Lily

Release v1.0.0-alpha.2

January 05, 2017

Contents

CHAPTER 1

Installation

Getting Lily up and running is straightforward using the included docker-compose setup.

1.1 Prerequisites

- you have docker and docker-compose installed
- you have git installed
- you have nodejs, npm and gulp installed

1.2 Docker environment

1. Checkout the Lily project and install gulp dependencies.

```
cd ~/projects/
git clone git@github.com:HelloLily/hellolily.git
cd hellolily
npm install
gulp build
```

2. Build the docker image. This takes a while the first time.

```
docker-compose build
```

Note: This command needs to run every time the Dockerfile, requirements or patches are adjusted. Good practice would be to run it every time the git repo is updated. If nothing changed, the command would complete almost instantly.

3. Do a first time migration of the models.

```
docker-compose run web python manage.py migrate
```

4. Create a search index for ElasticSearch.

```
docker-compose run web python manage.py index
```

5. Populate the database with some testdata.

```
docker-compose run web python manage.py testdata
```

6. Run the Django development server along with dependent containers.

```
docker-compose run --service-ports web
```

Open http://localhost:8003 in your browser to see Lily. You can login using user **superuser1@lily.com** and **admin** as password. Congratulations, you just completed the basic Lily installation!

1.3 Email integration

Lily uses Google email accounts(Gmail) and it's Gmail API to send email messages with. Customer emails are stored locally and indexed using ElasticSearch. This allows Lily users to search and find their customer's data very fast, using extended search queries.

In order to enable email in Lily, you first need to enable the Gmail API and create an OAuth 2.0 client ID for a web application. This sounds harder than it is; just proceed as following:

- Login to the Google APIs website
- From the *Overview* screen, fill in *Gmail API* in the Search bar and select it from the search results
- Click on the *Enable* button
- Now you need to create Credentials. Click on the *Go to Credentials* button
- Check if the following options are selected in the credentials form: * Which API are you using? *Gmail API* * Where will you be calling the API from? *Web server* * What data will you be accessing? *User data*
- Click on the *What credentials do i need?* button
- Give the credentials a name, e.g. *Lily*
- In *Authorized redirect URIs* fill in your development url, e.g. http://localhost:8003/messaging/email/callback/
- The restriction options can be kept empty. Just click on the *Create client ID* button
- You can skip step 3. Just click on the *Done* button at the bottom of the form
- The current screen should be the Credentials overview; click on *Lily*

The credentials are needed for our Lily GMail setup. Let's add them to the appropriate file. Open the environment settings file with an editor:

```
vim /path/to/lily/.env
```

Add the following settings and fill *Client ID* and *Client secret* as GA_CLIENT_ID and GA_CLIENT_SECRET:

```
GA_CLIENT_ID=your_client_id
GA_CLIENT_SECRET=your_client_secret
GMAIL_CALLBACK_URL=http://localhost:8003/messaging/email/callback/
```

That's it! Lily is now able to manage Gmail accounts. To test if Gmail integration works, go back to your running Lily instance and visit http://localhost:8003/#/preferences/emailaccounts

- Select *add an email account*

You should now be redirected to the Google OAuth login screen. Allow Lily to access your Gmail account. After that, fill in *From name* and *Label* and press the *Save* button. Your email account will now get synced to Lily.

CHAPTER 2

Reference

Available reference documentation:

2.1 Manual installation

Installing Lily with docker is recommended for most installations, but there may be cases where you find Docker not suitable. This installation describes how to install Lily without Docker.

2.1.1 Prerequisites

- you have python 2.7 and virtualenv installed
- you have git installed
- you have nodejs, npm and gulp installed
- you have postgresql up and running
- you have elasticsearch up and running
- you have rabbitmq up and running

2.1.2 Django environment

1. Make a virtualenv, checkout the Lily project and install gulp dependencies.

```
mkdir -p ~/projects/hellolily-env
cd ~/projects/hellolily-env
virtualenv2 .
. ./bin/activate
git clone git@github.com:HelloLily/hellolily.git
cd hellolily
npm install
gulp build
```

2. Install all related Python packages.

```
pip install -r requirements.txt
pip install -r requirements-dev.txt
```

3. Setup a postgresql database for Lily.

```
sudo su postgres
cd ~
createdb hellolily
createuser hellolily -s
echo "ALTER USER hellolily WITH ENCRYPTED PASSWORD 'c2cg63&(e';" | psql
echo "create extension unaccent;" | psql
exit
```

