Software Engineering Internet Database Project
Technical Report

Group Name | Flask Me Anything

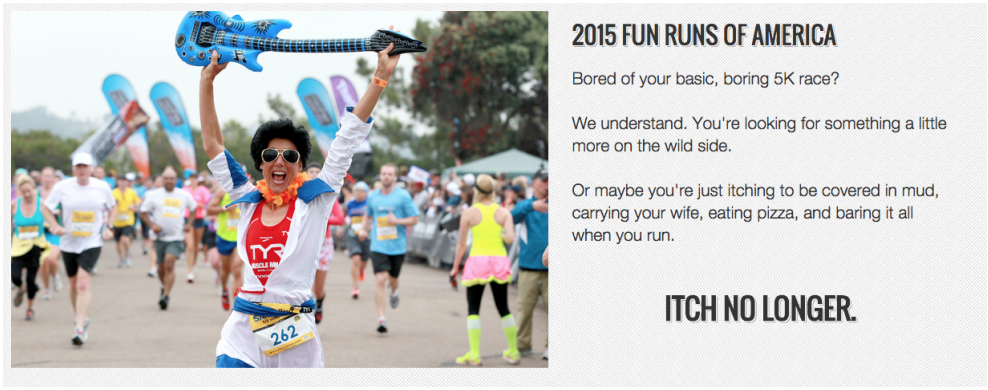
Alyssa Williams | Phase I Leader
Kurt Bixby | Team Member
Kevin Byers | Team Member
David Aguirre | Team Member
Edgar Treto | Team Member
Hector Garcia | Team Member

I. Introduction

I.A. The Objective

Races and running are not for everyone. The pain can be too overwhelming, the sun can be too harsh, and the intensity can just be too much. Those who are serious about their running train to attend marathons and walk away from the race with a sense of accomplishment. However, those who do not want to partake in miles of fatigue, have not trained, or do not find pure running of any interest are at a loss - how can they enjoy the benefits of exercise when it does not naturally please them?

America has shown an upswing of silly races popping up all over the country called “fun runs.” These fun runs are not your common marathon - they find ways to be quirky, different, and entertaining to the average Jane. These crazy competitions span in their ideas: running in your favorite costume, running then eating as fast as you can, running up the stairs of the



Empire State Building, and even running nude. We found these races to be an amazing way for die hard athletes and casuals alike to exercise and enjoy together in ways never experienced before, and therefore, made it the topic of our project.

I.B. Use Case

This site's intended audience is for hardcore athletes, casual runners, families, and children alike. Our group tried to design intuitively by including obvious navigation bars, highlighting the current position whenever possible, and remaining clutter-free. By picking a bootstrap that is able to handle a lot of content, we were not inhibited by the design and were free to include information we felt was pertinent. In many instances, the bootstrap's features were even helpful in displaying information without adding clutter - like slideshows and carousels.

Fun runs are great events to encourage those looking to be active but wanting alternatives to the core long-distance run; therefore, our design has to persuade such people to check out some runs by displaying information in an understandable, frustration-free hierarchy. The site sorts by our three pillars: the runs themselves, the types of challenges that one will face in these sorts of runs, and the common themes that these runs take on. By building this database, our team provides useful, entertaining information on these hilarious fun runs in a way that is convenient, user-friendly, and informative.

I.C. Workload Separation

A large part of working on a sizable development project is learning how to work with a team in an efficient and peaceful manner. Our difficulties were mainly in sharing a common codebase amongst so many contributors, but by utilizing GitHub, our team was able to resolve the occasional merge issues quickly because of our constant communication.

I.D. File Structure

Our repository exhibits natural hierarchy with app/ containing all of the relevant files to run our website. The templates folder includes the html files, static/ includes css/javascript/imgs/angular, and the app itself is funruns.py. We have models.py as a mock up of our future Flask Models; we did not implement a database this phase - instead we have json in api_helpers.py and our RESTful API calls in , but thought it would be a good exercise to think ahead.

