IDB Project

CS373 Spring 2014

Group Name: #yoloswag

Group Members:

- Sergio Escoto
- Comyar Zaheri
- Josh Gutierrez
- Steven Wilken

Introduction

The #yoloswag team's main goal is to provide a presentable web-based collection of mythological data that spans across multiple cultures - -e.g. Greek, Roman, Norse, Celtic, etc. We have collected mythological data from a variety of cultures, making it easy for our visitors to browse and search for the most relevant information they need.

Our motivation for this topic was our realisation that the modern world is becoming increasingly multicultural. The United States was founded on the basis of immigration, so we felt that it was especially relevant to provide a platform through which users could learn more about other cultures. Every culture has its own rich storytelling traditions, so why should one restrict themself to only their own stories? There is a lot we can gain from learning about each other, and most importantly, we can foster a better sense of harmony and co-existence.

Another factor in our choice of topic was the fact that there are numerous enormously

entertaining legendary tales that we may have missed out on simply because we were born in a different culture. We hope that our Mythos DB can provide both a learning experience and a certain amount of entertainment value through enabling easy access to a variety of cultures' wealth of epic stories.

An example use case would be an 8th grade student who has an assignment due for his world history class on Ancient Greece. Upon arriving at our website, he would merely have to click on the Cultures tab in our navbar to see a list of the cultures available on MDB (Mythos database). Upon clicking on the Greek culture entry, he would be presented with a variety of related Stories and Figures, as well as any other similar Cultures (Roman culture, for example).

Another possible example use case would an average college student is browsing the web for entertainment and he or she stumbles upon our Mythos database and is immediately enthused by both its contents and presentation. This average college student could be entertained for hours by the epic tales and myths of cultures spanning across the globe.

Much like IMDB, our structuring of the three categories allows a user to quickly and easily make links between data entries, even if they are in disparate categories. Just as a movie entry on IMDB would allow me to go to the entries for actors and crew involved in the film, MDB would enable easy access to mythological figures involved in certain stories, and vice versa. We hope that a user would further enhance their learning by going from story to story in a somewhat organic manner.

Design

RESTFUL API

The following figure illustrates our basic RESTful API structure.

