DATE:	Nov 29, 2020
TO:	Dr. Eberlein, EE 461L TAs
FROM:	Kevin Medina, kvnmedina59@gmail.com, Kvnmedina
	Samuel Yeboah, syeboahjr@utexas.edu, Samuel-Akwesi-Yeboah
	Alan Zhang, alan.zhang.2000@utexas.edu, adz00
	Edie Zhou, edie198@gmail.com, edie-zhou
SUBJECT:	Phase 4 Design Report
PROJECT:	Formula 1 Internet Database
GITHUB:	https://github.com/UT-SWLab/TeamE2
CANVAS TEAM:	Team E2
WEBSITE LINK:	https://f1stat-292509.uc.r.appspot.com/

Information Hiding

Our application has several design decisions that were implemented to improve the modularity of our system. We created a separate module for the database so we could easily change how we interact with the MongoDB database. There is a module that solely handles how we store and access pictures in the servers' file system and we have a module that implements various facades that are needed for the model pages.

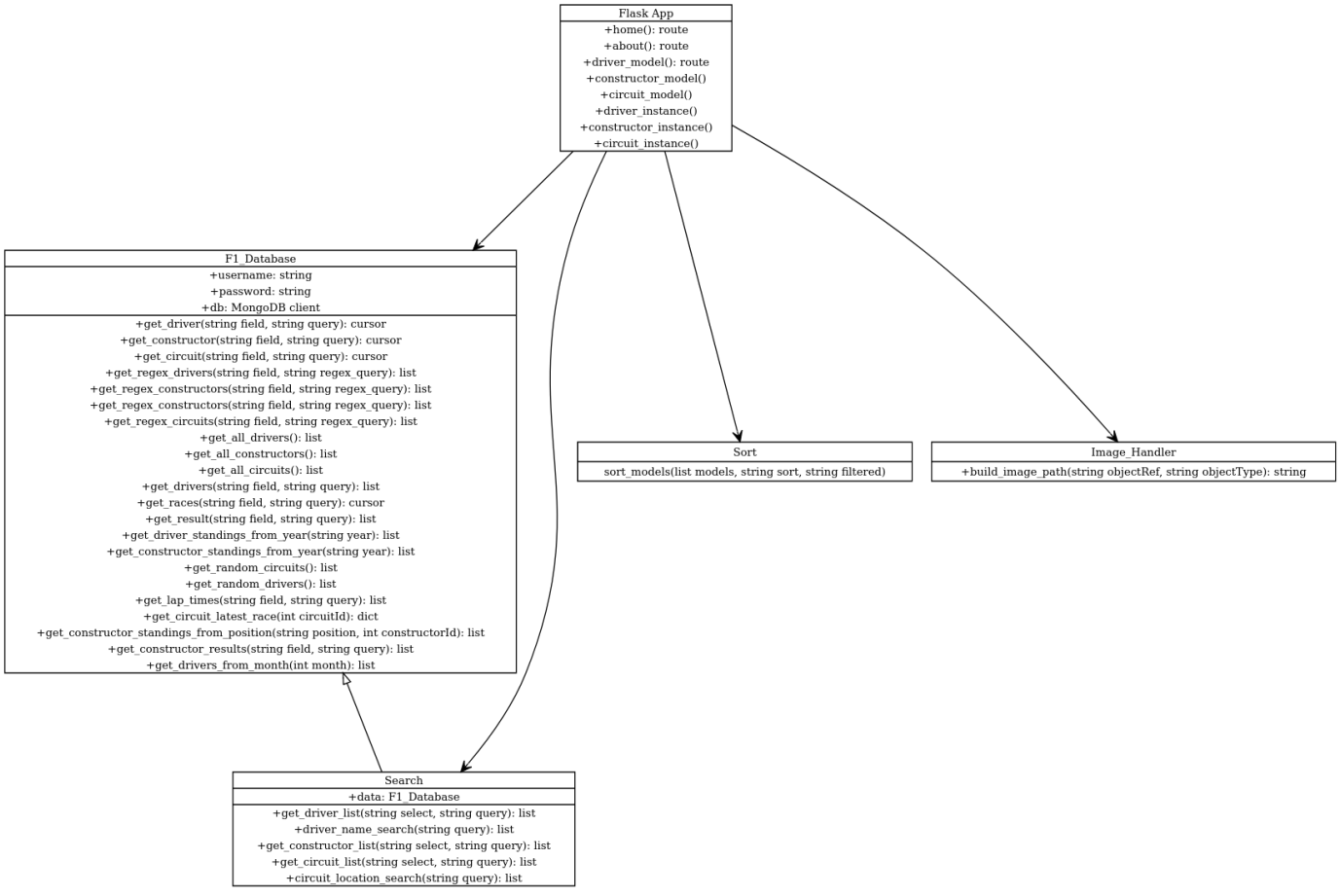
The module F1_Database implements various functions and abstracts the database from the app.py controller. We created a module for the database because we knew that there are a myriad of ways to make queries to the database and we wanted to make it easier for our team members to work on the controller without having to have a deep understanding of the database. The modularization also mitigates bugs that could arise from developers using the wrong collections for queries or developers updating articles' improperly. The disadvantage of creating a separate module for database queries comes from the fact that we have to create a singular function for each different type of database query. For instance, we have to create different functions for regex queries vs regular queries on fields. We can mitigate this by generalizing the functions within the module to be collection independent, which would reduce the need to write three functions per query type.

