IDB Project

CS373 Spring 2014

Group Name: #yoloswag

Group Members:

- Sergio Escoto
- Comyar Zaheri
- Josh Gutierrez
- Sidhant Srikumar
- 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.

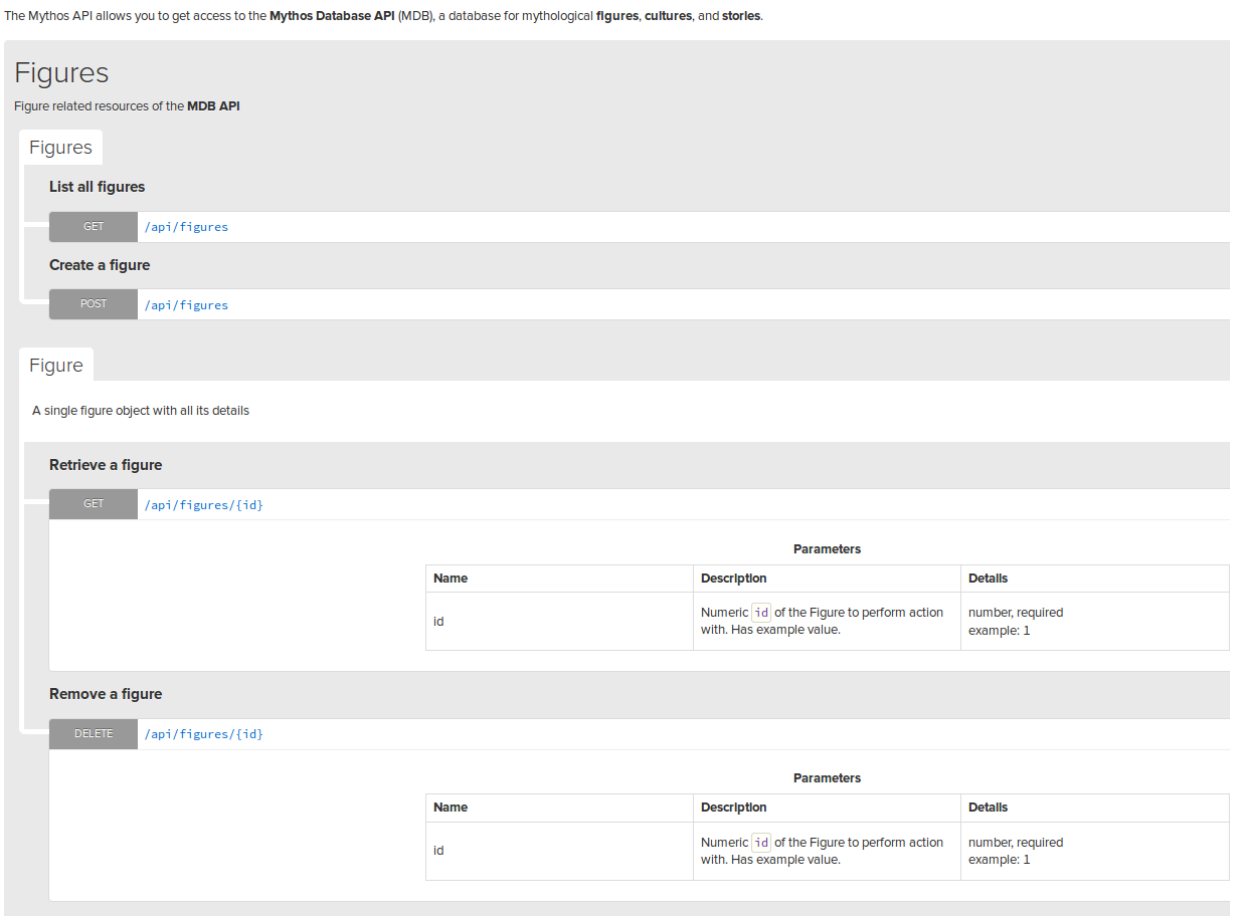


Figure 1. Our Apiary RESTful API

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 also 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.

