Sports Annotated Technical Report:

Phase III

Group Name: Team DGAF

Mission Statement: Don’t get an F…

Members: Amber Tucker (art653), Jerry Yune Han (jyh364)

Abstract:

Sports Annotated, created by Amber Tucker and Jerry Han (collectively known as Team DGAF), is a user-oriented data aggregation site providing information on various facets of professional American Football. Its full stack development will be implemented with Google Cloud Platform, Bootstrap, JavaScript, Python, HTML, Flask, SQLAlchemy, and PostgreSQL.

1: Introduction

Intended for the dedicated or casual National Football League fan, Sports Annotated details the three main pillars of football (games, teams, and individual players) within one comprehensive website. It allows a user to quickly and easily review information on the 32 existing teams, active athletes, and game results. Its searchable database facilitates a user’s ability to research specific games, athletes, and teams. Its attractive and easily navigable front end also enables easy and aimless browsing, allowing a user to explore at their leisure and come to a greater understanding and appreciation of the sport.

2: Front End

2.1: Design

The front end, designed with users in mind, is clean and responsive. Bootstrap’s pre-defined grid layout allows easy scaling depending on browser size. Bootstrap classes were used to format the navigation bar, collapsing it to a single drop-down menu if the viewport is under 576px (roughly the size of a smart phone screen). Bootstrap classes (primarily utilization of it subdivisions of sections into “rows” and “cols”) were also used to format the main content, stacking subsections vertically if the viewport size is under 576px. Bootstrap classes embedded within HTML endow Sports Annotated with responsive design and visibility regardless of a user’s device size. In addition, a search function is implemented on each page. The search function allows the user, regardless of what page he is on, to be able to pull data from the database by redirecting the user to a new page. If the user queries something that is not currently available or stored in the database, the user will be redirected to a page that says “Can’t find that one.”

2.2: Visual Elements

Images are used to provide context/clarification for textual information on the site as well as giving the site an ascetically pleasing quality. Image files of players, team logos, and a few miscellaneous football related images are stored in a directory as .jpg and .png files. The image locations for the images on the splash page are hard coded within the HTML, since Team DGAF has no plans to change the layout of the splash page. If desired, the image locations could be stored in a database and pulled dynamically and clickable to a new webpage. However, team DGAF feels this extra database query is unnecessary for the scope of this project.

The images of players and team logos are displayed dynamically like the rest of the model/instance data. The location of the image of each instance is stored in the appropriate table and inserted into template pages dynamically.

No videos and other visual elements are not implemented on this project. The only elements that the team used are the stadium images, player images, and team logos.

3: Back End

3.1: Data Collection:

Initially, team DGAF intended to collect data via an NFL API, store said data as key (model instances) value (instance attributes) pairs in a .json file, then insert the data from the .json file into tables on postgreSQL. Unfortunately, relevant APIs related to the National Football League are not available free of charge. Prior to committing their wallets, group members elected to explore alternative avenues of data collection. The website https://www.pro-football-reference.com provided this alternate avenue. The website is linked to the about page where other users may explore this resource.

Data was downloaded from <https://www.pro-football-reference.com> and exported as an xml file. Three different files were downloaded, each one corresponding to one of the models (games, players, teams). Extraneous information included by the website was deleted from the excel files. Names of each column were changed to parallel the names of each column in the tables created with postgresSQL. Once the changes were made, the files were converted from a .csv format to a .json format using an online “CSV to JSON Converter.” The Json files are stored on the repo in case graders wanted to see what format the team used.

3.2: Data Organization:

Data will be organized by model (Weeks, Teams, and Players) with their corresponding attributes. The attributes of teams are team name, city location, stadium name, total number of games played, stadium initial, unique id, and the last week the team played a game. The attributes of weeks are the two teams who participated, day/time of the game, final score of the game, overtime, and number of players who were injured that week. In the case of weeks, the instances are the games that were played and for teams are the unique teams. The instances of players are the individual players on the Patriots roster. The attributes of each player are their id, name, position, team, jersey number, and age. Note that in each table, the unique id act as primary keys for the table and cannot be foreign keys to other tables. Player names, because of their uniqueness, could be used as both primary keys and foreign keys, but the team did not go into creating joint tables thus foreign keys were not utilized in this project.

3.3: Data Storage:

In order to create dynamic web pages whose content is not embedded within the HTML, a database was required. The front-end of Sports Annotated interacts with a database created with Postgres SQL. The database was created via the Windows command prompt and populated with a python file: create\_db. First, PostgresSQL was accessed with the command “psql -U postgres -h localhost.” Once accessed, the database was created with the simple command “create db nfl; CREATE DATABASE.” The database was selected for use with the “use” command. Once selected, three relational tables were created, each representing the instances of each model and the attributes of each instance.

