# Phase IV Design Report

EE 461L

Github repo: https://github.com/chrisjoswin/EE461L_Project

Team Information:

| | | |
|---|---|---|
| Jacob Grimm | jacobgrimm@utexas.edu | jacobgrimm |
| Jerad Robles | sebastian.robles@utexas.edu | JSRobles |
| William Gu | williamgu@utexas.edu | Minalinnski |
| Christopher Erattuparambil | chris.joswin@utexas.edu | chrisjoswin |
| Josh Kall | joshuakall@utexas.edu | j-ka11 |
| Haosong Li | hl27346@utexas.edu | hdlee9885 |

Team Canvas Group: Morning-9

Project Name: Internet Comic Database (ICDb)

Project URL: https://icdb-phase1-270405.appspot.com/

## Part 1A Information Hiding:

To help with navigation on the frontend, we created a State Service. An example of how this service operates has to do with searching our models for instances. On every page within our application, there exists a search bar that the user can use to send a keyword to our database to search all three of our models and view a list of results. Using the State Service, any page can simply call the setKeyword(keyword: string) function that the state service provides, then use Angular's router to navigate to the search-page. The search-page is configured to, on initialization, get a keyword [getKeyword(keyword:string)] from the State Service, query the database with the keyword, and display the results. The State Service works in a similar fashion when a page wants to detail an instance in full by navigating to that model's detail page. Generally, the State Service is a temporary data container that pages can use to pass information to each other. It works by setting a variable when a set() method is called and returning that variable when the corresponding get() function is called.

We anticipate two major design aspects that might change that relate to the State Service. The first major aspect is the structure of the JSON object for a particular instance of a model. In the real world, it is very likely that we would eventually clean up our data points, as well as potentially add new fields. Instead of having the pages pass the full JSON object back and forth, the State Service allows us to simply pass an instance of an interface. In Angular, an interface can be used to define the structure of a JSON object. Interfaces are defined in a single file, so if the JSON object changes, we only have one place that the code needs to change for our State Service to continue functioning. The other major design aspect that could change is we could grow the number of models that we have. If this happens, the State Service would simply need to provide another function for that new model, again proving to be a single point of change.

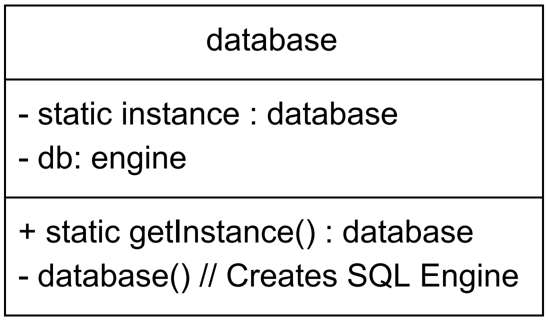
As of now, the internal data within the State Service can only be accessed by the State Service itself. We intend for the scope of the service to only handle data passage between pages, so vastly new functionality would be implemented with a completely different component or service. However, our State Service is poised to be able to handle extra modifications to the data being passed should our application ever require it. For example, on our set methods we could add a parameter that is an object where each field represents an operation on the data being passed. Once a page sets the state with a list of operations, the service could then perform said operations on the data before the next page retrieves the data. All this could be done under the hood after the set() function is called. In this sense, our State Service could function similarly to how streams function in Java.

The biggest disadvantage we found with using the State Service to pass data comes when the page is refreshed. A page refresh essentially resets an Angular application, so all the data the State Service was storing is lost. The way we got around this was if the page found the data in the State Service is null, it would just navigate to a page that doesn't need data from the State Service. This is not a perfect fix however, and to fix it completely we would need to store some data on the backend.

## Part 1B Design Patterns and Refactorings:

### Design Pattern 1 - Singleton Method:

UML:

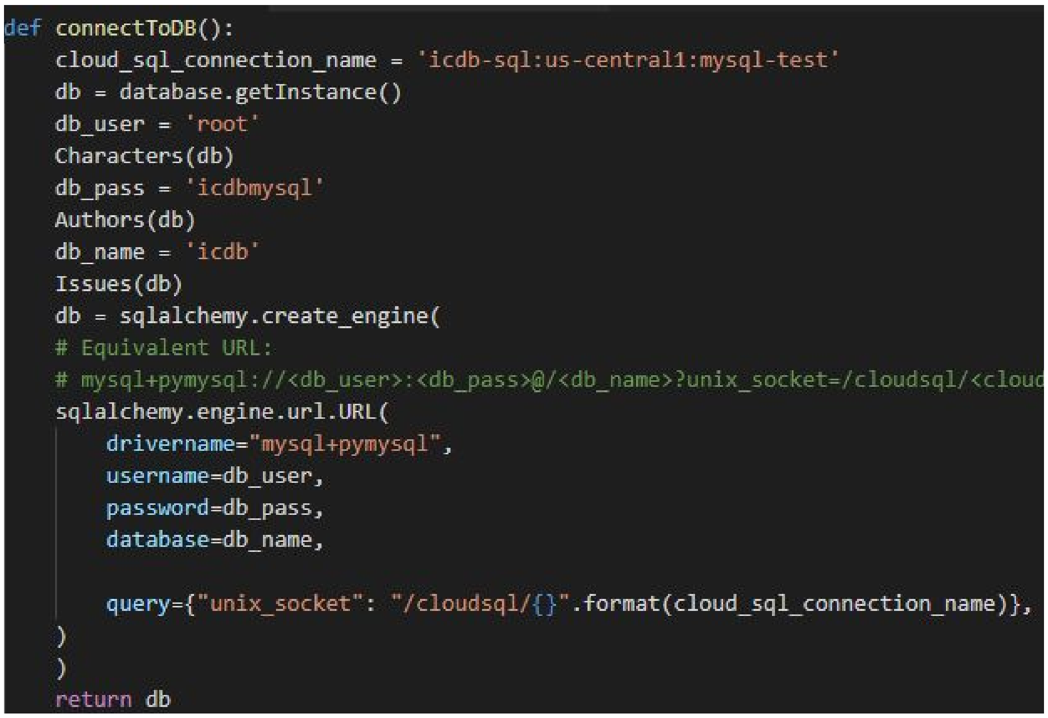

database
- static instance : database
- db: engine
+ static getInstance() : database
- database() // Creates SQL Engine