4. Override settings using your own environment settings.

```
vim ~/projects/hellolily-env/hellolily/.env
```

```
DEBUG=1
MULTI_TENANT=1

DEFAULT_FROM_EMAIL=info@mydomain.org
SERVER_EMAIL=info@mydomain.org
EMAIL_USE_TLS=1
EMAIL_HOST=smtp.gmail.com
EMAIL_HOST_USER=info@mydomain.org
EMAIL_HOST_PASSWORD=
EMAIL_PORT=587

ADMINS=(('Yourname', 'your@email.com'),)

DATAPROVIDER_API_KEY=
# Make SECRET_KEY unique for your own site!
SECRET_KEY=abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmn

DATABASE_URL=postgres://hellolily:c2cg63&(e@localhost/hellolily

AWS_ACCESS_KEY_ID=aws_key
AWS_SECRET_ACCESS_KEY=aws_secret_access_key
AWS_STORAGE_BUCKET_NAME=aws_bucket_name

REDISTOGO_URL=redis://localhost:6379/0
CELERY_SEND_TASK_ERROR_EMAILS=0
```

5. Do a first time migration of the models.

```
./manage.py migrate
```

6. Create a search index for ElasticSearch.

```
./manage.py index
```

7. Populate the database with some testdata.

```
./manage.py testdata
```

8. Run the Django development server.

```
./manage.py runserver 0:8000
```

Open http://localhost:8000 in your browser to see Lily. You can login using user **superuser1@lily.com** and **admin** as password. Congratulations, you just completed the basic Lily installation!

2.2 Migrating in Lily

We use the built-in Django migrations for both our schema & data migrations. For more detailed information you should take a look at the Django docs.

2.2.1 Caveats

Data migrations have two things you need to know about:

- The modified date
- Seperating schema and data changes

Modified date

It is important to disable the updating of modified fields on models during data migrations. The modified date should always represent the date a **user** last modified it.

Unfortunately as of writing, there is no way to set this behaviour automatically, so you'll have to do it yourself using *update_modified = False*.

Practical example:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models, migrations


def data_migration_forward(apps, schema_editor):
    Account = apps.get_model('accounts', 'Account')

    for account in Account.objects.all():
        account.update_modified = False
        # Do some mutations on the account here
        account.save()  # Because update_modified is False, this will not update the modified date o

class Migration(migrations.Migration):
    dependencies = []

    operations = [
        migrations.RunPython(data_migration_forward),
    ]
```

Seperation of schema/data changes

You should **never** combine a schema and data migration in a single migration file. This is because of consistency and error prevention. In some cases it is possible to combine the two, but you still shouldn't do it.

```
Thus, on PostgreSQL, for example, you should avoid combining schema changes and RunPython operations
in the same migration or you may hit errors like OperationalError: cannot ALTER TABLE "mytable"
because it has pending trigger events.
```

As can be read on the Django docs somewhat near the end of the block.

2.3 Running and debugging Celery tasks

Celery is a distributed task queue in Python that's being used to perform several queued tasks from within Lily. These are mostly tasking like syncing email and updating ElasticSearch indexes.

2.3.1 Running

```
celery worker -B --app=lily.celery --loglevel=info -Q celery,queue1,queue2,queue3 -n beat.%h -c 1
```

2.3.2 Debugging

Code that's being executed by a Celery worker can be PDB'ed with RDB. Add the following to your Celery code:

```
from celery.contrib import rdb
rdb.set_trace()
```

You should see a notification in the Celery console when a worker stumbles upon the rbd trace. At that point you can use telnet to login to the remote PDB session like:

