Composite Documentation

Release 1.0.0

James Steele, Keith Hamilton

Sep 27, 2017

Contents

Composite is an end-to-end framework for managing web-scale socket communication between devices in a shared session, allowing devices to communicate with each other, keep each other up-to-date, and to create a multi-screen, shared experience for the devices.

Think of it like a post office and a mail carrier in one - anyone connected to it can send messages to it, and it will deliver messages to anyone for whom the messages are intended. Yes, like the post office, but much, much faster.

Composite's backend is a Spring-based Java application, that relies on CouchDB for managing temporary data storage, RabbitMQ for message brokering and StompJS for sending/receiving messages with connected clients.

Composite's frontend client is a Javascript-based application that relies on StompJS for communicating between client and server.

CHAPTER 1

What can Composite be used for?

At Wieden+Kennedy, we have built a few web-scale multiplayer games with Composite, using both the proximity-based pairing and direct-pairing features discussed here, but games are not the only thing it could be used for.

Composite is a very simple thing. It doesn't know what is happening on the user end, which is its beauty. It only knows how to send and receive messages to a group of connected devices.

Whatever you can think of that can benefit from real-time multi-device communication can use Composite to manage the communication stream.

CHAPTER 2

Quickstart

The easiest way to check out Composite in action is to download and run our Vagrant demo. To do this, you will want to have both of the following tools installed:

- Vagrant
- Virtual Box

```
$ wget https://compositeframework.io/static/demo/Vagrantfile
$ vagrant up && vagrant ssh
```

This will download the Vagrant box, spin it up, and ssh into it. Once inside, you'll just need to run one more command to get some docker containers spun up and mapped back to your localhost.

```
$ sudo composite-demo
```

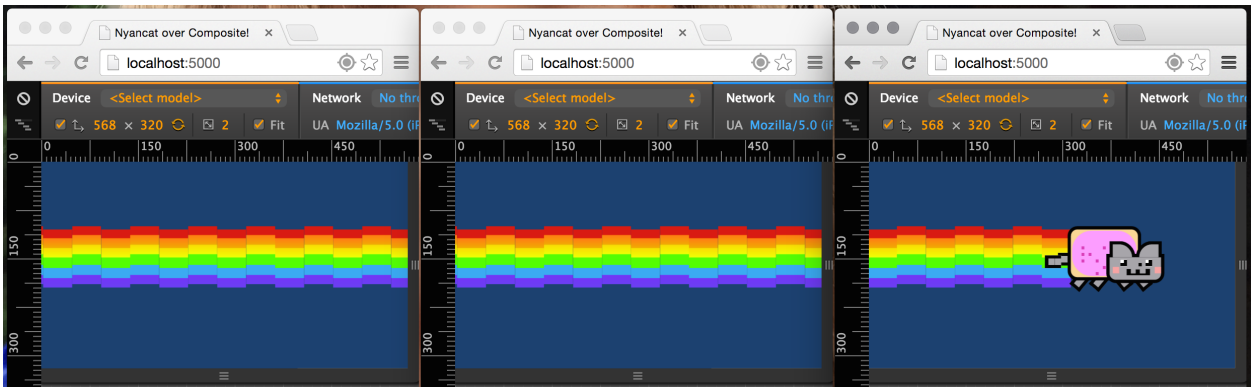
This command will import and spin up four Docker containers. While this is working, go grab a beverage of your choice. The imports can take a few minutes, especially if you're working on a slower connection.

For each container listed, port mapping goes from container to the Vagrant host to your local machine:

Container	Service	Mapped port
Composite	Tomcat	8080
Database	CouchDB	5984
Message Broker	RabbitMQ	61613
Web Client	NGINX	5000

Once these containers are up, you should be able to hit your system's address on port 5000 with a mobile device or a browser in emulation mode to get going. Once you hit localhost:5000 with three devices or browsers in emulation mode, the demo will start.

The demo is our favorite internet cat, Nyancat, streaming across the screens of the paired devices. Dragging Nyancat will move it up and down across the screens. Totally simple demo, but shows you that the device screens are linked via Composite messaging.


Nyancat over Composite!
localhost:5000
Device
<Select model>
Network
568 × 320
Fit
0
150
300
450

CHAPTER 3

Slowstart

Installation

Composite Server

Minimum Requirements

- Tomcat version 7.0.47+
- CouchDB (version 1.5.0 used in production)
- RabbitMQ (current)

Getting the Source

The latest Composite backend can be fetched from github:

```
$ git clone https://github.com/wieden-kennedy/composite
```

Setting up CouchDB

OS X

To set up CouchDB on OS X, we suggest you use Homebrew:

```
$ brew install couchdb
```

Ubuntu

We have created a setup script that works with Ubuntu 12.04 and 14.04. It's the easiest way to get CouchDB running:

```
$ git clone https://gist.github.com/8df5e450f34248ad1679.git couchdb_bootstrap
$ cd couchdb_bootstrap && /bin/bash run.bash
```

Setting up RabbitMQ

OS X

To set up RabbitMQ on OS X, we suggest you use Homebrew:

```
$ brew install rabbitmq
```

Ubuntu

We have created a setup script that works with Ubuntu 12.04 and 14.04. It's the easiest way to get RabbitMQ running:

```
$ git clone https://gist.github.com/keithhamilton/f2e20127f52618748266 rabbit_
↪bootstrap
$ /bin/bash rabbit_bootstrap/run.sh
$ rabbitmq-server start
```

Building Composite

The project is built using Maven, and is packaged as a war file. When building, the `-Denv` flag is used to indicate which environment should be used to build the project.