The ImageHandler module handles the picture file system that we have stored on the google cloud virtual machine. While the class only has one function, it provides flexibility for validating picture image paths and creating image paths. Because the image handling is in a singular module, every class that uses pictures must depend on this class, which increases the overall system coupling.

Design Patterns

For Phase 4, the team selected to enforce an MVC design pattern on the web application. Before MVC was implemented, every function was placed in the app.py file as show by the following UML class Diagram. Since there was no top level “main” class in the Flask application, the top level python application module is defined as “Flask App.”

This UML diagram shows no evidence of any design patterns implemented by the team, and the app.py file became increasingly challenging to work with as the size of the file grew. Thankfully python enforces singleton patterns by design, so this implementation already had some sort of design pattern enforced upon it. After reviewing the code, MVC was selected as the design pattern that the application would conform to. MVC was selected since the behavior of the database application was applicable to a Model, View, Controller design. The data retrieved from the database would be the model since this was the information that the application to be served to the user. The flask routing functions that controlled HTML templates would be the controller since the routing functions controlled how users viewed model data. Finally, the HTML templates would be the view since this was how users were able to view application data. MVC pattern was enforced on top of the existing Singleton Pattern provided with Python by encapsulating related functions within classes. After refactoring functions into classes, all model functions were removed from app.py and placed in models.py. A UML class diagram of the revised pattern is provided below:



Search stands out as the only class to inherit from another class, and this was done since many methods in the Search class depend on database calls defined in F1_Database. For example, the following snippet shows that `driver_name_search()` from the `Search` class entirely depends on methods from `F1_Database`:

```
def driver_name_search(self, query):
    """
    Purpose:
        Seach driver names with a query

    Args:
        query: {str} data to be sarched for in the collection

    Returns:
        {list} List of search results
    """

    # Search token in forenames and surnames
    tokens = query.split()
    driver_list = []
    if len(tokens) == 1:

        forname_list = self.get_regex_drivers('forename', query)
        surname_list = self.get_regex_drivers('surname' , query)

        for driver in surname_list:
            if driver not in forname_list:
                forname_list.append(driver)

        driver_list = forname_list

    else:

        # Search first token in forenames, search other tokens in surnames
        forename_list = self.get_regex_drivers('forename', query)
        surname_list = []

        for i in range(1, len(tokens)):
            surname_list += self.get_regex_drivers('surname' , tokens[i])

        driver_list = forename_list + surname_list

    return driver_list
```

However, the `Search` class also contains many methods that depend on each other, but are never referenced by methods in the `F1_Database` class. As a result, code that provided search functionality was encapsulated in its own class separate from `F1_Database`.

Refactorings

We refactored our application extensively because we did not create modules or implement any design patterns. We also had various code duplications and dead code that had accumulated through the various merges and project phases. The easiest refactoring was removing the dead code because we could use version control and the search function in our IDE to see which software was not being used. The screenshot below shows an example. The code on the left moved to modules, which made the entire highlighted red portion irrelevant. So, we deleted the code and replaced it. We also expunged the route “/dbTest” because it was unused.

```
app = Flask(__name__)
dbUsername = 'formulaOne'
dbPassword = '0WpPVH6LdcHiwdct'
- CONNECTION_STRING = "mongodb+srv://" + dbUsername + ":" + dbPassword +
"@formulaonedb.bue6f.gcp.mongodb.net/<dbname>?retryWrites=true&w=majority"
-
- client = pymongo.MongoClient(CONNECTION_STRING)
- db = client.get_database('FormulaOneDB')
-
- # Default filelr image
- NO_IMG = 'images/no_img.png'
-
-
- @app.route('/dbTest')
- def test():
-     db.db.collection.insert_one({"name": "John"})
-     return "Connected to the data base!"
-
```

```
+ # Self Written Modules
+ from database import FormulaOneDatabase
+ from image_handler import ImageHandler
+
app = Flask(__name__)
dbUsername = 'formulaOne'
dbPassword = '0WpPVH6LdcHiwdct'
+ db =  FormulaOneDatabase(dbUsername , dbPassword)

+ image_handler = ImageHandler()
```

Another refactoring that we completed was replacing a duplicated section of code with a module. The module will make future development a lot easier because it provides a singular location for all functions related to pictures. The segment of code that was duplicated is displayed below.

```
-               img_path = f'images/drivers/{driver_ref}.png'
-               if not os.path.exists(f'./static/{img_path}'):
-                   img_path = NO_IMG
```

