swe_t.py

Celebrity Rap Sheets

Jacob Brouwer
Parker Berg
Parker Timmins
Ian Daszkowski

Introduction

Given the goal of creating an IMDB style website that displays information about celebrities and the crimes they have allegedly committed, how should we model the data on the server side? Also, what is a reasonable API for querying this data? Note that there is an extension of this problem: we would like the platform to be reasonable not only for our internal use, but also for any arbitrary third-party that would like to query our database.

Data Representation

From a development perspective, we tackled this problem before defining the API. (This is a bit of a simplification as in practice there were additional iterations of both our models and our API after we finished drafting a “v0”.) We determined that having an existing model of our data would simplify the initial API design work. This may have been a less enticing option if the data to be represented were a bit more complex, but, given the nature of the website, the data unfolded naturally. Without discussing the modeling in too much detail (as we will dive down into this problem and our solution a bit more in the next section), we found that if we modeled our data in terms of the three pillars, the result was a natural division into columns.

API Design

The key to our API design iteration process was taking on the perspective of a client of our API. We wanted anyone using the platform to be able to interact with it as a black box for any arbitrary use of the data. What is interesting about this process is that, in reality, the design process involves defining what these arbitrary uses might look like. The end goal was to produce a *minimal* API; in other words, we wanted to minimize the total number of queries

and to make each response as concise as possible. Ultimately we determined that: the pillars would be queried in isolation; a client would either query *a pillar as a whole* or *an individual member of a pillar*; whole pillar queries were likely to be used for internal logic or listing and therefore should respond with minimum information; and individual member queries should respond with all data associated with a member (rather than dividing the member query into arbitrary permutations that unnecessarily increase complexity).

Use Cases

In the case of our website, the primary use case is purely for fun and to satisfy curiosity. Did you know George Clooney has been arrested for civil disobedience or that DMX seems to enjoy driving without a license?

In the case of the API, the primary use case is to populate a website such as the one we are building. However, it may also be used from a research standpoint (querying the data with no intention of displaying it in an IMDB-style website). Most end uses seem to segregate into one of these two camps.

Design

RESTful API

Celebrity Rap Sheet API

INTRODUCTION

The Celebrity Rap Sheet API is a platform for querying data about celebrities and their criminal charges.

REFERENCE

Celebrities

Celebrity related resources of the **Celebrity Rap Sheet API**

Celebrity Collection

List all Celebrities

+ Response 200 (application/json)

```
[{
  "uri": "http://23.253.252.30/api/celebrity/1", "name": "John Smith", "id": 1
}, {
  "uri": "http://23.253.252.30/api/celebrity/1", "name": "Jane Doe", "id": 2
}]
```


Celebrity

A single Celebrity object with all its details

Retrieve all data for a Celebrity

+ Response 200 (application/json)


```
{
    "aliases":
    [{
        "alias": "Jimmy John", "id": 1
    }, {
        "alias": "J", "id": 2
    }],
    "birthday": "10-20-2003",
    "charges":
    [{
        "id": 1, "uri": "http://23.253.252.30/api/charge/1"
    }, {
        "id": 2, "uri": "http://23.253.252.30/api/charge/2"
    }],
    "crimes":
    [{
        "id": 1, "name": "Rhymin",
        "uri": "http://23.253.252.30/api/crime/1"
    }, {
        "id": 3, "name": "Stealin",
        "uri": "http://23.253.252.30/api/crime/3"
    }],
    "description": "Description that may contain markup<br />.",
```

```
    "id": 1,
    "imdb_url": "http://www.imdb.com/name/nm0123456/",
    "name": "John Smith",
    "picture_url": "http://some.domain/resource",
    "twitter_handle": "@JSmith",
    "wiki_url": "http://en.wikipedia.org/wiki/JohnSmith"
}
```


Crimes

Crime related resources of the **Celebrity Rap Sheet API**

Crime Collection

List all Crimes

+ Response 200 (application/json)


```
    [{
        "uri": "http://23.253.252.30/api/crime/1", "name": "Rhymin", "id": 1
    }, {
        "uri": "http://23.253.252.30/api/crime/2", "name": "Cheatin", "id": 2
    }, {
        "uri": "http://23.253.252.30/api/crime/3", "name": "Stealin", "id": 3
    }]
```