Profiles

The maven POM is currently configured for building in the following environments:

- local
- dev
- test (i.e., staging)
- prod
- unit

The desired build environment is defined using the -Denv flag, as follows:

`` `mvn -Denv=local clean package` ``

The above will build the war using the properties files located in `` `src/main/classpath/local` ``.

When building for the `` `unit` `` profile (for unit testing), the properties files located in `` `src/main/test/resources` `` will be used.

Composite Client

Getting the Source

The latest Composite frontend can be fetched from github:

```
$ git clone https://github.com/wieden-kennedy/composite-client
```

Using the Client

To use the client, first grab the `/build/min/composite.min.js` file from the client source, then add the following into the head of your HTML document:

```
<script src="/path/to/composite.min.js"></script>
```

Building with Gulp

To build the client from source, you will need to first have the following installed:

- Nodejs + NPM
- Git
- Gulp

Once these are installed, building the client is easy:

```
$ git clone https://github.com/wieden-kennedy/composite-client
$ cd composite-client
$ npm install
```

This will build the human-readable `build/dev/composite.js` file and the production-ready `build/min/composite.min.js` file.

You can also compile on save with `gulp watch`

Autobuild (OS X + Ubuntu)

We've created an autobuild script you can use to get the client and build it, which can be run thusly:

```
$ wget https://raw.github.com/wieden-kennedy/composite-client/master/autobuild.sh
$ /bin/bash autobuild.sh
```

Autobuilding is supported on Ubuntu and OS X, but if you are running a different Debian flavor, you can attepmt to force the autobuild to run by adding the `--force` flag:

```
$ wget https://raw.github.com/wieden-kennedy/composite-client/master/autobuild.sh
$ /bin/bash autobuild.sh --force
```

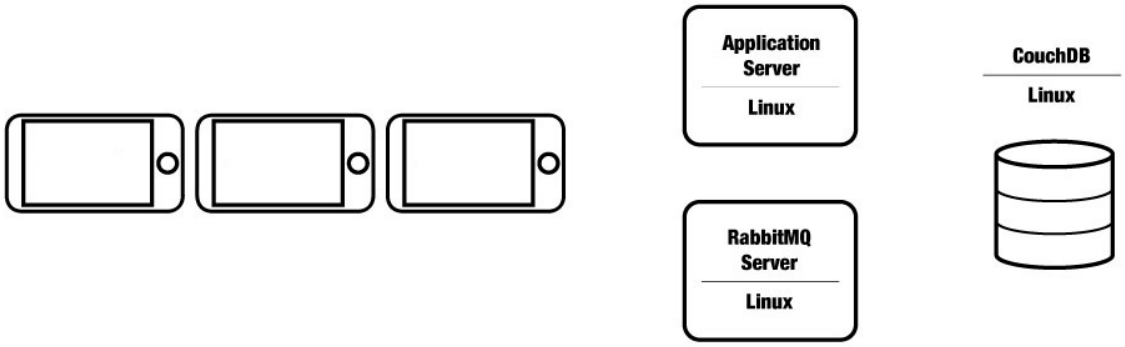
Application Landscape

This section describes the general application landscape and event chain for a Composite application.

Major Components

Four major components are part of the Composite application landscape:

- a group of mobile phones
- a Tomcat server
- a RabbitMQ message broker server
- a CouchDB NoSQL database server



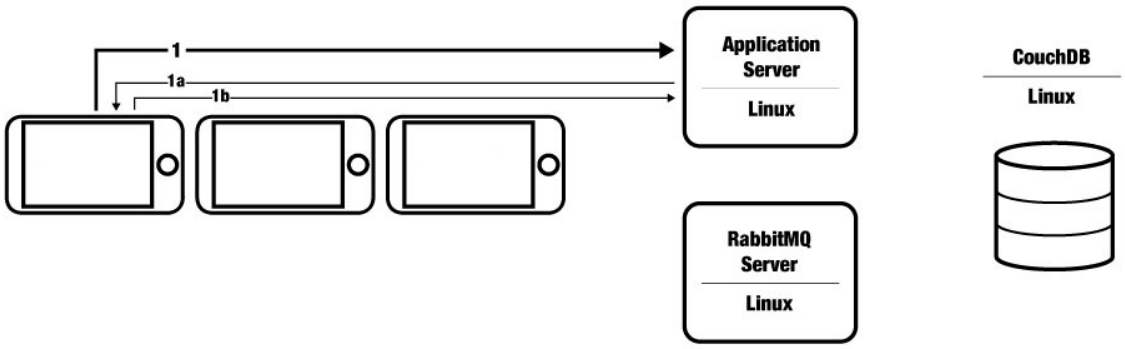
Ways to Join a Session

Composite supports two main ways of joining into a session:

- **Proximity-based**: devices are paired up with other devices in their proximity to create a shared session. In this mode, the session can have many devices, but is by default limited to eight.
- **Direct pairing**: devices are paired with the next available device for a shared session. In this mode, typically the session will be limited to two devices.

The key difference between proximity-based pairing and direct pairing is that in the former location information is used to prioritize which session a new device is paired with, whereas with latter, the first available open session is used for pairing. This difference occurs in step 2 below.