This is used to create an image path to the module specified by the ref. We duplicated this over 5 times in the app.py file for the three models that we have. To refactor this, we created a module called ImageHandler that contains any image related function. The function we wrote is the snippet below.

```
def build_image_path(self,objectRef,objectType):
    img_path = f'images/{objectType}/{objectRef}.png'
    if not os.path.exists(f'./static/{img_path}'):
        img_path = NO_IMG
    return img_path
```

The function takes in the modelType and the modelRef as parameters and creates a path to the image. We then check the path to make sure it exists before returning the path. After refactoring, creating an image path is extremely simple because we simply have to call the above function. Because the image handling was implemented in a separate module, the class structure of our application changed. An updated class diagram is listed below.

The last refactoring we completed was extracting methods from the various instance pages that we created. While we extracted methods for each route and view we created, I will only be

highlighting the home page in this design document. A snippet of the home page code shown below is from before the refactoring.

```
today = date.today()
currentMonth = str(today).split("-")[1]
currentMonthName = today.strftime('%B')
currentDay = str(today).split("-")[2]
currentYear = str(today).split("-")[0]

recentRaces = db.races.find({'year': int(currentYear)}).limit(5)
recentRaces = list(recentRaces)

regDate = "....-" + currentMonth + "-.."
drivers = db.drivers.find({'dob': {'$regex': regDate}}).limit(20)
drivers = list(drivers)
currentMonthDrivers = []
index = 0
for driver in drivers:
    driver_ref = driver['driverRef']
    name = driver['forename'] + "  " + driver['surname']
    image_path = f'images/drivers/{driver_ref}.png'
    if not os.path.exists(f'./static/{image_path}'):
        image_path = NO_IMG
    tempDict = {'driver_ref': driver_ref, 'image_path': image_path, 'name': name, 'id': driver['driverId']}
    currentMonthDrivers.append(tempDict)
    index += 1

 thisSeasonResults = db.results.find({'year': int(currentYear)})
 seasonDriverPoints = {}
 seasonConstructorPoints = {}
 for result in thisSeasonResults:
     driver = result['driverId']
     driverPoints = result['points']
     driverName = result['driverName']

     constructor = result['constructorId']
     constructPoints = result['points']
     constructorName = result['constructorName']
     if driver in seasonDriverPoints:
         currentPoints = seasonDriverPoints[driver][1]
         seasonDriverPoints[driver][1] = currentPoints + driverPoints
     else:
         seasonDriverPoints[driver] = [driverName, driverPoints, driver]

     if constructor in seasonConstructorPoints:
         currentPoints = seasonConstructorPoints[constructor][1]
         seasonConstructorPoints[constructor][1] = currentPoints + constructPoints
     else:
         seasonConstructorPoints[constructor] = [constructorName, constructPoints, constructor]

 sortedDriverSeasonLeaders = []
 for key in sorted(seasonDriverPoints.keys(), key=lambda k: seasonDriverPoints[k][1], reverse=True):
     sortedDriverSeasonLeaders.append(seasonDriverPoints[key])

 sortedConstructorSeasonLeaders = []

 for key in sorted(seasonConstructorPoints.keys(), key=lambda k: seasonConstructorPoints[k][1], reverse=True):
     sortedConstructorSeasonLeaders.append(seasonConstructorPoints[key])

 recentRaces = recentRaces[:5]
 sortedConstructorSeasonLeaders = sortedConstructorSeasonLeaders[:5]
 sortedDriverSeasonLeaders = sortedDriverSeasonLeaders[:5]
```

As you can see, the home page has a large amount of code that can be separated into various functions that fulfill specific tasks. These tasks can be moved into a separate module because they are mostly database calls that create varying lists of data. You can also see the duplicated code snippet mentioned earlier that we refactored. We refactored this page by creating a

function for each parameter that is specified in the render html function. For instance, we created a function called get_driver_standings_from_year that we used to create a season standings data structure. The function gathers all races from a year, calculates the total points for each individual that raced that year, sorts the list based on the point total for a driver, and returns the sorted list.

```
def get_driver_standings_from_year(self , year):
    results = self.db.results.find({'year' : int(year)})
    seasonDriverStandings = {}
    for result in results:
        driver = result['driverId']
        driverPoints = result['points']
        driverName = result['driverName']
        if driver in seasonDriverStandings:
            currentPoints = seasonDriverStandings[driver][1]
            seasonDriverStandings[driver][1] = currentPoints + driverPoints
        else:
            seasonDriverStandings[driver] = [driverName, driverPoints, driver]

    sortedDriverSeasonLeaders = []
    for key in sorted(seasonDriverStandings.keys(), key=lambda k: seasonDriverStandings[k][1], reverse=True):
        sortedDriverSeasonLeaders.append(seasonDriverStandings[key])
    return sortedDriverSeasonLeaders
```