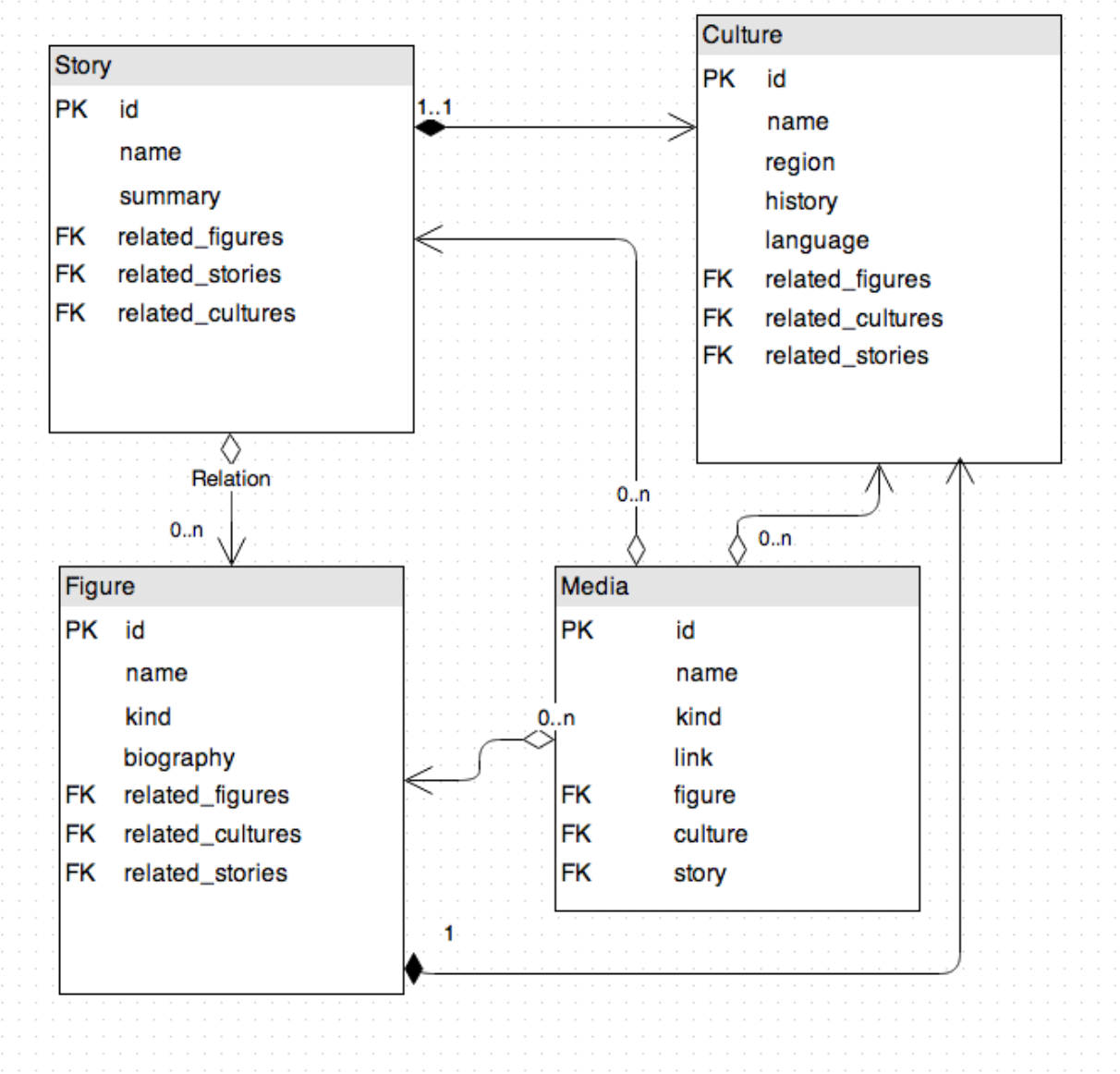
So a user who wanted to look into the Greek god of the Underworld, despite having forgotten his name, could request a list of all figures, of which the data


```
{
    "id": 1,
    "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.

We plan to allow this system to be more extensible in future phases of the project. For example, we plan to allow the user to combine categories for more precise searching of the data. This would allow a user to, for example, request a list of figures appearing in a certain culture or a list of figures involved in a certain story.

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

The Figures model has a many-to-many relationship to Culture, since we made the assumption

that the same figure could be related to many cultures. For cases where there might be analogous figures (e.g. Zeus in Greek mythology and Jupiter in Roman mythology), we will assume that they are separate instances and will have their own entries in the database. Lastly, Figure also has a many-to-many relationship with Story, since a figure can be associated with many historical events, wars, etc.

Media has a many-to-many relationship with Figure, Story and Culture, since any particular instance can have many media instances within its corresponding article page, and one media instance can be associated to various Figure, Story, or Culture instances.

The last relation between Culture and Story is many-to-many since a particular culture can have many stories but a story can only belong to one culture.

We chose the attributes for each category we thought were most relevant for a user browsing our website as an online encyclopedia. All categories require a name. Stories require a text description. Cultures require a region, and may have a language and/or a history attribute. Figures require a Kind (a Hero, God, Mortal, etc.) and may have a biography.

We also introduced three attributes to these categories: related_figures, related_stories, related_cultures, that we thought might be useful for any visitor who visited the article, who might also be looking for a figure, another culture or story article that is related to the original figure due to shared or similar stories and/or cultures.

The Figure Model

We decided to make a Figure instance contain a name, a short biographical description, and

three many to many attributes: to associate a figure instance to any related figures, to any other stories that this figure instance has been in, and to other related cultures (whenever a figure has an analogous figure in another culture). We decided on these attributes since we wanted to introduce such features where, upon visiting an article, the user can click on links leading to articles of relevant figures, stories, or cultures.

The Story Model

Stories denote notable mythological events such as wars, holidays, historical events, etc. An instance of a story has a required name attribute, a text description of the event, and three optional foreign keys similar to the Figure model, linking to relevant stories, figures, and cultures.

The Culture Model

A culture has a required name attribute, a region it is centered in, a short historical description, a primary language, and three attributes to associate a culture instance to any figures from that culture, any related cultures (whenever a culture influenced another and vice versa), and stories under that culture.

Media Model

The media model is used for any media that is associated with a specific figure, story, or culture. Each instance of a media has a required name to be displayed, the kind of media being used, an underlying link to the media itself, and three attributes to associate the media to a figure, culture, or story.

Django Templates & Bootstrap

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. Although the minimum requirements permit us to use static pages on phase 1, we went ahead and hard-coded dummy data within the models.py file in the form of the class P1_Models, which Django would then display that data as if it were fetched from a database. Despite this being a time-consuming process, we will save a lot of time once we begin linking our Django application to a database for the next two phases, since the remaining steps would simply consist of scraping Mythology data to populate our database.

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.

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. While we have yet to set up Django with a database, we set up our unit tests by creating an array of dummy data in JSON format. 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; and putting 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.

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.

Conclusion

In the next two phases, we will continue to add more features to our project as we implement a database and utilize the Django templates we created in a more formally correct manner than their current static population.