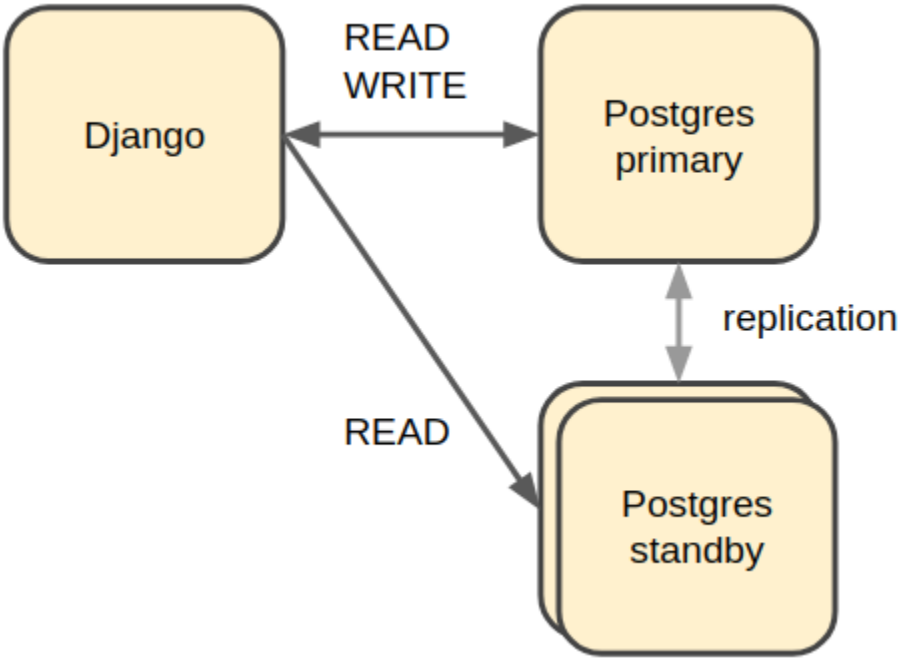
CHAPTER

SIXTYTWO

CONFIGURING SQL DATABASES IN COMMCARE

CommCare makes use of a number of logically different SQL databases. These databases can be all be a single physical database or configured as individual databases.

By default CommCare will use the *default* Django database for all SQL data.



62.1 Auditcare Data

Auditcare data may be stored in a separate database by specifying a mapping in the *LOCAL_CUSTOM_DB_ROUTING* setting. A database with the specified alias must also exist in the Django *DATABASES* setting. Example configuration:

```
LOCAL_CUSTOM_DB_ROUTING = {"auditcare": "auditcare"}
```

It is recommended to use a separate database in high-traffic environments when auditcare is enabled. Auditcare is enabled for a subset of URLs by default.

62.2 Synclog Data

Synclog data may be stored in a separate database specified by the *SYNCLOGS_SQL_DB_ALIAS* setting. The value of this setting must be a DB alias in the Django *DATABASES* setting.

62.3 UCR Data

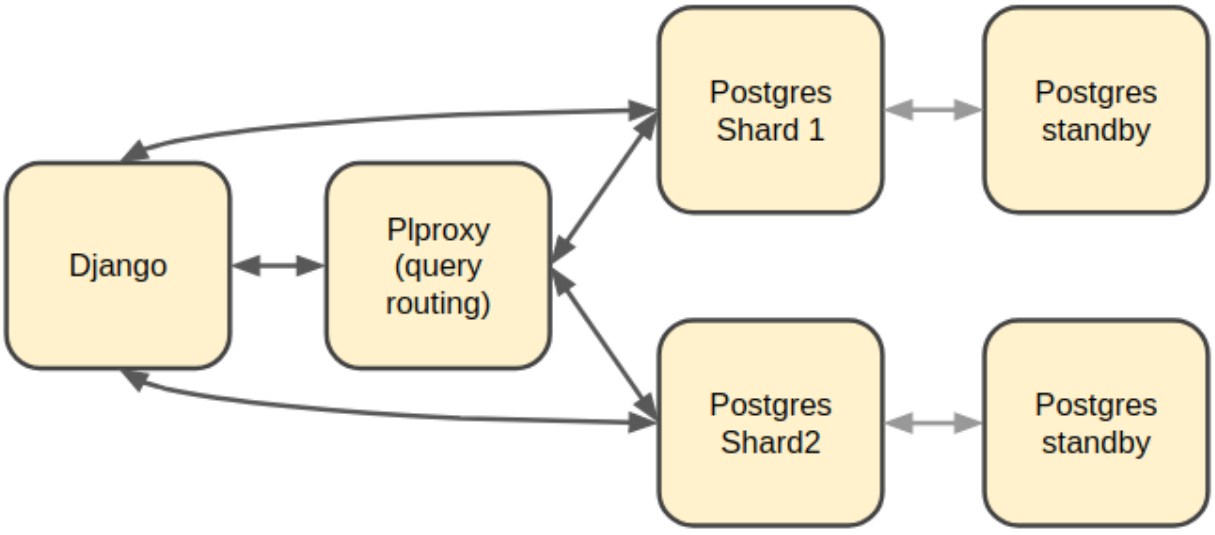
Data created by the UCR framework can be stored in multiple separate databases. Each UCR defines an *engine_id* parameter which tells it which configured database engine to use. These *engines* are defined in the *REPORTING_DATABASES* Django setting which maps the *engine_id* to a Django database alias defined in the *DATABASES* setting.

```
REPORTING_DATABASES = {
    'default': 'default',
    'ucr': 'ucr'
}
```

62.4 Sharded Form and Case data

It is recommended to have a separate set of databases to store data for Forms and Cases (as well as a few other models).

CommCare uses a combination of plproxy custom Python code to split the Form and Case data into multiple databases.



The general rule is that if a query needs to be run on all (or most) shard databases it should go through plproxy since plproxy is more efficient at running multiple queries and compiling the results.

The configuration for these databases must be added to the *DATABASES* setting as follows:

```
USE_PARTITIONED_DATABASE = True

DATABASES = {
    'proxy': {
```

(continues on next page)

(continued from previous page)

```
        ...
        'PLPROXY': {
            'PROXY': True
        }
    },
    'p1': {
        ...
        'PLPROXY': {
            'SHARDS': [0, 511]
        }
    },
    'p2': {
        ...
        'PLPROXY': {
            'SHARDS': [512, 1023]
        }
    }
}
```

62.4.1 Rules for shards

- There can only DB with *PROXY=True*
- The total number of shards must be a power of 2 i.e. 2, 4, 8, 16, 32 etc
- The number of shards cannot be changed once you have data in them so it is wise to start with a large enough number e.g. 1024
- The shard ranges must start at 0
- The shard ranges are inclusive
  - [0, 3] -> [0, 1, 2, 3]
- The shard ranges must be continuous (no gaps)

62.5 Sending read queries to standby databases

By including details for standby databases in the Django *DATABASES* setting we can configure CommCare to route certain READ queries to them.

Standby databases are configured in the same way as normal databases but may have an additional property group, *STANDBY*. This property group has the following sup-properties:

MASTER
: The DB alias of the master database for this standby. This must refer to a database in the *DATABASES* setting.

ACCEPTABLE_REPLICATION_DELAY
: The value of this must be an integer and configures the acceptable replication delay in seconds between the standby and the master. If the replication delay goes above this value then queries will not be routed to this database.

  The default value for *ACCEPTABLE_REPLICATION_DELAY* is 3 seconds.

```
DATABASES = {
    'default': {...}
    'standby1': {
        ...
        'STANDBY': {
            'MASTER': 'default',
            'ACCEPTABLE_REPLICATION_DELAY': 30,
        }
    }
}
```

Once the standby databases are configured in the *DATABASES* settings there are two additional settings that control which queries get routed to them.

REPORTING_DATABASES

The *REPORTING_DATABASES* setting can be updated as follows:

```
REPORTING_DATABASES = {
    'default': 'default',
    'ucr': {
        'WRITE': 'ucr',
        'READ': [
            ('ucr', 1),
            ('ucr_standby1', 2),
            ('ucr_standby2', 2),
        ]
    }
}
```

The tuples listed under the 'READ' key specify a database alias (must be in *DATABASES*) and weighting. In the configuration above 20% of reads will be sent to *ucr* and 40% each to *ucr_standby1* and *ucr_standby2* (assuming both of them are available and have replication delay within range).

LOAD_BALANCED_APPS

This setting is used to route read queries from Django models.

```
LOAD_BALANCED_APPS = {
    'users': {
        'WRITE': 'default',
        'READ': [
            ('default', 1),
            ('standby1', 4),
        ]
    }
}
```

In the configuration above all write queries from models in the *users* app will go to the *default* database as well as 20% or read queries. The remaining 80% of read queries will be sent to the *standby1* database.