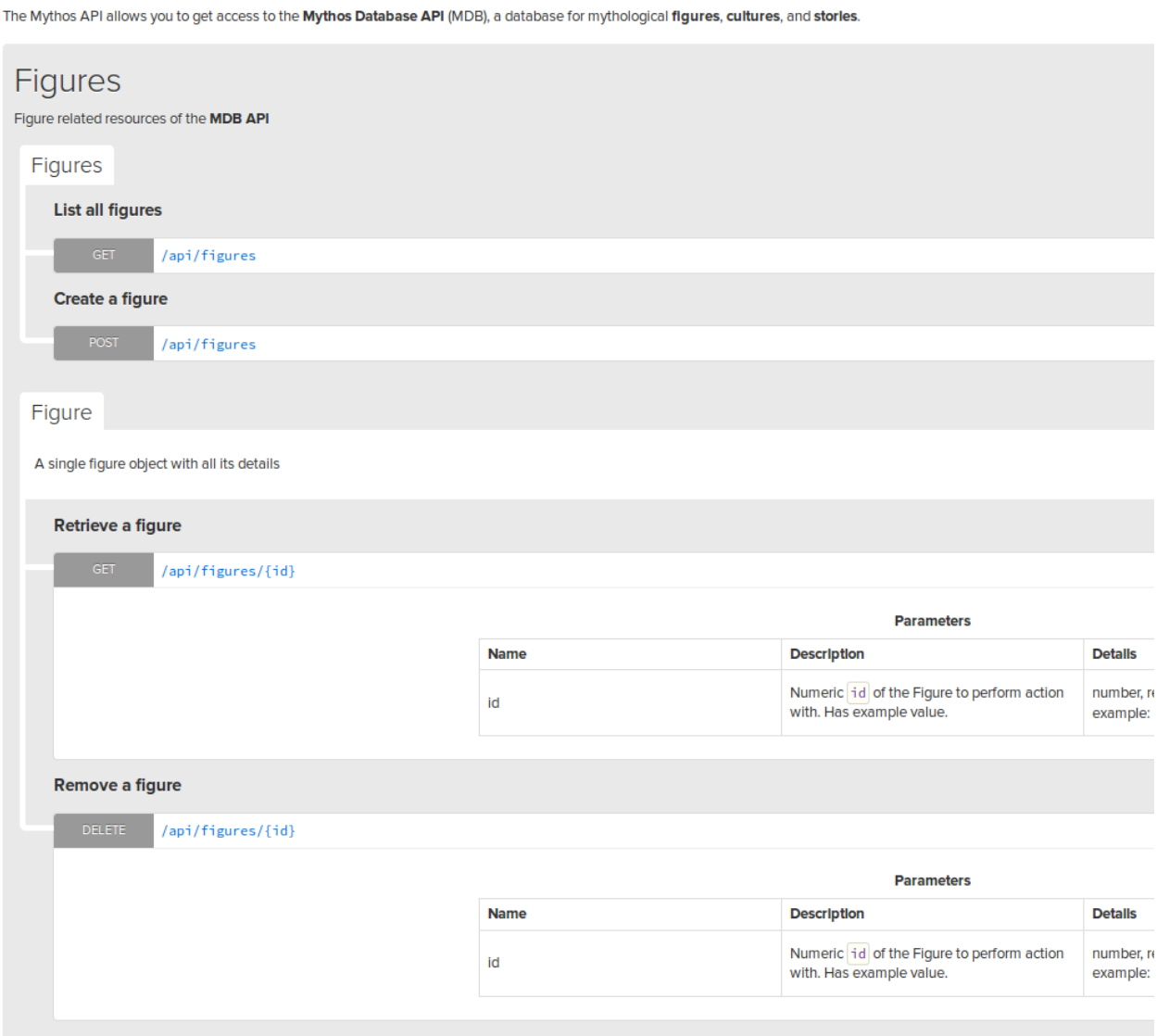


Figure 1. Our Apiary RESTful API

In order to conform to the specifications of the project, we implemented a RESTful API. More specifically, our API adheres to the four methods used according to the HTTP standard, namely GET, POST, PUT and DELETE. The main benefit of implementing a RESTful API for our web application is that any client or server can interact with it through HTTP requests without knowing its underlying structure. Any client can fetch any information from our database by performing a simple GET request such as "GET /figures" and our application will return a resource (e.g. a JSON resource) representing a list of all figures in our database. Our POST request implementation would allow any client to create a new entity in our database, e.g. a client performing a POST /figure with a correctly formatted resource would add a new "figure" to our database. Similarly, a PUT request should update an existing object within our database as long as the client performs the correctly formatted request. Lastly, a client can perform a DELETE request to remove the specified object from our database as long as the request conforms to our API (e.g. "DELETE /figures/4" should delete the figure with a primary key of 4 from the database, if it exists).

We structured the website backend so as to make the URLs of entries correspond to their ID attribute. Greek Culture, for example, is accessed at "cs373-mythos.herokuapp.com/cultures/1", as it corresponds with the first Culture entry in our database. The data is first separated into the three separate categories, then further separated into IDs, so a Culture and a Figure may both have the ID 1, but will point to separate entries as they will be displayed at "/cultures/1/" and "/figures/1/", respectively.

This hierarchy is mirrored by both our API frontend (which will be accessible at "/api/") and our HTML frontend. This means that "/cultures/1/" and "/api/cultures/1/" will return the same data, but formatted in user-readable html in the former case, and JSON in the latter. The hierarchy further allows us to present a full list of entries through the API (and the HTML frontend as well). For example,

Figure 1 shows that it is possible to make a GET request on both "/api/figures/" and "/api/figures/{id}/" (where {id} is some number corresponding to an entry in the figures category). The first request returns a list of all figures, with a small amount of truncated attributes for each entry in the list, while the second returns the complete entry for the figure that was requested.

We made the decision to structure our URL scheme in this way for two reasons. The first is that it provides clarity for a layman user of the website and for a developer taking advantage of our API. The user can quickly and easily construct URLS, both programmatically and in person, something that may have not been possible if we chose a less intuitive URL scheme.

The second reasons we chose this URL scheme is to allow searching of our entries based on attributes that are easier to understand than our numbering system. Because our numbering system is (for the most part) arbitrary and based on a FIFO system (i.e., entries inserted into the database first are given smaller numbers), it would not be feasible for a user or developer to navigate the database given only that information.

So a user who wanted to look into the Greek god of the Underworld, despite having forgotten his name, could request a list of all figures, of which the data

```
{
      ...
    "name": "Hades",
    "kind": "God",
    "biography": "Hades was the ancient Greek god of the underworld."
      …
}
```

would be present. Searching the provided data for the keyword “underworld” would give the user the ID of the entry they were looking for, in this case 1. Now that they have this information, all they would have to do is navigate to “/figures/1” (or “/api/figures/1”) to get the data they were after.