I.E. Important Commands

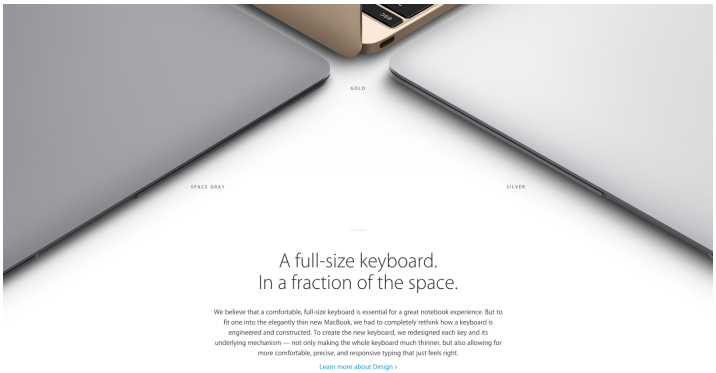
The website will be running from the virtual machine, served from Rackspace. However, this subsection highlights how we are able to run our app. To run the app through python, run `$ python3 funruns.py`. This will start the app and the website will be accessible on 104.239.130.43:8000. To use the web server, start nginx with `$ sudo /etc/init.d/nginx start` then execute `$ sudo uwsgi --http-socket :8000 --plugin python --callab app --wsgi-file funruns.py` to start uWSGI and serve our web-app. More explanation on these tools are in Section II.

II. Implementation

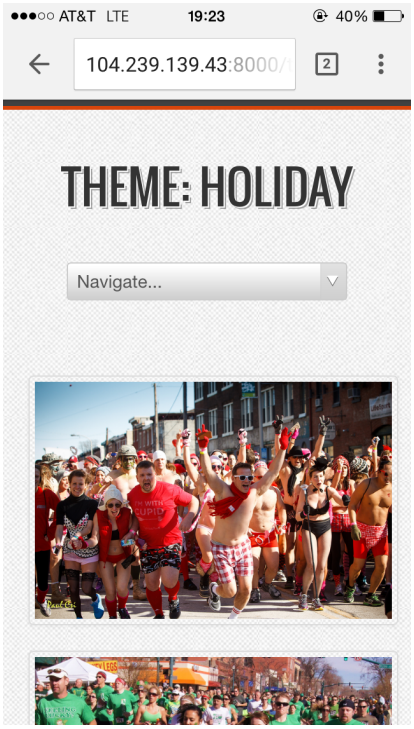
II.A. Bootstrap, Responsive Design, and Usability

The user interface is a core part of the end experience with our website. The final objective is to elevate the user's interest in exercising and getting involved in these fun runs as a creative form of exercise. With an end goal in mind, our team researched many bootstrap themes in the effort of finding a clean, user-friendly design. Many do not spend as much energy on

usability as they should - users have been known to become easily frustrated and abandon webpages in a heartbeat. Added features do not necessarily mean added value - if the new features clutter the screen or are invasive to the user, it will take away from the final website. We also tried to avoid the very “bootstrap-like bootstraps” (one page scrollers that have random animations as users scroll down for content) that have been popular, despite their lack of functionality. Unless done correctly (see: http://www.apple.com/macbook/), this overused theme loses its users in the endless scrolling, lack of navigation, and inability to display information larger than the window. It is not ideal for our uses.



Ultimately, we chose the theme Piccolo, as its ability to handle large amounts of data in a single page was appealing. Our implementation using the Piccolo bootstrap theme allows the user to click photos or titles of runs, themes, or challenges and be redirected to the corresponding page with ease. It is also responsive - certain content does not get shown on smaller devices as it clutters the website with unnecessary information. The navigation also changes to a more mobile-friendly drop down menu. In both the responsive and normal versions, the top nav bar, footer, and added javascript was helpful in making our page easy to use



with obvious navigation, javascript that helps users go back to the top, and appealing, simple animations to rotate through content. The nav bar changed into a drop down menu on smaller screen sizes - more friendly for mobile users. The span feature was also a lifesaver as it allows us to resize the content to the viewport without having to implement the sticky configuration ourselves. The spans range from 1 to 12, splitting the main page area into 12 sections and allowing the developer to use an appropriate width by specifying a certain span in a tag.