Justification:
In our original implementation every time an API endpoint was called to retrieve data from the database it would create a new SQLAlchemy Engine object and establish a new connection to our database. This was very wasteful as all endpoints could share one Engine instead of establishing a new one every time. We feel that this problem fits the Singleton design pattern since we only need one connection from our API to our SQL database.

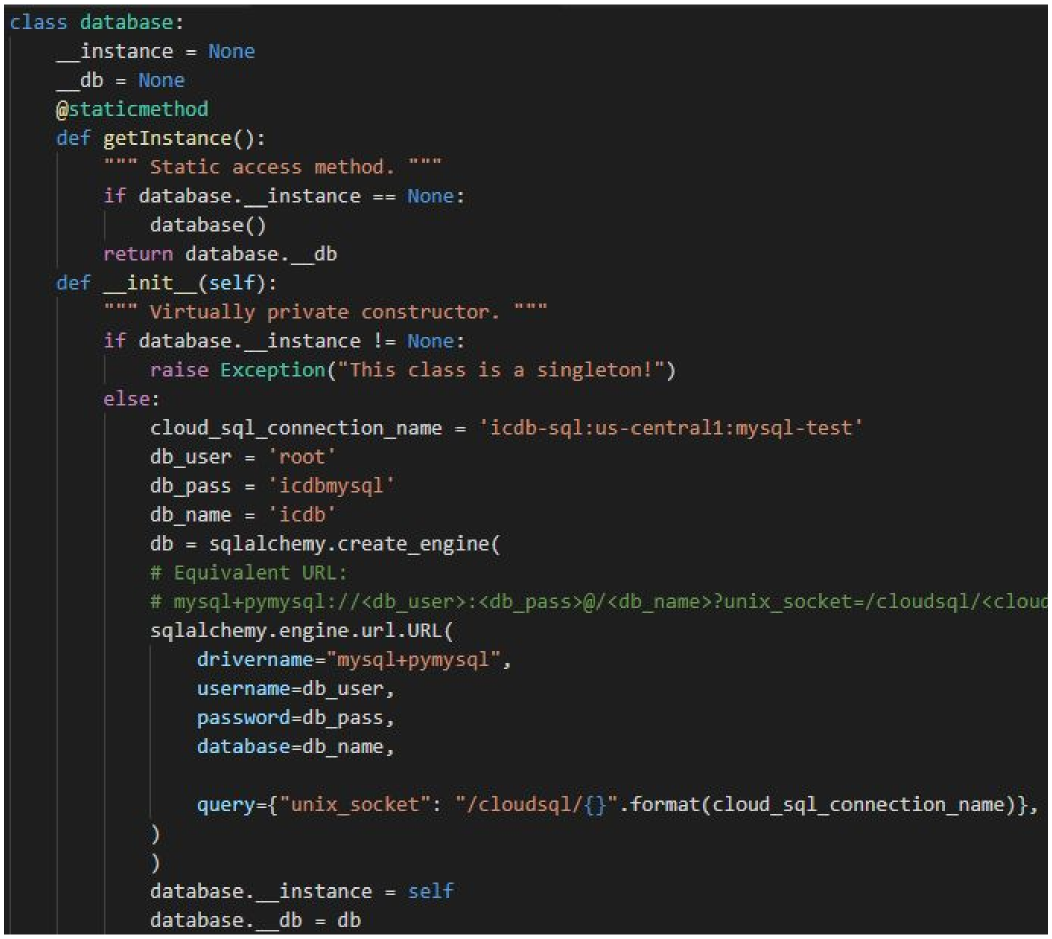
We achieved this by encapsulating the Engine object and its creation inside of a Singleton class named database using lazy instantiation. This way whenever an endpoint needs to retrieve data from our SQL database it can request the Engine object from the single instance of the database object by calling the getInstance() function.

The advantages of this are that the establishing of the connection to our database is much cleaner and there is only ever one connection from our API and the SQL database at a time. This will prevent excessive connection requests when our website is under heavy use. A disadvantage is that if we wanted to add the ability for our API to service multiple requests concurrently we would need to modify the design so each thread would have access to its own Engine object so that each thread could make its SQL queries concurrently.