Crime

A single Crime object with all its details

Retrieve all data for a Crime

+ Response 200 (application/json)


```
{
    "celebrities":
    [{
        "id": 1, "name": "John Smith", "uri": "http://23.253.252.30/api/celebrity/1"
    }, {
        "id": 2, "name": "Jane Doe", "uri": "http://23.253.252.30/api/celebrity/2"
    }],
    "charges":
    [{
        "id": 1, "uri": "http://23.253.252.30/api/charge/1"
    }, {
        "id": 2, "uri": "http://23.253.252.30/api/charge/2"
    }],
    "description": "Rhymin in the face of the law.",
    "id": 1,
    "name": "Rhymin",
    "wiki_url": "http://en.wikipedia.org/wiki/Rhymin"
}
```

Charges

Charge related resource of the **Celebrity Rap Sheet API**

Charge Collection

List all Charges

+ Response 200 (application/json)


```
[{
    "uri": "http://23.253.252.30/api/charge/1", "id": 1
}, {
    "uri": "http://23.253.252.30/api/charge/2", "id": 2
}]
```


Charge

A single Charge object with all its details

Retrieve all data for a Charge

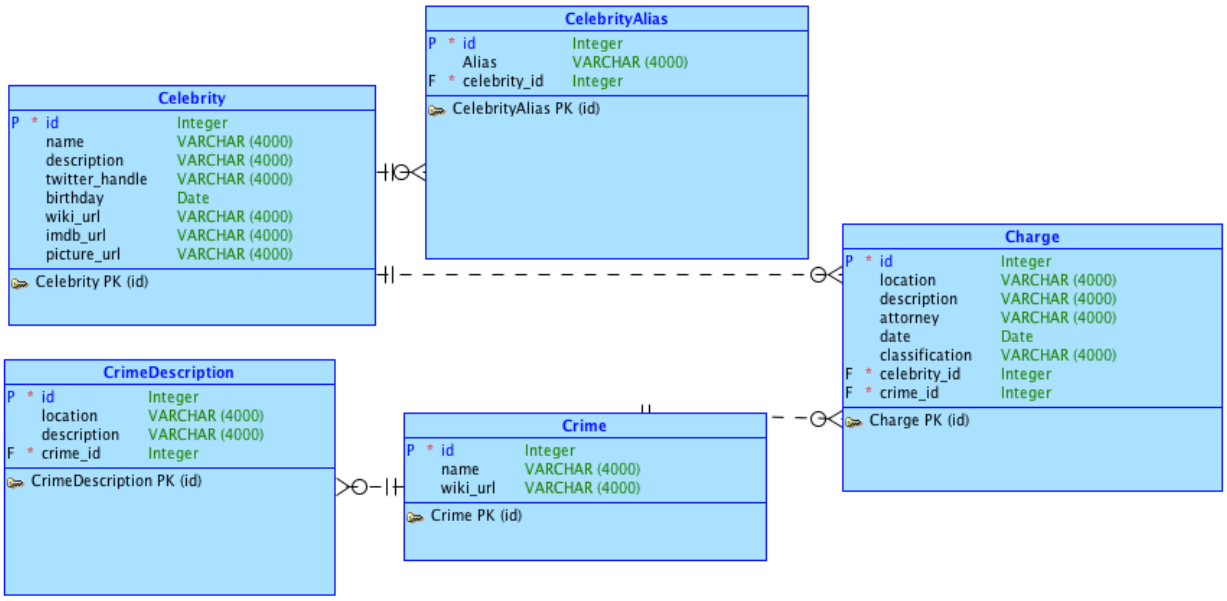
+ Response 200 (application/json)


```
{
```

```
    "attorney": "Name or Null",
    "celebrity":
    {
        "id": 1, "name": "John Smith", "uri": "http://23.253.252.30/api/celebrity/1"
    },
    "classification": "Class A Misdemeanor",
    "crime":
    {
        "id": 1, "name": "Rhymin", "uri": "http://23.253.252.30/api/crime/1"
    },
    "date": "10-20-2003",
    "description": "Actor John Smith was found Rhymin in the early morning.",
    "location": "Austin, Texas"
}
```


Models

The three pillars of the website are Celebrities, Crimes, and Charges; for each pillar there is a flask model. An instance of the Celebrity model, unsurprisingly, corresponds to a specific celebrity. A Crime instance is a type of crime. Lastly, a Charge is little more than a tuple containing a celebrity and a crime, and corresponds to a specific instance of the celebrity in question being charged with performing the crime in question. In addition to these models corresponding to the three pillars, there is another model: CelebrityAlias. This model is quite simple, containing a celebrity and a string alias; it is not discussed in detail. The following UML diagram shows the relationships between the models. The CrimeDescription table is extraneous and not in use.



