Design Report

I. Information Hiding

Before redesigning and refactoring our project, there was little to no information hiding. We had all our functionality for calling the APIs, initializing the database, rendering the web pages, and working with the 3 model pages all in one location. This was a bad design decision since there was not a lot of modularity, abstraction, or information hiding of any sort. We decided to incorporate the Facade pattern and make several refactorings in our project. When deciding which design pattern to use, we knew that we wanted to hide the majority of the logic used in initialization and rendering of our web pages and content. To use the Facade pattern, we created a ModelFacade class that is used as an interface to easily setup the database and render the web pages. The logic for setting up the database and rendering the web pages is hidden in our API model classes and backend classes. The API model classes have the actual code for initializing the mongoDB collections from the API. The backend classes include functionality for working with the model array, instances, filtering of the content, and rendering the model page. We purposely had weak coupling because we knew the API initialization process should be differentiated from main logic that services our website because we only call the API sparingly when we want to re-initialize our mongoDB collection. This prevents any unintended dependencies between code that should not be related in any way. The ModelBackend class along with the ModelBackendInterface interface modularize the code and functionality and improve cohesion within a module. We also have a class for each model that details the attributes we want to track for the model instance. By refactoring our code to include the API classes, backend classes and interface, model classes, and implementing the Facade design pattern, we have encapsulated our logic for initializing the database from the APIs and other logic needed to ensure that our web pages are displaying the correct information and functions as expected.