Code Before and After:



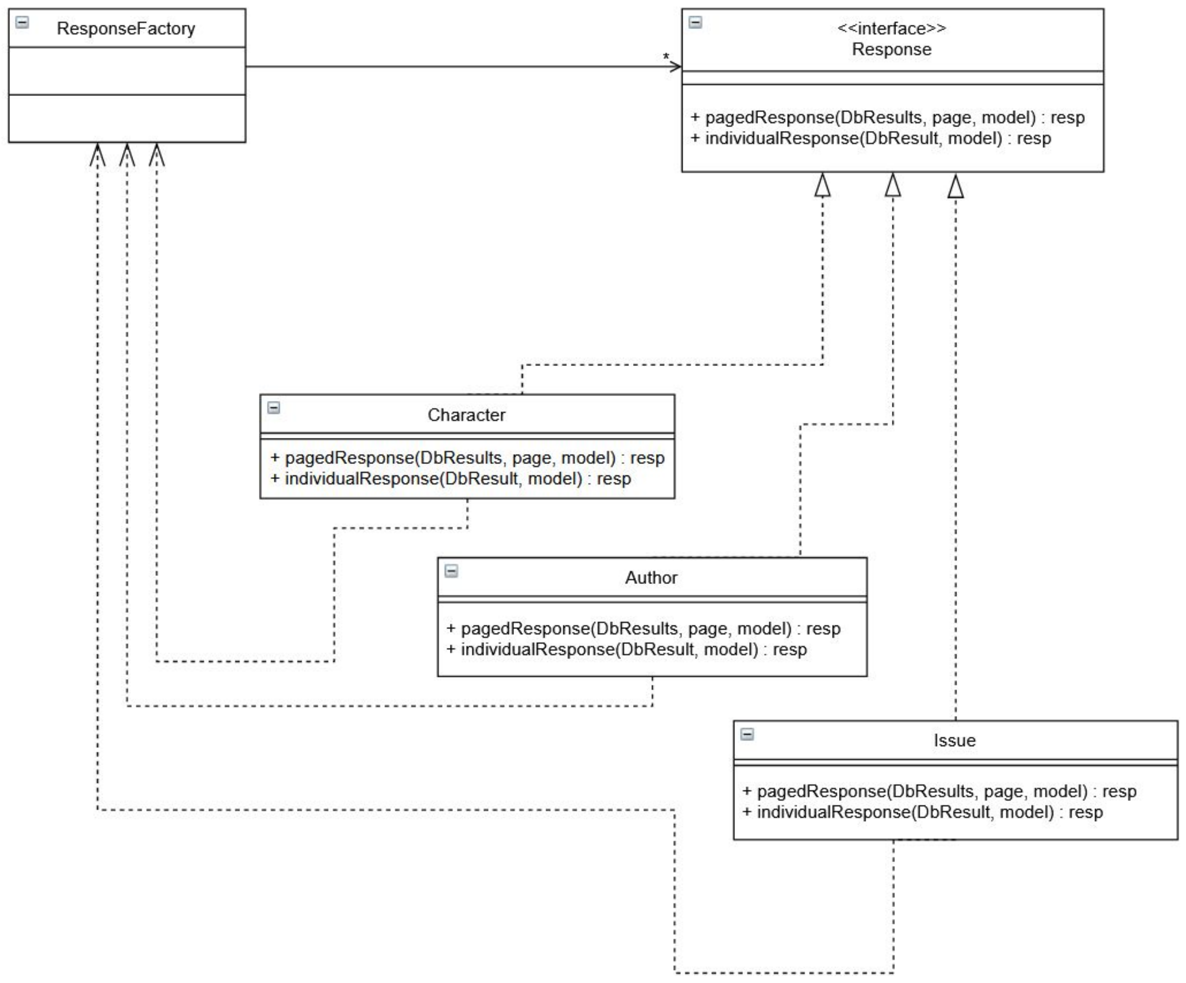
```
def connectToDB():
    cloud_sql_connection_name = 'icdb-sql:us-central1:mysql-test'
    db = database.getInstance()
    db_user = 'root'
    Characters(db)
    db_pass = 'icdbmysql'
    Authors(db)
    db_name = 'icdb'
    Issues(db)
    db = sqlalchemy.create_engine(
    # Equivalent URL:
    # mysql+pymysql://<db_user>:<db_pass>@/<db_name>?unix_socket=/cloudsql/<cloud
    sqlalchemy.engine.url.URL(
        drivername="mysql+pymysql",
        username=db_user,
        password=db_pass,
        database=db_name,

        query={"unix_socket": "/cloudsql/{}".format(cloud_sql_connection_name)},
    )
    )
    return db
```



```
class database:
    __instance = None
    __db = None
    @staticmethod
    def getInstance():
        """ Static access method. """
        if database.__instance == None:
            database()
        return database.__db
    def __init__(self):
        """ Virtually private constructor. """
        if database.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            cloud_sql_connection_name = 'icdb-sql:us-central1:mysql-test'
            db_user = 'root'
            db_pass = 'icdbmysql'
            db_name = 'icdb'
            db = sqlalchemy.create_engine(
            # Equivalent URL:
            # mysql+pymysql://<db_user>:<db_pass>@/<db_name>?unix_socket=/cloudsql/<cloud
            sqlalchemy.engine.url.URL(
                drivername="mysql+pymysql",
                username=db_user,
                password=db_pass,
                database=db_name,

                query={"unix_socket": "/cloudsql/{}".format(cloud_sql_connection_name)},
            )
            )
            database.__instance = self
            database.__db = db
```