1. Registering with the server

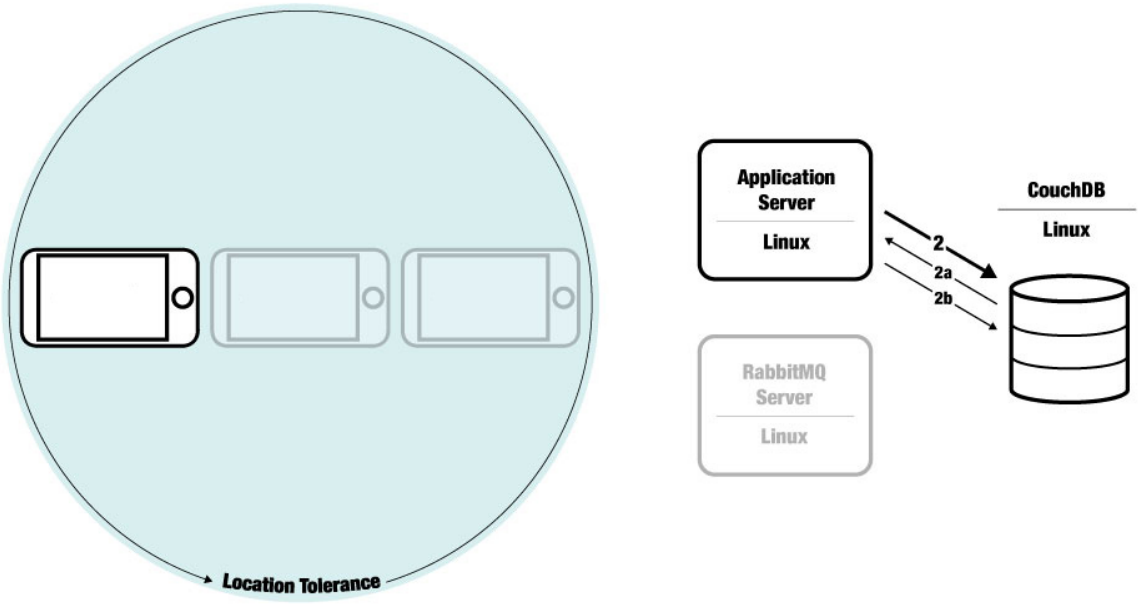


Three discrete sub-steps are involved here, denoted 1, 1a, and 1b respectively.

- Firstly, on mobile phone A, a Composite-driven web application is loaded in the browser. Once the user has approved the use of location services, the phone will try to make contact with the application server via Sockjs protocol (1).
- Next, the application server has acknowledged the device, and sent it a subscription id for sending and receiving socket messages (1a).
- Lastly, the phone again contacts the application server to say "I want to join a session." (1b)

---

2. The application server checks for existing sessions



The application server takes the phone's geolocation and uses a Haversine function to determine whether any existing, open sessions are within a tolerated radius of the new phone. If an open session is found, the new phone is added to it, otherwise, a new session is created. The first and second steps in this process differs depending on the type of pairing being performed:

For proximity-based pairing:

- First, it queries the CouchDB database for all unlocked (open) sessions (2).
- Next, it receives a list of open sessions back from the database, then either adds the new phone to an existing open session, if one is found within the tolerated range, or creates a new one, if no open sessions are found that are close enough (2a).

For direct pairing:

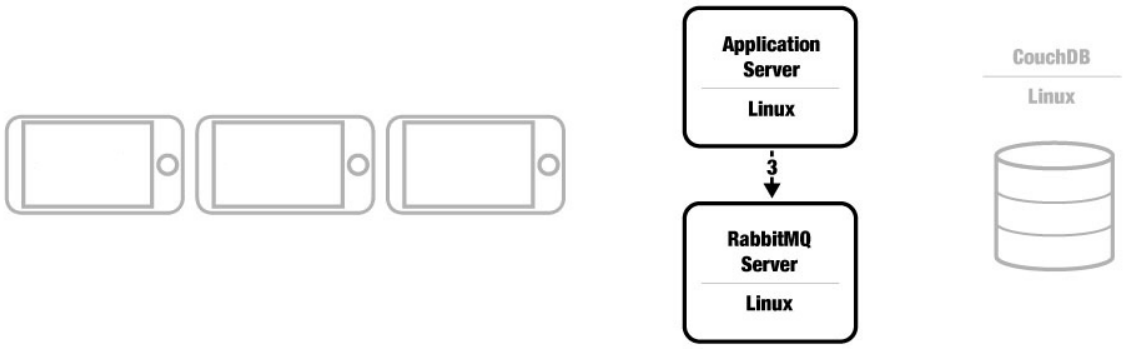
- First, the application server queries the database for one open session (it doesn't care which) (2).
- Second, if it finds an open session, it adds the new phone to that session. If it does not find an open session, a new session is created (2a).

In either pairing scenario, lastly the application server will:

- Update the CouchDB database either with the new session, or the updated session that was found (2b).

---

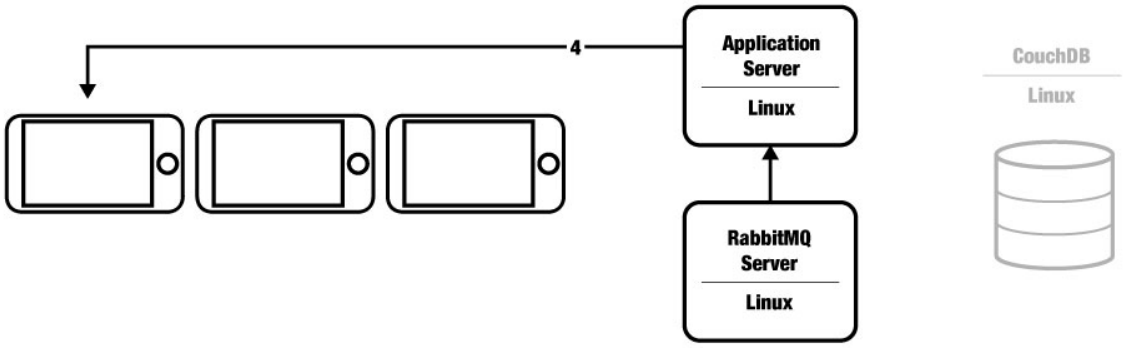
3. Pushing the message to the broker



With the phone added to a session, a socket message is pushed to the broker for delivery to the correct subscriber queue, which is the individual phone's queue. Other devices in the same session are notified via a topic channel that a new device has joined (see section 5 below, **Update lifecycle**)