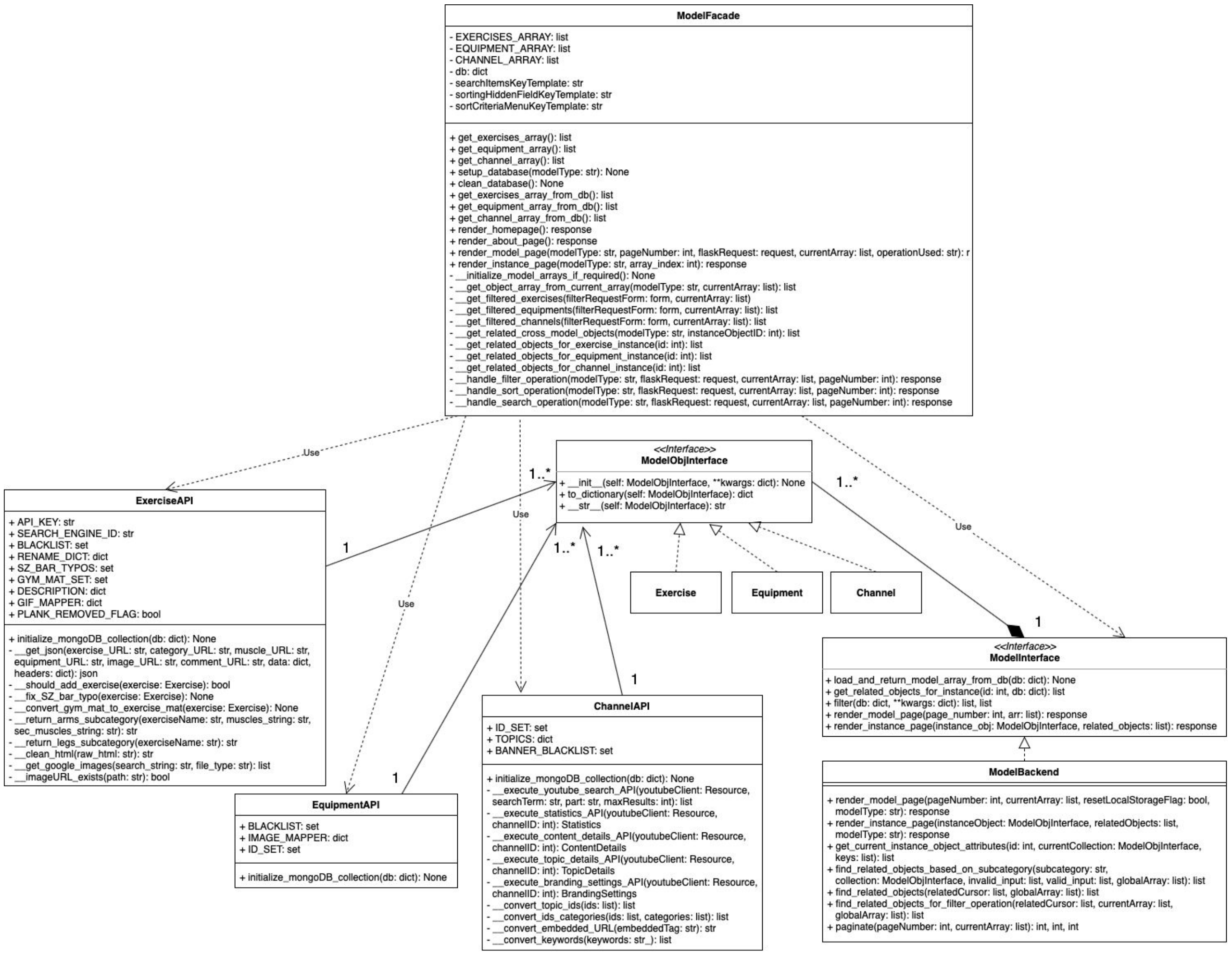
By modularizing our project, we have successfully applied the principle of information hiding. Logic that is likely to change depending on the implementation is hidden within the modules. This includes information like how our databases are initialized from calling the APIs, how related objects are found, how instances are filtered, etc. Information that is unlikely to change includes the functions used to render the html templates, functions each model class needs to implement, and logic that is shared across all current and future model pages. Our modularization scheme makes our program robust to changes since you can easily add new features and functionality without having to restructure the modules or make large changes to the code. For example, we can easily add a new model page by creating a new API class that is in charge of calling the API to initialize the mongoDB collection along with a model class that details the attributes of the model. We add an html file that inherits from the base html file that details how the new model will be displayed on our website. Finally, we add a function to the ModelFacade class and main class for rendering the web page. Since the backend model subclasses share common code, we won't need to implement it for the new model. By modularizing our code, we have made it easier to add new content and functionality to our project by quickly knowing where to add a new class or function. This has also allowed us to change functionality and code within one class without heavily affecting other classes (weak coupling).