Additionally, below are the three main models in a form very similar to their SQLAlchemy definitions. There are some things worth noting which are not evident from the UML diagram. First, id is a unique identifier which is created automatically at instantiation. Also, the only required fields in Celebrity and Crime are the name fields, and the only required fields in Charge are celebrity and crime. Any interesting use of the models will instantiate most of the optional fields, but leaving them out makes design and testing easier. Finally, the relationship fields are implicit in the UML diagram, but explicit in the definitions. These are not actually data held in the instances, but are database queries which are resolved dynamically. Thus they can only be called after all instances are added to the database. Note that the only relationships fields that are actually required to make the system work are Charge's celebrity and crime fields. All other relationships could have been obtained by explicitly querying the database tables associated with the models, but the field notation is easier to use.

Crime

id : integer (primary key, unique)
name : string
wiki_url : string (optional)

description : string (optional)
celebrities : relationship (Celebrity, though Charge)
charges : relationship (Charge)

Celebrity
id : integer (primary key, unique)
name : string
description : string (optional)
twitter_handle : string (optional)
birthday : datetime (optional)
wiki_url : string (optional)
imdb_url : string (optional)
picture_url : string (optional)
aliases : relationship (CelebrityAlias)
crimes : relationship (Crime, through Charge)
charges : relationship (Charge)

Charge
id : integer (primary key, unique)
date : datetime (optional)
location : string (optional)
description : string (optional)
attorney : string (optional)
classification : string (optional)
celebrity_id : integer (foreign key, Celebrity.id)
crime_id : integer (foreign key, Crime.id)
crime : relationship (Crime)
celebrity : relationship (Celebrity)

Code Source:

- api.py: Contains all route definitions annd json response generations that re required by the api.
- app.py: Contains the flask app definition and configurations.
- initialize_db.py: Containt the entirety of the database as a combination of raw data and defined SQLAlchemy models. Used to update the database. This is inelegant, however very useful for our small data size and quick data and model turn around.
- tests.py: Contains all tests, which are discussed in deatil later in this report.

- models.py: Contains detailed definitions of our three content pillars and four database tables.
- views.py: Eqivilant to api.py, except for routing the webpages and http responses.
- filters.py: Contains filter functions that are passed in to the jinja template evaluations. Use this to pass functions to templates.
- run.py: Runs the development server. Also used by uWSGI to define and import the application.
- uwsgi.sh: Used for mangement of the uWSGI application. ./uwsgi.sh start will start the application and ./uwsgi.stop will elegantly kill it.

Configurations

Our nginx routing is defined in /etc/nginx/sites-enabled/default and strictly routes any http requests made to the server directly to our uWSGI application which it expect to find at 127.0.0.1:5000. This is not flexible, and the correct way to do this would be to tell uWSGI to write a socket file somewhere and change the nginx configuration to point to that file instead of the string literal. We had issues getting uWSGI to run correctly with the socket file configuration, but this is a planned improvement.

Our uWSGI configuration is very simple and has been streamlined for server management. The entirety of the configuration is handled by uwsgi.sh. When running the start command, it initializes the application via nohup with the following flags:

- socket: Defines what ip address and port the application is listening on.
- wsgi-file: Defines the location of the python file which has the flask application imported globally (in our case, run.py).
- master: Sets up a master/worker configuration.
- processes: Sets up 4 worker processes.

- thread: Sets up two threads for those workers.
- callable: The name of the flask application as defined in the wsgi-file
- pidfile: Declares an (existing) file to overwrite with the pid of the master uWSGI process.

When uwsgi.sh is run with the stop command, it reads the defined pidfile and sends a SIGINT signal to the process there, which causes the master process to shutdown all worker processes then shut istelf down. The script starts the server via the nohup program, to keep it running after logout, however this is not the recommended method of keeping a uWSGI application running. This should eventually be migrated over to uWSGI-Emperor to manage startup and maintenance of the uWSGI application. We opted to use nohup for simplicity sake.