---

4. Device is notified it is part of a session



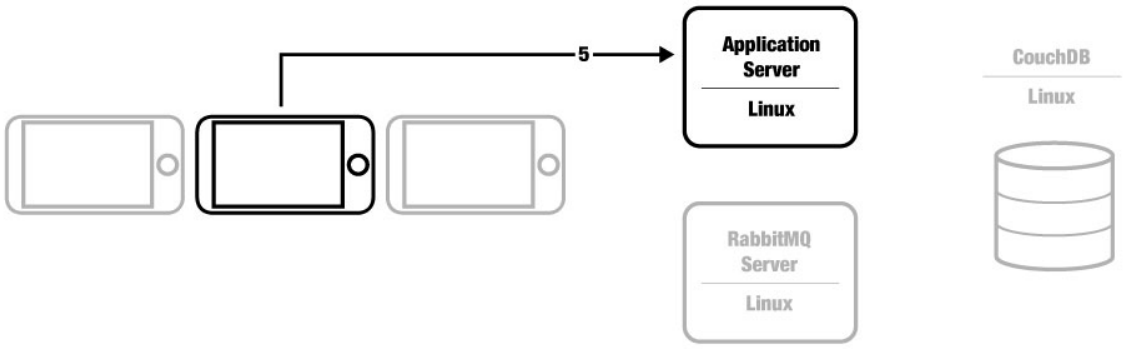
Once the device has been recorded in a session and subscribed to that session's topic, the messaging broker proxies a message through the Application Server, to the device's queue letting the device know it's part of a session.

This process is repeated each time a new phone joins a session.

Update Lifecycle

Once devices are in a session, the update cycle begins, where the devices send an update message to the application server at a rate of 20 messages per second. These updates are then pushed back out on the session's topic channel, so that all of the devices in the session are updated with instructions from each other.

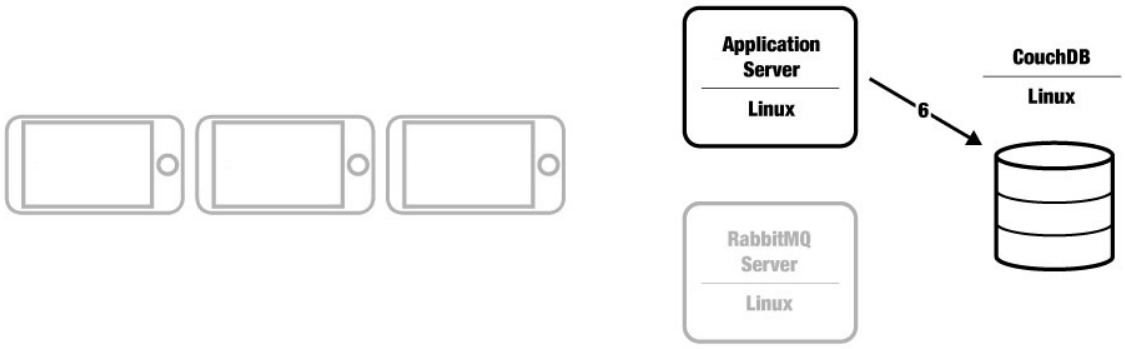
5. An update is sent



As an example, let's consider a situation in which a ball is moving across the screens in session. It would be helpful for each device to know if the ball was about to enter its screen, so on the client side, an update message containing the ball position should be fired periodically.

Here the second device is sending an update to the server.

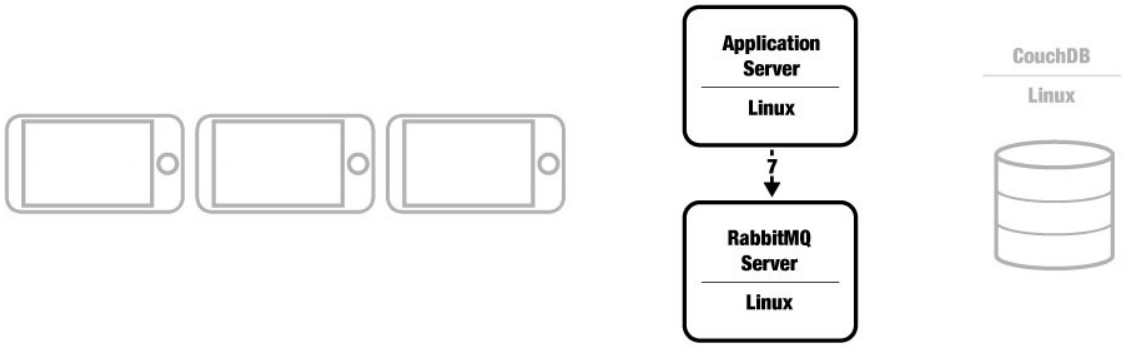
6. The session record is updated



Before sending the update message on to the broker, the application server updates the database record for the device's session. This helps distinguish stale sessions from active ones by updating a lastUpdated timestamp.

---

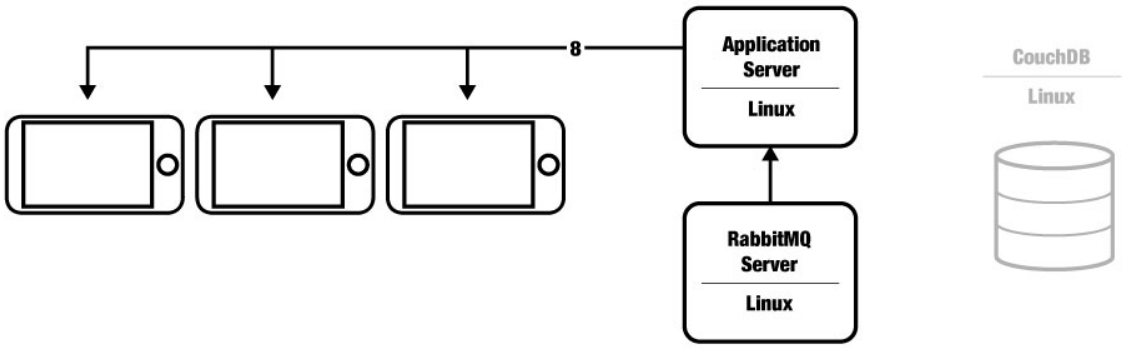
7. Pushing the message to the broker



The application then forwards the update message on to the message broker for queueing.