Our modularization approach has a few disadvantages. One disadvantage is that our main class only calls methods from the ModelFacade class which limits interaction with the subsystems. We can mitigate this by adding functions to the main class or ModelFacade class that helps us interact with the subsystems more similar to setter and getter functions. Another disadvantage is that our modularization approach was a bit over-modularized in some areas which caused us to pass many parameters into our ModelFacade functions that were passed to our backend classes. This required us to do a lot of tracing to figure out which object we were working with. When adding code to a file, it was easy for us to miss adding it to another file which used the same instance. In some cases, we had strong coupling. Having a centralized area of code would mitigate this disadvantage, so we refactored the code to have our main class be the main entrypoint for working with the objects.

II. Design Patterns

Facade Pattern

For the Facade Pattern, we added the ModelFacade python class to act as the higher-level interface that made it easier to use the backend and API subsystems. For this Facade Design Pattern, we were able to use the ModelFacade class to handle all actions including initializing the database with the APIs, loading data from the databases, and correctly rendering the site pages without having to access the subsystems directly. Note that because we were coding in Python, when defining an interface, it is still created with the class keyword. In addition to adding the ModelFacade class, this was also done by refactoring the two subsystems so that the API classes of the three models were separated from the backend class which implements the Model Interface. Both of these subsystems are also composed of instances of the three models -- Exercise, Equipment, Channel. This is displayed in Figure 2-1 in the UML class diagram below.



```
ModelFacade
- EXERCISES_ARRAY: list
- EQUIPMENT_ARRAY: list
- CHANNEL_ARRAY: list
- db: dict
- searchItemsKeyTemplate: str
- sortingHiddenFieldKeyTemplate: str
- sortCriteriaMenuKeyTemplate: str

+ get_exercises_array(): list
+ get_equipment_array(): list
+ get_channel_array(): list
+ setup_database(modelType: str): None
+ clean_database(): None
+ get_exercises_array_from_db(): list
+ get_equipment_array_from_db(): list
+ get_channel_array_from_db(): list
+ render_homepage(): response
+ render_about_page(): response
+ render_model_page(modelType: str, pageNumber: int, flaskRequest: request, currentArray: list, operationUsed: str): r
+ render_instance_page(modelType: str, array_index: int): response
- __initialize_model_arrays_if_required(): None
- __get_object_array_from_current_array(modelType: str, currentArray: list): list
- __get_filtered_exercises(filterRequestForm: form, currentArray: list)
- __get_filtered_equipments(filterRequestForm: form, currentArray: list): list
- __get_filtered_channels(filterRequestForm: form, currentArray: list): list
- __get_related_cross_model_objects(modelType: str, instanceObjectID: int): list
- __get_related_objects_for_exercise_instance(id: int): list
- __get_related_objects_for_equipment_instance(id: int): list
- __get_related_objects_for_channel_instance(id: int): list
- __handle_filter_operation(modelType: str, flaskRequest: request, currentArray: list, pageNumber: int): response
- __handle_sort_operation(modelType: str, flaskRequest: request, currentArray: list, pageNumber: int): response
- __handle_search_operation(modelType: str, flaskRequest: request, currentArray: list, pageNumber: int): response
```

```
<<Interface>>
ModelObjInterface
+ __init__(self: ModelObjInterface, **kwargs: dict): None
+ to_dictionary(self: ModelObjInterface): dict
+ __str__(self: ModelObjInterface): str
```

```
ExerciseAPI
+ API_KEY: str
+ SEARCH_ENGINE_ID: str
+ BLACKLIST: set
+ RENAME_DICT: dict
+ SZ_BAR_TYPOS: set
+ GYM_MAT_SET: set
+ DESCRIPTION: dict
+ GIF_MAPPER: dict
+ PLANK_REMOVED_FLAG: bool

+ initialize_mongoDB_collection(db: dict): None
- __get_json(exercise_URL: str, category_URL: str, muscle_URL: str,
  equipment_URL: str, image_URL: str, comment_URL: str, data: dict,
  headers: dict): json
- __should_add_exercise(exercise: Exercise): bool
- __fix_SZ_bar_typo(exercise: Exercise): None
- __convert_gym_mat_to_exercise_mat(exercise: Exercise): None
- __return_arms_subcategory(exerciseName: str, muscles_string: str,
  sec_muscles_string: str): str
- __return_legs_subcategory(exerciseName: str): str
- __clean_html(raw_html: str): str
- __get_google_images(search_string: str, file_type: str): list
- __imageURL_exists(path: str): bool
```

```
EquipmentAPI
+ BLACKLIST: set
+ IMAGE_MAPPER: dict
+ ID_SET: set

+ initialize_mongoDB_collection(db: dict): None
```

```
ChannelAPI
+ ID_SET: set
+ TOPICS: dict
+ BANNER_BLACKLIST: set

+ initialize_mongoDB_collection(db: dict): None
- __execute_youtube_search_API(youtubeClient: Resource,
  searchTerm: str, part: str, maxResults: int): list
- __execute_statistics_API(youtubeClient: Resource,
  channelID: int): Statistics
- __execute_content_details_API(youtubeClient: Resource,
  channelID: int): ContentDetails
- __execute_topic_details_API(youtubeClient: Resource,
  channelID: int): TopicDetails
- __execute_branding_settings_API(youtubeClient: Resource,
  channelID: int): BrandingSettings
- __convert_topic_ids(ids: list): list
- __convert_ids_categories(ids: list, categories: list): list
- __convert_embedded_URL(embeddedTag: str): str
- __convert_keywords(keywords: str_): list
```

```
<<Interface>>
ModelInterface
+ load_and_return_model_array_from_db(db: dict): None
+ get_related_objects_for_instance(id: int, db: dict): list
+ filter(db: dict, **kwargs: dict): list, list
+ render_model_page(page_number: int, arr: list): response
+ render_instance_page(instance_obj: ModelObjInterface, related_objects: list): response
```