JSON

Because we implemented a RESTful API, our API returns JSON to clients requesting data from our database. In order to create the requested JSON, we had to take into account how the Media objects for each Figure, Culture, and Story object would be represented in the JSON. We decided that we could represent all the Media objects that map to a specific Figure, Culture, and Story as an array in the JSON object. From a design standpoint, this makes sense because given the JSON object for a specific Figure, Culture, or Story object, you could easily iterate over the list of Media objects that map to it. Rather than representing the entire Media object in the array, we decided to simply list the resource URI.

We also had to take into account how we would represent the many-to-many relations between all of our objects. For example, the Figure model has multiple many-to-many relations, so we had to decide how we would want to represent those relations in our JSON object. We decided that the simplest representation of our many-to-many relations in our objects was a list. So, for a many-to-many relation, we simply created a list of URIs of the objects being mapped to. For example, if Athena maps to Zeus and Atlas, then the *related_figure* key for Athena would map to a list of URIs representing Zeus and Atlas.

Tastypie

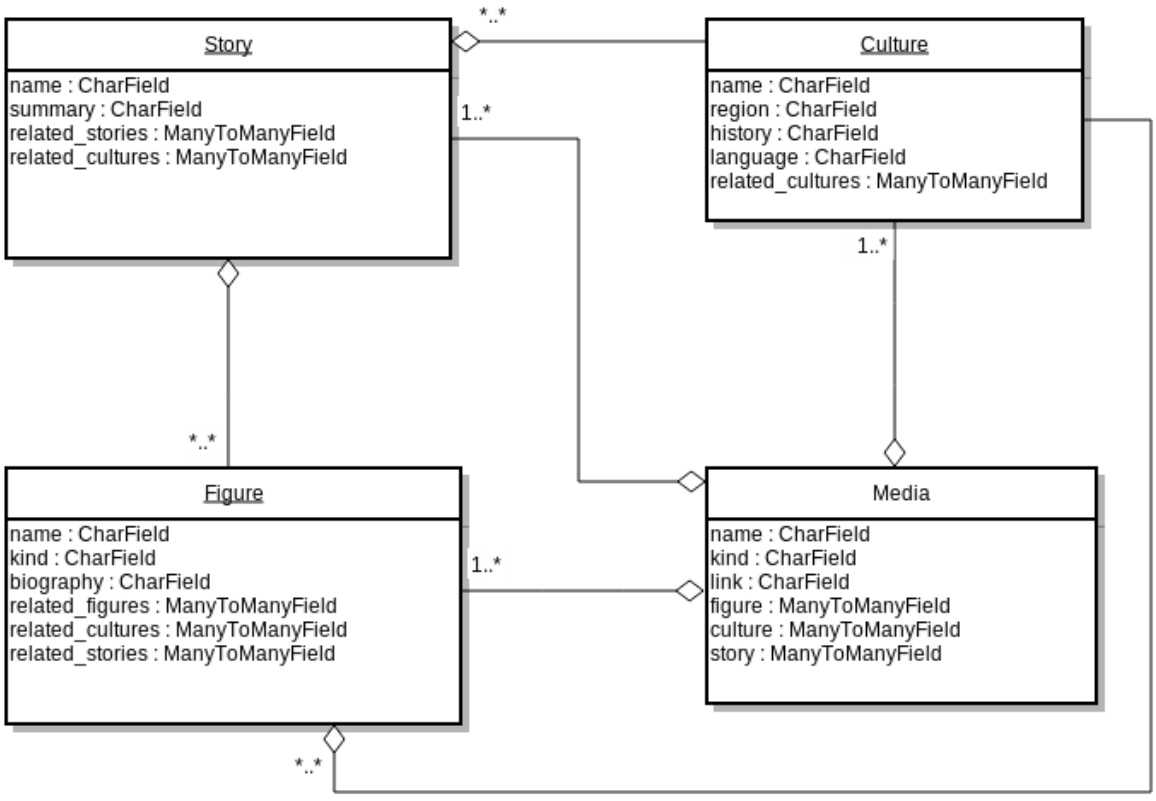
We decided to implemented our RESTful API using the Tastypie module. This module is a webservice API framework for Django. It provides a customizable abstraction for creating REST-style interfaces. This module made implementing the API very easy as it automatically takes care of serialization, server requests, and mirrors our models. The only thing that had to be setup to get the module to work was setting up foreign keys and many to many associations between the models and what set the API would be querying for specific models.