```
telnet localhost 6902
help
```

CHAPTER 3

Community

Lily is being developed at Devhouse Spindle. Several developers are committing to the project:

- Allard Stijnman https://github.com/snoepkast
- Arjen Vellinga https://github.com/arjenfvellinga
- Bob Voorneveld https://github.com/bobvoorneveld
- Cornelis Poppema https://github.com/cpoppema
- Ednan Pasagic https://github.com/epasagic
- Ferdy Galema https://github.com/ferdynice
- Jeroen van Veen https://github.com/jvanveen
- Luuk Hartsema https://github.com/luukhartsema
- Marco Vellinga https://github.com/m-vellinga
- Redmer Loen https://github.com/spindleredmer
- Stefan Strijker https://github.com/00stefan00
- Tom Offringa https://github.com/TomOffringa

3.1 Contributing

As an open source project, anyone is welcome to contribute to Lily, which can include taking part in discussions, filing bug reports, contributing code or documentation or proposing improvements. Please read the following sections carefully if you would like to contribute.

This project adheres to the Open Code of Conduct. By participating, you are expected to honor this code.

3.1.1 Code of Conduct

This code of conduct outlines our expectations for participants within the **Lily** community, as well as steps to reporting unacceptable behavior. We are committed to providing a welcoming and inspiring community for all and expect our code of conduct to be honored. Anyone who violates this code of conduct may be banned from the community.

Our open source community strives to:

- **Be friendly and patient.**

- **Be welcoming**: We strive to be a community that welcomes and supports people of all backgrounds and identities. This includes, but is not limited to members of any race, ethnicity, culture, national origin, colour, immigration status, social and economic class, educational level, sex, sexual orientation, gender identity and expression, age, size, family status, political belief, religion, and mental and physical ability.
- **Be considerate**: Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions. Remember that we're a world-wide community, so you might not be communicating in someone else's primary language.
- **Be respectful**: Not all of us will agree all the time, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one.
- **Be careful in the words that we choose**: we are a community of professionals, and we conduct ourselves professionally. Be kind to others. Do not insult or put down other participants. Harassment and other exclusionary behavior aren't acceptable.
- **Try to understand why we disagree**: Disagreements, both social and technical, happen all the time. It is important that we resolve disagreements and differing views constructively. Remember that we're different. The strength of our community comes from its diversity, people from a wide range of backgrounds. Different people have different perspectives on issues. Being unable to understand why someone holds a viewpoint doesn't mean that they're wrong. Don't forget that it is human to err and blaming each other doesn't get us anywhere. Instead, focus on helping to resolve issues and learning from mistakes.

Definitions

Harassment includes, but is not limited to:

- Offensive comments related to gender, gender identity and expression, sexual orientation, disability, mental illness, neuro(a)typicality, physical appearance, body size, race, age, regional discrimination, political or religious affiliation
- Unwelcome comments regarding a person's lifestyle choices and practices, including those related to food, health, parenting, drugs, and employment
- Deliberate misgendering. This includes deadnaming or persistently using a pronoun that does not correctly reflect a person's gender identity. You must address people by the name they give you when not addressing them by their username or handle
- Physical contact and simulated physical contact (eg, textual descriptions like "*hug*" or "*backrub*") without consent or after a request to stop
- Threats of violence, both physical and psychological
- Incitement of violence towards any individual, including encouraging a person to commit suicide or to engage in self-harm
- Deliberate intimidation
- Stalking or following
- Harassing photography or recording, including logging online activity for harassment purposes
- Sustained disruption of discussion
- Unwelcome sexual attention, including gratuitous or off-topic sexual images or behaviour
- Pattern of inappropriate social contact, such as requesting/assuming inappropriate levels of intimacy with others
- Continued one-on-one communication after requests to cease

- Deliberate "outing" of any aspect of a person's identity without their consent except as necessary to protect others from intentional abuse
- Publication of non-harassing private communication

Our open source community prioritizes marginalized people's safety over privileged people's comfort. We will not act on complaints regarding:

- 'Reverse' -isms, including 'reverse racism,' 'reverse sexism,' and 'cisphobia'
- Reasonable communication of boundaries, such as "leave me alone," "go away," or "I'm not discussing this with you"
- Refusal to explain or debate social justice concepts
- Communicating in a 'tone' you don't find congenial
- Criticizing racist, sexist, cissexist, or otherwise oppressive behavior or assumptions

Diversity Statement

We encourage everyone to participate and are committed to building a community for all. Although we will fail at times, we seek to treat everyone both as fairly and equally as possible. Whenever a participant has made a mistake, we expect them to take responsibility for it. If someone has been harmed or offended, it is our responsibility to listen carefully and respectfully, and do our best to right the wrong.

Although this list cannot be exhaustive, we explicitly honor diversity in age, gender, gender identity or expression, culture, ethnicity, language, national origin, political beliefs, profession, race, religion, sexual orientation, socioeconomic status, and technical ability. We will not tolerate discrimination based on any of the protected characteristics above, including participants with disabilities.

Reporting Issues

If you experience or witness unacceptable behavior—or have any other concerns—please report it by contacting us via **hello@wearespindle.com**. All reports will be handled with discretion. In your report please include:

- Your contact information.
- Names (real, nicknames, or pseudonyms) of any individuals involved. If there are additional witnesses, please

include them as well. Your account of what occurred, and if you believe the incident is ongoing. If there is a publicly available record (e.g. a mailing list archive or a public IRC logger), please include a link. - Any additional information that may be helpful.

After filing a report, a representative will contact you personally, review the incident, follow up with any additional questions, and make a decision as to how to respond. If the person who is harassing you is part of the response team, they will recuse themselves from handling your incident. If the complaint originates from a member of the response team, it will be handled by a different member of the response team. We will respect confidentiality requests for the purpose of protecting victims of abuse.

Attribution & Acknowledgements

We all stand on the shoulders of giants across many open source communities. We'd like to thank the communities and projects that established code of conducts and diversity statements as our inspiration:

- Django
- Python

- Ubuntu
- Contributor Covenant
- Geek Feminism
- Citizen Code of Conduct

3.1.2 Lily Angular style guide

The Angular part of Lily was built with the John Papa Angular style guide serving as the basis and the Airbnb JavaScript style guide for the JavaScript part. This Lily Angular (& JavaScript) style guide will give you an overview of how we use various Angular components and the coding style in Lily. Not all examples might be representative of the actual code and some code might be missing to highlight the important bits. Any improvements are welcome of course.

Note: This isn't meant as a full Angular tutorial/guide, so I assume you have at least know the basics of Angular.

Basics

Let's start with the basics; the coding style. You can check our ESLint rules in the `.eslintrc` file and the scss-lint rules in the `.scss-lint.yml` file. Both located in the root of the Lily app. Here's a small excerpt of some general coding style rules.

Naming conventions

Our naming conventions differ slightly from the one used in the John Papa guide.

Element	Style	Example
Controllers	Functionality + 'Controller'	ListWidgetController
Directives	camelCase	listWidget
Filters	camelCase	customSanitize
Services	PascalCase	HLResource
Factories	PascalCase	Account

Comments

Comments start with a capital letter and end with a period. Inline comments (comments on the same line as the code) are written in lowercase and without period.

Directives

Directives are used a lot in Lily. Most things we use more than a few times will get converted to a directives. Even a simple thing like displaying a date is a directive, because we want to be consistent throughout the whole application. Let's take a random directive and break it down.