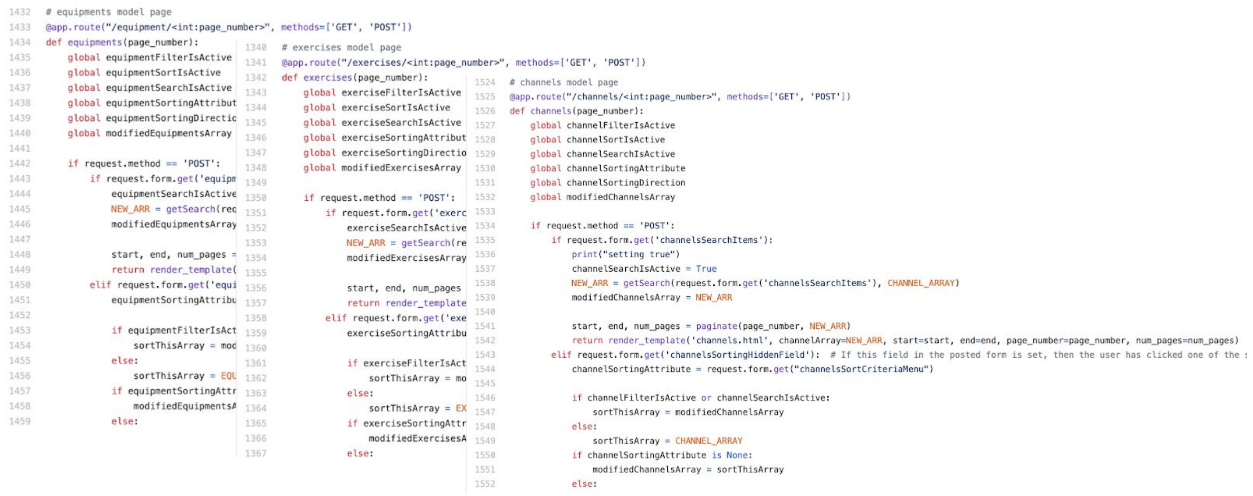
```
ModelBackend

+ render_model_page(pageNumber: int, currentArray: list, resetLocalStorageFlag: bool,
  modelType: str): response
+ render_instance_page(instanceObject: ModelObjInterface, relatedObjects: list,
  modelType: str): response
+ get_current_instance_object_attributes(id: int, currentCollection: ModelObjInterface,
  keys: list): list
+ find_related_objects_based_on_subcategory(subcategory: str,
  collection: ModelObjInterface, invalid_input: list, valid_input: list, globalArray: list): list
+ find_related_objects(relatedCursor: list, globalArray: list): list
+ find_related_objects_for_filter_operation(relatedCursor: list, currentArray: list,
  globalArray: list): list
+ paginate(pageNumber: int, currentArray: list): int, int, int
```

Fig. 2-1: UML diagram demonstrating implementation of Facade design pattern.

Our problem fits the Facade Pattern because there only needs to be one class that is accessed to run this site properly. This means that main.py need only communicate with the ModelFacade class without knowing the structure of the other subsystems and how they interact.

Additionally, we can see the benefits of implementing this design pattern in our class diagram. Although this project is complex and has many working parts, such as all of the methods and classes in both the API and backend subsystems, this simplifies the code so that main.py only needs to interact with the model_facade.py file. Additionally, because there is only one point of entry for the code, coupling is decreased. This provides benefits for programmers as the interdependency between modules is limited. Before implementing the pattern, you can see in Figure 2-2, that in main.py these methods are very similar and yet still in separate functions. You can also see that our file is thousands of lines long. However, in Figure 2-3 you can see that the methods are called from one entry point, ModelFacade. Note that all three functions to render the model pages are the same length as the method displayed.



```
# equipments model page
@app.route("/equipment/<int:page_number>", methods=['GET', 'POST'])
def equipments(page_number):
    global equipmentFilterIsActive
    global equipmentSortIsActive
    global equipmentSearchIsActive
    global equipmentSortingAttribut
    global equipmentSortingDirectic
    global modifiedEquipmentsArray

    if request.method == 'POST':
        if request.form.get('equipm
            equipmentSearchIsActive
            NEW_ARR = getSearch(req
            modifiedEquipmentsArray

            start, end, num_pages =
            return render_template(
        elif request.form.get('equi
            equipmentSortingAttribu

            if equipmentFilterIsAct
                sortThisArray = moc
            else:
                sortThisArray = EQU
            if equipmentSortingAttr
                modifiedEquipmentsA
            else:
```

```
# exercises model page
@app.route("/exercises/<int:page_number>", methods=['GET', 'POST'])
def exercises(page_number):
    global exerciseFilterIsActive
    global exerciseSortIsActive
    global exerciseSearchIsActive
    global exerciseSortingAttribut
    global exerciseSortingDirectio
    global modifiedExercisesArray

    if request.method == 'POST':
        if request.form.get('exerc
            exerciseSearchIsActive
            NEW_ARR = getSearch(re
            modifiedExercisesArray

            start, end, num_pages
            return render_template
        elif request.form.get('exe
            exerciseSortingAttribu

            if exerciseFilterIsAct
                sortThisArray = mo
            else:
                sortThisArray = EX
            if exerciseSortingAttr
                modifiedExercisesA
            else:
```