After implementing TastyPie for our RESTful API, our web site allows queries to our database based on the URL scheme described earlier. So, if a user wanted to query our database for information regarding Athena and knew that the Primary Key for Athena was "1", they could simply make an HTTP request to `http://cs373-mythos.herokuapp.com/api/figures/1`. The resulting JSON object for that request would like this:


```
{
        "id": 1,
        "biography": "In Greek religion and mythology, Athena or Athene (/əˈθiːnə/ or
/əˈθiːniː/; Attic: Ἀθηνᾶ, Athēnā or Ἀθηναία, Athēnaia; Epic: Ἀθηναίη, Athēnaiē; Ionic:
Ἀθήνη, Athēnē; Doric: Ἀθάνα, Athānā), also referred to as Pallas Athena/Athene (/ˈpæləs/;
Παλλὰς Ἀθηνᾶ; Παλλὰς Ἀθήνη), is the goddess of wisdom, courage, inspiration,
civilization, law and justice, just warfare, mathematics, strength, strategy, the arts,
crafts, and skill. Minerva is the Roman goddess identified with Athena.\nAthena is
portrayed as a shrewd companion of heroes and is the patron goddess of heroic endeavour.
She is the virgin patroness of Athens.",
        "kind": "Olympian",
        "name": "Athena",
        "related_cultures": ["/api/cultures/1/", "/api/cultures/8/"],
         "related_figures": ["/api/figures/2/"],
```

```
        "related_stories": ["/api/stories/1/"],
        "resource_uri": "/api/figures/1/"
}
```


Django Models



Example UML diagram depicting the relationship between our chosen categories.

We chose three categories for our models: Figures, Stories, and Cultures. We also introduced another model, the Media model, as an abstraction that would allow us to optionally associate instances

of our three main categories (Story, Culture, Figure) with many Media objects such as images, videos, links, etc.

Media Model

The Media object has the attributes *name*, *kind*, and *link*. Each of these attributes is a CharField with a maximum length of 200 characters. We decided to limit the length of these character fields in order to save storage space in our final database.

Along with the *name*, *kind*, and *link* attributes, the Media object also has three ForeignKeys that relate a Media object to a figure, culture, and/or story. We decided to make each of these ForeignKeys optional because a Media object does not necessarily relate to a specific figure, culture, *and* story. For example, if we had a Media object that represented an image of Zeus, it's possible for that image to not relate to a culture or story. Because of this, making the *figure*, *culture*, and *story* ForeignKeys optional made the most sense.

Figure Model

The Figure object has the attributes *name*, *kind*, and *biography*. Each of these attributes is a CharField with a maximum length of 200 and 10000 for *name*/*kind* and *biography*, respectively. Just like we did in our Media object, we decided to limit the length of these character fields in order to save storage space in our database. However, we decided to increase the maximum length of the *biography* attribute because of the potential for very long figure biographies.

In addition to the *name*, *kind*, and *biography* attributes, we included many-to-many relations from Figure to Culture, Story, and Figure. This allowed for us to relate a given Figure object to other

Figure objects, as well as Story or Culture objects. For example, if we had a Figure object representing Athena, we could relate Athena to another Figure object, such as Zeus, as well as relating Athena to a Story such as “The Birth of Athena.”

Story Model

The Story object has the attributes *name* and *summary*. Both of these attributes are CharFields with a maximum length of 200 characters and 10000 characters, respectively. Just as in our other Model objects, we decided to provide a maximum length in order to minimize the amount of space each Story object reserves in our database. We decided to increase the maximum length of the *summary* attribute because of the potential for very long story summaries.

In addition to the *name* and *summary* attributes, we included many-to-many relates from Story to both Culture and Story. This allowed us to relate a given Story object to other Story objects as well as Culture objects. For example, the story “The Birth of Athena” originates from both Greek and Roman mythology, so it would make sense to allow a relation between the story to multiple cultures. Also, the story “The Birth of Athena” may also relate to other stories including Athena, so allowing a relation between those stories makes sense as well.

Also, because of the many-to-many relation we defined in the Figure model between Figure and Story, any particular story can also relate to many figures. For example, if the story “The Birth of Athena” involves the figures Athena and Zeus, the story can relate to both the Athena and Zeus Figure objects.

Culture Model

The Culture object has the attributes *name*, *region*, *history*, and *language*. Each of these

attributes is a CharField with a maximum length of 200 characters, except for *history* which has a maximum length of 10000 characters. Just as in our other Model objects, we decided to provide a maximum length in order to minimize the amount of space each Story object reserves in our database. The *history* attribute has a greater maximum length than any of the other attributes in order to accommodate for potentially long histories.