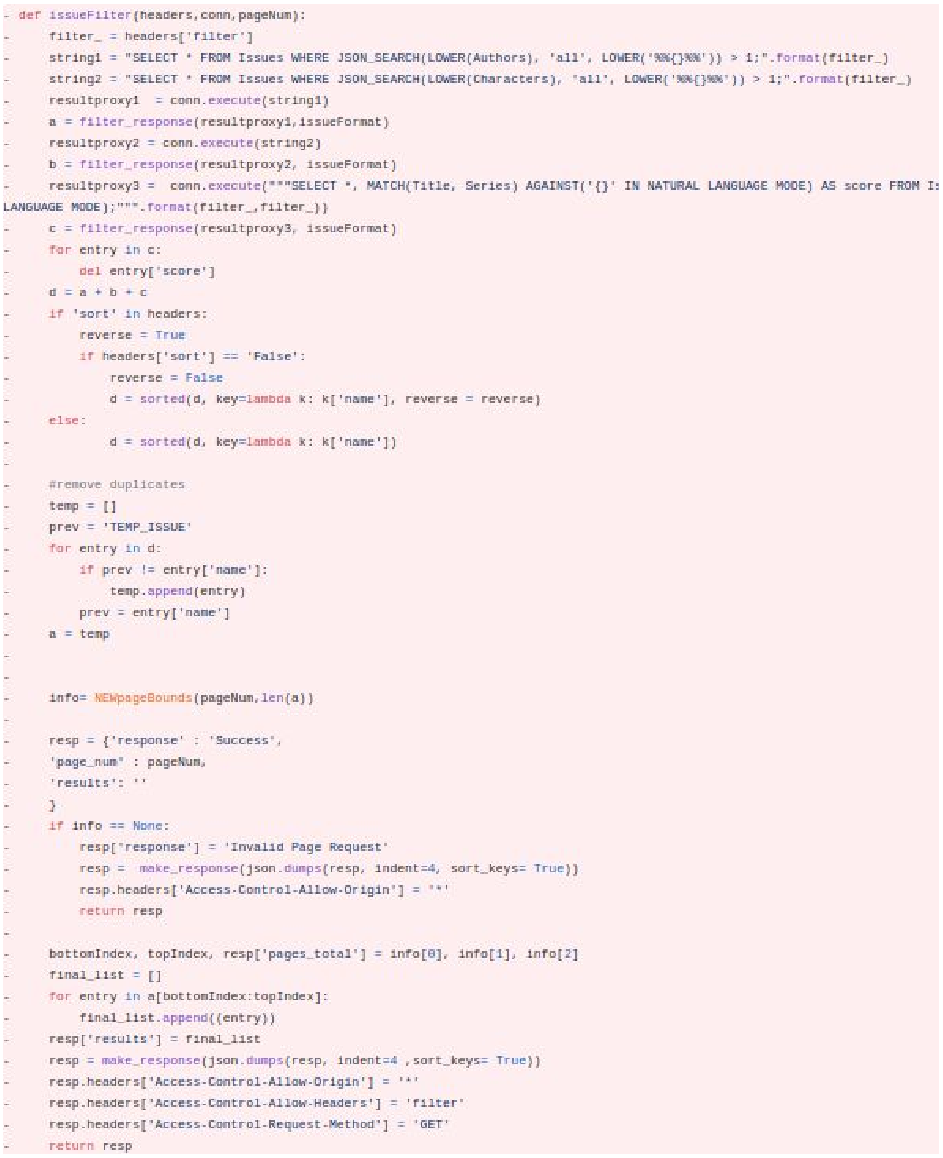
## Design Pattern 2- Factory Method:

UML:

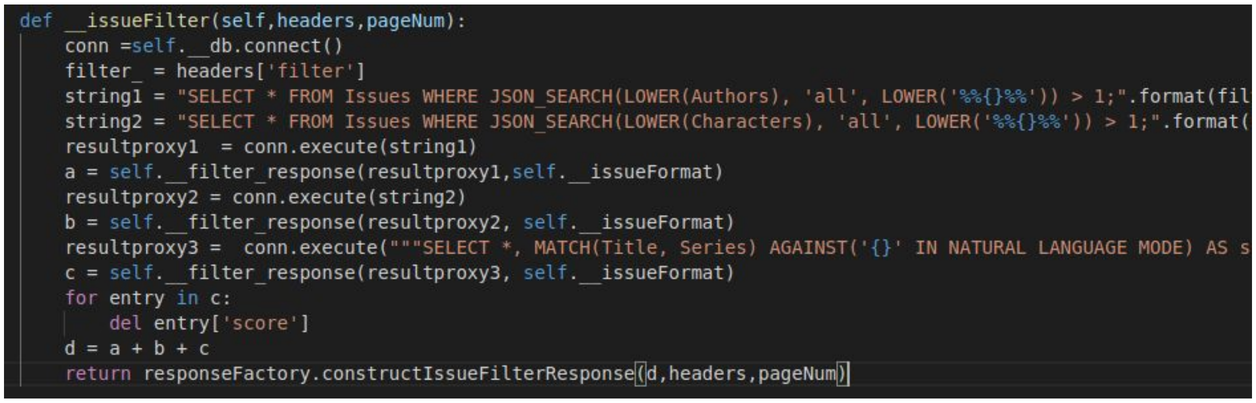

ResponseFactory
<<interface>>
Response
*
+ pagedResponse(DbResults, page, model) : resp
+ individualResponse(DbResult, model) : resp
Character
+ pagedResponse(DbResults, page, model) : resp
+ individualResponse(DbResult, model) : resp
Author
+ pagedResponse(DbResults, page, model) : resp
+ individualResponse(DbResult, model) : resp
Issue
+ pagedResponse(DbResults, page, model) : resp
+ individualResponse(DbResult, model) : resp

Justification:

In the original implementation everytime a data retrieval endpoint was called each model had its own functions for formulating the API's response object. With each endpoint having its own way of writing responses it made it difficult to have a unified and easy to process API response. We felt that this problem would fit the Factory Method Pattern. Using the factory method pattern we were able to create a factory to formulate response objects for the API to return. The response factory would be responsible for making the response objects uniform across all models. The advantage of this was that it greatly simplified the backend a lot making it easier to make changes to how responses were formatted by only changing the response factory and not having to rewrite the why every model formulated a response. The disadvantage is that not every model contains the same data so a lot of extra work must be done on each model in order to make it fit what the response factory is asking for.



```
- def issueFilter(headers,conn,pageNum):
-     filter_ = headers['filter']
-     string1 = "SELECT * FROM Issues WHERE JSON_SEARCH(LOWER(Authors), 'all', LOWER('%%{}%%')) > 1;".format(filter_)
-     string2 = "SELECT * FROM Issues WHERE JSON_SEARCH(LOWER(Characters), 'all', LOWER('%%{}%%')) > 1;".format(filter_)
-     resultproxy1  = conn.execute(string1)
-     a = filter_response(resultproxy1,issueFormat)
-     resultproxy2 = conn.execute(string2)
-     b = filter_response(resultproxy2, issueFormat)
-     resultproxy3 =  conn.execute("""SELECT *, MATCH(Title, Series) AGAINST('{}' IN NATURAL LANGUAGE MODE) AS score FROM I
LANGUAGE MODE);""".format(filter_,filter_))
-     c = filter_response(resultproxy3, issueFormat)
-     for entry in c:
-         del entry['score']
-     d = a + b + c
-     if 'sort' in headers:
-         reverse = True
-         if headers['sort'] == 'False':
-             reverse = False
-             d = sorted(d, key=lambda k: k['name'], reverse = reverse)
-     else:
-             d = sorted(d, key=lambda k: k['name'])
-
-     #remove duplicates
-     temp = []
-     prev = 'TEMP_ISSUE'
-     for entry in d:
-         if prev != entry['name']:
-             temp.append(entry)
-         prev = entry['name']
-     a = temp
-
-
-     info= NEWpageBounds(pageNum,len(a))
-
-     resp = {'response' : 'Success',
-     'page_num' : pageNum,
-     'results': ''
-     }
-     if info == None:
-         resp['response'] = 'Invalid Page Request'
-         resp =  make_response(json.dumps(resp, indent=4, sort_keys= True))
-         resp.headers['Access-Control-Allow-Origin'] = '*'
-         return resp
-
-     bottomIndex, topIndex, resp['pages_total'] = info[0], info[1], info[2]
-     final_list = []
-     for entry in a[bottomIndex:topIndex]:
-         final_list.append((entry))
-     resp['results'] = final_list
-     resp = make_response(json.dumps(resp, indent=4 ,sort_keys= True))
-     resp.headers['Access-Control-Allow-Origin'] = '*'
-     resp.headers['Access-Control-Allow-Headers'] = 'filter'
-     resp.headers['Access-Control-Request-Method'] = 'GET'
-     return resp
```

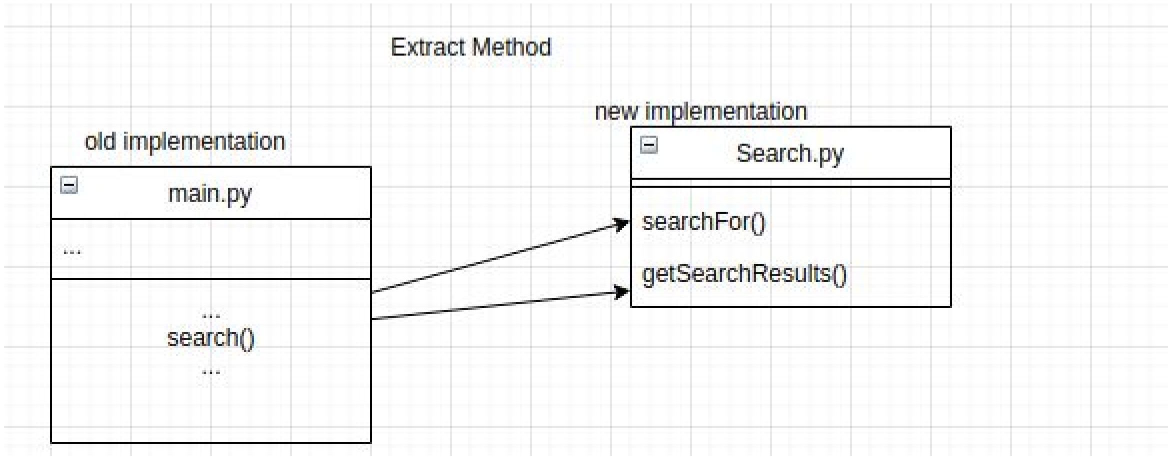


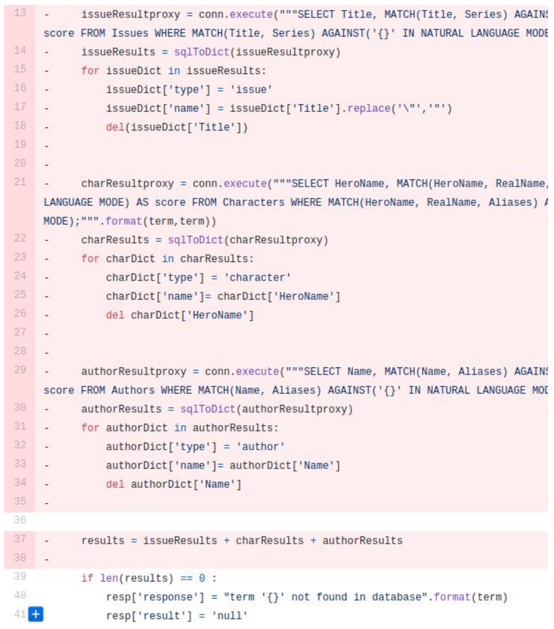
```
def __issueFilter(self,headers,pageNum):
    conn =self.__db.connect()
    filter_ = headers['filter']
    string1 = "SELECT * FROM Issues WHERE JSON_SEARCH(LOWER(Authors), 'all', LOWER('%%{}%%')) > 1;".format(fil
    string2 = "SELECT * FROM Issues WHERE JSON_SEARCH(LOWER(Characters), 'all', LOWER('%%{}%%')) > 1;".format(
    resultproxy1  = conn.execute(string1)
    a = self.__filter_response(resultproxy1,self.__issueFormat)
    resultproxy2 = conn.execute(string2)
    b = self.__filter_response(resultproxy2, self.__issueFormat)
    resultproxy3 =  conn.execute("""SELECT *, MATCH(Title, Series) AGAINST('{}' IN NATURAL LANGUAGE MODE) AS s
    c = self.__filter_response(resultproxy3, self.__issueFormat)
    for entry in c:
        del entry['score']
    d = a + b + c
    return responseFactory.constructIssueFilterResponse(d,headers,pageNum)
```

