Star Lords: Phase I

Group Members:

Andres Avila

Megan Knocke

Olivia Lin

Sydney Mercier

Chris Metcalf

<u>Table of Contents</u>

Introduction

Problem

Star Lords exists to rule over (and map) the universe's heavenly bodies. The cosmos is infinitely vast and the Earth just one small part of it. Few people are actively aware of the many varied worlds that exist both within and outside of our solar system, much less aware of the hierarchies and categorizations used to define them. The purpose of Star Lords is to make this wider universe accessible by showing the relationships between planets and their stars, stars and their star clusters, and to show how even bodies separated by thousands of light-years may still be related as "families" - members of the same cosmic neighborhood.

Use Cases

Users can search Star Lords for information on moons, planets, stars, constellations, and families of constellations; a ranking from smallest to largest. Every object page links to other celestial objects which are linked to that object based on this hierarchy. For example, the page for the star WASP-1 contains links to its planet and the constellation where it can be found. In this case, the constellation is Andromeda, whose page also contains a link to the family of constellations it belongs to, as well as the stars and planets that it contains.

Our knowledge about the universe is incomplete, so some adjustments had to be made for the data presented. Separate categories were made for planets in our own solar system and exoplanets - planets orbiting other stars. Not as much data is known for these exoplanets, but users should be able to lookup as much as is currently known.

Even though our own solar system, aptly called *the* Solar System, is not in a constellation (constellations being arbitrary concepts that are only visible from *inside* our own system), Star Lords has conveniently included it in its mapping so that users can explore the scientific attributes of the sun and its planets.