In addition to the *name*, *region*, *history*, and *language* attributes, we defined a many-to-many relation between Culture and Culture. This allows us to relate similar cultures to one another. For example, Greek and Roman cultures are very similar to one another so it makes sense to relate them to each other. However, Roman culture also shares some similarities with Norse culture so we needed to allow for a many-to-many relation between Cultures.

Also, because of the many-to-many relations defined in Figure and Story to Culture, every culture can be related to multiple figures and stories. For example, the Greek culture could relate to the figures Zeus and Athena, as well as the stories “The Birth of Athena” and “Trojan War.”

Populating the Database

Wikiparser.py

In order to populate our database, we first had to retrieve all of the information we needed to use. To do this, we used a python module called *wikipedia*. The python module *wikipedia* is a wrapper around the *BeautifulSoup* python library, used for scraping web pages. We used the *wikipedia* module to scrape all the information we needed.

We hard-coded the names of the pages we wanted to parse information and ran the scraper. Once a particular page’s information was scraped from Wikipedia’s website, we pulled out each piece of

information we needed. For every object we wanted to insert into our database, we constructed a python dictionary to contain the data in a format that matched the requirements of Django fixtures.

Using Django Fixtures

Once we had all of our data, we wrote it all to a file in JSON format. The JSON consisted of an array of dictionary objects. Each dictionary object contained a key “pk” that mapped to an integer representing the primary key of the object in its corresponding database table. Along with the primary key, each dictionary object contained values for each of the fields of a Model object. After writing the data to file in JSON format, we simply had run a command for Django to load all of our data into our database automatically.

Django Templates & Bootstrap

Overview

The front-end for this project extensively used the Twitter Bootstrap library to create the look and layout of our pages. We set up Django to dynamically serve this information using its template system. Django templates give us the option to serve the same shared HTML content to any number of pages according to their category. All article entries share the same template that fetches and displays a text description, any relevant images, videos, and links.

For each category, the site has a page linking to a list of objects of that category. For example, coming from the main index page, the user can click on the Figures link and be presented with a list of all figures contained within the database. The page listing for all three categories share the same template. Within this page listing, the user clicks on the name of a figure, a culture, or a story and is then

presented to an article page which shares the same base template that fetches the article name, text description, and any media associated with that article. This design decision simplifies the management of our database, making it scalable in terms of adding new content, deleting content, or updating content, where Django takes care of displaying or deleting the Figure, Story, or Culture articles since Django automatically generates, deletes or updates the needed links and article pages for us.

Displaying Specific Models

In order to display the correct information for any given URL, we needed to query our database and retrieve the relevant Model object. Because our URL scheme uses the primary key of objects in our database tables, this was a simple process. Given a primary key for a specific Model object, we attempt to get the object from our database using that primary key.

If the object exists in our database, we construct a dictionary mapping variable names from our HTML templates to the values of the Model object. For example, the *title* variable in our HTML template could be set to the value of the *name* attribute of a Figure object. Once the dictionary is constructed, we simply pass it off to Django which renders our HTML pages for us automatically.

If the objects does not exist in our database, the "get" query will throw an exception. In that case, we catch the exception and render a "404 not found" page.

Displaying Listing of Models

In addition to view specific pages, it's also possible for a user to view a listing of all pages of a particular type. For example, a user can browse the list of all Figures on our site. In order to display this list, we simply use the ManagerObject for a Model to retrieve the set of all objects of that type. Once

we have the set of all objects of a particular type, we simply pass that our Django templates which iterate over the set and render the appropriate HTML.

Implementing Search

In order to implement search, we used a python package called *django-watson*. *django-watson* provides a simple-to-use interface for implementing search by allowing us to register our models with its search system.

Before we could use *django-watson*, we had to do some initial setup. First we synced our database using *python manage.py syncdb*. Then, we initialized *django-watson* by running *python manage.py installwatson*. Because we were implementing *django-watson* on a system that already contained data, we had to also run *python manage.py buildwatson*. From here, using *django-watson* was very easy. We implemented a search template page in order to show search results and we made the search bar located in the navigation bar automatically redirect to the search page on enter, passing the typed search query as a GET parameter.

In our *views.py*, we just grabbed the GET parameter from the URL and queried our models using *watson.search(<search query here>)*. *django-watson* simply returns the results of the search, and we simply pass that to our search page template.