---

8. Broadcasting the message



The update message is sent out to the session topic, so that all session devices are updated with the new information. Now they all know where the ball is relative to their screens, and if they need to do anything.

Configuration

Composite is set up to be built for local, development, test, and production environments by means of configuration properties. Each environment has its own set of configuration/properties files, and they can be found in one of two locations:

- `src/main/resources/`.
- `pom.xml`

In each of the properties files, defaults have been left in place, and context-specific values have been omitted.

application.properties

This is where global application settings are stored, such as maximium distance between two devices that should be considered to be in the same session. Properties that can be edited in this file are:

Global Composite Properties

Property	Set in	Description
staleSessionMinutes	application.properties	the number of minutes after which a session should be considered stale and eligible for deletion
minDistanceThresholdBetweenDevices	application.properties	the preferred maximum distance between devices that should be considered in the same session. When a new device contacts the server, this distance will be checked to find sessions within this range that the device may join.
maxDistanceThresholdBetweenDevices	application.properties	the maximum distance between devices that should be considered to be in the same session. This distance will be used to find a session when a device connects, and no sessions are found within the preferred maximum distance range are found.
regex.applicationId	application.properties	a regex pattern containing all apps running on a Composite instance, e.g., regex.applicationId=(appOne|appTwo)
deleteUnhealthyDevices	application.properties	whether to delete unhealthy devices from the database when the periodic cleanup task runs.

Application-Specific Properties

Property	Set in	Description
appId.applicationId	application.properties	the name of the application
appId.maxDevicesPerSession	application.properties	the maximum number of devices that can exist in a session
appId.roomNames	application.properties	comma-separated list of names identifying rooms assigned to sessions. Room names do not have any bearing on session activity and are used to distinguish between multiple sessions in close proximity.

couchdb.properties

This is where global settings for the CouchDB connection will be defined, such as host, port, and maximum number of connections. Properties that can be edited in this file are:

Property	Set in	Description
couchdb.host	pom.xml	the host IP or DNS name for the CouchDB server
couchdb.port	application.properties	the port CouchDB is served over
couchdb.username	pom.xml	the username used to connect to a CouchDB database that is password protected. Can be blank if not needed.
couchdb.password	pom.xml	the password used to conenct to a CouchDB database that is password protected. Can be blank if not needed.
couchdb.max.connections	application.properties	the maximum number of connections to allow in the connection pool.
couchdb.createdb.if-not-exist	application.properties	whether to create databases if they do not already exist when the application first connects to the CouchDB host.
couchdb.protocol	application.properties	the protocol to use when connecting to the CouchDB database.
couchdb.sessions.database.name	application.properties	the name of the database where session information will be stored.

rabbitmq.properties

This is where global settings for the RabbitMQ broker can be configured, such as host, port, and login. Properties that can be edited in this file are:

Property	Set in	Description
rabbitmq.host	pom.xml	the host IP or DNS name for the RabbitMQ server.
rabbitmq.port	rabbitmq.properties	the port RabbitMQ is served over
rabbitmq.clientLogin	pom.xml	the login name used for the RabbitMQ broker. Can be left blank if not needed.
rabbitmq.clientPasscode	pom.xml	the password used for the RabbitMQ broker. Can be left blank if not needed.
rabbitmq.systemLogin	pom.xml	the login name used for the RabbitMQ host system.Can be left blank if not needed.
rabbitmq.systemPasscode	pom.xml	the password used for the RabbitMQ host system. Can be left blank if not needed.
rabbitmq.systemHeartbeatSendInterval	rabbitmq.properties	interval, in milliseconds, on which to send a heartbeat
rabbitmq.systemHeartbeatReceiveInterval	rabbitmq.properties	interval, in milliseconds, on which to receive heartbeats
rabbitmq.heartbeatTime	rabbitmq.properties	heartbeat interval, in milliseconds, for the Stomp client to send on
rabbitmq.inboundChannelCorePoolSize	rabbitmq.properties	initial number of executor threads for inbound message processing
rabbitmq.outboundChannelCorePoolSize	rabbitmq.properties	initial number of executor threads for outbound message processing
rabbitmq.brokerChannelCorePoolSize	rabbitmq.properties	initial number of executor threads for broker message processing.
rabbitmq.sendTimeLimit	rabbitmq.properties	the message timeout value in milliseconds
rabbitmq.sendBufferSizeLimit	rabbitmq.properties	the maximum number of bytes to buffer when sending messages in KB.
rabbitmq.messageSizeLimit	rabbitmq.properties	the maximum message size in KB

Test Properties

The same three properties files are available for unit tests as well, under `src/test/resources`. Unlike the main run configuration files, however, the test files have all of their values set in the actual files, not in the pom.

By default, the CouchDB and RabbitMQ hosts are listed as `localhost`. Update these accordingly, if needed.

Appendices

Composite Client Command Reference

This is a companion guide for using the Composite Client, found here.