For both the settings above, the following rules apply to the databases listed under *READ*:

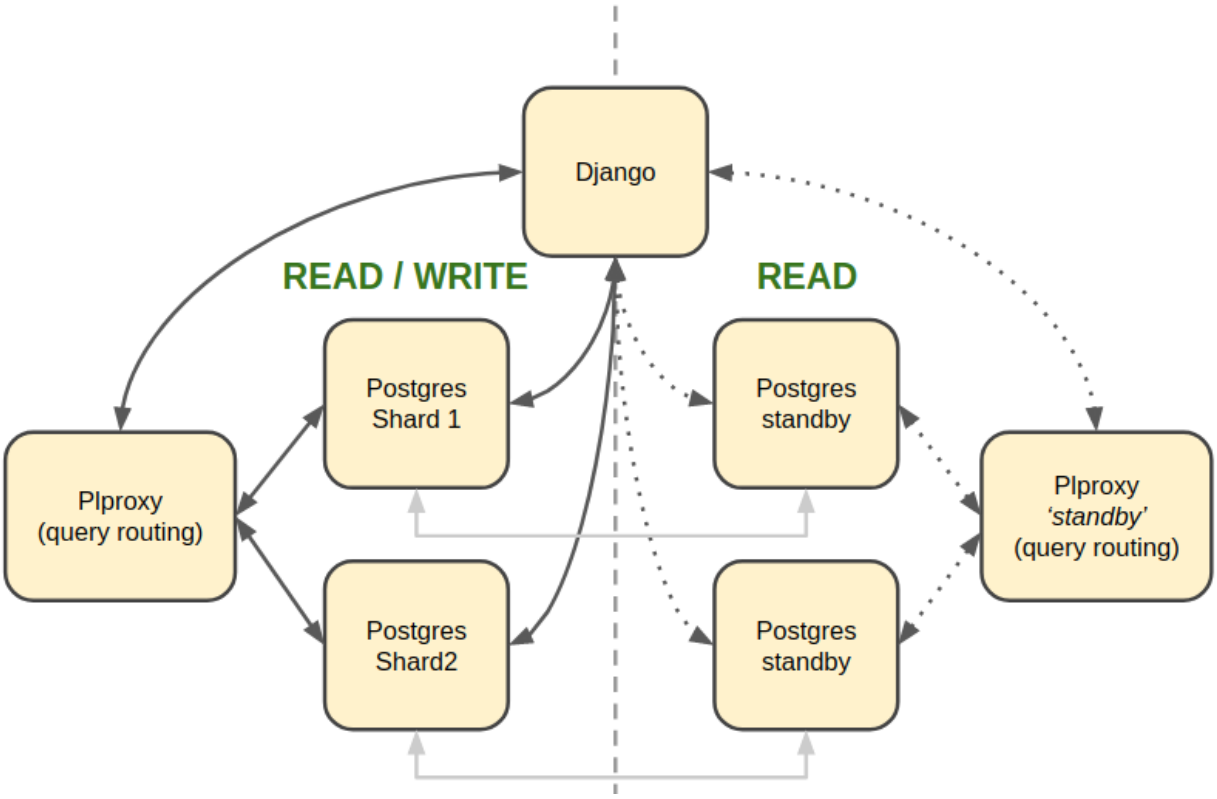
- There can only be one master database (not a standby database)
- All standby databases must point to the same master database
- If a master database is in this list, all standbys must point to this master

62.5.1 Using standbys with the plproxy cluster

The plproxy cluster needs some special attention since the queries are routed by plproxy and not by Django. In order to do this routing there are a number of additional pieces that are needed:

1. Separate plproxy cluster configuration which points the shards to the appropriate standby node instead of the primary node. 2. Duplicate SQL functions that make use of this new plproxy cluster.

In order to maintain the SQL function naming the new plproxy cluster must be in a separate database.



Example usage

```
# this will connect to the shard standby node directly
case = CommCareCase.objects.partitioned_get(case_id)

# this will call the `get_cases_by_id` function on the 'standby' proxy which in turn
# will query the shard standby nodes
cases = CommCareCase.objects.get_cases(case_ids, domain)
```

These examples assume the standby routing is active as described in the *Routing queries to standbys* section below.

Steps to setup

1. Add all the standby shard databases to the Django *DATABASES* setting as described above.

2. Create a new database for the standby plproxy cluster configuration and SQL accessor functions and add it to *DATABASES* as shown below:

```
DATABASES = {
    'proxy_standby': {
        ...
```

(continues on next page)

(continued from previous page)

```
        'PLPROXY': {
            'PROXY_FOR_STANDBYS': True
        }
    }
}
```

3. Run the *configure_pl_proxy_cluster* management command to create the config on the 'standby' database.
4. Run the Django migrations to create the tables and SQL functions in the new standby proxy database.

Routing queries to standbys

The configuration above makes it possible to use the standby databases however in order to actually route queries to them the DB router must be told to do so. This can be done it one of two ways:

1. Via an environment variable

```
export READ_FROM_PLPROXY_STANDBYS=1
```

This will route ALL read queries to the shard standbys. This is mostly useful when running a process like pillowtop that does is asynchronous.

2. Via a Django decorator / context manager

```
# context manager
with read_from_plproxy_standbys():
    case = CommCareCase.objects.partitioned_get(case_id)

# decorator
@read_from_plproxy_standbys()
def get_case_from_standby(case_id)
    return CommCareCase.objects.partitioned_get(case_id)
```

CHAPTER

SIXTYTHREE

METRICS

63.1 CommCare Infrastructure Metrics

CommCare uses Datadog and Prometheus for monitoring various system, application and custom metrics. Datadog supports a variety of applications and is easily extendable.

Below are a few tables tabulating various metrics of the system and service infrastructure used to run CommCare. The list is not absolute nor exhaustive, but it will provide you with a basis for monitoring the following aspects of your system:

- Performance
- Throughput
- Utilization
- Availability
- Errors
- Saturation

Each table has the following format:

Metric	Metric type	Why care	User impact	How to measure
Name of metric	*Category or aspect of system the metric speaks to*	*Brief description of why metric is important*	*Explains the impact on user if undesired reading is recorded*	*A note on how the metric might be obtained. Please note that it is assumed that Datadog will be used as a monitoring solution unless specified otherwise.*

63.1.1 General Host

The Datadog Agent ships with an integration which can be used to collect metrics from your base system. See the System Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
CPU usage (%)	Utilization	Monitoring server CPU usage helps you understand how much your CPU is being used, as a very high load might result in overall performance degradation.	Lagging experience	system.cpu.idle system.cpu.system system.cpu.iowait system.cpu.user
Load averages 1-5-15	Utilization	Load average (CPU demand) over 1 min, 5 min and 15 min which includes the sum of running and waiting threads. What is load average	User might experience trouble connecting to the server	system.load.1 system.load.5 system.load.15
Memory	Utilization	It shows the amount of memory used over time. Running out of memory may result in killed processes or more swap memory used, which will slow down your system. Consider optimizing processes or increasing resources.	Slow performance	system.mem.usable system.mem.total
Swap memory	Utilization	This metric shows the amount of swap memory used. Swap memory is slow, so if your system depends too much on swap, you should investigate why RAM usage is so high. Note that it is normal for systems to use a little swap memory even if RAM is available.	Server unresponsiveness	system.swap.free system.swap.used
Disk usage	Utilization	Disk usage is important to prevent data loss in the event that the disk runs out of available space.	Data loss	system.disk.in_use
Disk latency	Throughput	The average time for I/O requests issued to the device to be served. This includes the time spent by the requests in queue and the time spent servicing them. High disk latency will result in slow response	Slow performance	system.io.await

63.1.2 Gunicorn

The Datadog Agent ships with an integration which can be used to collect metrics. See the Gunicorn Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Requests per second	Throu put	This metric shows the rate of requests received. This can be used to give an indication of how busy the application is. If you're constantly getting a high request rate, keep an eye out for bottlenecks on your system.	Slow user experience or trouble accessing the site.	guni-corn.requests
Request duration	Throu put	Long request duration times can point to problems in your system / application.	Slow experience and timeouts	guni-corn.request.duration.*
Http status codes	Per-formanc	A high rate of error codes can either mean your application has faulty code or some part of your application infrastructure is down.	User might get errors on pages	guni-corn.request.status.*
Busy vs idle Gunicorn workers	Uti-liza-tion	This metric can be used to give an indication of how busy the gunicorn workers are over time. If most of the workers are busy most of the time, it might be necessary to start thinking of increasing the number of workers before users start having trouble accessing your site.	Slow user experience or trouble accessing the site.	guni-corn.workers

63.1.3 Nginx

The Datadog Agent ships with an integration which can be used to collect metrics. See the Nginx Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Total requests	Throughput	This metric indicates the number of client requests your server handles. High rates means bigger load on the system.	Slow experience	nginx.requests.total
Requests per second	Throughput	This metric shows the rate of requests received. This can be used to give an indication of how busy the application is. If you're constantly getting a high request rate, keep an eye out for services that might need additional resources to perform optimally.	Slow user experience or trouble accessing the site.	nginx.net.request_per_s
Dropped connections	Errors	If NGINX starts to incrementally drop connections it usually indicates a resource constraint, such as NGINX's worker_connections limit has been reached. An investigation might be in order.	Users will not be able to access the site.	nginx.connections.dropped
Server error rate	Error	Your server error rate is equal to the number of 5xx errors divided by the total number of status codes. If your error rate starts to climb over time, investigation may be in order. If it spikes suddenly, urgent action may be required, as clients are likely to report errors to the end user.	User might get errors on pages	nginx.server_zone.responses.5xx nginx.server_zone.responses.total_coun
Request time	Performance	This is the time in seconds used to process the request. Long response times can point to problems in your system / application.	Slow experience and timeouts	Need to include in NGINX configuration file

63.1.4 PostgreSQL

PostgreSQL has a statistics collector subsystem that collects and reports on information about the server activity.