Design

<u>Flask/UML Model</u>

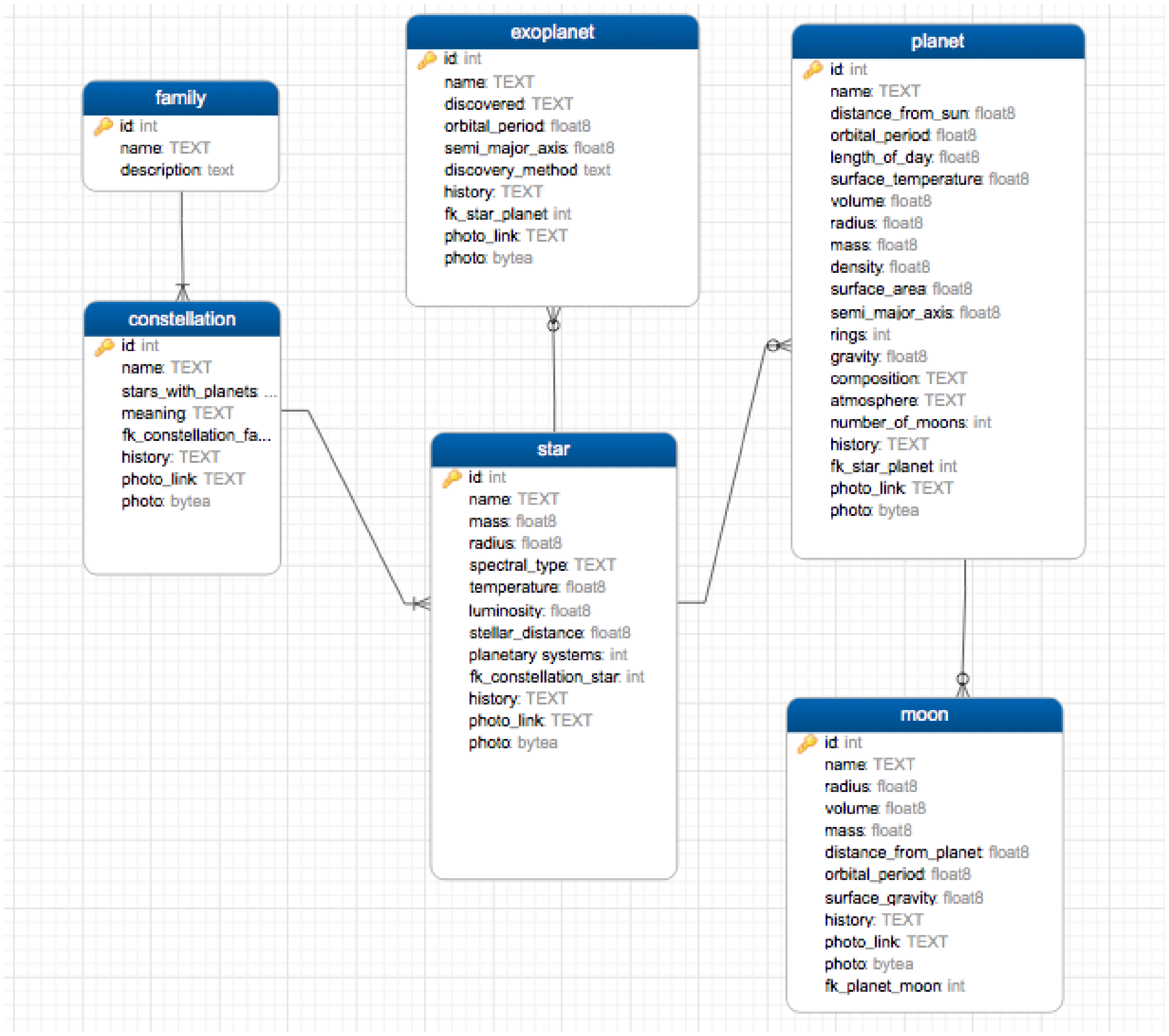


Figure 1: shows the models and tables

Data Model Relationships:

We have six tables: family, constellation, exoplanet, planet, and moon (Figure 1).

Family: Family represents the eight named families of constellations. Each family has a one to many relationship with the constellation table, meaning that one family contains one or many constellations. We broke with the traditional classification schema in order to give our own Solar System a family of “Home”, which isn’t a family of constellations, but we wanted a special category for our own Solar System.

Constellation*:* The constellation table represents the 88 named constellations - a group of stars forming a recognizable pattern that is traditionally named after its apparent form or identified with a mythological figure. The fk_constellation_family column in the constellation table is a foreign key pointing to the id column in the family table. It has a one to many relationship with the star table, meaning a constellation has one or more stars. For our own Solar System, we will list its constellation as “Our Solar System” because it is not in a constellation. Constellations are viewed from our perspective into the night sky here on Earth, so our sun is not a part of any, hence our special categorization.

Star: The star table represents named stars in the galaxy. The number of stars is quite large, so we will limit the number of entries in this table only to stars with special significance. The fk_constellation_star column in the star table is a foreign key pointing to the id column in the constellation table. The star table has a zero to many relationship with both the exoplanet table and also the planet table.

Exoplanet: The exoplanet table represents planets outside our Solar System. The fk_star_planet column is a foreign key pointing to the id column in the star table.

Planet: The planet table represents planets in our Solar System. The fk_star_planet column is a foreign key pointing to the id column in the star table. The planet table has a zero to many relationship with the moon table.

Moon: The moon table represents moons in our Solar System. The fk_planet_moon column is a foreign key pointing to the id column in the planet table.

Data Model Attributes:

Family: Each family of constellations has an id and a name, representing the eight named families of constellations plus our Solar System, which we will represent as Home, in the family table. It also has a description.

Constellation: Each constellation has several attributes, including id, name, stars with planets, meaning, history, fk_constellation_family, history, photo link, and photo. “Meaning” holds the definition of the constellation name while “history” gives a brief description of each constellation’s discovery and history since its creation, to the extent known. “Photo” holds a blob or bytea type in postgresql, which should be a jpg file. “Photo link” is a url to the photo. Our own solar system will be listed under a row for Our Solar System, as it is not in a traditional constellation.

Star: Each star has an id, name, mass, radius, spectral_type, temperature, luminosity, stellar_distance, planetary systems, fk_constellation_star, history, photo link, and photo. Most stars are currently classified under the Morgan–Keenan (MK) system using the letters O, B, A, F, G, K, and M, in a sequence from the hottest (O type) to the coolest (M type). Each letter class is then subdivided using a numeric digit with 0 being hottest and 9 being coolest. The sequence has been expanded with classes for other stars and star-like objects that do not fit in the classical system, such as class D for white dwarfs and class C for carbon stars.