Methods

Constructor

The global `Composite` constructor. Takes no config options and is used for instantiation purposes.

```
var app = new Composite();
```

---

connect:function (string:url)

Connects to the Composite service. The handshake is over HTTP, so a good example would be:

```
app.connect('http://localhost:8081/composite');
```

---

on:function (string:event, function:callback)

Registers an event handler for a given event.

```
app.on('app_start', function(){ start_your_app(); });
```

The following are valid events (`Automatic` denotes that the event is triggered internally, whilst `Manual` requires a client-triggered action):

Property	Description
`init`	Fires after a successful connect and a valid UUID has been assigned. Automatic.
`synced`	Fires once the app has determined the average latency between it and the service. Automatic.
`app_start`	Fires after the `host` device in the same session queries the `start` endpoint. Manual.
`app_end`	Fires after the `host` device in the same session queries the `end` endpoint. Manual.
`session_joined`	Fires after a successful `join` query and only to the client that queried it. Manual.
`device_update`	Fires after a new device has joined the session. Manual
`device_disconnect`	Fires if a device in the session suddenly disconnects. Manual.
`data`	Fires when a client sends a payload to the `data` endpoint. Manual.
`update`	Fires when a client sends a payload to the `update` endpoint. Manual.

---

off:function (string:event, function:callback)

Removes an event handler for a given event.

```
app.off('app_start', function(){ start_your_app(); }); // Must pass the function you␣
↪called earlier
```

---

syncTime:function

Sends a message to the service with the current time in order to determine latency. This also happens automatically, but is exposed in case your application needs to check more frequently.

```
app.syncTime()
```

---

join:function (object)

Sends a join request to the service to get a session, requires that the client has already set its location (lat/lon) under the `location` array (see location).

The Object passed can have the following parameters: - type: String 'enter' or 'exit'. Defaults to 'exit'. - geo: Array with two elements corresponding to a devices longitude/latitude. Defaults to the local lon/lat if not present.

```
app.join({type: 'exit', geo: [0.1234123, 1.123123]});
```

---

sendData:function (object)

Sends a message to all clients in the same session with a data payload. The passed object is the payload you wish to send to all clients. Requires that clients have ''join''ed successfully prior to sending.

```
app.sendData({ ballPosition: [103, 234], ballSpeed: 23, activeDevice: 2 });
```

---

sendUpdate:function (object)

Sends a message to all clients in the same session with a data payload. The passed object is the payload you wish to send to all clients. Requires that clients have ''join''ed successfully prior to sending.

```
app.sendUpdate({ ballPosition: [103, 234], ballSpeed: 23, activeDevice: 2 });
```

---

startApp:function

Triggers the `app_start` event in all other clients. Must be the "host" client to trigger (see `host` below).

```
app.startApp()
```

---

endApp:function

Triggers the `app_end` event in all the other clients. Must be the "host" client to trigger (see `host` below).

```
app.endApp();
```

---

disconnect:function

Disconnects cleanly from the Composite service. Other clients in the same session will be notified of the disconnect.

```
app.disconnect()
```

Properties

connected

Type: `boolean`

Container indicating if the client is connected to the composite service.

```
app.connected; // true if connected, false if not
```

---

uuid

Type: `string`

The UUID of the device given from the service, can be used as a way to find the device(s) order.

```
app.uuid; // "7040550a-3834-4974-a19c-c7d39749a7e5"
```

---

timeDifference

Type: `number`

The median time difference between the client's Date.now and the services Date.now.

```
app.timeDifference; // 1121
```

---

latency

Type: `number (milliseconds)`

The average time it takes to send and receive a message through composite. Updated periodically throughout the application.

```
app.latency; // 10
```

---

host

Type: `boolean`

If the device is the `host` device. This is determined by order `join``ed, and the first device to join is given ``host` privileges. This is updated during the `device_update` event as it's possible for the `host` to drop connection. `host``s can trigger the ``app_start` and `app_end` events.

```
app.host; // true
```

---

location

Type: `array[float]`

The container for the devices geographic position in the form of `[{latitude}, {longitude}]`. Location is not captured automatically, and must be implemented manually, and is used for session management.

Example:

```
navigator.geolocation.getCurrentPosition(function(position) {
    // Setting the position
    app.location = [position.coords.latitude, position.coords.longitude];
});

app.location; // [45.523452, -122.67620699999999]
```

---

session

Type: `string`

The session the device is currently a part of. This is set automatically after successfully `join``ing, and is required when broadcasting ``update``s and ``data`.

```
app.session; // "7040550a-3834-4974-a19c-c7d39749a7e5"
```

---

active

Type: `boolean` If the app is currently in the `start` state. This happens automatically after the `app_start` event and is set to false after the `app_end` event.

```
app.active; // true
```

Composite (Backend) Messaging Reference

Endpoints

/init

Direct message handler for an init message sent by a device. The init message will follow directly after the device has made a successful socket connection to the server, and indicates that the device would like to start interacting with Composite.

Parameters	Description
principal	The device principal that sent the request

Returns to: `/queue/device`

/join

Direct message handler that receives a join message from a device. When a device joins, a session is attempted to be found within a tolerated geo-proximity. if one is found, the device is added to it and returned, if not, a new session is created.

Params

Parameters	Description
j	Stringified JoinMessage sent from the device seeking to join a session

Returns to: `/queue/device`

/sync

Direct message handler for assisting connected devices in calculating the latency between when messages are sent by the server and when they are received by the client. Each client should hit this endpoint a number of times just after the initial connect response is received, and will calculate an average latency time based off of the server responses.