Implementing Tests

Using a combination of Django's own testing framework (specifically the django.test.TestCase) and Python's urllib and json modules, our team wrote a set of unit tests to verify the correctness of our

own API implementation. Our unit tests are located under project_folder/idb/tests.py. We tested getting a list of all figures, cultures, and stories; getting information on a specific figure, culture, and story; deleting a specific figure, culture, and story; putting a new figure, culture, and story, and posting a new figure, culture, and story..

Our design decision of having the URL correspond to a resource (e.g cs373-mythos.herokuapp.com/category/instance) made testing extremely simple. We made test cases for each of our categories' models. Each of our test cases made a RESTful request to our Apiary API which performed a given action. This API documentation contained expected responses for what should be returned from our server.

For each of our tests, we attempted to use our own API the same way any other developer would. So, in the cases that our API returned JSON-formatted data, we would parse the API's response, create a JSON object (a python dictionary or a python list depending on the specific request response), and compare the content of that newly created JSON object to the expected python dictionary or python list object.

In order to make the requests, we used Python's urllib module to make HTTP requests to our server. For each of the RESTful methods (GET, PUT, POST, and DELETE), we simply set the request headers and specified the method and used `urlopen` to send the requests to our server. We decided to implement our tests in this way because this is how we would expect any developer trying to access our API to do it.

An example of a test for a GET request for a figure looks like this:

```
request = Request(self.url+"/api/figures/1/?format=json")
response = urlopen(request)
response_body = response.read().decode("utf-8")
self.assertEqual(response.getcode(), 200)
```

```
response_data = loads(response_body)
for key in response_data:
        if type(response_data[key]) == list:
        response_data[key] = sorted(response_data[key])
self.assertEqual(expected_response, response_data)
```

User Interface Enhancements

For our final phase of the project, we updated the user interface of the entire site. In order to use static files with Django, we had to set-up the *static* directory in our project directory and establish the path to that directory in our *settings.py*.

Navigation Bar

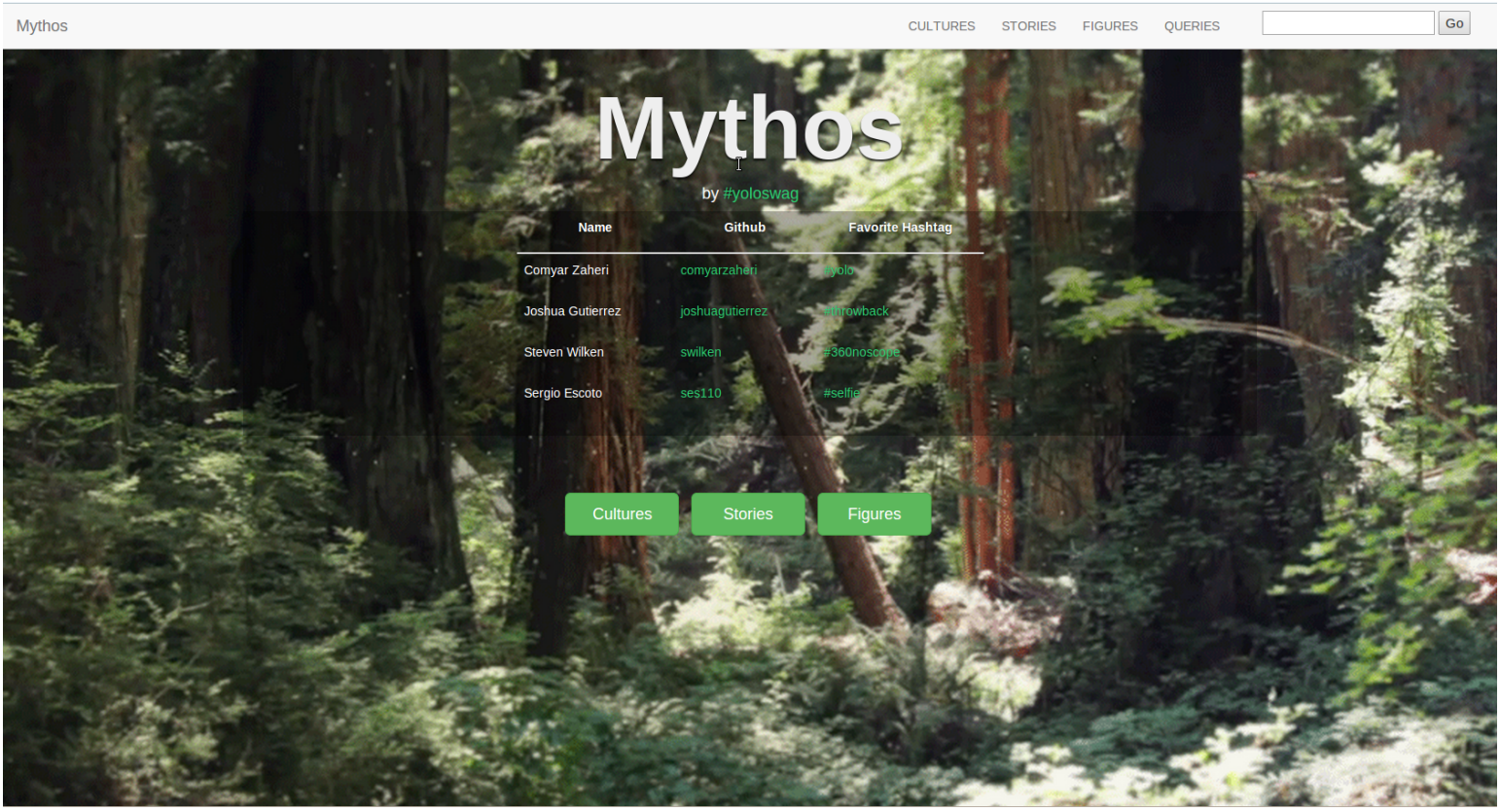
We decided to keep the navigation bar mostly as-is because we felt that it was the most intuitive way to navigate through all of our pages. The one addition we made to the navigation bar was adding a search bar. Putting the search bar in the navigation bar was the best decision for our website because it allows a user to search all of our pages from anywhere on our site.