Three tables were created to represent instances of each model (players, teams, and games) with the SQL command “create table.” Each instance has a unique identification (represented as an arbitrarily assigned integer). The attributes of each instance are represented in columns. The “teams” table has attributes with names id, stadium\_name, stadium\_init, games\_played, city, team, and week. The “players” table has attributes with id, name, jersey\_num, player, age, pos, and team. Each instance of team has a one-to-many relationship with the instances in table “players,” as a team has many different players. Each instance of player has a one-to-one relationship with the instances in table “teams,” as each player can only belong to one team at any given moment. The “weeks” table has attributes: score, week, team, id, overtime, and injuries.

Once the tables were created, they were populated with data stored in the .csv file by invoking the “COPY” command, with delimiters specified as commas. The SQL “COPY” command facilitated an easy transfer of data from the three comma separated value files to tables stored in the group database. However, the team later realized that they could use a python file to populate the database, thus they utilized the create\_db.py file. Reference section 3.4.1 for further information on create\_db.py.

3.4: Data Display:

Originally, each model page (teams, players, games/weeks) displayed data hard-coded within the HTML. Due to the increasing amount of data displayed on the site, a relational database was created. Using a database to create dynamic content on a website makes a website more scalable. In addition, the use of a database allowed templates to be more accommodating and automated the process of displaying relevant data onto the website. Additional data can be added with little to no alteration of the HTML or CSS of the website. The team also realize that linking different pages with the ‘{{url\_for()}}’ could also be automated and simplified the process of linking different webpages together.

3.4.1: Database:

The name and location of the database was specified in the .gitlab-ci.yml file. The name was specified with the “POSTGRES\_DB” identifier, the location was specified with the “DB\_STRING:” identifier. The name was set to “postgres” and location set to “postgres://postgres:Hello!123@/postgres?host=35.226.209.166.” The team used the IP address displayed on the database instance instead of the database name because the errors occurred when the team did not use the IP address.

An additional file was added to create the database (create\_db.py). The file create\_db.py contains the methods load\_json (filename), create\_players (), create\_teams (), and create\_weeks (). The create\_teams () /create\_players (), and create\_weeks () methods all call the load\_json (filename) helper function. The “create” methods pass the name of the relevant file to the “load” method. Once called, load\_json () opens and returns the file. The create function traverses through each line in the file, assigning each attribute to a variable, then creating an object with those variables. After the object is created, it is added to the session with the command “db.session.add (object)” and committed with the command “db.session.commit ().” The create\_db.py is executed when the mainfile.py is executed.

3.4.2: Database/Front-End Interface:

During development, the team set app.run() to app.run(debug=true) so changes could be made fluid without having to rerun the python mainfile.py

SQLalchemy and flask were used to interface between the database and website. For the players model page, the database was queried about player instances and attributes, returning a generator that was converted into a list by invoking the simple list() built-in python method. The list was then looped through and reorganized into a dictionary with the player “position” attribute as keys and the player objects as values. The dictionary was passed via flask when the players.html was rendered, allowing the players to be grouped together by the attribute “position.” The model player page can now be scaled to include a massive number of player instances added to the database without further alterations to the format or style of the page. Note that to run all flask modules, the environment of the machine requires certain python modules that are noted on the requirements.txt file on the repo. To verify that the build is successful, the team utilized Gitlab’s continuous integration to verify that the pipelines passed. To ensure that all the correct modules are on the text file, the team ran ‘pip freeze > requirements.txt’ to export all the modules that are installed on the virtual environment.

To implement the search function, the team used flask\_whooshalchemy which is a module that integrates text-search functionality of ‘whoosh’ with the ORM of SQLAlchemy. Note that errors will occur if the an engineer uses any version of python 3.0 or later because flask libraries were updated from ‘flask.ext.SQLAlchemy’ to ‘flask\_SQLAlchemy’. Therefore, the team imported the ‘correct’ flask\_whooshAlchemy from the public github repo: <https://github.com/gyllstromk/Flask-WhooshAlchemy>. The team ran ‘pip install git+git://github.com/gyllstromk/Flask-WhooshAlchemy/blob/master/flask\_whooshalchemy.py’ on the virtual environment and installed a working version of the module (the correct version is noted on requirements.txt). From there, the team created a new template called the search.html file that would display entries that are related to the text-search input data. Using the flask\_whooshalchemy to index the player table, the team was able to create an index that could be referenced in other methods in the mainfile.py. The team also implemented the search function onto all the other html template files so that a user could search the database at any time. Reference 2.1 for design on invalid inputs on search bar.