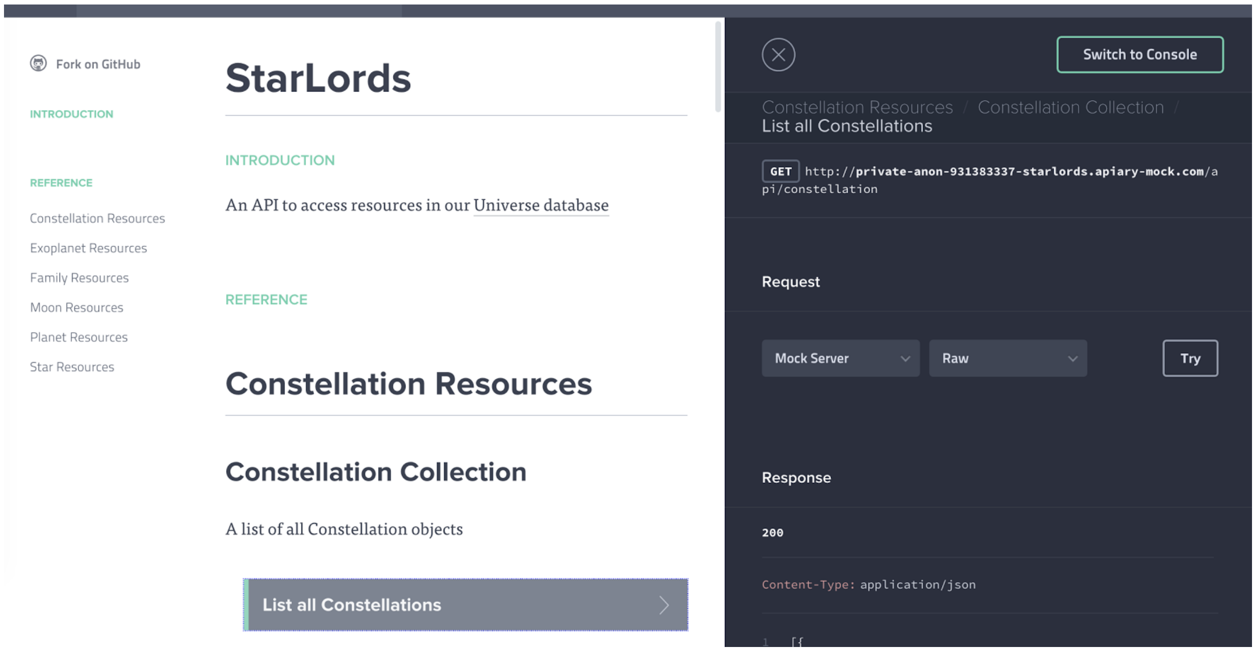
In the MK system a luminosity class is added to the spectral class using Roman numerals. This is based on the width of certain absorption lines in the star's spectrum which vary with the density of the atmosphere, distinguishing giant stars from dwarfs. Luminosity class 0 or Ia+ stars are hyper-giants, class I stars are supergiants, class II stars are bright giants, class III stars are regular giants, class IV stars are sub-giants, class V stars are main-sequence stars, class sd is used for sub-dwarfs, and class D is used for white dwarfs.

Exoplanet: Each exoplanet has an id, name, when discovered, orbital period, semi-major axis, discovery method, history, fk_star_planet, photo link, and photo. The semi-major axis is the radius of an orbit at the orbit's two most distant points.

Planet: Each planet has an id, name, distance from the sun, orbital period, length of day, surface temperature, volume, mass, density, surface area, semi-major axis, number of rings, gravity, composition, atmosphere, number of moons, history, fk_star_planet, photo link, and photo.

Moon: Each moon has an id, name, radius, volume, mass, distance from planet, orbital period, surface gravity, history, fk_planet_moon, photo link, and photo.

<u>RESTful API (Apiary)</u> (Documented at http://docs.starlords.apiary.io/)



Fork on GitHub

INTRODUCTION

REFERENCE

Constellation Resources

Exoplanet Resources

Family Resources

Moon Resources

Planet Resources

Star Resources

StarLords

INTRODUCTION

An API to access resources in our Universe database

REFERENCE

Constellation Resources

Constellation Collection

A list of all Constellation objects

List all Constellations

Switch to Console

Constellation Resources / Constellation Collection / List all Constellations

```
GET http://private-anon-931383337-starlords.apiary-mock.com/api/constellation
```