```
angular.module('app.directives').directive('editableSelect', editableSelect);

function editableSelect() {
    return {
        restrict: 'E',
        scope: {
            viewModel: '=',
```

```
            field: '@',
            type: '@',
            choiceField: '@',
            selectOptions: '=?', // contains any custom settings for the select
        },
        templateUrl: 'base/directives/editable_select.html',
        controller: EditableSelectController,
        controllerAs: 'es',
        transclude: true,
        bindToController: true,
    };
}

EditableSelectController.$inject = ['$scope', '$filter', 'HLResource'];
function EditableSelectController($scope, $filter, HLResource) {
    var es = this;

    es.getChoices = getChoices;
    es.updateViewModel = updateViewModel;

    activate();

    ...

    <other code>
```

Let's start at the top.

```
angular.module('app.directives').directive('editableSelect', editableSelect);
```

We set up the module and say what name the directive has and what function we call to invoke the directive. The directive can then be used like this (as seen on the `deals/controllers/detail.html` page)

```
<editable-select field="next_step" view-model="vm" type="Deal">
    {{ vm.deal.next_step.name }}
</editable-select>
```

Once the directive is called it invokes the `function editableSelect()`. Let's take the contents of that function and break it down (see comments).

```
return {
    // This directive can only be used as an HTML element (so by invoking <editable-select></editable
    restrict: 'E',
    // This directive has an isolated scope and accepts the following parameters:
    scope: {
        // Two way binded param. Changes to this param get reflected in the parent too.
        viewModel: '=',
        // One way binded param, so just pass the value so it can be used in this directive. Changes
        field: '@',
        type: '@',
        choiceField: '@',
        // Two way binded optional param.
        selectOptions: '=?',
    },
    templateUrl: 'base/directives/editable_select.html', // The template to be used.
    controller: EditableSelectController, // The controller which contains any logic for this directi
    controllerAs: 'es', // What variable is used to call the current directive. Is usually 'vm', but
    transclude: true, // Any content put between the directive's HTML tags will be put in the right s
    bindToController: true,
```

```
};
```

The directive then knows what controller to use and calls that controller (`EditableSelectController` in this case).

```
// Inject any dependencies for this controller (such as utility functions).
EditableSelectController.$inject = ['$scope', '$filter', 'HLResource'];
function EditableSelectController($scope, $filter, HLResource) {
    // Set the controller's scope to an easier to use variable. Using `this` could given conflicts.
    var es = this;

    // Bind functions to the scope.
    es.getChoices = getChoices;
    es.updateViewModel = updateViewModel;

    // Not required, but used as an 'init' function for the controller.
    activate();

    ...