Params

Parameters	Description
s	Stringified SyncMessage sent by client device

Returns to: `/queue/device`

/{id}

Multiplex handler inbound messages from session devices that are sent to the session topic channel. Uses the following handlers to determine what information to broadcast back across the session topic:

Handler	Description
update	Broadcasts an update response back to the session when a device principal sends an update
data	Broadcasts a data response back to the session when a device principal sends a data message
start	Broadcasts a start event back to the session when a device principal initiates a start event
stop	Broadcasts a stop event back to the session when a device principal initiates a stop event
devices	Broadcasts a list of devices found in a session back to the session topic

Parameters	Description
principal	Device principle sending the message
id	UUID of the session to which the inbound message need be returned
obj	Map<String, Object> message data for the endpoint. Contains the type of message that corresponds to one of the above endpoints as well as arbitrary String:Object pairs that contain the main message body

Returns to: `/queue/device`

/ping

Message handler that receives a ping from a connected client device, and in turn adds the device to a list of "healthy" session devices, thereby preventing it from being automatically deleted from the session. If a device fails to ping the server within a specified timeframe, it will be marked as unhealthy, and subsequently deleted.

Parameters	Description
principal	The device principal that sent the request

Returns to: None. Logs the device in a registry that determines which devices to boot if they do not ping regularly.

/disconnect

Handles disconnect messages sent by a client device by removing the device from its associated session. if there are still devices in its session, they are notified of the disconnect, otherwise, the session is removed.

Parameters	Description
principal	The device principal that sent the request

Returns to: `/topic/{id}` where id is the session id to which the device belongs.

Composite Model Reference

Composite keeps track of two objects:

- Sessions
- Devices

The relationship between sessions and devices is one-to-many, and devices are stored in an array in a given session object.

Session Model

All types given are Java types.

Member	Type	Description
_id	String	CouchDB unique id for the session record.
_rev	String	CouchDB revision version for the session record.
devices	ArrayList<Device>	Array of devices in the session.
geoLocation	float[]	Array of lat/lon coordinates for the session location.
inserted	long	Unix timestamp of when the session was created.
locked	boolean	Indicates if the session is unlocked or not. If locked, no more devices join the session until it is unlocked.
room	string	The name of the "room" for the session. Can be used on the client end to disambiguate multiple sessions in close proximity.
sessionStarted	long	Unix timestamp for when a session's activity begun. For example, when a shared game begins.
sessionEnded	long	Unix timestamp for when a session's activity ended.
updated	long	Unix timestamp of the last update to the session object.
uuid	UUID	Unique identifier for the session.

Device Model

All types given are Java types.

Member	Type	Description
uuid	UUID	Unique identifier for the device.
width	int	Device screen width.
height	int	Device screen height.
performance	int	Figure representing relative performance of device. This property is used on the client-side to manage screen redrawing.
instructions	int	Figure representing client-side instructions that tell the device what to do within the context of the session and other devices.

Sample Record (CouchDB)

Records are saved as JSON in the CouchDB instance(s). Below is a sample taken from the Composite Demo app.

```
{
   "_id": "39d4e19c25964bb3a6c3b2f806e8fa33",
   "_rev": "3-32d6973d965f40261053b2cfb523212f",
   "devices": [
       {
           "uuid": "5c6f9f63-dc2f-4f4c-81dc-b722eb4b0a1f",
           "width": 375,
           "height": 667,
           "performance": 0,
           "instructions": 0
       },
       {
           "uuid": "7c528f92-9223-4802-bede-b8b01e8e5aa6",
           "width": 375,
           "height": 667,
           "performance": 0,
           "instructions": 0
       },
```

```
        {
            "uuid": "b969d6e3-94f8-4c1a-a8e9-efa7b4b58962",
            "width": 375,
            "height": 667,
            "performance": 0,
            "instructions": 0
        }
    ],
    "geoLocation": [
        45.524426,
        -122.68396
    ],
    "inserted": 1412097229725,
    "locked": false,
    "room": "default_room",
    "sessionStarted": 1412097232288,
    "sessionEnded": 0,
    "updated": 1412097232288,
    "uuid": "28171d9c-05e4-422b-9dd9-9e8b8ff84609"
}
```

Composite CouchDB View Reference

Composite instances use the below views to retrieve documents from the CouchDB database via the SessionRepository class.

application-id

Used to retrieve sessions by application id. Used when multiple Composite-based applications are working from the same server or set of resources.

```
function(doc) {
    if(doc.applicationId && !doc.locked){
        emit(doc.applicationId, doc)
    }
}
```

locked-sessions

Used to retrieve a list of sessions, keyed by their lock status.

```
function(doc){
    emit(doc.locked, doc)
}
```

session-by-device

Used to retrieve a set of sessions keyed by device UUID.

```
function(doc) {
    if(doc.devices){
        for(var i in doc.devices){
```

```
            emit(doc.devices[i].uuid, doc)
        }
    }
}
```

session-devices

Used to retrieve a set of devices in a session.

```
function(doc) {
    if(doc.devices){
        emit(doc.uuid, doc.devices)
    }
}
```

session-by-timestamp

Used to retrieve a session by its inserted timestamp.

```
function(doc) {
    if(doc.inserted){
        emit(doc.inserted, doc)
    }
}
```