II.B. Pillars Pages

There are 3 pillars we currently have: Fun Runs, Fun Themes, Fun Challenges. Each has their own landing page, along with 'Locations', which is an extra area of information that may potentially turn into a 4th pillar in the next phase. On each page, there is a photo that responds to roll-over on large devices and a short description. This page and its photos also resize amicably with smaller devices. The Fun Runs tab is great to find a specific race. The themes tab is perfect for looking for information about certain types of fun runs. The challenges tab is the best way to see the common trials one would face in their run. Finally, they could choose the fun locations tab to see all of the places these fun runs occur and hopefully find a race near them or at somewhere they would like to visit.

II.C. Angular vs Jinja2

We decided to serve our pages' content using AngularJS, a web development framework that helps fill inherently-static HTML dynamically. Angular allows us to use our own RESTful API calls, grab the jsonified data, and parse it for easy filling. Initially, we used Jinja2 to fill our pages by pushing information straight through `render_template` in funruns.py, but this felt like bad web design, where our front end is tightly coupled with our back end. We instead opted to

use our own API (which is public for now, but would ideally be private and optimized for internal use) so that when the back end changes to a database - instead of our jsonified dictionaries - the front end does not have to account for the change.

A big problem was that Jinja2 and AngularJS both use double curly brackets to fill in the page with their information. Jinja2 would go through the page, find a double curly brace annotation that was meant for Angular, and would error out, thinking the tag was for it. We had to specify AngularJS to instead use double brackets for the correct behavior from both.

Angular was also able to seamlessly loop through the related runs/themes/challenges IDs and fill in the boxes in the web page appropriately.

II.D. Embedding Media

Some features we implemented to help provide useful information and encourage participation in these runs were Google maps and associated Facebook pages. Google maps provides a frame of reference to the visitor and can help them pinpoint the starting location of the race by interacting with the map. The embedded facebook page for each run highlights that in this social media age, potential fun-runners can follow the run and keep updated with any developments from the race-organizers themselves. These were served via Angular by including a field for each race where the URL necessary to embed these were included.

II.E. Nginx and uWSGI

Setting up Nginx and uWSGI was a huge blocker to the team. While the web server was being configured to serve the funruns.py app for us, fellow team members were unable to see the

changes they made reflected on the website/run the unit tests as executing the usual `$ python3 funruns.py` would take the port that the web server was also trying to use. Eventually, by creating a toy program and configuring to run that, we were able to figure out the exact arguments we needed to successfully serve our web-app.

II.F. Other Main Tools

We used Flask as our main framework to build our web-app. Flask is a micro framework for python that we used to launch our application before we started using Nginx and uWSGI. Using its decorators, we created functions that respond to certain URLs and functions that allowed template rendering with specified IDs. It would then push the ID through to the page with Jinja2 and the .html template page utilized Angular to make a GET request for the appropriate data and fill the page.

III. Design

III.A. RESTful API Overview

The main design tenets of the API were to create a very flexible API, under the RESTful standards, with enough endpoints to allow users to easily access any cross-section of our data without having to do much filtering on their end. This is done to help ensure symmetry across the different branches of the API, and to let a user efficiently and effectively use the API without having to read the documentation.

III.B. RESTful API Levels

The API is structured into multiple different levels, each of which has more options in a hierarchical manner. Currently, there is a root '/' level, which serves as the entry-point; a pillar

level (including ‘runs’, ‘themes’, and ‘challenges’), which acts as the entry-point for each pillar; an ID or object level (such as ‘run index 1’), which allows information retrieval about a particular object; and a related layer, which shows objects in other pillars related to the current object.

III.B.a. Root Level

The root level of the API exists at /api/ and lets users get information about how to use the API. Sending a GET request to this resource returns a JSON object containing the resource URLs for the different pillar API end-points. This level not only structures the API, but acts as a user-friendly introduction to using our API. The details: `{"runs_url": "/runs", "themes_url": "/themes", "challenges_url": "/challenges"}`.

The keys were chosen as such because they are descriptive and informs the user what they are able to do. These could be improved by adding API, entry-point, or another descriptor, but they are acceptable in their current state for Phase I.

III.B.b. Pillar Level

The pillar level is arguably the most interesting level of our API. It was initially designed to simply return the entire collection and then have resources below them that would act as filters. We decided to stray from that initial design because it would have ballooned the API and have been both less flexible and more confusing than the design we ultimately ended up with.