Home Page

Our home page before the final phase of the project was very bland, so we decided to spice it up with a nice background image and some better colors. We increased the font size of our project title and added shadow to emphasize it. Beneath the title, we updated our table of project members by changing the font sizes, adding shadow, and adding a translucent black overlay beneath the entire table to help emphasize the information. Links were changed to be a nice green color in order to match our

background image, and three buttons linking to the cultures, stories, and figure pages respectively were added to the homepage to encourage a user to begin exploring.

Here's a screenshot of our homepage:



Backgrounds

We wanted to make the backgrounds of all of our pages on the site interesting, yet subtle enough not to distract from the content. We decided to use *Cinemagraphs* for the background of our site. Cinemagraphs are perfectly looping GIFS where only a small portion of the image is actually animating. Using Cinemagraphs for the background of our home page and list page brought our pages to life.

GIFs usually carry with them a significant performance hit, but Cinemagraphs avoid this having each frame only contain changes from the previous frame, whereas normal GIFs are a series of

complete images. Because of this, we knew that using Cinemagraphs would not result in a significant increase in our loading times.

For our pages regarding a specific item, we decided to use a blurred version of the cinemagraph used on the item's corresponding list page. We felt this showed a clear relation between an item and its category and provided a user with spatial clues regarding where they were on our site.

Interesting SQL Statements

I. Get all figures who are in Greek and Roman mythology

We decided this would be an important query to use since there are some figures who are in both Greek and Roman mythology and it often becomes confusing to figure out which Roman figures have a greek counterpart or vice versa. This is a common question that comes up when studying these cultures so we wanted to have a quick way to clarify confusion on this.

II. Cultures with more than 1 story

Given that our database is not that large at this point in time, we also were interested in finding out which of the cultures on our site contained the most robust information. It could be useful for users, but it is especially useful for us when designing the website because we could discover where we are lacking in information.

III. Find all stories that have a relation between at least one figure, one culture, and one other story

This query can be useful if we needed to discover which stories have the most related information. It lends itself to a google-like ranking where stories that have the most related items become more prominent.

IV. Return count of figures, stories, and cultures related to each culture

We use this query to get a comprehensive view of each culture quickly. It gives the basic count of figures, stories, and cultures related to each culture without having too much detail.