3.4.3: Front-End:

In the scope of this project, the team created ten template html files that display various data from the database. In phase one, the team hard coded several more templates to display each model and their respective instances and attributes. In phase two, in order to avoid hard coding an excessive number of web pages to display each instance of each model, each instance listed on a model page links to only one dynamically-generated page. Originally, the group planned to create a session/set a cookie using JavaScript when the user selected an instance of a model to view specific attributes of that instance. Thankfully, Team DGAF realized that Flask is perfectly capable of receiving arguments and dynamically generating content. Not only does it provide a method to dynamically generate content with a small amount of code, it removes a potential user annoying feature. If using cookie/sessions that are destroyed when the user navigates away from the web page, the user is not able to return to the previous page. Flask does not remove the content that a user viewed, so they will be able to use the keyboard’s “backspace” button to review previous data they looked at.

Flask was employed to achieve seamless content generation and display. When a user selects an instance by clicking on the href link of an instance of a player (denotated by the player’s name), two variables are passed to flask. The “url\_for” flask function sends two variables, the location of the template that should be rendered, and the unique identification number of a player, team, or game to the “render\_template” function for the specific model being viewed. When flask receives the unique id for the instance, it queries the database for the instance object that matches the unique id. Flask then renders the specific template for the model whose instance is being displayed. Flask also passed the instance object as a variable to the template. The template renders the attributes for the instance object which has been specified.

Disclaimer: Note that this description holds only for some templates. Not all templates passed unique ids as an argument in the ‘url\_for’ function. However, without loss of generality, the process holds as long as the argument that that is being passed matches what is needed in the mainfile.py file.

4: Hosting

Sports Annotated is hosted on Google Cloud Platforms (GCP). A new project titled “cs329e-idb” was created on App Engine. Using Google Cloud Shell, the gitlab repository was cloned onto GCP with the command “git clone.” Prior to cloning the repository, the app.yaml file was created and stored in the repository. This file configures the App Engine app’s settings and designates the cloud instance and database that the web application needs to run smoothly. Prior to deployment, the team ran ‘python mainfile.py’ on google shell to ensure that everything was running smoothly. During this process the team ran into an error that stated that the app infrastructure was unhealthy and was waiting for it to become healthy again. To fix this issue, the team checked to make sure that the quota limits were not past their limits and checked to make sure that they had enough credits to deploy. The team discovered that they were low on credit and added a new coupon. After waiting two days, the app infrastructure was healthy and the team moved to deploy the app on GCP.

Once the repository is cloned, “cd [directory of repository]” and type “gcloud app deploy”. To ensure that the app can be deployed, a ‘requirements.txt’ was created so that when GCP creates a virtual environment, Flask and Gunicorn will run inside the newly create environment and deploy the website. For phase two, more flask modules were noted on the requirements.txt and can be referenced on the repo. Once the app is deployed, the team typed in “gcloud app browse” so that the team could render a viewable website link from GCP. The current URL link hosted on GCP is https://cs329e-idb-220319.appspot.com/. Namecheap was utilized to so that the website could point to a custom domain. The team followed the *Namecheap* tutorial by Chuchu Ma and produced a new and easier to remember URL: teamdgaf.me.

Note: As explained in the “data” section, the database is also hosted on Google Cloud Platforms. Refer back to 3.4.1/2.

5: Testing

In phase one, no unit tests were written, as the data was hard-coded in the HTML and it was easier to view each page and visually confirm that the correct data was displayed. Phase two includes the addition of dynamic pages whose content is pulled from the group database. Unit testing has been employed to test the functionality of the web page’s content acquisition from the database. The setup of the database was tested by assigning known attributes of instances to a variable “s.” Variable s was added and committed to the session. At this point, variable “s” should be an instance of the model in the database. A database query was initialized under variable “r.” The attributes returned by r were compared to the attributes sent to the database (s) with the “assert equal” method. If the values are equal, an instance and its attributes have been successfully added to the database and the test passes. If the values are not equal, either the creation or querying of the database was unsuccessful. The instance that was added to the database was then deleted from the database. All tables in the database were tested.

6: Integration Methodology

Team members conform to a continuous integration model. Though they occasionally work separately, alterations to project files were tracked via frequent pushing to gitlab. Each team member “pulls” to update their own files with the most current version of the team repo prior to altering any files. Once the project was completed, the team merged the dev branch to the master branch and then pulled all changes on google shell and then deployed the website on Google Cloud Platforms.