The Datadog Agent ships with an integration which can be used to collect metrics. See the PostgreSQL Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Sequential scans on table vs. Index scans on table	Other	This metric speaks directly to the speed of query execution. If the DB is making more sequential scans than indexed scans you can improve the DB's performance by creating an index.	Tasks that require data to be fetched from the DB will take a long time to execute.	***PostgreSQL*:** pg_stat_user_tables ***Datadog integration*:** postgresql.seq_scans postgresql.index_scans
Rows fetched vs. returned by queries to DB	Throughput	This metric shows how effectively the DB is scanning through its data. If many more rows are constantly fetched vs returned, it means there's room for optimization.	Slow experience for tasks that access large parts of the database.	***PostgreSQL*:** pg_stat_database ***Datadog integration*:** postgresql.rows_fetched postgresql.rows_returned
Amount of data written temporarily to disk to execute queries	Saturation	If the DB's temporary storage is constantly used up, you might need to increase it in order to optimize performance.	Slow experience for tasks that read data from the database.	***PostgreSQL*:** pg_stat_database ***Datadog integration*:** postgresql.temp_bytes
Rows inserted, updated, deleted (by database)	Throughput	This metric gives an indication of what type of write queries your DB serves most. If a high rate of updated or deleted queries persist, you may want to keep an eye out for increasing dead rows as this will degrade DB performance.	No direct impact	***PostgreSQL*:** pg_stat_database ***Datadog integration*:** postgresql.rows_inserted postgresql.rows_updated postgresql.rows_deleted
Locks	Other	A high lock rate in the DB is an indication that queries could be long-running and that future queries might start to time out.	Slow experience for tasks that read data from the database.	***PostgreSQL*:** pg_locks ***Datadog integration*:** postgresql.locks
Deadlocks	Other	The aim is to have no	Slow experience for	

63.1.5 Elasticsearch

The Datadog Agent ships with an integration which can be used to collect metrics. See the Elasticsearch Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Query load	Utilization	Monitoring the number of queries currently in progress can give you a rough idea of how many requests your cluster is dealing with at any particular moment in time.	A high load might slow down any tasks that involve searching users, groups, forms, cases, apps etc.	elasticsearch.primaries.search.query.current
Average query latency	Throughput	If this metric shows the query latency is increasing it means your queries are becoming slower, meaning either bottlenecks or inefficient queries.	Slow user experience when generating or reports, filtering groups or users, etc.	elasticsearch.primaries.search.query.tota elasticsearch.primaries.search.query.tim
Average fetch latency	Throughput	This should typically take less time than the query phase. If this metric is constantly increasing it could indicate problems with slow disks or requesting of too many results.	Slow user experience when generating or reports, filtering groups or users, etc.	elasticsearch.primaries.search.fetch.total elasticsearch.primaries.search.fetch.time
Average index latency	Throughput	If you notice an increasing latency it means you may be trying to index too many documents simultaneously.Increasing latency may slow down user experience.	Slow user experience when generating or reports, filtering groups or users, etc.	elasticsearch.indexing.index.total elasticsearch.indexing.index.time
Average flush latency	Throughput	Data is only persisted on disk after a flush. If this metric increases with time it may indicate a problem with a slow disk. If this problem escalates it may prevent you from being able to add new information to your index.	Slow user experience when generating or reports, filtering groups or users, etc. In the worst case there may be some data loss.	elasticsearch.primaries.flush.total elasticsearch.primaries.flush.total.time
Percent of JVM heap currently in use	Utilization	Garbage collections should initiate around 75% of heap use. When this value is consistently going above 75% it indicates that the rate of garbage collection is not	Users might experience errors on some pages	jvm.mem.heap_in_use

63.1.6 CouchDB

The Datadog Agent ships with an integration which can be used to collect metrics. See the CouchDB Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Open databases	Availability	If the number of open databases are too low you might have database requests starting to pile up.	Slow user experience if the requests start to pile up high.	couchdb.couchdb.open_databas
File descriptors	Utilization	If this number reaches the max number of available file descriptors, no new connections can be opened until older ones have closed.	The user might get errors on some pages.	couchdb.couchdb.open_os_files over
Data size	Utilization	This indicates the relative size of your data. Keep an eye on this as it grows to make sure your system has enough disk space to support it.	Data loss	couchdb.by_db.file_size
HTTP Request Rate	Throughput	Gives an indication of how many requests are being served.	Slow performance	couchdb.couchdb.httpd.requests
Request with status code of 2xx	Performanc	Statuses in the 2xx range are generally indications of successful operation.	No negative impact	couchdb.couchdb.httpd_status_c
Request with status code of 4xx and 5xx	Performanc	Statuses in the 4xx and 5xx ranges generally tell you something is wrong, so you want this number as low as possible, preferably zero. However, if you constantly see requests yielding these statuses, it might be worth looking into the matter.	Users might get errors on some pages.	couchdb.couchdb.httpd_status_c
Workload - Reads & Writes	Performanc	These numbers will depend on the application, but having this metric gives an indication of how busy the database generally is. In the case of a high workload, consider ramping up the resources.	Slow performance	couchdb.couchdb.database_read
Average request latency	Throughput	If the average request latency is rising it means somewhere exists a bottleneck that needs to be addressed.	Slow performance	couchdb.couchdb.request_time.a
Cache hits	Other	CouchDB stores a fair amount of user credentials in memory to speed up the authentication process. Monitoring usage of the authentication cache can alert you for possible attempts to gain unauthorized access.	A low number of hits might mean slower performance	couchdb.couchdb.auth_cache_h
Cache misses	Error	If CouchDB reports a high number of cache misses, then either the cache is undersized to service the volume of legitimate user requests, or a brute force password/username attack is taking place.	Slow performance	couchdb.couchdb.auth_cache_m

63.1.7 Kafka

The Datadog Agent ships with a Kafka Integration to collect various Kafka metrics. Also see Integrating Datadog, Kafka and Zookeper.

Broker Metrics

Metric	Metric type	Why care	User impact	How to measure
UnderReplicated-Partitions	Availability	If a broker becomes unavailable, the value of UnderReplicatedPartitions will increase sharply. Since Kafka's high-availability guarantees cannot be met without replication, investigation is certainly warranted should this metric value exceed zero for extended time periods.	Fewer in-sync replicas means the reports might take longer to show the latest values.	kafka.replication.under_replicated_parti
IsrShrinksPerSec	Availability	The rate at which the in-sync replicas shrinks for a particular partition. This value should remain fairly static. You should investigate any flapping in the values of these metrics, and any increase in *IsrShrinksPerSec* without a corresponding increase in *IsrExpandsPerSec* shortly thereafter.	As the in-sync replicas become fewer, the reports might take longer to show the latest values.	kafka.replication.isr_shrinks.rate
IsrExpandsPerSec	Availability	The rate at which the in-sync replicas expands.	As the in-sync replicas become fewer, the reports might take longer to show the latest values.	kafka.replication.isr_expands.rate
TotalTimeMs	Performance	This metrics reports on the total time taken to service a request.	Longer servicing times mean data-updates take longer to propagate to the reports.	kafka.request.produce.time.avg kafka.request.consumer.time.avg kafka.request.fetch_follower.time.avg
ActiveController-Count	Error	The first node to boot in a Kafka cluster automatically becomes the controller, and there can be only one. You should alert on any other value that lasts for longer than one second. In the case	Reports might not show new updated data, or even break.	kafka.replication.active_controller_coun

Producer Metrics

Metric	Metric type	Why care	User impact	How to measure
Request rate	Throughput	The request rate is the rate at which producers send data to brokers. Keeping an eye on peaks and drops is essential to ensure continuous service availability.	Reports might take longer to reflect the latest data.	kafka.producer.request_rate
Response rate	Throughput	Average number of responses received per second from the brokers after the producers sent the data to the brokers.	Reports might take longer to reflect the latest data.	kafka.producer.response_rat
Request latency average	Throughput	Average request latency (in ms). Read more	Reports might take longer to reflect the latest data.	kafka.producer.request_laten
Outgoing byte rate	Throughput	Monitoring producer network traffic will help to inform decisions on infrastructure changes, as well as to provide a window into the production rate of producers and identify sources of excessive traffic.	High network throughput might cause reports to take a longer time to reflect the latest data, as Kafka is under heavier load.	kafka.net.bytes_out.rate
Batch size average	Throughput	To use network resources more efficiently, Kafka producers attempt to group messages into batches before sending them. The producer will wait to accumulate an amount of data defined by the batch size. Read more	If the batch size average is too low, reports might take a longer time to reflect the latest data.	kafka.producer.batch_size_a

Consumer Metrics

Metric	Metric type	Why care	User impact	How to measure
Reco lag	Performanc	Number of messages consumers are behind producers on this partition. The significance of these metrics' values depends completely upon what your consumers are doing. If you have consumers that back up old messages to long-term storage, you would expect records lag to be significant. However, if your consumers are processing real-time data, consistently high lag values could be a sign of overloaded consumers, in which case both provisioning more consumers and splitting topics across more partitions could help increase throughput and reduce lag.	Reports might take longer to reflect the latest data.	kafka.consumer_lag
Reco consume rate	Throu put	Average number of records consumed per second for a specific topic or across all topics.	Reports might take longer to reflect the latest data.	kafka.consumer.records_
Fetch rate	Throu put	Number of fetch requests per second from the consumer.	requests per second from the consumer.	kafka.request.fetch_rate

63.1.8 Zookeeper

The Datadog Agent ships with an integration which can be used to collect metrics. See the Zookeeper Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Outstanding requests	Saturation	This shows the number of requests still to be processed. Tracking both outstanding requests and latency can give you a clearer picture of the causes behind degraded performance.	Reports might take longer to reflect the latest data.	zookeeper.outstanding_requests
Average latency	Throu put	This metric records the amount of time it takes to respond to a client request (in ms).	Reports might take longer to reflect the latest data.	zookeeper.latency.avg
Open file descriptors	Utilization	Linux has a limited number of file descriptors available, so it's important to keep an eye on this metric to ensure ZooKeeper can continue to function as expected.	Reports might not reflect new data, as ZooKeeper will be getting errors.	zookeeper.open_file_descriptor

63.1.9 Celery

The Datadog Agent ships with a HTTP Check integration to collect various network metrics. In addition, CommCare HQ reports on many custom metrics for Celery. It might be worth having a look at Datadog's Custom Metrics page. Celery also uses Celery Flower as a tool to monitor some tasks and workers.