## Refactorings:

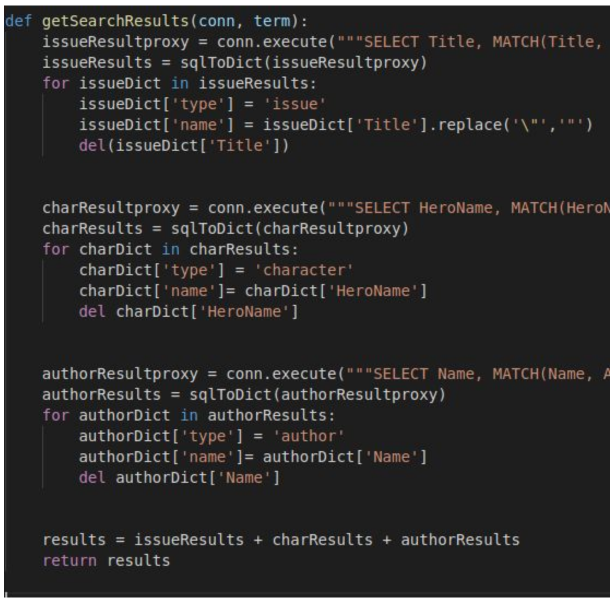
### 1. Extract Method:

Here we moved a big code block that makes a few queries to our database to search for a given term and returns an array of results into its own function. We did this for a variety of reasons, primarily for modularization and information hiding. First off, it certainly is ugly just to look at that giant code block that does multiple things and not break it up. So we split it on the most logical split, with the part where we make the database queries and the part that disects them.
As well, we figured that if we were to continue to work on this codebase, in the future we may want to implement a different way of getting the information from our database when making a search call, so now getSearchResults() is a totally independent function that takes as input a search term and the database connection, and as a black box to outsiders, just returns an array of search results. So inthe future, we could implement a more interesting search manner in our database, without having to change any code that calls this function.





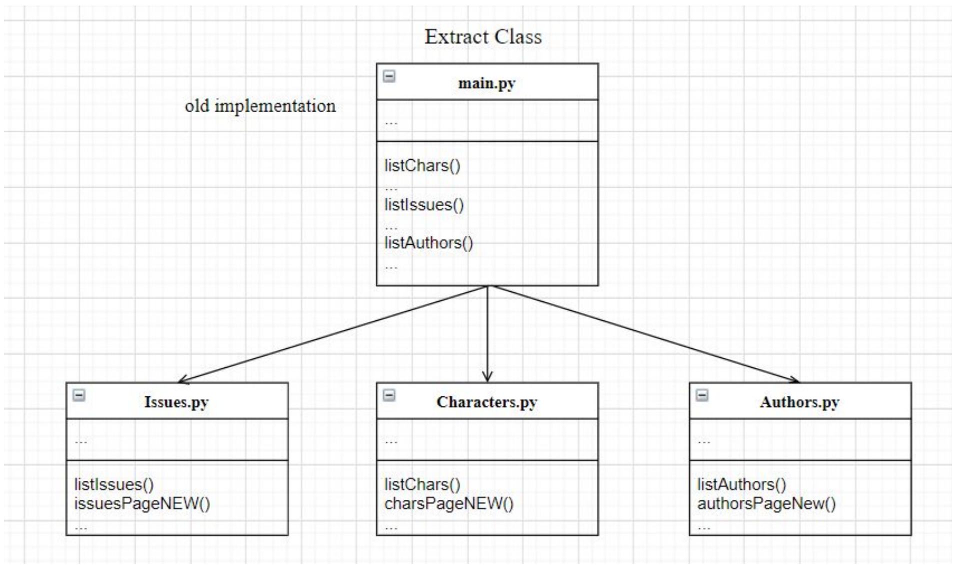
```
-     issueResultproxy = conn.execute("""SELECT Title, MATCH(Title, Series) AGAINS
score FROM Issues WHERE MATCH(Title, Series) AGAINST('{}' IN NATURAL LANGUAGE MODE
-     issueResults = sqlToDict(issueResultproxy)
-     for issueDict in issueResults:
-         issueDict['type'] = 'issue'
-         issueDict['name'] = issueDict['Title'].replace('\"','"')
-         del(issueDict['Title'])
-
-
-     charResultproxy = conn.execute("""SELECT HeroName, MATCH(HeroName, RealName,
LANGUAGE MODE) AS score FROM Characters WHERE MATCH(HeroName, RealName, Aliases) A
MODE);""".format(term,term))
-     charResults = sqlToDict(charResultproxy)
-     for charDict in charResults:
-         charDict['type'] = 'character'
-         charDict['name']= charDict['HeroName']
-         del charDict['HeroName']
-
-
-     authorResultproxy = conn.execute("""SELECT Name, MATCH(Name, Aliases) AGAINS
score FROM Authors WHERE MATCH(Name, Aliases) AGAINST('{}' IN NATURAL LANGUAGE MOD
-     authorResults = sqlToDict(authorResultproxy)
-     for authorDict in authorResults:
-         authorDict['type'] = 'author'
-         authorDict['name']= authorDict['Name']
-         del authorDict['Name']
-

-     results = issueResults + charResults + authorResults
-
      if len(results) == 0 :
          resp['response'] = "term '{}' not found in database".format(term)
          resp['result'] = 'null'
```