Our postgres configuration is particularly inelegant solution that was decided upon to reduce development overhead. We have two postgres users, postgres and parker. Parker is the actual owner of the celebs database, however the account goes mostly untouched. All processes intending to touch the server must be run as the postgres user. I repeat, you must ‘su postgres’ (password cs373) then execute commands like ‘python3 initialize_db.py’, ‘python3 run.py -p 5050’, or ‘./uwsgi.sh start’, otherwise the process will not hae adaquet permissions to read from the database.

Technologies Used

Flask - A python framework for building websites. Flask also contains other useful frameworks like jinja and SQLAlchemy. An instance of the Flask application contains all logic necessary to run the server, http routing to functions, static file routing, and http responses. This server can be run directly,, this is used as the development server.

SQLAlchemy - An object relational mapper and database wrapper. In the running flask application the models are reified as python objects. We also wish to hold the model instances in a database for persistent storage and for speed and ease of manipulation. A possible implementation would define the python class for each model separately from the database schema for each model, as well as a mechanism for mapping between these model representations. SQLalchemy lets us define both the class and schema simultaneously and performs the mapping automatically.

PostgreSQL - A relational database management system. This is used to hold the model instances which are dynamically injected into the web pages.

Jinja - A template system for injecting values computed from python code into html pages.

Bootstrap - A css library which provides the look and feel of the website.

APIary - This is not actually a part of the running system. It provides a way to design the API interface before working on the implementation, and hosts nicely formatted documentation of the API.

uWSGI - The protocol in which our flask application serves local urls (127.0.0.1:5000). Most uWSGI setup is performed and handled by the flask application behind the scenes, however when running the application directly through uWSGI instead of python, certain configurations need to be supplied.

nginx - The server routing software that listens on port 80 for HTTP requests and forwards them to our custom defined server directive, our uWSGI application. This allows serving our server through external urls like celebrapsheet.tk

Tests

As it is difficult to test visual aspects of a website, the unit tests for this project are only used to verify the correctness of the back-end. Specifically, there are unit tests which check that the models are correct, tests which check the database interactions, and tests which check the api functions.

Because the application has a database and a running server instance, the unit tests require setUp and tearDown methods to handle configuration. One problematic aspect of testing the server is that there is a single global server instance, the flask app. This makes it difficult to spin up multiple instances of the server application with different configurations, which would make testing simpler and would be a valuable change in future iterations. The key part of the flask app that needs to be configured for testing is the app's setting the app's database. The database is simply a separate database from the production database; the tables in the database are created before each test and deleted after each test. The remainder of this section consists of a description of the different types of tests.

Model Attribute Tests - These tests do not consider the database aspects of the model instances, but treat them as regular objects. This just means that to test a given model we simply create an instance of the model and verify that the field values are correct. Because we do not add the model instance to the database, any fields which correspond to queries on

other model tables will not have a meaningful value. So we check that all regular data fields contain the correct data, and all query fields return the empty list.

Model database tests - These tests involve instantiating a single object of each model type, adding it to the database, then verifying that it may be obtained by querying the database. In addition to testing the models, this type of test gives confidence that the database is working correctly.

Query Attribute Tests - As mentioned above, in addition to containing fields which are instance variables like in regular objects, model instances have fields which correspond to database queries. These tests check the correctness of these query fields. As these tests involve queries, all of the pertaining model instances must be added to the database before any assertions are checked. Note that the regular data fields are left as default values in these model instantiations as they are irrelevant and would only serve to obfuscate the code.For example, in one such test we create two charges each with the same celebrity, but with two different crimes. After adding them to the database, we check that calling the ‘crimes’ query field on the celebrity returns both crimes.

Query Duplicate Tests - These tests are very similar to the previous type of test, except that they check that the query calls do not return duplicates. For example, we create two charges with the same celebrity and the same crime, then verify that calling the ‘crimes’ query on the celebrity only returns a single copy of the crime.

Local API Tests - Ideally we would unit test the api calls on a running server instance through the public internet. Unfortunately, we found this difficult with the flask application's configuration, so we used a FlaskClient to test locally. This object simply takes a call with a url suffix, for example '/api/celebrity', maps this to the appropriate routing function and returns the result of the function call. We then test the returned value against expected JSON results.