Metric	Metric type	Why care	User impact	How to measure
Celery uptime	Availability	The uptime rating is a measure of service availability.	Background tasks will not execute (sending of emails, periodic reporting to external partners, report downloads, etc)	network.http.can_connect
Celery uptime by queue	Availability	The uptime rating as per queue.	Certain background or asynchronous tasks will not get executed. The user might not notice this immediately.	CommCare HQ custom metric
Time to start	Other	This metric shows the time (seconds) it takes a task in a specific queue to start executing. If a certain task consistently takes a long time to start, it might be worth looking into.	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	CommCare HQ custom metric
Blockage duration by queue	Throughput	This metric indicates the estimated time (seconds) a certain queue was blocked. It might be worth it to alert if this blockage lasts longer than a specified time.	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	CommCare HQ custom metric
Task execution rate	Throughput	This metric gives a rough estimation of the amount of tasks being executed within a certain time bracket. This can be an important metric as it will indicate when more and more tasks take longer to execute, in which case an investigation might be appropriate.	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	CommCare HQ custom metric
Celery tasks by host	Throughput	Indicates the running time (seconds) for celery tasks by host.	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	CommCare HQ custom metric
Celery tasks by queue	Throughput	Indicates the running time (seconds) for celery tasks by queue. This way you can identify slower queues.	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	CommCare HQ custom metric
Celery tasks by task	Throughput	Indicates the running time (seconds) for celery tasks by each respective task. Slower tasks can be identified.	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	CommCare HQ custom metric
Tasks queued	Saturation	Indicates the number of tasks queued by each respective queue. If this becomes increasingly large, keep an	For the most part this might go unnoticed for the user, but there	Celery

63.1.10 RabbitMQ

The Datadog Agent ships with an integration which can be used to collect metrics. See the RabbitMQ Integration for more information.

Metric	Metric type	Why care	User impact	How to measure
Queue depth	Saturation	Using queue depth, messages ready and messages unacknowledged	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	rabbitmq.queue.messages
Messages ready	Other	Using queue depth, messages ready and messages unacknowledged	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	rabbitmq.queue.messages_read
Messages unacknowledged	Error	Using queue depth, messages ready and messages unacknowledged	Certain background tasks will fail to execute, like sending emails, SMS's, alerts, etc.	rabbitmq.queue.messages_una
Queue memory	Utilization	RabbitMQ keeps messages in memory for faster access, but if queues handle a lot of messages you could consider using lazy queues in order to preserve memory. Read more	For the most part this might go unnoticed for the user, but there will be a delay in the execution of background tasks, like sending emails, SMS's, alerts, etc.	rabbitmq.queue.memory
Queue consumer	Other	The number of consumers is configurable, so a lower-than-expected number of consumers could indicate failures in your application.	Certain background tasks might fail to execute, like sending emails, SMS's, alerts, etc.	rabbitmq.queue.consumers
Node sockets	Utilization	As you increase the number of connections to your RabbitMQ server, RabbitMQ uses a greater number of file descriptors and network sockets. Since RabbitMQ will block new connections for nodes that have reached their file descriptor limit, monitoring the available number of file descriptors helps you keep your system running.	Background tasks might take longer to execute as, or in the worst case, might not execute at all.	rabbitmq.node.sockets_used
Node file descriptors	Utilization	As you increase the number of connections to your RabbitMQ server, RabbitMQ uses a greater number of file descriptors and network sockets. Since RabbitMQ will block new connections for nodes that have reached their file descriptor limit, monitoring the available number of file descriptors helps you keep your system running.	Background tasks might take longer to execute as, or in the worst case, might not execute at all.	rabbitmq.node.fd_used

CHAPTER

SIXTYFOUR

COMMCARE EXTENSIONS

This document describes the mechanisms that can be used to extend CommCare's functionality. There are a number of legacy mechanisms that are used which are not described in this document. This document will focus on the use of pre-defined *extension points* to add functionality to CommCare.

64.1 Where to put custom code

The custom code for extending CommCare may be part of the main *commcare-hq* repository or it may have its own repository. In the case where it is in a separate repository the code may be 'added' to CommCare by cloning the custom repository into the *extensions* folder in the root of the CommCare source:

```
/commcare-hq
  /corehq
  /custom
  ...
  /extensions
    /custom_repo
      /custom_app1/models.py
      /custom_app2/models.py
```

64.2 Extensions Points

The *corehq/extensions* package provides the utilities to register extension points and their implementations and to retrieve the results from all the registered implementations.

64.2.1 Create an extension point

```
from corehq import extensions

@extensions.extension_point
def get_things(arg1: int, domain: str, keyword: bool = False) -> List[str]:
    '''Docs for the extension point'''
    pass


@extensions.extension_point
```

(continues on next page)

(continued from previous page)

```
def get_other_things():
    "Default implementation of ``get_other_things``. May be overridden by an extension"
    return ["default1", "default2"]
```

The extension point function is called if there are no registered extensions or none that match
the call args.

64.2.2 Registering an extension point implementation

Registering an extension point implementation is as simple as creating a function with the same signature as the extension point and adding a decorator to the function.

To guarantee that the extension point implementation is registered during startup you should also add the module path to the *COMMCARE_EXTENSIONS* list in settings.

The convention is to name your module *commcare_extensions* and place it in the root package of your Django app.

```
# in path/to/myapp/commcare_extensions.py

from xyz import get_things

@get_things.extend()
def some_things(arg1, domain, keyword=False):
    return ["thing2", "thing1"]


# in localsettings.py
COMMCARE_EXTENSIONS = ["custom.myapp.commcare_extensions"]
```

Extensions may also be limited to specific domains by passing the list of domains as a keyword argument (it must be a keyword argument). This is only supported if the extension point defines a *domain* argument.

```
from xyz import get_things

@get_things.extend(domains=["cat", "hat"])
def custom_domain_things(arg1, domain, keyword=False):
    return ["thing3", "thing4"]
```

64.2.3 Calling an extension point

An extension point is called as a normal function. Results are returned as a list with any *None* values removed.

```
from xyz import get_things

results = get_things(10, "seuss", True)
```

Formatting results

By default the results from calling an extension point are returned as a list where each element is the result from each implementation:

```
> get_things(10, "seuss", True)
[["thing2", "thing1"], ["thing3", "thing4"]]
```

Results can also be converted to a flattened list or a single value by passing a *ResultFormat* enum when defining the extension point.

Flatten Results

```
@extensions.extension_point(result_format=ResultFormat.FLATTEN)
def get_things(...):
    pass

> get_things(...)
["thing2", "thing1", "thing3", "thing4"]
```

First Result

This will return the first result that is not None. This will only call the extension point implementations until a value is found.

```
@extensions.extension_point(result_format=ResultFormat.FIRST)
def get_things(...):
    pass

> get_things(...)
["thing2", "thing1"]
```

CHAPTER

SIXTYFIVE

LIST EXTENSION POINTS

You can list existing extension points and their implementations by running the following management command:

```
python manage.py list_extension_points
```

CHAPTER

SIXTYSIX

CUSTOM MODULES

CommCare HQ includes some code written for specific projects.

Most of this consists of custom reports or customized messaging logic.

66.1 COVID: Available Actions

The following actions can be used in messaging in projects using the `covid` custom module.

`custom.covid.rules.custom_actions.`**`close_cases_assigned_to_checkin`**(*checkin_case*, *rule*)

For any associated checkin case that matches the rule criteria, the following occurs:

1. For all cases of a given type, find all assigned cases. An assigned case is a case for which all of the following are true:
   - Case type patient or contact
   - Exists in the same domain as the user case
   - The case property assigned_to_primary_checkin_case_id equals an associated checkin case's case_id
2. For every assigned case, the following case properties are blanked out (set to ""):
   - assigned_to_primary_checkin_case_id
   - is_assigned_primary
   - assigned_to_primary_name
   - assigned_to_primary_username

`custom.covid.rules.custom_actions.`**`set_all_activity_complete_date_to_today`**(*case*, *rule*)

For any case matching the criteria, set the all_activity_complete_date property to today's date, in YYYY-MM-DD format, based on the domain's default time zone.

66.2 COVID: Available Criteria

The following criteria can be used in messaging in projects using the `covid` custom module.

`custom.covid.rules.custom_criteria.`**`associated_usercase_closed`**(*case*, *now*)

Is this an open checkin case where the associated usercase has been closed?

CHAPTER

SIXTYSEVEN

MIGRATIONS IN PRACTICE

67.1 Background

67.1.1 Definitions

Schema migration - Modifies the database schema, say by adding a new column or changing properties of an existing column. Usually pretty fast but can get complicated if not backwards-compatible.

Data migration - Modifies data already in the database. This has the potential to be quite slow

Django migration - A migration (either kind) that is run automatically on deploy by `./manage.py migrate`. These are in files like `corehq/apps/<app>/migrations/0001_my_migration.py`

Management command - Some data migrations are written as management commands. These are then run either manually via a commcare-cloud command or automatically from inside a django migration using `call_command`. These are in files like `corehq/apps/<app>/management/commands/my_command.py`

Private release - If you need to run code from a branch that's not currently deployed, use a private release.

67.1.2 General Principles

Don't block deploys - If you write a migration that will take more than 15 minutes to run on any server (typically production will have the most data and therefore be the slowest), take care to run it outside of a deploy, otherwise that deploy will hang. How do you know if your migration will take too long? Run it on staging and compare the amount of data on staging to the amount of data on prod to estimate. If in any kind of doubt, err on the side of caution. In practice, any migration that touches common models - users, apps, domains - will need to be run outside of a deploy, while migrations to tiny tables (thousands of rows) or flagged features with few users may be able to run in a deploy.