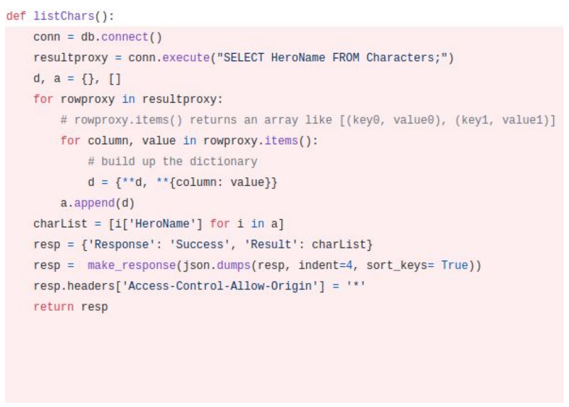


```
def getSearchResults(conn, term):
    issueResultproxy = conn.execute("""SELECT Title, MATCH(Title,
    issueResults = sqlToDict(issueResultproxy)
    for issueDict in issueResults:
        issueDict['type'] = 'issue'
        issueDict['name'] = issueDict['Title'].replace('\"','"')
        del(issueDict['Title'])


    charResultproxy = conn.execute("""SELECT HeroName, MATCH(HeroN
    charResults = sqlToDict(charResultproxy)
    for charDict in charResults:
        charDict['type'] = 'character'
        charDict['name']= charDict['HeroName']
        del charDict['HeroName']


    authorResultproxy = conn.execute("""SELECT Name, MATCH(Name, A
    authorResults = sqlToDict(authorResultproxy)
    for authorDict in authorResults:
        authorDict['type'] = 'author'
        authorDict['name']= authorDict['Name']
        del authorDict['Name']


    results = issueResults + charResults + authorResults
    return results
```

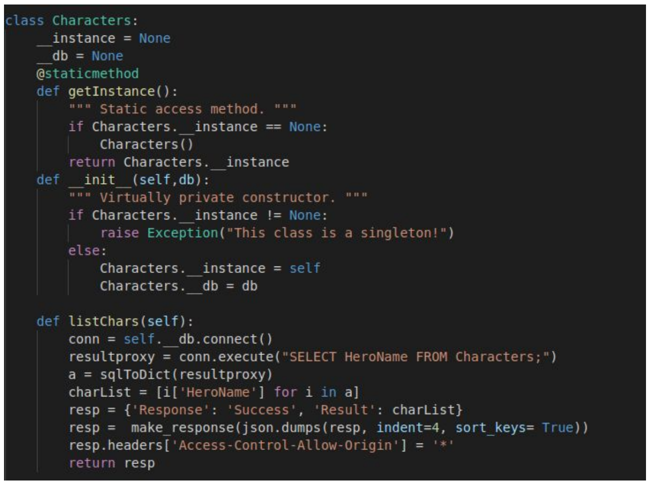
## 2. Extract Class:

We realized that when we developed endpoints in the past three phases, we are piling all methods into main.py. We extracted all required endpoint calls for each specific model: character, issues and authors into respective classes. With this structure, we would not need to find the specific method from a single messy thousand-line class. It is more understandable and easier for future changes. Here I will write briefly about the Characters class.
First we still keep the endpoint structure in main.py as before, but instead of starting to write the method, we call the corresponding method in each respective class. Of course, since connection to database and cleaning response text are somehow redundant, we also move it to response_functions.py so that the three modules wouldn't have to contain these methods separately.





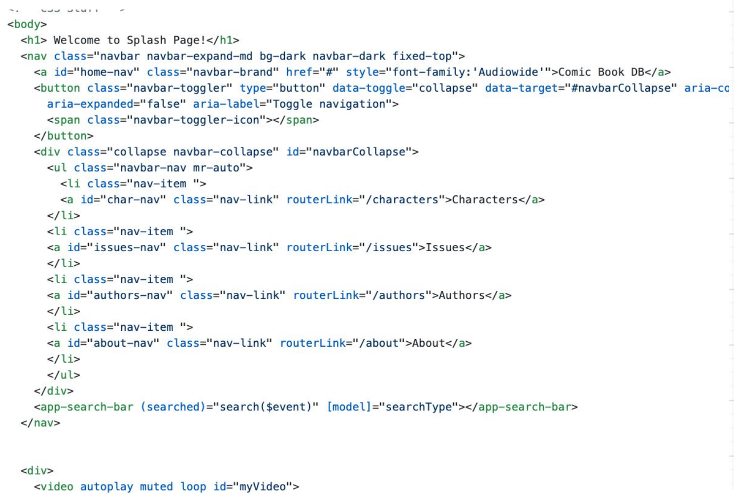
```python
def listChars():
    conn = db.connect()
    resultproxy = conn.execute("SELECT HeroName FROM Characters;")
    d, a = {}, []
    for rowproxy in resultproxy:
        # rowproxy.items() returns an array like [(key0, value0), (key1, value1)]
        for column, value in rowproxy.items():
            # build up the dictionary
            d = {**d, **{column: value}}
        a.append(d)
    charList = [i['HeroName'] for i in a]
    resp = {'Response': 'Success', 'Result': charList}
    resp =  make_response(json.dumps(resp, indent=4, sort_keys= True))
    resp.headers['Access-Control-Allow-Origin'] = '*'
    return resp
```