IV. Figures that are related to only 1 culture

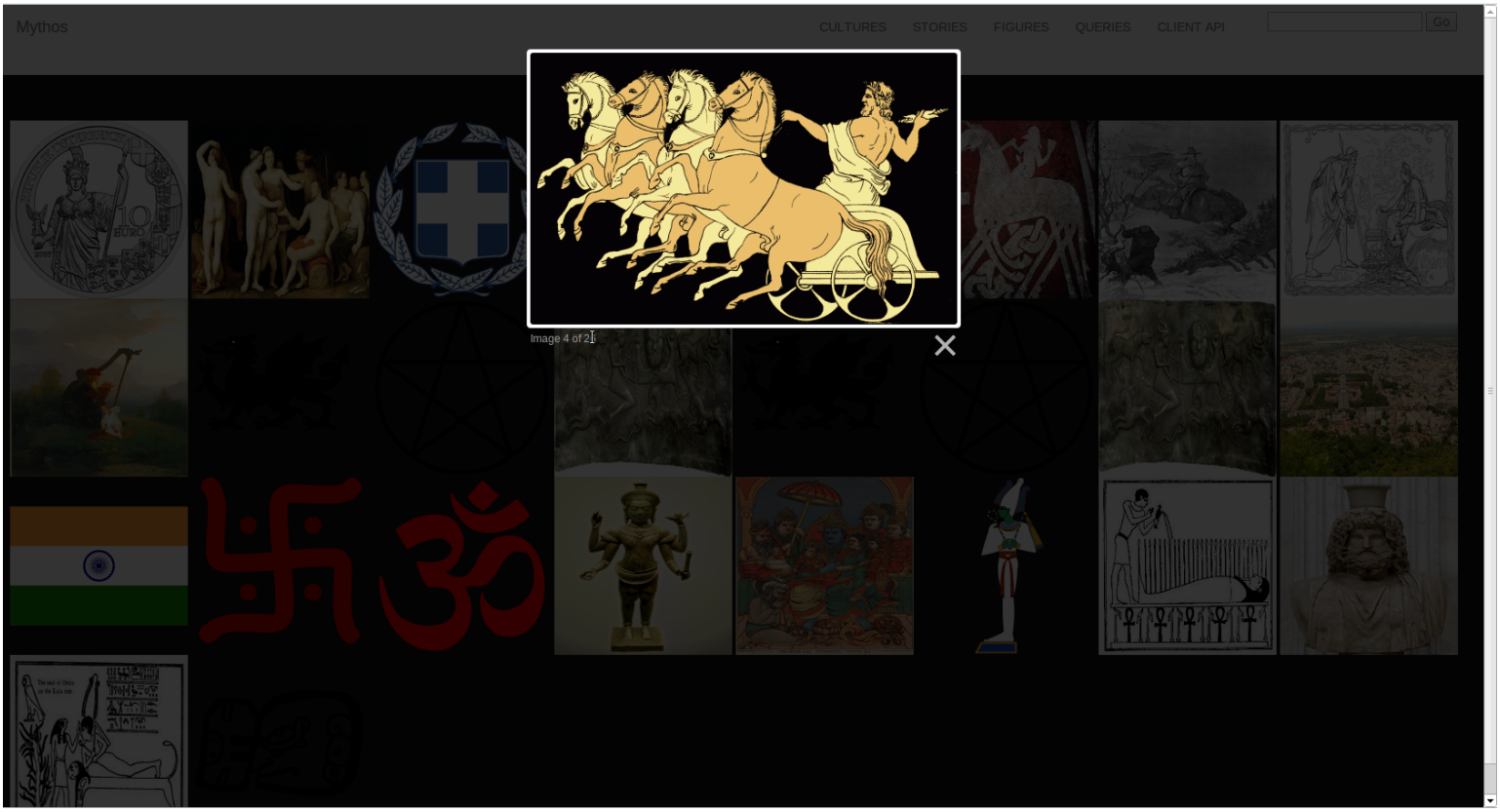
This query achieves a similar goal to our first query. Instead of picking specifically at Greek and Roman mythology, a common case where there is a figure related to more than one culture, this query will find figures only in one culture; you might consider these to be "unique" figures. In our database, it is more common for a figure to only be related to one culture, so we receive many results from this query.

Client-Side API Usage

For our client-side usage of our API, we developed a mythology gallery! The mythology gallery uses the GET API call to get the URLs of all image Media objects in our database and then displays them as thumbnails on our website. To view an image at a higher resolution, you can simply click on its thumbnail.

To make the RESTful calls to our API, we used jQuery's getJSON function to request up to 95 Media objects from our database. With the given JSON, we grabbed all of the image Media objects and appended the images onto our HTML DOM.

To display the higher resolution images once a user has clicked on a thumbnail, we implemented a javascript library called *Lightbox*. *Lightbox* allows for beautiful presentation of images in series. The following is a screenshot of *Lightbox* in action in our mythology gallery!



Conclusions

In the final phase of the project, we did a complete UI overhaul on our site, demonstrated the use of our client-side API, performed interesting SQL queries on our database, and implemented search on our website.