The old design would have had both `/runs/prices/max/700/` and `/runs/prices/min/20/` as resources; those two different filters would have been mutually exclusive. This pattern of mutual exclusivity and API enlarging would have been non-ideal, so we chose to take the 30+ API endpoints and filter them down to a few more flexible ones.

The current design allows for users to essentially query our database and get exactly what they expect. Each pillar has a series of parameters that can be specified when talking to the pillar resource that will filter the list of objects down to just those that match the query. If no parameters are supplied, the API simply returns the entire list. This design allows for more flexible querying, reducing the excess data on the receiving end and letting users be very granular with their requests.

For example, the runs resource can take any combination consisting between a theme ID, a challenge ID, a location ID, a minimum price, a maximum price, a minimum length, and a maximum length. These parameters are all optional, so the end-user can obtain the exact amount of data they want and essentially search our database.

In addition to these queries, these resources can be used to retrieve a single object of that pillar. By simply adding a `/<id>/` to the end of the resource, the resource will deliver the object with that particular ID. This treats the pillar-level resource and its objects as, abstractly, a folder with its own files and folders.

Querying the API is not currently supported as there is a fundamental difference in structure between our current JSON representation and future database representation. Therefore, we are planning for the next phase by creating a more enriched API than we currently support.

III.B.c. Object Level

The object level is the smallest level of API. It exists solely so users can see and traverse the relationships connecting objects from different pillars, e.g. retrieving ‘Challenge’ objects that are related to a specific ‘Theme’.

III.C. Improvements to the API

There is room for improvement and extension upon the API than Phase I needed.

III.C.a. Locations

A useful addition is the locations pillar to our database and API. It would essentially be another full pillar and act similarly to the rest where a user could filter down the locations by a number of criteria. The locations would have a one-to-many relationship with runs and would allow users to find specific runs in their area and find things to do there. The object would contain weather, landmark, and altitude data to let users find cities they would like to run in and activities they could do while they are visiting.

III.C.b. Overhaul Runs Data Structure

As it currently stands, almost all of the data stored in the runs objects are very long strings that have little demarcation, and are not stored in a way that is conducive to filtering, searching, or manipulating. The changes would consist of taking the current data and converting them to dictionaries and lists so they can be used quickly and easily by the users. Changes to be made in next phases:

- Changing the hosts, sponsors, and charities to arrays that contain the lists of those organizations.
- Changing how the dates, addresses, and cities are stored from strings to dictionaries with fields so users can grab just the fields they want, and so we personally can search and query our own data easier.
- Changing distance to be stored as a float in miles which can then be converted to whatever units we would want.

- Changing how the prices for a run are stored. Since there are different prices for different dates, we would need to figure out a way to store that data. We would likely use an array with dictionaries containing the start and end dates for each price, in addition to the price for that range. The dictionaries would be sorted by start date of each range.

III.C.c. Add More User-Friendly Queries Similar To The Root Level

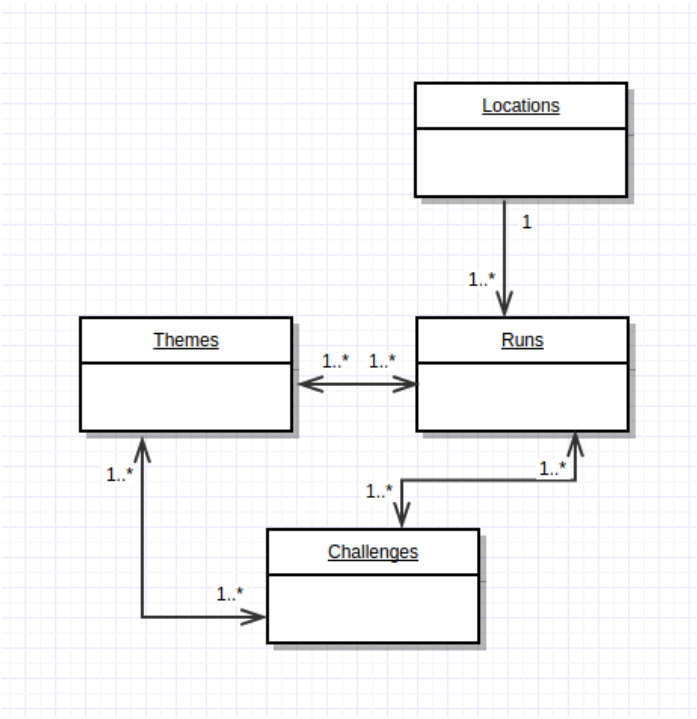
A previous point in the overview of the API stated that users who had not read the documentation could use the API effectively. This has only been implemented for the root level request and not the lower ones. It would be ideal to include requests at each level that would let the user explore what they can do at each level - similar to navigating a file system. The help queries would return a list of resources they can access and what actions or parameters can be specified there.