uuid

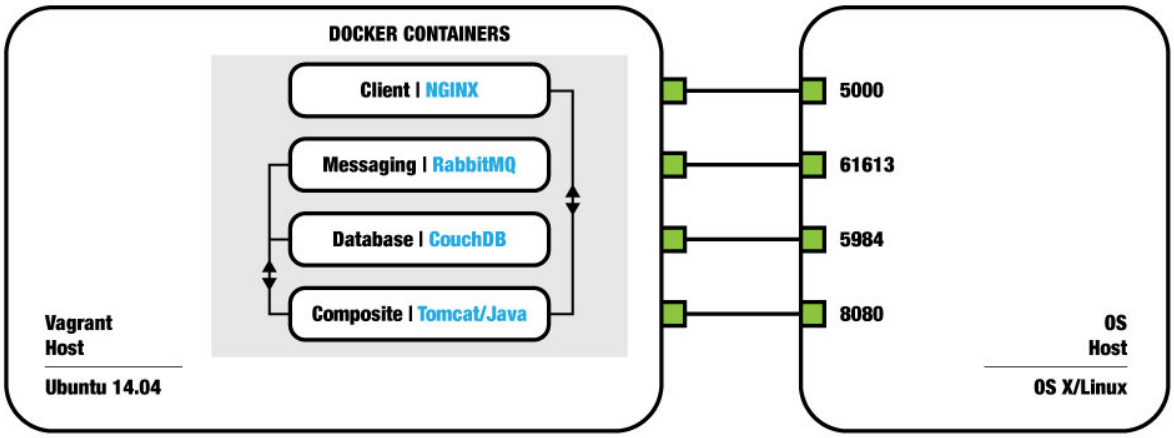
Used to retrieve a session by its UUID.

```
function(doc) {
    if(doc.uuid){
        emit(doc.uuid, doc)
    }
}
```

Composite Demo Reference

Architecture

The basic concept behind the Composite demo's architecture is that of Docker containers on a Vagrant host. Nothing new.



Each Docker container maps the main service port/s back to the Vagrant host, which in turns forwards those ports up to the OS host, either a Linux or Mac OS X host. The service port mapping is as follows:

Service	Container < - > Vagrant	Vagrant < - > OS
Client (NGINX)	5000	5000
RabbitMQ	61613, 5672	61613
CouchDB	5984	5984
Composite	8080	8080

Base Container

All above containers are built off of a container called `local/base`. This container is simply an Ubuntu 14.04 container with some basic tools installed:

- vim
- curl
- wget
- openssh-server
- pip
- puppet
- fabric

Importing Composite Containers

If you followed the quickstart guide, after you got the Vagrant host up and running you ran:

```
$ sudo composite-demo
```

The underlying mechanics of this command, if you haven't already peeked, is simply:

1. Clone the composite-demo repository.
2. Import the Docker containers from S3.
3. Run the Docker containers, linking appropriately, as indicated in the diagram above.

The underlying mechanism for performing steps two and three is a Fabric file at the root of the composite-demo repository. It contains the following method for importing the Composite Docker containers:

```
def import_container(container_name=None):
    import_containers = CONTAINERS

    if container_name:
        import_containers = [x for x in import_containers if x['name'] == container_
↪name]

    for c in import_containers:
        container = Container(env.environment, c)
        print(_white("==> Importing %s container from S3" % container.name))
        local('sudo curl %s | sudo docker import - %s/%s' % (container.s3_path,
                                                             env.environment.lower(),
                                                             container.name))
```

Taking this into account, should you want to re-import any or all containers, you can navigate to the composite-demo directory at `/home/vagrant/composite-demo` and run one or more of the following commands:

```
# import client, rabbitmq, couchdb, or composite
$ fab import_container:[client|rabbitmq|couchdb|composite]

# import client container
$ fab import_container:client

# import all containers
$ fab import_container
```

Note that if you ran the quickstart, and imported the Composite containers, the base container will not be built.

Building Composite Containers

By running `sudo composite-demo` when bringing up the Vagrant host, you may have run into a prompt to build the containers locally, in the event that your internet connection was considered too slow to make effective use of downloading/importing all containers. In this event, the containers would have been individually built using another Fabric method:

```
def build(container_name=None):
    killall_containers()

    build_containers = CONTAINERS
    base_container = local("sudo docker images | grep base | awk '{print $1}'",␣
↪capture=True)

    # if container_name was passed, reduce build_containers to just that container
    if container_name:
        build_containers = [x for x in build_containers if x['name'] == container_
↪name]
        # if the base container doesn't yet exist, build it
        if container_name != 'base' and not base_container == 'local/base':
            build('base')
```

```
    # build the service and composite containers
    for c in build_containers:
        build_container(c, env.environment)
```

Similar to `fab import`, containers can be built all at once or individually:

```
# build one container
$ fab build:[base|client|rabbitmq|couchdb|composite]

# build client container
$ fab build:client

# build all containers
$ fab build
```

Fabfile Reference

The `import_container` and `build` methods are the most commonly used. Below is a reference of the methods available to you from within the Fabric file included with the composite-demo[1].

Methods

import_container(container_name=None)

Imports one or more containers from S3 by name.

```
# import client
$ fab import_container:client

# import rabbitmq
$ fab import_container:rabbitmq

# import couchdb
$ fab import_container:couchdb

# import composite
$ fab import_container:composite

# import all containers
$ fab import_container
```

build(container_name=None)

Builds one or more containers using the Dockerfiles included in the demo repository.

```
# build client
$ fab build:client

# build rabbitmq
```

[1] Technically there are more methods available in the Fabric file, but the remaining few methods are used internally, for the most part. To see the rest, check out the fabfile.

```
$ fab build:rabbitmq

# build couchdb
$ fab build:couchdb

# build composite
$ fab build:composite

# build all containers
$ fab build
```

run()

Runs the Composite demo containers, linking the service containers to the web application container

```
# run all containers
$ fab run
```

killall_containers()

Kills and removes all running or exited Docker processes.

```
# kill and remove all Docker processes
$ fab killall_containers
```

delete_image(container_name=None)

Deletes one or more container images on the Vagrant host machine.

```
# delete client
$ fab delete_image:client

# delete rabbitmq
$ fab delete_image:rabbitmq

# delete couchdb
$ fab delete_image:couchdb

# delete composite
$ fab delete_image:composite

# delete all Docker images
$ fab delete_image
```

CHAPTER 4

License

This repository and its code are licensed under the BSD 3-Clause license, which can be found here.