Request

Mock Server | Raw | Try

Response

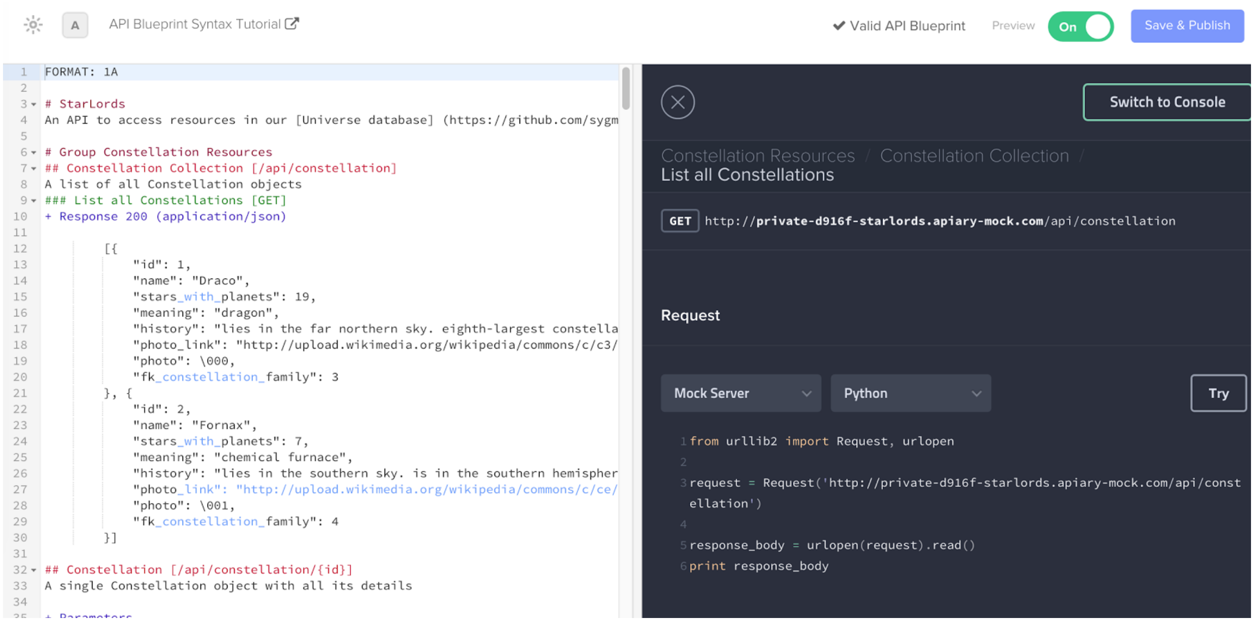
```
200

Content-Type: application/json

[{
```

Figure 2: API documentation

Set up apiary: http://apiary.io/

<u>Apiary Summary</u>

Apiary helps document API design (an API mockup) and allows team members to contribute to the shared API editor (Figure 3). Apiary can connect with GitHub, so that when team members press “Save and Publish”, the API BluePrint gets saved onto GitHub and allows version control.



```
FORMAT: 1A

# StarLords
An API to access resources in our [Universe database] (https://github.com/sygm

# Group Constellation Resources
## Constellation Collection [/api/constellation]
A list of all Constellation objects
### List all Constellations [GET]
+ Response 200 (application/json)

        [{
            "id": 1,
            "name": "Draco",
            "stars_with_planets": 19,
            "meaning": "dragon",
            "history": "lies in the far northern sky. eighth-largest constella
            "photo_link": "http://upload.wikimedia.org/wikipedia/commons/c/c3/
            "photo": \000,
            "fk_constellation_family": 3
        }, {
            "id": 2,
            "name": "Fornax",
            "stars_with_planets": 7,
            "meaning": "chemical furnace",
            "history": "lies in the southern sky. is in the southern hemispher
            "photo_link": "http://upload.wikimedia.org/wikipedia/commons/c/ce/
            "photo": \001,
            "fk_constellation_family": 4
        }]

## Constellation [/api/constellation/{id}]
A single Constellation object with all its details

+ Parameters
```

Constellation Resources / Constellation Collection /
List all Constellations

```
GET http://private-d916f-starlords.apiary-mock.com/api/constellation
```

Request

```
from urllib2 import Request, urlopen

request = Request('http://private-d916f-starlords.apiary-mock.com/api/constellation')

response_body = urlopen(request).read()
print response_body
```

Figure 3: API editor

The API design teaches the user about the attributes of our tables, what format and information users will get with an HTTP GET request, and try the call requests on a Mocker Server. The requests return dictionaries that are formatted in JSON and can be parsed by the user for information.

Our API has resources for our pages: constellations, exoplanets, family, moon, planet, and star. Using our API, users can get a list of those resource objects or a single resource object.

Dependencies

Flask==0.10.1

Flask-SQLAlchemy==2.0

psycopg2==2.6

SQLAlchemy==0.9.9

The database behind the application is running on Postgresql and being managed by SQLAlchemy's ORM. SQLAlchemy is defining and providing the data models, which are described in more detail on the data model's relationships section. Psycop2 is the driver required for SQLAlchemy to work with the Postgres database.