Unfortunately, this creates a problem regarding the default behavior at each starting node in each level. It is an interesting problem because having the default at each level be a help request would overload their functionality and make the API possibly more confusing for experienced users, while catering to a small subset of people who decide to use our API without reading the documentation.

III.D. Database Representation Through Pillars

For this first phase we wanted to formulate an idea of what the database for our next phase will look like. This is why our flask model shows the objects that will be used to populate our database. We made four classes that represent each of the pillars: “Fun Runs”, “Themes”, “Challenges”, and “Locations”. Each of these pillars have attributes that correspond to the information we need in our webpage. We then created relationships between the pillars. There is

a one-to-many relationship between Location and Fun Runs since every Fun Run has one Location and a Location can have several Run Runs. The next relationship we created was between Fun Runs and Themes, since every Theme can have many Races, and a Race can have multiple Themes; we created a many-to-many relationship and to do this we needed to create a table that has a one-to-many relationship between the two pillars. We also needed many-to-many relationships between Fun Runs and Challenges as well as Challenges and Themes. For this, we also added a table for each of the many-to-many relationships.



IV. Unit Tests

Since models.py is not required, we created unit tests to test our API calls to guarantee that we are delivering what we expect to be delivering. This ensures that we know what the end users are receiving when they consume our API. We were a little confused at first trying to figure out how to test our API, since the decorated functions we would be testing took in requests and

faking those seemed somewhat daunting. What we decided upon was simply sending an HTTP request to our live server and checking that we were receiving what we expected to be sending. We debated different approaches to unit testing, such as using a bash script and some output to show that the tests ran, but we ended up choosing a Python script so it would act as a tests.py.

We unit tested all of our currently implemented API end-points. This was performed by sending a GET request to each resource and ensuring what we returned is what we sent out by comparing the JSON objects.

The API end-points we tests are the following:

- `/api` : The root of our API
- `/api/funruns` : The resource for grabbing collections of Fun Run objects
- `/api/funruns/{id}` : The resource for grabbing a specific Fun Run
- `/api/funruns/{id}/themes` : The resource for grabbing Theme objects that are related to a particular Fun Run
- `/api/funruns/{id}/challenges`: The resource for grabbing Challenge objects that are related to a particular Fun Run
- `/api/themes` : The resource for grabbing collections of Theme objects
- `/api/themes/{id}` : The resource for grabbing a specific Theme object
- `/api/themes/{id}/funruns` : The resource for grabbing Fun Run objects that are related to a particular Theme
- `/api/themes/{id}/challenges`: The resource for grabbing Challenge objects that are related to a particular Theme
- `/api/challenges` : The resource for grabbing collections of Challenge objects

- `/api/challenges/{id}` : The resource for grabbing a specific Challenge object
- `/api/challenges/{id}/funruns` : The resource for grabbing Fun Run objects that are related to a particular Challenge
- `/api/challenges/{id}/themes`: The resource for grabbing Theme objects that are related to a particular Challenge

---

V. Conclusion

Our team thus far has enjoyed creating the web app and working together to make a functional website with an organized user interface. The topic itself is interesting and has an enormous amount of related media, making this project inherently entertaining. The most frustrating parts of this project have been mainly configuration and dependency issues - not the implementation itself. Therefore, having a VM to work on with a working state has been a good method to avoid a scenario where TA's have to set up environments just to look through and execute any files we provide. Our team's Read Me for the GitHub better describes the files we have in our repository and how to run the web app, if interested (we will keep our web app running, however). The freedom to choose what tools we want to use has been a great way to explore web-app development, while this project itself has pushed us to implement, tinker, and create to the best of our ability.