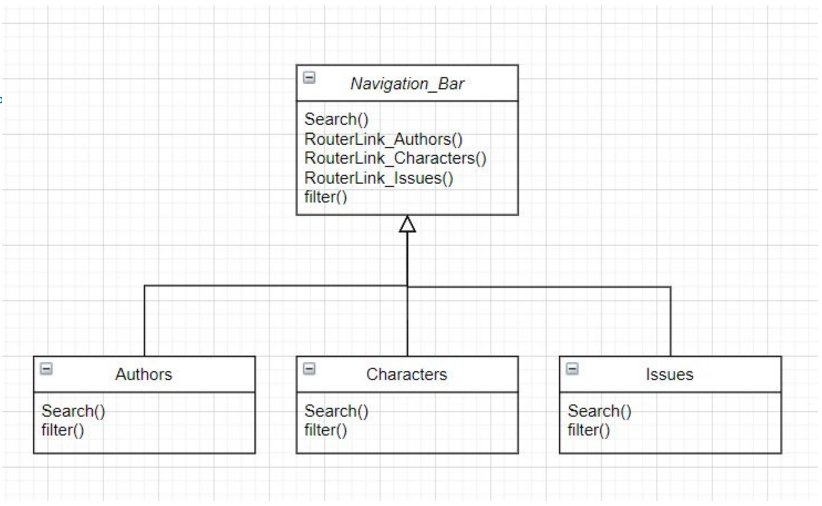
```python
class Characters:
    __instance = None
    __db = None
    @staticmethod
    def getInstance():
        """ Static access method. """
        if Characters.__instance == None:
            Characters()
        return Characters.__instance
    def __init__(self,db):
        """ Virtually private constructor. """
        if Characters.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            Characters.__instance = self
            Characters.__db = db

    def listChars(self):
        conn = self.__db.connect()
        resultproxy = conn.execute("SELECT HeroName FROM Characters;")
        a = sqlToDict(resultproxy)
        charList = [i['HeroName'] for i in a]
        resp = {'Response': 'Success', 'Result': charList}
        resp =  make_response(json.dumps(resp, indent=4, sort_keys= True))
        resp.headers['Access-Control-Allow-Origin'] = '*'
        return resp
```

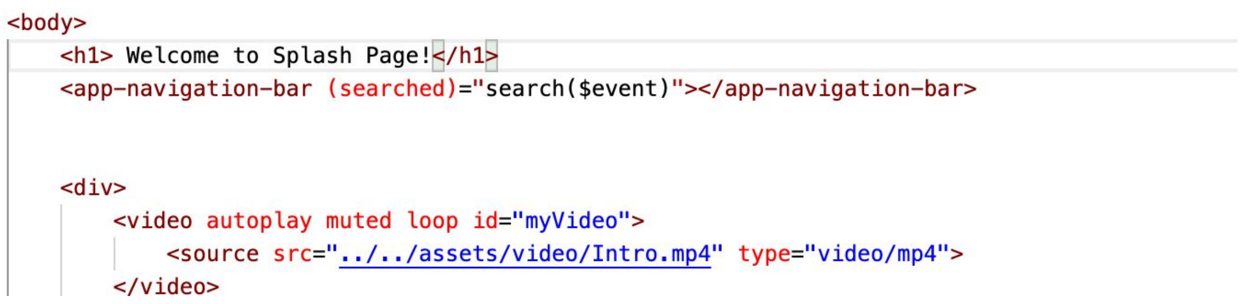
## 3. Template Method:

For the front-end we decided to implement the Template Method with our navigation bar.  We initially had the navigation bar html code and functionality on every single page. We then created a new navigation bar component and put our navigation bar html code as well as the typescript file in the new component.We imported the navigation bar in every individual page that needed it by inserting <app-navigation-bar></app-navigation-bar> instead of having the whole html written out for every file. The parent class for us would be the navigation-bar component, and the subclasses would be each of the specific pages. The functions that the subclasses did not have to implement were the button redirection functions. The subclasses only had to implement the search redirection function because if you were already on the search page, we did not want to redirect and refresh the page. This really benefits us because when we create new classes we can easily implement a navigation bar with the code above.
Code Before and After with UML:



```
<body>
  <h1> Welcome to Splash Page!</h1>
  <nav class="navbar navbar-expand-md bg-dark navbar-dark fixed-top">
    <a id="home-nav" class="navbar-brand" href="#" style="font-family:'Audiowide'">Comic Book DB</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarCollapse" aria-co
      aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarCollapse">
      <ul class="navbar-nav mr-auto">
        <li class="nav-item ">
        <a id="char-nav" class="nav-link" routerLink="/characters">Characters</a>
      </li>
      <li class="nav-item ">
      <a id="issues-nav" class="nav-link" routerLink="/issues">Issues</a>
      </li>
      <li class="nav-item ">
      <a id="authors-nav" class="nav-link" routerLink="/authors">Authors</a>
      </li>
      <li class="nav-item ">
      <a id="about-nav" class="nav-link" routerLink="/about">About</a>
      </li>
      </ul>
    </div>
    <app-search-bar (searched)="search($event)" [model]="searchType"></app-search-bar>
  </nav>


  <div>
    <video autoplay muted loop id="myVideo">
```





```
<body>
    <h1> Welcome to Splash Page!</h1>
    <app-navigation-bar (searched)="search($event)"></app-navigation-bar>


    <div>
        <video autoplay muted loop id="myVideo">
            <source src="../../assets/video/Intro.mp4" type="video/mp4">
        </video>
```