Deploys are not instantaneous - Deploys will push new code, run the migrations in that new code, and *then* switch the servers to the new code. The site will be active this whole time. Users or tasks can add or modify data after the start of the migration and before the code switch and you need to account for that.

All migration states should be valid - Similar to the above, you must consider the states before, during, and after the migration. Will the active code handle all three states correctly?

Master should always be deployable - If you have a PR with a migration that requires manual handling, don't merge it until you are prepared to handle it.

Remember third parties - We'll often manage migrations manually for at least prod and india, but third parties run environments that we can't manage directly. Be sure that whatever changes are necessary will be applied automatically on these environments, though it will likely require running data migrations during deploy. If the change may be disruptive or *requires* manual handling, we'll need to communicate it out in advance.

67.1.3 Practical Considerations

Handling new data - There's likely a code change that writes to the database in the new way going forward. It cannot be deployed until any requisite schema changes have been implemented.

Migrating old data - This will be handled via a django migration, a management command, or both. Typically, small/simple migrations are handled by a django migration and large/complex ones use a django migration that runs a management command. It cannot run until any schema changes are deployed.

Dealing with the gap - We generally can't pause the servers, put everything to rights, then restart. Rather, we must ensure that we're saving new data properly before migrating old data, or otherwise ensure that all data from before, during, and after the migration is handled correctly.

Backwards-incompatible changes - These are best avoided. A common workaround is to treat it as two migrations - one to store the data (in duplicate, with any necessary syncing code) the new way, then a later migration to remove the old way from the schema. With couchdb, this is a little easier, since you don't need to remove the old schema once all the data is migrated.

Idempotence and resumability - If at all possible, you should design your management command to be run multiple times without changing the result, breaking, or redoing work. This means it should expect that some data in the db might already be migrated, and only operate on unmigrated data. This should happen performantly. This will help with some of the below migration strategies and make dealing with unexpected failures much smoother.

Shim code for the transition - Another useful pattern in some circumstances is to write short-lived code that can deal with both versions of the data in the db. This can make the transition much easier. One example of this is overriding the `wrap` method on a couch Document. Be sure to make a clean-up PR that drops support for the old version to be merged later once it's no longer needed.

Migration-dependent code - This is the reason the migration is being performed. You need to make sure all data is migrated before code depending on it is released.

Testing - Complex migrations can justify unit tests. These tests are often short-lived, but they can protect against highly disruptive and lengthy data cleanup caused by a bug. With migrations, plan for the worst. Example tests for a management command.

Staging - Prod data is usually much more complex than what you have locally or what you might write for test cases. Before running your migration on prod, run it on staging. However, this can cause issues if you need to change your migration later, or if another migration in the same app conflicts with yours. Be sure to leave staging in a good state.

67.2 Example Migration: User Logging

Let's speak about this a bit more concretely with a specific example migration in mind. This is based on a real example, but I've idealized it somewhat here for illustration. Here's a brief description of a migration which will be referred back to throughout this document.

We log all changes to users, keeping track of who changed what. We currently store the ID of the affected user, but now we want to store the username too. This means we'll need to make the following four changes:

1. **Schema migration**: Add a new `user_repr` field to the log model to hold the username
2. **Data migration**: Populate that field for all existing log rows
    a. If this will be run multiple times (more on that below), it should be idempotent and resumable.
        i. Resumability: Rather than update *all* user changelogs, it should filter out those that already have `user_repr` set, so subsequent runs can be much much faster.
        ii. Idempotence: Running the command multiple times should behave the same as if it were run only once. For example, the command shouldn't error if it encounters an already migrated log. It also shouldn't

apply a modification where unnecessary, like if the migration appended the `user_repr` to a string, then running it twice might result in something like `"user@example.comuser@example.com"`

3. **Handle new data**: Modify the logging code to populate the `user_repr` field going forward.
4. **Migration-dependent code**: Update the UI to display the `user_repr` and make it filterable. We can't turn this on until all existing logs have `user_repr` set, or at least they'll need to anticipate that some rows will be missing that field.

Because this example exclusively adds new data, there's no cleanup step. Some migrations will need to remove an "old" way of doing things, which is frequently done in an additional PR. For low-risk, simple, single-PR migrations, cleanup might be included in the single PR.

67.3 Common types of migrations

67.3.1 Simple

If you're adding a new model or field in postgres that doesn't need to be back-populated, you can just put the schema migration in the same PR as the associated code changes, and the deploy will apply that migration before the new code goes live. In couch, this type of change doesn't require a migration at all.

User Logging Example

A "simple" migration would not be suitable for the example user logging migration described above. If you tried to make all those changes in a single PR and let it get deployed as-is, you risk missing data. During deploy, the data migration will be run before the code handling new data properly goes live. Any users modified in this period would not have the `user_repr` populated. Additionally, the migration might take quite a while to run, which would block the deploy.

67.3.2 Multiple deploys

This is the most robust approach, and is advocated for in the couch-to-sql pattern. You make two PRs:

- **PR 1**: Schema migration; handle new data correctly; data migration management command
- **PR 2**: Django migration calling the management command; actual code relying on the migration

After the first PR is deployed, you can run the migration in a management command on whatever schedule is appropriate. The Django migration in the second PR calls the command again so we can be sure it's been run at least once on every environment. On production, where the command has been run manually already, this second run should see that there are no remaining unmigrated rows/documents in the db and be nearly a noop.

Although using two deploys eliminates the risk of an indeterminate state on environments that you control, this risk is still present for third party environments. If the third party doesn't deploy often and ends up deploying the two PRs together, there's still a risk of changes happening in the gap between the migration and the new code going live. The magnitude of this risk depends on the functionality being migrated - how much data it touches and how frequently it is used. If necessary, you can mitigate this risk by spacing the deploys so that third parties are likely to deploy them separately. See guidelines for third parties running CommCare.

User Logging Example

Splitting the example user logging migration across two deploys would be a good way to ensure everything is handled correctly. You'd split the changes into two PRs as described above and deploy them separately. The steps would be:

1. **First PR deployed**: Now we have the schema change live, and all new changes to users have the `user_repr` field populated. Additionally, the management command is available for use.
2. **Run the management command**: This can be done in a private release any time before the second deploy. This should almost certainly be done on prod. Whether or not it needs to be done on the other Dimagi-managed environments (india, swiss) depends on how much data those environments have.
3. **Second PR deployed**: This will run the management command again, but since all logs have already been migrated, it won't actually make any changes and should run fast - see the migrations best practices section below. This will also make sure third party environments have the change applied. This second PR also finally contains user-facing references to the `user_repr` field, since by the time the code switch happens, everything will have been migrated.

67.3.3 Single Deploy

While this single-deploy option is tempting compared to waiting weeks to get out a multi-deploy migration, it's really only suitable for specific situations like custom work and unreleased features, where we can be confident the drawbacks are insignificant.

The main drawbacks are:

- This method requires manually running the Django migrations which are normally only run during deploy. Running migrations manually on a production environment is generally a bad idea.
- It is possible that there will be a gap in data between the final run of the data migration command and the new going live (due to the sequence of events during a deploy).

If you decide to go down this route you should split your changes into two PRs:

- **PR 1**: Schema migration; data migration management command
- **PR 2**: Handle new data correctly; Django migration calling the management command; actual code relying on the migration

Once the PRs have both been approved, **merge PR 1**, then set up a private release containing that change. Merging the PR first will prevent migration conflicts with anyone else working in the area, and it's a good idea that anything run on a production environment is on the master branch.

Run your schema migration and management command directly:

```
cchq <ENV> django-manage --release=<NAME> migrate <APP_NAME>          cchq <ENV>
django-manage --release=<NAME> my_data_migration_command
```

Then merge PR 2. The subsequent deploy will run your management command again, though it should be very quick this time around, since nearly all data has been migrated, and finally the code changes will go live.

The big limitation here is that there's a gap between the final run of the management command and go-live (especially with the variation). Any changes in the interim won't be accounted for. This is sometimes acceptable if you're confident no such changes will have happened (eg, the migration pertains only to a custom feature, and we know that project won't have relevant activity during that period).

User Logging Example

Consider attempting to apply our example user logging migration with a single deploy. Make two PRs as described, so they can be merged independently. Then while coordinating with the team, merge the first PR, deploy a private release, and run the schema migration, then the management command.

The second PR can be merged and go live with the next deploy. This django migration will re-run the management command, picking up any new changes since it was previously run. In our case, this should be a small enough data set that it won't hinder the deploy. *However*, any changes in the window between that run and go-live will not be migrated. To pick up those changes, you can run the management command a third time after the deploy, which will ensure all user logs have been migrated.

This is still not ideal, since for the period between go-live and this third run, there will be missing data in the DB and that data will be in-use in the UI. Remember also that third party environments will have the management command run only once, on the second deploy (unless we announce this as a required maintenance operation), which would mean their data would have a gap in it.

67.4 Best practices for data migrations in Python

Consider codifying boundaries for your migration - This is especially useful for large migrations that might require manual intervention or special handling on third party environments. See detailed instructions in the *Auto-Managed Migration Pattern* doc.

Don't fetch all data at once - Instead, use an iterator that streams data in chunks (note that django queryset's `.iter()` method does not do this). Some models have their own performant getters, for others, consider `queryset_to_iterator` for SQL models, `iter_update` or `IterDB` for couch models. The `chunked` function is also helpful for this.

Don't write all data at once - Instead, write data in chunks (ideally) or individually (if needed, or if performance isn't a concern). For couch, use `IterDB`, `iter_update`, or `db.bulk_save`. For SQL, use `django_bulk_update`. Remember though that these bulk options won't call the `save()` method of your model, so be sure to check for any relevant side effects or signals that happen there and either trigger them manually or use individual saves in this instance.

