The h Documentation

Release 0.0.2

Hypothes.is Project and contributors

Sep 19, 2018

Contents

h is the web app that serves most of the https://hypothes.is/ website, including the web annotations API at https://hypothes.is/api/. The Hypothesis client is a browser-based annotator that is a client for h's API, see the client's own documentation site for docs about the client.

This documentation is for:

- Developers working with data stored in h
- Contributors to h

CHAPTER 1

Contents

1.1 The Hypothesis community

Please be courteous and respectful in your communication on Slack (request an invite or log in once you've created an account), IRC (#hypothes.is on freenode.net), the mailing list (subscribe, archive), and GitHub. Humor is appreciated, but remember that some nuance may be lost in the medium and plan accordingly.

If you plan to be an active contributor please join our mailing list to coordinate development effort. This coordination helps us avoid duplicating efforts and raises the level of collaboration. For small fixes, feel free to open a pull request without any prior discussion.

1.2 Advice for publishers

If you publish content on the web and want to allow people to annotate your content, the following documents will help you get started.

1.2.1 Generating authorization grant tokens

> **Warning:** This document describes an integration mechanism that is undergoing early-stage testing. The details of the token format may change in the future.

In order to allow your users (i.e. those whose accounts and authentication status you control) to annotate using a copy of Hypothesis embedded on your pages, you can ask us to register your site as a special kind of OAuth client.

We will issue you with a *Client ID* and a *Client secret*. These will allow you generate time-limited "authorization grant tokens" which can be supplied as configuration to the Hypothesis sidebar, and which will allow your users to annotate without needing to log in again. This document describes how to generate those tokens.

Overview

You will have been provided with the following:

Client ID A unique identifier for your client account. It's a UUID and will look something like this: `4a2fa3b4-c160-4436-82d3-148f602c9aa8`

Client secret A secret string which you MUST NOT reveal publicly, and which is used to cryptographically sign the grant tokens you will generate.

In addition, you will have provided us with what we call an "authority" for your account. The authority is a DNS domain name and acts as a unique namespace for your user's accounts. For example, if your site lives at `https://example.com`, you may choose to use `example.com` as your authority, although we do not currently require any particular correspondence between your web address and your account authority.

You will use these three pieces of information, in combination with your user's unique usernames, to generate the grant token.

The grant token is a JSON Web Token (JWT) and we strongly recommend that you use an existing JWT library for your programming environment if you can. You can find a list of JWT client libraries here.

Token format

A grant token is a JWT, signed with the *Client secret* using the `HS256` algorithm, with a payload in a specific format, given below. Let's assume that:

- Your *Client ID* is `4a2fa3b4-c160-4436-82d3-148f602c9aa8`.
- Your authority is `customwidgets.com`.
- The user has a username of `samina.mian`.
- The current time is `2016-11-08T11:35:45Z`, which corresponds to a UTC Unix timestamp of `1478601345`.
- The token should be valid for a few minutes, e.g. until `1478601645`, expressed as a UTC Unix timestamp. The server limits the lifetime of a token (the difference between the `nbf` and `exp` timestamps) to 10 minutes.
- The token should be valid for the annotation service running at `hypothes.is`.

With these data, we can construct a token payload. It should look like the following:

```
{
  "aud": "hypothes.is",
  "iss": "4a2fa3b4-c160-4436-82d3-148f602c9aa8",
  "sub": "acct:samina.mian@customwidgets.com",
  "nbf": 1478601345,
  "exp": 1478601645
}
```

You should sign this payload using the `HS256` JWT-signing algorithm, using the *Client secret* as the key. The result will look something like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJpc3MiOiI0QTJGQTNCNC1DMTYwLTQ0MzYtODJEMy0xNDhGNjAyQzlBQTgiLCJuYmYiOjE0Nzg2MDEzNDUsImF1ZCI6Imh5cG
↪65-ZErbLu1q8LpT_K8FAOQO984hAyN1XFBe1rC3lgfk
```

See also:

RFC7523, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants". Note that we currently only support the `HS256` signing algorithm, and not the public-key `RS256` signing algorithm mentioned in the RFC.

Examples

This section contains complete example code for generating a JWT in various common programming environments.

Python

We recommend using PyJWT:

```
import datetime
import jwt

# IMPORTANT: replace these values with those for your client account!
CLIENT_AUTHORITY = 'customwidgets.com'
CLIENT_ID        = '4a2fa3b4-c160-4436-82d3-148f602c9aa8'
CLIENT_SECRET    = '5SquUVG0Tpg57ywoxUbPPgjtK0OkX1ttipVlfBRRrpo'

def generate_grant_token(username):
   now = datetime.datetime.utcnow()
   userid = 'acct:{username}@{authority}'.format(username=username,
                                                 authority=CLIENT_AUTHORITY)
   payload = {
      'aud': 'hypothes.is',
      'iss': CLIENT_ID,
      'sub': userid,
      'nbf': now,
      'exp': now + datetime.timedelta(minutes=10),
   }
   return jwt.encode(payload, CLIENT_SECRET, algorithm='HS256')
```

Ruby

We recommend using ruby-jwt:

```
require 'jwt'

# IMPORTANT: replace these values with those for your client account!
CLIENT_AUTHORITY = 'customwidgets.com'
CLIENT_ID        = '4a2fa3b4-c160-4436-82d3-148f602c9aa8'
CLIENT_SECRET    = '5SquUVG0Tpg57ywoxUbPPgjtK0OkX1ttipVlfBRRrpo'

def generate_grant_token(username)
  now = Time.now.to_i
  userid = "acct:#{username}@#{CLIENT_AUTHORITY}"
  payload = {
    aud: "hypothes.is",
    iss: CLIENT_ID,
    sub: userid,
    nbf: now,
    exp: now + 600
```

(continues on next page)

(continued from previous page)

```
  }
  JWT.encode payload, CLIENT_SECRET, 'HS256'
end
```

1.3 Using the Hypothesis API

The Hypothesis API enables you to create applications and services which read or write data from the Hypothesis service.

1.3.1 Authorization

API requests which only read public data do not require authorization.

API requests made as a particular user or which manage user accounts or groups require authorization.

Using OAuth

OAuth is an open standard that enables users to grant an application (an "OAuth client") the ability to interact with a service as that user, but without providing their authentication credentials (eg. username and password) directly to that application.

OAuth clients follow an authorization process, at the end of which they get an access token which is included in API requests to the service. This process works as follows:

1. The user clicks a login button or link in the application.
2. The application sends them to an authorization page from the h service (`/oauth/authorize`) via a redirect or popup, where the user will be asked to approve access.
3. If they approve, the authorization endpoint will send an authorization code back to the application (via a redirect).
4. The application exchanges the authorization code for a pair of tokens using the h service's `POST /api/token` endpoint: A short-lived access token to authorize API requests, and a long-lived refresh token.
5. When the access token expires, the application obtains a new access token by submitting the refresh token to the `POST /api/token` endpoint.

To build an application for Hypothesis that uses OAuth, there are two steps:

1. Register an OAuth client for your application in the h service.
2. Implement the client-side part of the OAuth flow above in your application. You may be able to use an existing OAuth 2 client library for your language and platform.

Registering an OAuth client

To register an OAuth client on an instance of the h service for which you have admin access, go to the */admin/oauthclients* page.

To register a new client as an admin of the "h" service:

1. Go to */admin/oauthclients* and click "Register a new OAuth client".

2. Enter a name for a client, select "authorization_code" as the grant type and enter the URL where your client will listen for the authorization code as the "redirect URL".
3. Click "Register client" to create the client. Make a note of the randomly generated client ID.

Implementing the OAuth flow

The h service implements the "Authorization code grant" OAuth flow, with the following endpoints:

- Authorization endpoint: `/oauth/authorize`
- Token endpoint: `/api/token`

In order to implement the flow, your application must do the following:

1. When a user clicks the "Login" link, the application should open the h service's authorization page at `/oauth/authorize` using the query parameters described in 4.1.1 Authorization Request.

   Example request:

```
GET /oauth/authorize?client_id=510cd02e-767b-11e7-b34b-ebcff2e51409&redirect_
↪uri=https%3A%2F%2Fmyapp.com%2Fauthorize&response_type=code&
↪state=aa3d3062b4dbe0a1 HTTP/1.1
```

2. After the user authorizes the application, it will receive an authorization code via a call to the redirect URI. The application must exchange this code for an access token by making a request to the `POST /api/token` endpoint as described in 4.1.3 Access Token Request.

   Example request:

```
POST /api/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

client_id=631206c8-7792-11e7-90b3-872e79925778&
↪code=V1bjcvKDivRUc6Sg1jhEc8ckDwyLNG&grant_type=authorization_code
```

   Example response:

```
{
  "token_type": "Bearer",
  "access_token": "5768-mfoPT52ogx0Si7NkU8QFicj183Wz1O4OQmbNIvBhjTQ",
  "expires_in": 3600,
  "refresh_token": "4657-dkJGNdVn8dmhDvgCHVPmIJ2Zi0cYQgDNb7RWXkpGIZs",
  "scope": "annotation:read annotation:write"
}
```

3. Once the application has an access token, it can make API requests and connect to the real time API. See *Authorization* for details of how to include this token in requests.
4. The access token expires after a while, and must be refreshed by making a request to the `POST /api/token` endpoint as described in 6. Refreshing an access token.

   Example request:

```
POST /api/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=4657-
↪diyCpZ9oPRBaBkaW6ZrKgI0yagvZ9yBgLmxJ9k4HfeM
```

Example response:

```
{
  "token_type": "Bearer",
  "access_token": "5768-8CHodeMUAPCLmuBooabXolnpHReBUI5cC3txCXk7sQA",
  "expires_in": 3600,
  "refresh_token": "4657-11f1CUrhZs29QvXpywDpsXFwlfl_wPEIY5N8whwUrRw",
  "scope": "annotation:read annotation:write"
}
```

Revoking tokens

If your application no longer needs an OAuth token, for example because a user has logged out of your application which uses Hypothesis accounts, it is good practice to revoke the access and refresh tokens.

Hypothesis implements the OAuth 2 Token Revocation endpoint at `/oauth/revoke`.

Example request:

```
POST /oauth/revoke HTTP/1.1
Content-Type: application/x-www-form-urlencoded

token=5768-yXoTA2R94b5fB0dTBbXHSvc_IX4I1Gc_bGQ4KyjM5dY
```

Further reading

- "OAuth 2 simplified" is a good introduction for developers.
- The OAuth specification describes the standard in detail.
- The OAuth Token Revocation specification describes an extension to support revoking tokens.

Access tokens

API requests which read or write data as a specific user need to be authorized with an access token. Access tokens can be obtained in two ways:

1. By generating a personal API token on the Hypothesis developer page (you must be logged in to Hypothesis to get to this page). This is the simplest method, however these tokens are only suitable for enabling your application to make requests as a single specific user.
2. By *registering* an "OAuth client" and *implementing* the OAuth authentication flow in your application. This method allows any user to authorize your application to read and write data via the API as that user. The Hypothesis client is an example of an application that uses OAuth.

   See *Using OAuth* for details of how to implement this method.

Once an access token has been obtained, requests can be authorized by putting the token in the `Authorization` header.

Example request:

```
GET /api HTTP/1.1
Host: hypothes.is
Accept: application/json
Authorization: Bearer $TOKEN
```

(Replace `$TOKEN` with your own API token or OAuth access token.)

Client credentials

Endpoints for managing user accounts are authorized using a client ID and secret ("client credentials"). These can be obtained by *registering an OAuth client* with the grant type set to `client_credentials`.

Once a client ID and secret have been obtained, requests are authorized using HTTP Basic Auth, where the client ID is the username and the client secret is the password.

For example, with client details as follows

```
Client ID: 96653f8e-80be-11e6-b32b-c7bcde86613a
Client Secret: E-hReVMuRyZbyr1GikieEw4JslaM6sDpb18_9V59PFw
```

you can compute the Authorization header [as described in RFC7617](https://tools.ietf.org/html/rfc7617):

```
$ echo -n '96653f8e-80be-11e6-b32b-c7bcde86613a:E-hReVMuRyZbyr1GikieEw4JslaM6sDpb18_
↪9V59PFw' | base64
OTY2NTNmOGUtODBiZS0xMWU2LWIzMmItYzdiY2RlODY2MTNhOkUtaFJlVk11UnlaYnlyMUdpa2llRXc0SnNsYU02c0RwYjE4XzlW
```

Example request:

```
POST /users HTTP/1.1
Host: hypothes.is
Accept: application/json
Content-Type: application/json
Authorization: Basic␣
↪OTY2NTNmOGUtODBiZS0xMWU2LWIzMmItYzdiY2RlODY2MTNhOkUtaFJlVk11UnlaYnlyMUdpa2llRXc0SnNsYU02c0RwYjE4Xzl

{
  "authority": "example.com",
  "username": "jbloggs1",
  "email": "jbloggs1@example.com"
}
```

1.3.2 API Reference Stub

This is a stub document for the API reference to allow creating internal links to it from the TOC tree.

In the compiled documentation it is replaced by the contents of _extra/api-reference.

See https://github.com/sphinx-doc/sphinx/issues/701

1.3.3 Real Time API

Warning: This document describes an API that is in the early stages of being documented and refined for public use. Details may change, and your systems may break in the future.

In addition to the HTTP API Hypothesis has a WebSocket-based API that allows developers to receive near real-time notifications of annotation events.

Overview

To use the Real Time API, you should open a WebSocket connection to the following endpoint:

```
wss://hypothes.is/ws
```

Communication with this endpoint consists of JSON-encoded messages sent from client to server and vice versa.

Authorization

Clients that are only interested in receiving notifications about public annotations on a page do not need to authenticate. Clients that want to receive notifications about all updates relevant to a particular user must authenticate.

Server-side clients can authenticate to the Real Time API by providing an access token in an Authorization header:

```
Authorization: Bearer <token>
```

Browser-based clients are not able to set this header due to limitations of the the browser's `WebSocket` API. Instead they can authenticate by setting an `access_token` query parameter in the URL when connecting:

```
var socket = new WebSocket(`wss://hypothes.is/ws?access_token=${token}`)
```

Server messages

Each messages from the server will be either an *event* or a *reply*:

event An event is sent to clients as a result of an action taken elsewhere in the system. For example: if an annotation is made which matches one of the client's subscriptions, the client will receive an event message. All event messages have a `type` field.

reply A reply is sent in response to a message sent by the client. All replies have an `ok` field which indicates whether the server successfully processed the client's message, and a `reply_to` field which indicates which client message the server is responding to.

Clients should ignore events with types that they do not recognise, as this will allow us to add new events in future without breaking your client.

Note: We will add documentation for specific event types as we upgrade the protocol.

Sending messages

All messages sent to the server must have a numeric ID which is unique for the connection. The ID should be sent with the message in the `id` field. In addition, every message sent to the server must have a valid `type` field. See below for the different types of message you can send.

Message types

- *ping*
- *whoami*

ping

To verify that the connection is still open, clients can (and are encouraged to) send a "ping" message:

```
{
   "id": 123,
   "type": "ping"
}
```

The server replies with a `pong` message:

```
{
   "ok": true,
   "reply_to": 123,
   "type": "pong"
}
```

whoami

Primarily for debugging purposes, you can send the server a "who am I?" message to check whether you have authenticated correctly to the WebSocket.

```
{
   "id": 123,
   "type": "whoami"
}
```

The server will respond with a `whoyouare` message:

```
{
   "ok": true,
   "reply_to": 123,
   "type": "whoyouare",
   "userid": "acct:joe.bloggs@hypothes.is"
}
```

1.4 Developing Hypothesis

The following sections document how to setup a development environment for h and how to contribute code or documentation to the project.

1.4.1 Contributor License Agreement

Before submitting significant contributions, we ask that you sign one of our Contributor License Agreements. This practice ensures that the rights of contributors to their contributions are preserved and protects the ongoing availability of the project and a commitment to make it available for anyone to use with as few restrictions as possible.

If contributing as an individual please sign the CLA for individuals:

- CLA for individuals, HTML
- CLA for individuals, PDF

If making contributions on behalf of an employer, please sign the CLA for employees:

- CLA for employers, HTML
- CLA for employers, PDF

A completed form can either be sent by electronic mail to license@hypothes.is or via conventional mail at the address below. If you have any questions, please contact us.

```
Hypothes.is Project
2261 Market St #632
SF, CA 94114
```

1.4.2 Website dev install

The code for the https://hypothes.is/ website and API lives in a Git repo named h. To get this code running in a local development environment the first thing you need to do is install h's system dependencies.

See also:

This page documents how to setup a development install of h. For installing the Hypothesis client for development see https://github.com/hypothesis/client/, and for the browser extension see https://github.com/hypothesis/browser-extension.

Follow either the *Installing the system dependencies on Ubuntu* or the *Installing the system dependencies on macOS* section below, depending on which operating system you're using, then move on to *Getting the h source code from GitHub* and the sections that follow it.

Installing the system dependencies on Ubuntu

This section describes how to install h's system dependencies on Ubuntu. These steps will also probably work with few or no changes on other versions of Ubuntu, Debian, or other Debian-based GNU/Linux distributions.

Install the following packages:

```
sudo apt-get install -y --no-install-recommends \
    build-essential \
    git \
    libevent-dev \
    libffi-dev \
    libfontconfig \
    libpq-dev \
    libssl-dev \
    python-dev \
    python-pip \
    python-virtualenv
```

Install node by following the instructions on nodejs.org (the version of the nodejs package in the standard Ubuntu repositories is too old).

Upgrade pip, virtualenv and npm:

```
sudo pip install -U pip virtualenv
sudo npm install -g npm
```

Installing the system dependencies on macOS

This section describes how to install h's system dependencies on macOS.

The instructions that follow assume you have previously installed Homebrew.

Install the following packages:

```
brew install \
    libevent \
    libffi \
    node \
    postgresql \
    python
```

Note: Unfortunately you need to install the `postgresql` package, because Homebrew does not currently provide a standalone `libpq` package.

Upgrade pip and virtualenv:

```
pip install -U pip virtualenv
```

Getting the h source code from GitHub

Use `git` to download the h source code:

```
git clone https://github.com/hypothesis/h.git
```

This will download the code into an `h` directory in your current working directory.

Change into the `h` directory from the remainder of the installation process:

```
cd h
```

Installing the services

h requires the following external services:

- PostgreSQL 9.4+
- Elasticsearch v6, with the Elasticsearch ICU Analysis plugin
- RabbitMQ v3.5+

You can install these services however you want, but the easiest way is by using Docker and Docker Compose. This should work on any operating system that Docker can be installed on:

1. Install Docker and Docker Compose by following the instructions on the Docker website.
2. Run Docker Compose:

   ```
   docker-compose up
   ```

   You'll now have some Docker containers running the PostgreSQL, RabbitMQ, and Elasticsearch services. You should be able to see them by running `docker ps`. You should also be able to visit your Elasticsearch service by opening http://localhost:9200/ in a browser, and connect to your PostgreSQL by running `psql postgresql://postgres@localhost/postgres` (if you have psql installed).

Note: If at any point you want to shut the containers down, you can interrupt the `docker-compose` command. If you want to run the containers in the background, you can run `docker-compose up -d`.

3. Create the *htest* database in the `postgres` container. This is needed to run the h tests:

```
docker-compose exec postgres psql -U postgres -c "CREATE DATABASE htest;"
```

Tip: You can use Docker Compose image to open a psql shell in your Dockerized database container without having to install psql on your host machine. Do:

```
docker-compose exec postgres psql -U postgres
```

Tip: Use the `docker-compose logs` command to see what's going on inside your Docker containers, for example:

```
docker-compose logs rabbit
```

For more on how to use Docker and Docker Compose see the Docker website.

Installing the gulp command

Install `gulp-cli` to get the `gulp` command:

```
sudo npm install -g gulp-cli
```

Creating a Python virtual environment

Create a Python virtual environment to install and run the h Python code and Python dependencies in:

```
virtualenv .venv
```

Activating your virtual environment

Activate the virtual environment that you've created:

```
source .venv/bin/activate
```

Tip: You'll need to re-activate this virtualenv with the `source .venv/bin/activate` command each time you open a new terminal, before running h. See the Virtual Environments section in the Hitchhiker's guide to Python for an introduction to Python virtual environments.

Running h

Start a development server:

```
make dev
```

The first time you run `make dev` it might take a while to start because it'll need to install the application dependencies and build the client assets.

This will start the server on port 5000 (http://localhost:5000), reload the application whenever changes are made to the source code, and restart it should it crash for some reason.

Running h's tests

There are test suites for both the frontend and backend code. To run the complete set of tests, run:

```
make test
```

To run the frontend test suite only, run the appropriate test task with gulp. For example:

```
gulp test
```

When working on the front-end code, you can run the Karma test runner in auto-watch mode which will re-run the tests whenever a change is made to the source code. To start the test runner in auto-watch mode, run:

```
gulp test-watch
```

To run only a subset of tests for front-end code, use the `--grep` argument or mocha's .only() modifier.

```
gulp test-watch --grep <pattern>
```

Debugging h

The pyramid_debugtoolbar package is loaded by default in the development environment. This will provide stack traces for exceptions and allow basic debugging. A more advanced profiler can also be accessed at the /_debug_toolbar path.

> http://localhost:5000/_debug_toolbar/

Check out the pyramid_debugtoolbar documentation for information on how to use and configure it.

You can turn on SQL query logging by setting the `DEBUG_QUERY` environment variable (to any value). Set it to the special value `trace` to turn on result set logging as well.

Feature flags

Features flags allow admins to enable or disable features for certain groups of users. You can enable or disable them from the Administration Dashboard.

To access the Administration Dashboard, you will need to first create a user account in your local instance of H and then give that account admin access rights using H's command-line tools.

See the *Accessing the admin interface* documentation for information on how to give the initial user admin rights and access the Administration Dashboard.

Troubleshooting

Cannot connect to the Docker daemon

If you get an error that looks like this when trying to run `docker` commands:

```
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
Error: failed to start containers: postgres
```

it could be because you don't have permission to access the Unix socket that the docker daemon is bound to. On some operating systems (e.g. Linux) you need to either:

- Take additional steps during Docker installation to give your Unix user access to the Docker daemon's port (consult the installation instructions for your operating system on the Docker website), or
- Prefix all `docker` commands with `sudo`.

1.4.3 Accessing the admin interface

To access the admin interface, a user must be logged in and have admin permissions. To grant admin permissions to a user, run the following command:

```
hypothesis user admin <username>
```

For example, to make the user 'joe' an admin in the development environment:

```
hypothesis --dev user admin joe
```

When this user signs in they can now access the adminstration panel at `/admin`. The administration panel has options for managing users and optional features.

1.4.4 An introduction to the h codebase

If you're new to the team, or to the Hypothesis project, you probably want to get up to speed as quickly as possible so you can make meaningful improvements to `h`. This document is intended to serve as a brief "orientation guide" to help you find your way around the codebase.

This document is a living guide, and is at risk of becoming outdated as we continually improve the software. If you spot things that are out of date, please submit a pull request to update this document.

This guide was last updated on 11 Apr 2017.

A lightning guide to Pyramid

The `h` codebase is principally a Pyramid web application. Pyramid is more of a library of utilities than a "framework" in the sense of Django or Rails. As such, the structure (or lack thereof) in our application is provided by our own conventions, and not the framework itself.

Important things to know about Pyramid that may differ from other web application frameworks you've used:

- Application setup is handled explicitly by a distinct configuration step at boot. You'll note `includeme` functions in some modules – these are part of that configuration system.
- The `request` object is passed into views explicitly rather than through a threadlocal (AKA "global variable"), and is often passed around explicitly to provide request context to other parts of the application. This has a number of advantages but can get a bit messy if not managed appropriately.

You can read more about the distinguishing features of Pyramid in the excellent Pyramid documentation.

Application components

The important parts of the `h` application can be broken down into:

Models SQLAlchemy models representing the data objects that live in our database. These live in `h.models`.

Views (and templates) Views are code that is called in response to a particular request. Templates can be used to render the output of a particular view, typically as HTML.

With a few exceptions, views live in `h.views`, and templates live in the `h/templates/` directory.

Services Putting business logic in views can quickly lead to views that are difficult to test. Putting business logic in models can lead to model objects with a large number of responsibilities.

As such, we put most business logic into so-called "services." These are objects with behaviour and (optionally) state, which can be retrieved from the `request` object.

Services live in `h.services`.

Tasks Tasks are bits of code that run in background workers and which can be easily triggered from within the context of a request.

We use Celery for background tasks, and task definitions can be found in `h.tasks`.

There are a number of other modules and packages in the `h` repository. Some of these (e.g. `h.auth`, `h.settings`) do one-off setup for a booting application. Others may be business logic that dates from before we introduced the services pattern, and thus might be more appropriately moved into a service in the future.

1.4.5 Submitting a Pull Request

To submit code or documentation to h you should submit a pull request.

For trivial changes, such as documentation changes or minor errors, PRs may be submitted directly to master. This also applies to changes made through the GitHub editing interface. Authors do not need to sign the CLA for these, or follow fork or branch naming guidelines.

For any non-trivial changes, please create a branch for review. Fork the main repository and create a local branch. Later, when the branch is ready for review, push it to a fork and submit a pull request.

Discussion and review in the pull request is normal and expected. By using a separate branch, it is possible to push new commits to the pull request branch without mixing new commits from other features or mainline development.

Some things to remember when submitting or reviewing a pull request:

- Your pull request should contain one logically separate piece of work, and not any unrelated changes.
- When writing commit messages, please bear the following in mind:
  - http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
  - https://github.com/blog/831-issues-2-0-the-next-generation

  Please minimize issue gardening by using the GitHub syntax for closing issues with commit messages.
- We recommend giving your branch a relatively short, descriptive, hyphen-delimited name. `fix-editor-lists` and `tabbed-sidebar` are good examples of this convention.
- Don't merge on feature branches. Feature branches should merge into upstream branches, but never contain merge commits in the other direction. Consider using `--rebase` when pulling if you must keep a long-running

branch up to date. It's better to start a new branch and, if applicable, a new pull request when performing this action on branches you have published.

- Code should follow our *coding standards*.
- All pull requests should come with code comments. For Python code these should be in the form of Python docstrings. For AngularJS code please use ngdoc. Other documentation can be put into the `docs/` subdirectory, but is not required for acceptance.
- All pull requests should come with unit tests. For the time being, functional and integration tests should be considered optional if the project does not have any harness set up yet.

  For how to run the tests, see *Running h's tests*.

1.4.6 Code style

This section contains some code style guidelines for the different programming languages used in the project.

Python

Follow PEP 8, the linting tools below can find PEP 8 problems for you automatically.

Docstrings

All public modules, functions, classes, and methods should normally have docstrings. See PEP 257 for general advice on how to write docstrings (although we don't write module docstrings that describe every object exported by the module).

The `pep257` tool (which is run by `prospector`, see below) can point out PEP 257 violations for you.

It's good to use Sphinx references in docstrings because they can be syntax highlighted and hyperlinked when the docstrings are extracted by Sphinx into HTML documentation, and because Sphinx can print warnings for references that are no longer correct:

- Use Sphinx Python cross-references to reference other Python modules, functions etc. from docstrings (there are also Sphinx domains for referencing objects from other programming languages, such as JavaScript).
- Use Sphinx info field lists to document parameters, return values and exceptions that might be raised.
- You can also use reStructuredText to add markup (bold, code samples, lists, etc) to docstrings.

Linting

We use Flake8 for linting Python code. Lint checks are run as part of our continuous integration builds and can be run locally using `make lint`. You may find it helpful to use a flake8 plugin for your editor to get live feedback as you make changes.

Automated code formatting

You can use YAPF (along with the YAPF configuration in this git repo) to automatically reformat Python code. We don't strictly adhere to YAPF-generated formatting but it can be a useful convenience.

Additional reading

- Although we don't strictly follow all of it, the Google Python Style Guide contains a lot of good advice.

Front-end Development

See the Hypothesis Front-end Toolkit repository for documentation on code style and tooling for JavaScript, CSS and HTML.

We use ESLint for linting front-end code. Use `gulp lint` to run ESlint locally. You may find it helpful to install an ESLint plugin for your editor to get live feedback as you make changes.

1.4.7 Testing

This section covers writing tests for the `h` codebase.

Getting started

Sean Hammond has written up a guide to getting started running and writing our tests, which covers some of the tools we use (`tox` and `pytest`) and some of the testing techniques they provide (factories and parametrization).

Unit and functional tests

We keep our functional tests separate from our unit tests, in the `tests/functional` directory. Because these are slow to run, we will usually write one or two functional tests to check a new feature works in the common case, and unit tests for all the other cases.

Using mock objects

The `mock` library lets us construct fake versions of our objects to help with testing. While this can make it easier to write fast, isolated tests, it also makes it easier to write tests that don't reflect reality.

In an ideal world, we would always be able to use real objects instead of stubs or mocks, but sometimes this can result in:

- complicated test setup code
- slow tests
- coupling of test assertions to non-interface implementation details

For new code, it's usually a good idea to design the code so that it's easy to test with "real" objects, rather than stubs or mocks. It can help to make extensive use of value objects in tested interfaces (using `collections.namedtuple` from the standard library, for example) and apply the functional core, imperative shell pattern.

For older code which doesn't make testing so easy, or for code that is part of the "imperative shell" (see link in previous paragraph) it can sometimes be hard to test what you need without resorting to stubs or mock objects, and that's fine.

1.4.8 Writing documentation

To build the documentation issue the `make dirhtml` command from the `docs` directory:

```
cd docs
make dirhtml
```

When the build finishes you can view the documentation by running a static web server in the newly generated `_build/dirhtml` directory. For example:

```
cd _build/dirhtml; python -m SimpleHTTPServer; cd -
```

API Documentation

The Hypothesis API documentation is rendered using ReDoc, a JavaScript tool for generating OpenAPI/Swagger reference documentation.

The documentation-building process above will regenerate API documentation output without intervention, but if you are making changes to the API specification (*hypothesis.yaml*), you may find it convenient to use the ReDoc CLI tool, which can watch the spec file for changes:

```
npm install -g redoc-cli
redoc-cli serve [path-to-spec] --watch
```

1.4.9 Serving h over SSL in development

If you want to annotate a site that's served over HTTPS then you'll need to serve h over HTTPS as well, since the browser will refuse to load external scripts (eg. H's bookmarklet) via HTTP on a page served via HTTPS.

To serve your local dev instance of h over HTTPS:

1. Generate a private key and certificate signing request:

   ```
   openssl req -newkey rsa:1024 -nodes -keyout .tlskey.pem -out .tlscsr.pem
   ```

2. Generate a self-signed certificate:

   ```
   openssl x509 -req -in .tlscsr.pem -signkey .tlskey.pem -out .tlscert.pem
   ```

3. Run `hypothesis devserver` with the `--https` option:

   ```
   hypothesis devserver --https
   ```

4. Since the certificate is self-signed, you will need to instruct your browser to trust it explicitly by visiting https://localhost:5000 and selecting the option to bypass the validation error.

Troubleshooting

Insecure Response errors in the console

The sidebar fails to load and you see `net::ERR_INSECURE_RESPONSE` errors in the console. You need to open https://localhost:5000 and tell the browser to allow access to the site even though the certificate isn't known.

Server not found, the connection was reset

When you're serving h over SSL in development making non-SSL requests to h won't work.

If you get an error like **Server not found** or **The connection was reset** in your browser (it varies from browser to browser), possibly accompanied by a gunicorn crash with `AttributeError: 'NoneType' object has no attribute 'uri'`, make sure that you're loading https://localhost:5000 in your browser, not `http://`.

WebSocket closed abnormally, code: 1006

If you see the error message **Error: WebSocket closed abnormally, code: 1006** in your browser, possibly accompanied by another error message like **Firefox can't establish a connection to the server at wss://localhost:5001/ws**, this can be because you need to add a security exception to allow your browser to connect to the websocket. Visit https://localhost:5001 in a browser tab and add a security exception then try again.

403 response when connecting to WebSocket

If your browser is getting a 403 response when trying to connect to the WebSocket along with error messages like these:

- WebSocket connection to 'wss://localhost:5001/ws' failed: Error during WebSocket handshake: Unexpected response code: 403
- Check that your H service is configured to allow WebSocket connections from https://127.0.0.1:5000
- WebSocket closed abnormally, code: 1006
- WebSocket closed abnormally, code: 1001
- Firefox can't establish a connection to the server at wss://localhost:5001/ws

make sure that you're opening https://localhost:5000 in your browser and *not* https://127.0.0.1:5000.

1.4.10 Making changes to model code

Guidelines for writing model code

No length limits on database columns

Don't put any length limits on your database columns (for example `sqlalchemy.Column(sqlalchemy.Unicode(30), ...)`). These can cause painful database migrations.

Always use `sqlalchemy.UnicodeText()` with no length limit as the type for text columns in the database (you can also use `sqlalchemy.Text()` if you're sure the column will never receive non-ASCII characters).

When necessary validate the lengths of strings in Python code instead. This can be done using SQLAlchemy validators in model code.

View callables for HTML forms should also use Colander schemas to validate user input, in addition to any validation done in the model code, because Colander supports returning per-field errors to the user.

Creating a database migration script

If you've made any changes to the database schema (for example: added or removed a SQLAlchemy ORM class, or added, removed or modified a `sqlalchemy.Column` on an ORM class) then you need to create a database migration script that can be used to upgrade the production database from the previous to your new schema.

We use Alembic to create and run migration scripts. See the Alembic docs (and look at existing scripts in h/migrations/versions) for details. The `hypothesis migrate` command is a wrapper around Alembic. The steps to create a new migration script for h are:

1. Create the revision script by running `bin/hypothesis migrate revision`, for example:

   ```
   bin/hypothesis migrate revision -m "Add the foobar table"
   ```

   This will create a new script in `h/migrations/versions/`.

2. Edit the generated script, fill in the `upgrade()` and `downgrade()` methods.

   See https://alembic.readthedocs.io/en/latest/ops.html#ops for details.

   Note: Not every migration should have a `downgrade()` method. For example if the upgrade removes a max length constraint on a text field, so that values longer than the previous max length can now be entered, then a downgrade that adds the constraint back may not work with data created using the updated schema.

   ---

3. Stamp your database.

   Before running any upgrades or downgrades you need to stamp the database with its current revision, so Alembic knows which migration scripts to run:

   ```
   bin/hypothesis migrate stamp <revision_id>
   ```

   `<revision_id>` should be the revision corresponding to the version of the code that was present when the current database was created. The will usually be the `down_revision` from the migration script that you've just generated.

4. Test your `upgrade()` function by upgrading your database to the most recent revision. This will run all migration scripts newer than the revision that your db is currently stamped with, which usually means just your new revision script:

   ```
   bin/hypothesis migrate upgrade head
   ```

   After running this command inspect your database's schema to check that it's as expected, and run h to check that everything is working.

   Note: You should make sure that there's some repesentative data in the relevant columns of the database before testing upgrading and downgrading it. Some migration script crashes will only happen when there's data present.

   ---

5. Test your `downgrade()` function:

   ```
   bin/hypothesis migrate downgrade -1
   ```

   After running this command inspect your database's schema to check that it's as expected. You can then upgrade it again:

```
bin/hypothesis migrate upgrade +1
```

Batch deletes and updates in migration scripts

It's important that migration scripts don't lock database tables for too long, so that when the script is run on the production database concurrent database transactions from web requests aren't held up.

An SQL `DELETE` command acquires a `FOR UPDATE` row-level lock on the rows that it selects to delete. An `UPDATE` acquires a `FOR UPDATE` lock on the selected rows *if the update modifies any columns that have a unique index on them that can be used in a foreign key*. While held this `FOR UPDATE` lock prevents any concurrent transactions from modifying or deleting the selected rows.

So if your migration script is going to `DELETE` or `UPDATE` a large number of rows at once and committing that transaction is going to take a long time (longer than 100ms) then you should instead do multiple `DELETE`s or `UPDATE`s of smaller numbers of rows, committing each as a separate transaction. This will allow concurrent transactions to be sequenced in-between your migration script's transactions.

For example, here's some Python code that deletes all the rows that match a query in batches of 25:

```
query = <some sqlalchemy query>
query = query.limit(25)
while True:
    if query.count() == 0:
        break
    for row in query:
        session.delete(row)
    session.commit()
```

Separate data and schema migrations

It's easier for deployment if you do *data migrations* (code that creates, updates or deletes rows) and *schema migrations* (code that modifies the database *schema*, for example adding a new column to a table) in separate migration scripts instead of combining them into one script. If you have a single migration that needs to modify some data and then make a schema change, implement it as two consecutive migration scripts instead.

Don't import model classes into migration scripts

Don't import model classes, for example `from h.models import Annotation`, in migration scripts. Instead copy and paste the `Annotation` class into your migration script.

This is because the script needs the schema of the `Annotation` class as it was at a particular point in time, which may be different from the schema in `h.models.Annotation` when the script is run in the future.

The script's copy of the class usually only needs to contain the definitions of the primary key column(s) and any other columns that the script uses, and only needs the name and type attributes of these columns. Other attributes of the columns, columns that the script doesn't use, and methods can usually be left out of the script's copy of the model class.

Troubleshooting migration scripts

(sqlite3.OperationalError) near "ALTER"

SQLite doesn't support `ALTER TABLE`. To get around this, use Alembic's batch mode.

Cannot add a NOT NULL column with default value NULL

If you're adding a column to the model with `nullable=False` then when the database is upgraded it needs to insert values into this column for each of the already existing rows in the table, and it can't just insert `NULL` as it normally would. So you need to tell the database what default value to insert here.

`default=` isn't enough (that's only used when the application is creating data, not when migration scripts are running), you need to add a `server_default=` argument to your `add_column()` call.

See the existing migration scripts for examples.

1.4.11 Environment Variables

This section documents the environment variables supported by h.

CLIENT_URL
: The URL at which the Hypothesis client code is hosted. This is the URL to the client entrypoint script, by default https://cdn.hypothes.is/hypothesis.

CLIENT_OAUTH_ID
: The OAuth client ID for the Hypothesis client on pages that embed it using the service's /embed.js script.

CLIENT_RPC_ALLOWED_ORIGINS
: The list of origins that the client will respond to cross-origin RPC requests from. A space-separated list of origins. For example: `https://lti.hypothes.is https://example.com http://localhost.com:8001`.

Index