Commands for Running our Application on our Rackspace Server

Our application runs on the Rackspace server with IP: http://104.130.244.239

We SSH onto our server using: ssh -i .ssh/id_rsaRack root@104.130.244.239

Once we're logged in, we can pull the newest files from GitHub onto our server using git pull.

We can activate the virtual environment with source /my-venv/bin/activate.

Go to the cs373-idb file. Then, from within that file we can call these following commands for nginx and uwsgi:

```
sudo /etc/init.d/nginx stop
sudo /etc/init.d/nginx start
uwsgi --ini universe_uwsgi.ini
```

The site can then be navigated with http://104.130.244.239.

Serving Pages with Flask

Flask is managed by uWSGI. uWSGI talks to nginx, and nginx handles contact with the outside world. When a client connects to the server trying to reach our Flask app:

- nginx opens the connection and proxies it to uWSGI
- uWSGI handles the Flask instances we have and connects one to the client
- Flask talks to the client

We have a nginx configuration file, nginx.conf that works with uwsgi on port 4242. The communication between nginx and uWSGI is done via a socket file, called universe_uwgi.sock. We create this with our uWSGI configuration file, universe_uwsgi.ini. We execute uWSGI and pass it the newly created configuration file with the uwsgi --ini universe_uwsgi.ini command. Now nginx can connect to the uWSGI process.

Right now, all of the pages are static for the web application. We include the following in the __init__.py file to allow the project to work:

```
from flask import Flask, render_template, url_for, g, request, session, redirect, abort, flash
app = Flask(__name__)
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080)
```

We are running the application using nginx and uwsgi right now, but if we just wanted to run the python file __init__.py, the above app.run line would be necessary. We have port 8080 open if we wanted to do this.

These allow Flask to work with our application and run. To render each page, there must be a separate function for each template page. The following example is the standard format for serving a page through flask.

```
@app.route('/example')
def examplepage():
    return render_template("example.html")
```

All html files are templates and are stored in the /templates directory. Flask is also used in these html files. Right now we are using it for the following cases:

- The ability to insert our navbar in each template.

Example:

```
{% include 'navbar.html' %}
```

This is included at the top of each body element in each static page. It will include html from another file relative to the location in the file.

- Inserting static files into the html.

  Examples:

  ```
  {{ url_for('static', filename='css/bootstrap.min.css') }}
  {{ url_for('static', filename='images/andromeda.jpg') }}
  {{ url_for('static', filename='js/jquery-2.1.3.min.js') }}
  ```

  The first parameter 'static' is the folder relative to the root where the content is stored, which is static for this project. Filename is the relative path to a file with the data in the static folder. Flask will use this file as the source of the data.

BootStrap

Bootstrap is a CSS, HTML, and JS framework that helps in developing responsive web applications. To install, go to http://getbootstrap.com/.

At the moment, all we needed to get Bootstrap working on our site were the following files, with the following locations in the file structure:

/373-idb/static/css/bootstrap.min.css

/373-idb/static/bootstrap-theme.min.css

/373-idb/static/bootstrap.min.js

/373-idb/static/jquery-2.1.3.min.js

Bootstrap does not come with everything we needed for styling though, so there exists an option to add a custom CSS file. In this project, the custom file is called styles.css. There are a lot of template examples and documentation online for Bootstrap. Good websites to start with are at http://www.w3schools.com/bootstrap/ or http://getbootstrap.com/getting-started/.

Unit Tests

Before the tests are run, the setUp() function creates a new Sqlite database and fills it with the tables inside our SQLAlchemy data model, which is contained inside models.py. Sqlite is used instead of Postgres to allow the tests run from any machine without any restrictions of connection. This is due to the fact that Postgres requires a connection in order to be accessed unlike Sqlite, which creates a single local file with the data and can be easily accessed. A lock is also created to prevent any concurrency problems within the test cases. The unit tests handle four different types of cases within the data model and SQLAlchemy ORM system:

1. The first case type tests our ability to write information to the database. Using SQLAlchemy we insert a new entry to the table and check the quantity of rows in the table to make sure it was added.
2. The second case type tests our ability to read data from the database. This test inserts a new row with a specific name that we then pull in order to check that it exists.
3. The third case type tests our ability to delete specific rows from the database. To test this we insert a new row into the desired table, then assert that the data has been inserted. The row is then queried through SQLAlchemy and deleted. At the end we assert that the data does not exist in the database anymore.

4. The last case type tests our ability to query other data and data types from the tables. For this case we insert a row with more than one single variable and data type and assert that we are obtaining the same data we submitted and in the same format.

These four case types are run over all our tables to test if they are being built and accessed correctly.