Don't hold all data in memory - Since you're submitting in chunks anyways, consider writing your changes in chunks as you iterate through them, rather than saving them all up and submitting at the end.

Don't write from elasticsearch - It's sometimes necessary to use ES to find the data that needs to be modified, but you should only return the ids of the objects you need, then pull the full objects from their primary database before modifying and writing.

Check your assumptions - Consider what could go wrong and encode your assumptions about the state of the world. Eg: if you expect a field to be blank, check that it is before overwriting. Consider what would happen if your migration were killed in the middle - would that leave data in a bad state? Would the migration need to redo work when run again? Couch data in particular, since it's less structured than SQL, can contain surprising data, especially in old documents.

Only run migrations when needed - All historical migrations are run whenever a new environment is set up. This means your migrations will be run in every future test run and in every future new production or development environment. If your migration is only relevant to environments that already have data in the old format, decorate it with `@skip_on_fresh_install` so that it is a noop for new environments.

CHAPTER

SIXTYEIGHT

AUTO-MANAGED MIGRATION PATTERN

A re-entrant data migration management command can be a useful way to perform large-scale data migrations in environments where the migration takes a long time to complete due to the volume of data being migrated. A management command is better than a simple Django migration because it can be designed to be stopped and started as many times as necessary until all data has been migrated. Obviously the migration must be performed prior to the deployment of any code depending on the finished migration, so it must be applied to all environments before that can happen.

However, it would be tedious and error prone to require everyone running smaller CommCare HQ environments, including developers who are working on other parts of the project, to learn about and follow the painstaking manual process used to migrate large environments. This document outlines a pattern that can be used to ensure a smooth rollout to everyone running any size environment with minimal overhead for those running small environments.

68.1 Pattern Components

- A management command that performs the data migration.
  - Unless downtime will be scheduled, the command should be written in a way that allows legacy code to continue working while the migration is in progress. Techniques for achieving this are out of scope here.
  - May accept a `--dbname=xxxx` parameter to limit operation to the given database.
- Change log entry in CommCare Cloud describing the steps to perform the migration manually by running the management command.
- A Django migration that will
  - Check if there are any items that need to be migrated
  - Run the management command if necessary
  - Verify management command success/failure
  - Display an error and stop on failure
  - Continue with next migration on success

68.2 Django Migration Code Example

Edit as necessary to fit your use case. The constants at the top and the migration dependencies are the most important things to review/change.

This example does a migration that only affects SQL data, but that is not required. It is also possible to apply this pattern to migrations on non-SQL databases as long as the necessary checks (does the migration need to be run? did it run successfully?) can be performed in the context of a Django migration.

```
import sys
import traceback

from django.core.management import call_command, get_commands
from django.db import migrations

from corehq.util.django_migrations import skip_on_fresh_install


COUNT_ITEMS_TO_BE_MIGRATED = "SELECT COUNT(*) FROM ..."
GIT_COMMIT_WITH_MANAGEMENT_COMMAND = "TODO change this"
AUTO_MIGRATE_ITEMS_LIMIT = 10000
AUTO_MIGRATE_COMMAND_NAME = "the_migration_management_command"
AUTO_MIGRATE_FAILED_MESSAGE = """
This migration cannot be performed automatically and must instead be run manually
before this environment can be upgraded to the latest version of CommCare HQ.
Instructions for running the migration can be found at this link:

https://commcare-cloud.readthedocs.io/en/latest/changelog/0000-example-entry.html
"""
AUTO_MIGRATE_COMMAND_MISSING_MESSAGE = """
You will need to checkout an older version of CommCare HQ before you can perform this␣
↪migration
because the management command has been removed.

git checkout {commit}
""".format(commit=GIT_COMMIT_WITH_MANAGEMENT_COMMAND)


@skip_on_fresh_install
def _assert_migrated(apps, schema_editor):
    """Check if migrated. Raises SystemExit if not migrated"""
    num_items = count_items_to_be_migrated(schema_editor.connection)

    migrated = num_items == 0
    if migrated:
        return

    if AUTO_MIGRATE_COMMAND_NAME not in get_commands():
        print("")
        print(AUTO_MIGRATE_FAILED_MESSAGE)
        print(AUTO_MIGRATE_COMMAND_MISSING_MESSAGE)
        sys.exit(1)
```

(continues on next page)

(continued from previous page)

```
    if num_items < AUTO_MIGRATE_ITEMS_LIMIT:
        try:
            # add args and kwargs here as needed
            call_command(AUTO_MIGRATE_COMMAND_NAME)
            migrated = count_items_to_be_migrated(schema_editor.connection) == 0
            if not migrated:
                print("Automatic migration failed")
        except Exception:
            traceback.print_exc()
    else:
        print("Found %s items that need to be migrated." % num_items)
        print("Too many to migrate automatically.")

    if not migrated:
        print("")
        print(AUTO_MIGRATE_FAILED_MESSAGE)
        sys.exit(1)


def count_items_to_be_migrated(connection):
    """Return the number of items that need to be migrated"""
    with connection.cursor() as cursor:
        cursor.execute(COUNT_ITEMS_TO_BE_MIGRATED)
        return cursor.fetchone()[0]


class Migration(migrations.Migration):

    dependencies = [
        ...
    ]

    operations = [
        migrations.RunPython(_assert_migrated, migrations.RunPython.noop)
    ]
```

68.3 Real-life example

XForm attachments to blob metadata migration.

CHAPTER

SIXTYNINE

MIGRATING DATABASE DEFINITIONS

There are currently three persistent data stores in CommCare that can be migrated. Each of these have slightly different steps that should be followed.

69.1 General

For all ElasticSearch and CouchDB changes, add a "reindex/migration" flag to your PR. These migrations generally have some gotchas and require more planning for deploy than a postgres migration.

69.2 Adding Data

69.2.1 Postgres

Add the column as a nullable column. Creating NOT NULL constraints can lock the table and take a very long time to complete. If you wish to have the column be NOT NULL, you should add the column as nullable and migrate data to have a value before adding a NOT NULL constraint.

69.2.2 ElasticSearch

You only need to add ElasticSearch mappings if you want to search by the field you are adding. There are two ways to do this:

a. Change the mapping's name, add the field, and using ptop_preindex.

b. Add the field, reset the mapping, and using ptop_preindex with an *in-place* flag.

If you change the mapping's name, you should add reindex/migration flag to your PR and coordinate your PR to run ptop_preindex in a private release directory. Depending on the index and size, this can take somewhere between minutes and days.

69.2.3 CouchDB

You can add fields as needed to couch documents, but take care to handle the previous documents not having this field defined.

69.3 Removing Data

69.3.1 General

Removing columns, fields, SQL functions, or views should always be done in multiple steps.

1. Remove any references to the field/function/view in application code
2. Wait until this code has been deployed to all relevant environments.
3. Remove the column/field/function/view from the database.

Step #2 isn't reasonable to expect of external parties locally hosting HQ. For more on making migrations manageable for all users of HQ, see the "Auto-Managed Migration Pattern" link below.

It's generally not enough to remove these at the same time because any old processes could still reference the to be deleted entity.

69.3.2 Couch

When removing a view, procedure depends on whether or not you're removing an entire design doc (an entire *_design* directory). If the removed view is the last one in the design doc, run *prune_couch_views* to remove it. If other views are left in the design doc, a reindex is required.

69.3.3 ElasticSearch

If you're removing an index, you can use *prune_es_indices* to remove all indices that are no longer referenced in code.

69.4 Querying Data

69.4.1 Postgres

Creating an index can lock the table and cause it to not respond to queries. If the table is large, an index is going to take a long time. In that case:

1. Create the migration normally using django.
2. On all large environments, create the index concurrently. One way to do this is to use ./manage.py run_sql ... to apply the SQL to the database.
3. Once finished, fake the migration. Avoid this by using *CREATE INDEX IF NOT EXISTS ...* in the migration if possible.
4. Merge your PR.

69.4.2 Couch

Changing views can block our deploys due to the way we sync our couch views. If you're changing a view, please sync with someone else who understands this process and coordinate with the team to ensure we can rebuild the view without issue.

69.5 Migration Patterns and Best Practices

- *Migrations in Practice*
- *Auto-Managed Migration Pattern*
- *Migrating models from couch to postgres*

CHAPTER

SEVENTY

MIGRATING MODELS FROM COUCH TO POSTGRES

This is a step by step guide to migrating a single model from couch to postgres.

70.1 Conceptual Steps

This is a multi-deploy process that keeps two copies of the data - one in couch, one in sql - in sync until the final piece of code is deployed and the entire migration is complete. It has three phases:

1. Add SQL models and sync code
    - Define the new SQL models, based on the existing couch classes and using the SyncSQLToCouchMixin to keep sql changes in sync with couch.
    - Add the SyncCouchToSQLMixin to the couch class so that changes to couch documents get reflected in sql.
    - Write a management command that subclasses PopulateSQLCommand, which will create/update a corresponding SQL object for every couch document. This command will later be run by a django migration to migrate the data. For large servers, this command will also need to be run manually, outside of a deploy, to do the bulk of the migration.
2. Switch app code to read/write in SQL
    - Update all code references to the couch classes to instead refer to the SQL classes.
    - Write a django migration that integrates with `PopulateSQLCommand` to ensure that all couch and sql data is synced.
3. Remove couch
    - Delete the couch classes, and remove the `SyncSQLToCouchMixin` from the SQL classes.

70.2 Practical Steps

Even a simple model takes several pull requests to migrate, to avoid data loss while deploys and migrations are in progress. Best practice is a minimum of three pull requests, described below, each deployed to all large environments before merging the next one.

Some notes on source control:

- It's best to create all pull requests at once so that reviewers have full context on the migration.
- It can be easier to do the work in a single branch and then make the branches for individual PRs later on.