    <other code>
```

The rest of this directive's code isn't relevant and won't be covered.

There's one more thing we need to create a directive: the template. The template for the above controller isn't very complicated and contains everything a normal template contains.

```
<span editable-select="es.selectModel" onshow="es.getChoices()" e-ng-options="item.id as item[es.opti
      onbeforesave="es.updateViewModel($data)" buttons="no">
    <ng-transclude></ng-transclude>
</span>
```

This template might be confusing, but you can pretty much ignore all the attributes in the `span` tag. They are there to call a third party library (Angular x-editable), but you can see how the controller's variables and function get used to set up the template. The `ng-transclude` you see is what I referred to in the intro to this directive. The `{{ vm.deal.next_step.name }}` is what will be put in the place of the `ng-transclude`. This transclusion allows you to have generic templates (like we do with the `widget` directive).

Note: Yes, another `editableSelect` directive gets called here, but this is the `editableSelect` provided by the Angular x-editable library.

Services

We use services to provide generic code to the app. Below is the `HLResource` service, which provides some useful functions related to resources.

```
// Make the service available and provide the name of the function which contains the logic.
angular.module('app.services').service('HLResource', HLResource);

// Inject any dependencies.
HLResource.$inject = ['$injector'];
function HLResource($injector) {
    this.patch = function(model, args) {
        // Function code.
    };

    ...
```

```
    <other code>
}
```

This function provides a generic way to `PATCH` a resource. It also provides generic error and success message once the request is done. An example of it's usage can be found below.

```
// Inject the HLResource service.
DealDetailController.$inject = ['Deal', 'HLResource'];
function DealDetailController(Deal, HLResource) {
    // DealDetailController code.

    function updateModel() {
        // updateModel code.

        return HLResource.patch('Deal', args);
    }
}
```

Resources/Factories

To retrieve data from the backend and to share data across the app we use factories. Below is an excerpt of the Deal factory.

```
angular.module('app.deals.services').factory('Deal', Deal);

Deal.$inject = ['$resource', 'HLUtils', 'HLForms', 'User'];
function Deal($resource, HLUtils, HLForms, User) {
    // 'private' variable to show it's only supposed to be used in this scope.
    // Factory can be used by calling `Deal.<function>`.
    var _deal = $resource(
        '/api/deals/:id/',
        null,
        {
            // Overwrite the built-in patch function Angular provides so we can overwrite the transfo
            // and do stuff like cleaning our data.
            patch: {
                method: 'PATCH',
                params: {
                    id: '@id',
                },
                transformRequest: function () {
                    // transformRequest code.
                },
            },
            // Allows us to search deals through ElasticSearch.
            query: {
                url: '/search/search/',
                method: 'GET',
                params: {
                    // Set url GET parameters.
                    type: 'deals_deal',
                },
            },
            // This could be its own resource, but since it's so tightly connected to deals we just
            // provide it in the Deal service.
            getNextSteps: {
                url: 'api/deals/next-steps/',
```

```
            },
        }
    );

    return _deal;
```

Angular tips & tricks

This section provides a couple of tips & tricks which can save a lot of Googling and wondering why your code isn't working.

Passing resources to directive

Make sure you either resolve promises before passing them to a directive or resolve them in the directive's controller. An example of this is the `listWidget` directive. Here it's not always sure if we're passing a list or passing a promise. So we do the following check and resolve the promise if needed and then execute our code.

```
if (vm.collapsableItems) {
    // Certain list widgets have collapsable cells, so set the default state to collapsed.
    if (!vm.list.hasOwnProperty('$promise')) {
        // Array was passed, so just pass the list.
        _setCollapsed(vm.list);
    } else {
        vm.list.$promise.then(function(response) {
            // List hasn't fully loaded, so wait and pass the response.
            _setCollapsed(response);
        });
    }
}
```

Building un-minified files

By default the `gulp build` and `gulp watch` commands will provide you with minified files. This is nice for production, but when developing it can lead to a lot of frustration because of unclear errors. You can use the following commands to make sure you build un-minified files.

```
NODE_ENV=dev gulp build
```

```
NODE_ENV=dev gulp watch
```

Linting

Make sure your editor has ESLint and preferably scss-lint set up so you can instantly see any violations. A pre-commit hook which runs the linters is nice to have as well in case you miss a violation during development.

ng-inspector

[ng-inspector](http://ng-inspector.org/): Tired of doing *console.log()* everywhere just to see what you models contain? Use ng-inspector and you get a real-time overview of all variables currently available. If needed you can click one to *console.log()* it.