- If you don't typically run a linter before PRing, let the linter run on each PR and fix errors before opening the next one.
- Avoid having more than one migration happening in the same django app at the same time, to avoid migration conflicts.

70.2.1 PR 1: Add SQL model and migration management command, write to SQL

This should contain:

- A new model and a management command that fetches all couch docs and creates or updates the corresponding SQL model(s).
  - Start by running the management command `evaluate_couch_model_for_sql django_app_name MyDocType` on a production environment. This will produce code to add to your models file, a new management command and also a test which will ensure that the couch model and sql model have the same attributes.
    - The reason to run on production is that it will examine existing documents to help determine things like `max_length`. This also means it can take a while. If you have reasonable data locally, running it locally is fine - but since the sql class will often have stricter data validation than couch, it's good to run it on prod at some point.
    - If the script encounters any list or dict properties, it'll ask you if they're submodels. If you say no, it'll create them as json columns. If you say yes, it'll skip them, because it doesn't currently handle submodels. For the same reason, it'll skip SchemaProperty and SchemaListProperty attributes. More on this subject below.
    - Properties found on documents in Couch that are not members of the Couch model class will be added to the SQL model. In most cases they can be dropped (and not migrated to SQL).
    - Properties that are present in the Couch model, but always `null` or not found in Couch will be added to the SQL model as `unknown_type(null=True)`. These fields may be able to be dropped (and not migrated to SQL).
  - Add the generated models code to your models file. Edit as needed. Note the TODOs marked in the code:
    - The new class's name will start with "SQL" but specify table name `db_table` that does not include "sql." This is so that the class can later be renamed back to the original couch class's name by just removing the `db_table`. This avoids renaming the table in a django migration, which can be a headache when submodels are involved.
    - The new class will include a column for couch document id.
    - The generated code uses SyncCouchToSQLMixin and SyncSQLToCouchMixin. If your model uses submodels, you will need to add overrides for `_migration_sync_to_sql` and `_migration_sync_to_couch`. If you add overrides, definitely add tests for them. Sync bugs are one of the easiest ways for this to go terribly wrong.
      - For an example of overriding the sync code for submodels, see the CommtrackConfig migration, or the CustomDataFields migration which is simpler but includes a P1-level bug fixed here.
      - Beware that the sync mixins capture exceptions thrown while syncing in favor of calling `notify_exception`. If you're overwriting the sync code, this makes bugs easy to miss. The branch `jls/sync-mixins-hard-fail` is included on staging to instead make syncing fail hard; you might consider doing the same while testing locally.
    - Consider if your new model could use any additional `db_index` flags or a `unique_together`.

    * Some docs have attributes that are couch ids of other docs. These are weak spots easy to forget when the referenced doc type is migrated. Add a comment so these show up in a grep for the referenced doc type.

  - Run `makemigrations`
  - Add the test that was generated to it's respective place. - The test file uses a *ModelAttrEquality* util which has methods for running the equality tests. - The test class that is generated will have two attributes *couch_only_attrs*, *sql_only_attrs* and one method *test_have_same_attrs*. - Generally during a migration some attributes and methods are renamed or removed as per need. To accomodate the changes you can update *couch_only_attrs* and *sql_only_attrs*. - *couch_only_attrs* should be a set of attributes and methods which are either removed, renamed or not used anymore in SQL. - *sql_only_attrs* should be a set of attributes and methods that are new in the SQL model. - *test_have_same_attrs* will test the equality of the attributes. The default implementation should work if you have populated *couch_only_attrs* and *sql_only_attrs* but you can modify it's implementation as needed.
  - Add the generated migration command. Notes on this code:
    * The generated migration does not handle submodels. Support for submodels with non-legacy bulk migrations might just work, but has not been tested. Legacy migrations that implement `update_or_create_sql_object` should handle submodels in that method.
    * Legacy mode: each document is saved individually rather than in bulk when `update_or_create_sql_object` is implemented. `update_or_create_sql_object` populates the sql models based on json alone, not the wrapped document (to avoid introducing another dependency on the couch model). You may need to convert data types that the default `wrap` implementation would handle. The generated migration will use `force_to_datetime` to cast datetimes but will not perform any other wrapping. Similarly, if the couch class has a `wrap` method, the migration needs to manage that logic. As an example, `CommtrackActionConfig.wrap` was defined here and handled in this migration. **WARNING**: migrations that use `update_or_create_sql_object` have a race condition.
      · A normal HQ operation loads a Couch document.
      · A `PopulateSQLCommand` migration loads the same document in a batch of 100.
      · The HQ operation modifies and saves the Couch document, which also syncs changes to SQL (the migration's copy of the document is now stale).
      · The migration calls `update_or_create_sql_object` which overwrites above changes, reverting SQL to the state of its stale Couch document.
    * The command will include a `commit_adding_migration` method to let third parties know which commit to deploy if they need to run the migration manually. This needs to be updated **after** this PR is merged, to add the hash of the commit that merged this PR into master.

- Most models belong to a domain. For these:
  - Add the new model to DOMAIN_DELETE_OPERATIONS so it gets deleted when the domain is deleted.
  - Update tests in *test_delete_domain.py*. Sample PR that handles several app manager models.
  - Add the new model to sql/dump.py so that it gets included when a domain is exported.

To test this step locally:

- With master checked out, make sure you have at least one couch document that will get migrated.
- Check out your branch and run the populate command. Verify it creates as many objects as expected.
- Test editing the pre-existing object. In a shell, verify your changes appear in both couch and sql.
- Test creating a new object. In a shell, verify your changes appear in both couch and sql.

Automated tests are also a good idea. Automated tests are definitely necessary if you overrode any parts of the sync mixins. Example of tests for sync and migration code.

The migration command has a `--verify` option that will find any differences in the couch data vs the sql data.

The `--fixup-diffs=/path/to/migration-log.txt` option can be used to resolve differences between Couch and SQL state. Most differences reported by the migration command should be transient; that is, they will eventually be resolved by normal HQ operations, usually within a few milliseconds. **The ``–fixup-diffs`` option should only be used to fix persistent differences caused by a bug in the Couch to SQL sync logic after the bug has been fixed.** If a bug is discovered and most rows have diffs and (important!) PR 2 has not yet been merged, it may be more efficient to fix the bug, delete all SQL rows (since Couch is still the source of truth), and redo the migration.

Once this PR is deployed - later, after the whole shebang has been QAed - you'll run the migration command in any environments where it's likely to take more than a trivial amount of time. If the model is tied to domains you should initially migrate a few selected domains using `--domains X Y Z` and manually verify that the migration worked as expected before running it for all the data.

70.2.2 PR 2: Verify migration and read from SQL

This should contain:

- A django migration that verifies all couch docs have been migrated and cleans up any stragglers, using the auto-managed migration pattern.
  - This should be trivial, since all the work is done in the populate command from the previous PR.
  - The migration does an automatic completeness check by comparing the number of documents in Couch to the number of rows in SQL. If the counts do not match then the migration is considered incomplete, and the migration will calculate the difference and either migrate the remaining documents automatically or prompt for manual action. **NOTE**: if the automatic migration route is chosen (in the case of a small difference) the migration may still take a long time if the total number of documents in Couch is large since the migration must check every document in Couch (of the relevant doc type) to see if it has been migrated to SQL. A count mismatch is more likely when documents are written (created and/or deleted) frequently. One way to work around this is to use the `--override-is-migration-completed` option of `PopulateSQLCommand` to force the migration into a completed state. **WARNING**: careless use of that option may result in an incomplete migration. It is recommended to only force a completed state just before the migration is applied (e.g., just before deploying), and after checking the counts with `--override-is-migration-completed=check`.
  - Sample migration for RegistrationRequest.
- Replacements of all code that reads from the couch document to instead read from SQL. This is the hard part: finding **all** usages of the couch model and updating them as needed to work with the sql model. Some patterns are:
  - Replacing couch queries with SQL queries.
  - Unpacking code that takes advantage of couch docs being json.
  - Replacing `get_id` with `id` - including in HTML templates, which don't typically need changes - and `MyModel.get(ID)` with `SQLMyModel.objects.get(id=ID)`.

For models with many references, it may make sense to do this work incrementally, with a first PR that includes the verification migration and then subsequent PRs that each update a subset of reads. Throughout this phase, all data should continue to be saved to both couch and sql.

After testing locally, this PR is a good time to ask the QA team to test on staging. Template for QA request notes:

```
This is a couch to sql migration, with the usual approach:
- Set up <workflow to create items in couch>.
- Ping me on the ticket and I'll deploy the code to staging and run the migration
- Test that you can <workflows to edit the items created earlier> and also <workflow to␣
↪create new items>.
```

70.2.3 PR 3: Cleanup

This is the cleanup PR. Wait a few weeks after the previous PR to merge this one; there's no rush. Clean up:

- If your sql model uses a `couch_id`, remove it. Sample commit for HqDeploy
- Remove the old couch model, which at this point should have no references. This includes removing any syncing code.
- Now that the couch model is gone, rename the sql model from `SQLMyModel` to `MyModel`. Assuming you set up `db_table` in the initial PR, this is just removing that and running `makemigrations`.
- Add the couch class to `DELETABLE_COUCH_DOC_TYPES`. Blame deletable_doc_types.py for examples.
- Remove any couch views that are no longer used. Remember this may require a reindex; see the main db migration docs.

70.3 Current State of Migration

The current state of the migration is available internally here, which outlines approximate LOE, risk level, and notes on the remaining models.

For a definitive account of remaining couch-based models, you can identify all classes that descend from `Document`:

```
from dimagi.ext.couchdbkit import Document

def all_subclasses(cls):
    return set(cls.__subclasses__()).union([s for c in cls.__subclasses__() for s in all_
↪subclasses(c)])

sorted([str(s) for s in all_subclasses(Document)])
```

To find how many documents of a given type exist in a given environment:

```
from corehq.dbaccessors.couchapps.all_docs import get_doc_ids_by_class, get_deleted_doc_
↪ids_by_class

len(list(get_doc_ids_by_class(MyDocumentClass) + get_deleted_doc_ids_by_
↪class(MyDocumentClass)))
```

There's a little extra value to migrating models that have dedicated views:

```
grep -r MyDocumentClass . | grep _design.*map.js
```

There's a lot of extra value in migrating areas where you're familiar with the code context.

Ultimately, all progress is good.

CHAPTER

SEVENTYONE

1. RECORD ARCHITECTURE DECISIONS

Date: 2018-07-04

71.1 Status

Accepted

71.2 Context

We need to record the architectural decisions made on this project.

71.3 Decision

We will use Architecture Decision Records, as described by Michael Nygard.

71.4 Consequences

See Michael Nygard's article, linked above. For a lightweight ADR toolset, see Nat Pryce's adr-tools.

CHAPTER

SEVENTYTWO

2. KEEP STATIC UCR CONFIGURATIONS IN MEMORY

Date: 2018-07-04

72.1 Status

Accepted

72.2 Context

As part of the UCR framework configurations for data sources and reports may be stored in the database or as static files shipped with the code.

These static files can apply to many different domains and even different server environments.

When a data source or report configuraiton is requested the static configuration is read from disk and converted into the appropriate JsonObject class.

During some performance testing on ICDS it was noted that the process or readying the static configuration files from disk and converting them to the JsonObject classes was taking up significant time (14% of restore time for ICDS).

72.3 Decision

To improve the performance (primarily of restores) it was decided to maintain the list of configurations in memeory rather than re-read them from disk for each request.

In order to keep the memory footprint to a minimum only the static configurations are kept in memory and not the generated classes. This also serves to ensure that any modifications that may get made to the classes do not persist.

There are still some places that re-read the configurations from disk each time but these not called in places that require high performance. An example of this is the UCR pillow bootstrapping.

72.4 Consequences

Although this may raise the memory usage of the processes (after the configurations have been loaded) it should be noted that even in the current setup all the configurations are loaded on first request in order to generate the list of available data sources / reports. It may be that memory get's released at some point after the initial load.

In terms of actual memory footprint the figures are as follows:

Base memory: 285Mb Data sources: 60Mb Report configs: 35Mb

CHAPTER

SEVENTYTHREE

3. REMOVE WAREHOUSE DATABASE

Date: 2019-10-16

73.1 Status

Accepted

73.2 Context

The data warehouse was intended to house data for all CommCare HQ reports. The warehouse would replace Elasticsearch in almost all contexts that it is currently used. The migration began in 2017 with the Application Status report and the effort to move the report to the warehouse and ensure it is stable, performs well and provides the same features as the ES-backed reports was much higher than anticipated.

73.3 Decision

To reduce our infrastructure dependencies and focus our efforts on existing databases, we have decided to remove the warehouse and stop any efforts to iterate on it.

This decision is not because we believe that the warehouse is a worse implementation than Elasticsearch. This decision is because we believe that with our current priorities, we will not be able to spend the appropriate amount of time to make the warehouse a robust solution for generic reports in the near future. Because no current reports are backed by the warehouse, it is an important time to reconsider our approach and decide on what will be appropriate long term.

When there are more dedicated resources for generic reports, we believe that a warehouse-style approach should be considered when implementing.

73.4 Consequences

The warehouse was intended to reduce our usage of Elasticsearch and assist in an effort to remove many dependencies on our cluster. No matter the short term status of the warehouse, we need to improve our management of ES soon. This will include upgrading to more recent versions, re-indexing indexes to contain more shards, and supporting aliases that consist of multiple indexes.

The Application Status report also uniquely adds a lot of load on our CouchDB cluster. This load comes from the pillows for the report updating the user doc to contain the latest metadata. There will be a separate change that batches these updates to CouchDB into chunks.

CHAPTER

SEVENTYFOUR

DOCUMENTING

Documentation is awesome. You should write it. Here's how.

All the CommCare HQ docs are stored in a `docs/` folder in the root of the repo. To add a new doc, make an appropriately-named rst file in the `docs/` directory. For the doc to appear in the table of contents, add it to the `toctree` list in `index.rst`.

Sooner or later we'll probably want to organize the docs into sub-directories, that's fine, you can link to specific locations like so: `Installation <intro/install>`.

For a nice example set of documentation, check out Django's docs directory. This is used to build docs.djangoproject.com.

74.1 Index

1. *Sphinx* is used to build the documentation.
2. *Read the Docs* is used for hosting.
3. *Writing Documentation* - Some general tips for writing documentation
4. *reStructuredText* is used for markup.
5. *Editors* with RestructuredText support

74.2 Sphinx

Sphinx builds the documentation and extends the functionality of rst a bit for stuff like pointing to other files and modules.

To build a local copy of the docs (useful for testing changes), navigate to the `docs/` directory and run `make html`. Open `<path_to_commcare-hq>/docs/_build/html/index.html` in your browser and you should have access to the docs for your current version (I bookmarked it on my machine).

- Sphinx Docs
- Full index

74.3 Read the Docs

Dimagi maintains the hosted version of the documentation at readthedocs.io. For Dimagi employees, the credentials are maintained in our internal password manager under the "readthedocs" entry.

The configuration for *Read the Docs* lives in `.readthedocs.yml`, which calls the `docs/conf.py` script.

Due to problematic dependencies that need to be mocked, we cannot properly setup django apps until after `docs/conf.py` has been applied. We then must be aware that we are performing a docs build, at which point we can run *django.setup()* in `corehq/__init__.py`. We use an environment variable (*DOCS_BUILD*) to convey this information, which is set in the Admin UI of our readthedocs.io account.

74.3.1 Troubleshooting

The docs are built with every new merge to master. This build can fail completely, or "succeed" with errors. If you made a change that's not appearing, or if `autodoc` doesn't seem to be working properly, you should check the build.

On *Read the Docs*, in the bottom left, you should see "v: latest". Click to expand, then click "Builds". There you should see a build history (you don't need to log in for this). Click on the latest build. I find the "view raw" display to be more useful. That should show logs and any tracebacks.

Running `autodoc` or `automodule` requires that sphinx be able to load the code to import docstrings. This means that ~all of the source code's requirements to be installed, and the code cannot do complex stuff like database queries on module load. Build failures are likely caused by issues there.

Replicating the build environment

Read the Docs builds in an environment that doesn't have any support services, so turn those off. Next, make a new virtual environment with just the docs requirements. Finally, build the docs, which should surface any errors that'd appear on the build server.

```
$ cd commcare-hq/
$ mkvirtualenv --python=python3.9 hq-docs
$ pip install -r requirements/docs-requirements.txt
$ cd docs/
$ make html
```

74.4 Writing Documentation

For some great references, check out Jacob Kaplan-Moss's series Writing Great Documentation and this blog post by Steve Losh. Here are some takeaways:

- Use short sentences and paragraphs
- Break your documentation into sections to avoid text walls
- Avoid making assumptions about your reader's background knowledge
- Consider three types of documentation:
    1. Tutorials - quick introduction to the basics
    2. Topical Guides - comprehensive overview of the project; everything but the dirty details
    3. Reference Material - complete reference for the API

One aspect that Kaplan-Moss doesn't mention explicitly (other than advising us to "Omit fluff" in his Technical style piece) but is clear from both his documentation series and the Django documentation, is *what not to write*. It's an important aspect of the readability of any written work, but has other implications when it comes to technical writing.

Antoine de Saint Exupéry wrote, "… perfection is attained not when there is nothing more to add, but when there is nothing more to remove."

Keep things short and take stuff out where possible. It can help to get your point across, but, maybe more importantly with documentation, means there is less that needs to change when the codebase changes.

Think of it as an extension of the DRY principle.

74.5 reStructuredText

reStructuredText is a markup language that is commonly used for Python documentation. You can view the source of this document or any other to get an idea of how to do stuff (this document has hidden comments). Here are some useful links for more detail:

- rst quickreference
- Sphinx guide to rst
- reStructuredText full docs
- Referencing arbitrary locations and other documents

74.6 Editors

While you can use any text editor for editing RestructuredText documents, I find two particularly useful:

- PyCharm (or other JetBrains IDE, like IntelliJ), which has great syntax highlighting and linting.
- Sublime Text, which has a useful plugin for hard-wrapping lines called Sublime Wrap Plus. Hard-wrapped lines make documentation easy to read in a console, or editor that doesn't soft-wrap lines (i.e. most code editors).
- Vim has a command `gq` to reflow a block of text (`:help gq`). It uses the value of `textwidth` to wrap (`:setl tw=75`). Also check out `:help autoformat`. Syntastic has a rst linter. To make a line a header, just `yypVr=` (or whatever symbol you want).

74.6.1 Examples

Some basic examples adapted from 2 Scoops of Django:

Section Header

Sections are explained well here

emphasis (bold/strong)

italics

Simple link: http://commcarehq.org

Inline link: CommCare HQ

Fancier Link: CommCare HQ

1. An enumerated list item
2. Second item

- First bullet
- **Second bullet**
  - Indented Bullet
  - Note carriage return and indents

Literal code block:

```
def like():
    print("I like Ice Cream")

for i in range(10):
    like()
```

Python colored code block (requires pygments):

```
# You need to "pip install pygments" to make this work.

for i in range(10):
    like()
```

JavaScript colored code block:

```
console.log("Don't use alert()");
```

CHAPTER

SEVENTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W