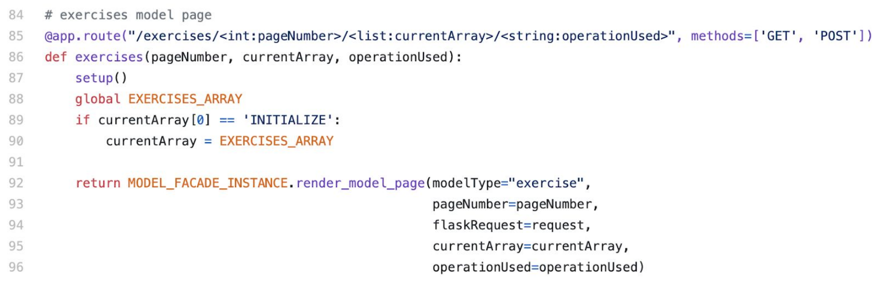
```
# channels model page
@app.route("/channels/<int:page_number>", methods=['GET', 'POST'])
def channels(page_number):
    global channelFilterIsActive
    global channelSortIsActive
    global channelSearchIsActive
    global channelSortingAttribute
    global channelSortingDirection
    global modifiedChannelsArray

    if request.method == 'POST':
        if request.form.get('channelsSearchItems'):
            print("setting true")
            channelSearchIsActive = True
            NEW_ARR = getSearch(request.form.get('channelsSearchItems'), CHANNEL_ARRAY)
            modifiedChannelsArray = NEW_ARR

            start, end, num_pages = paginate(page_number, NEW_ARR)
            return render_template('channels.html', channelArray=NEW_ARR, start=start, end=end, page_number=page_number, num_pages=num_pages)
        elif request.form.get('channelsSortingHiddenField'):  # If this field in the posted form is set, then the user has clicked one of the
            channelSortingAttribute = request.form.get("channelsSortCriteriaMenu")

            if channelFilterIsActive or channelSearchIsActive:
                sortThisArray = modifiedChannelsArray
            else:
                sortThisArray = CHANNEL_ARRAY
            if channelSortingAttribute is None:
                modifiedChannelsArray = sortThisArray
            else:
```

Fig. 2-2: functions for rendering model pages in main.py before refactoring



```
# exercises model page
@app.route("/exercises/<int:pageNumber>/<list:currentArray>/<string:operationUsed>", methods=['GET', 'POST'])
def exercises(pageNumber, currentArray, operationUsed):
    setup()
    global EXERCISES_ARRAY
    if currentArray[0] == 'INITIALIZE':
        currentArray = EXERCISES_ARRAY

    return MODEL_FACADE_INSTANCE.render_model_page(modelType="exercise",
                                                   pageNumber=pageNumber,
                                                   flaskRequest=request,
                                                   currentArray=currentArray,
                                                   operationUsed=operationUsed)
```

Fig. 2-3: function for rendering a model page in main.py after refactoring

However, there are also some disadvantages to applying this design pattern. For example, using the Facade Pattern limits how you directly interact with the subsystem. Because main.py only interacts with the ModelFacade class, we can only execute the methods in ModelFacade. Another disadvantage is that if one of the subsystem's structure changes, then a significant amount of code may have to be changed in main.py, model_facade.py, and any other files affected by the change.

IV. Refactorings

A. Refactoring API

Firstly, we refactored the code for the API. Originally, our API code was all in one file and was not separated by class. This greatly limited our information hiding and greatly reduced our modularity. However, after refactoring, the API code for each of the models were each split into their own classes. This can be seen in Figure 3-1.

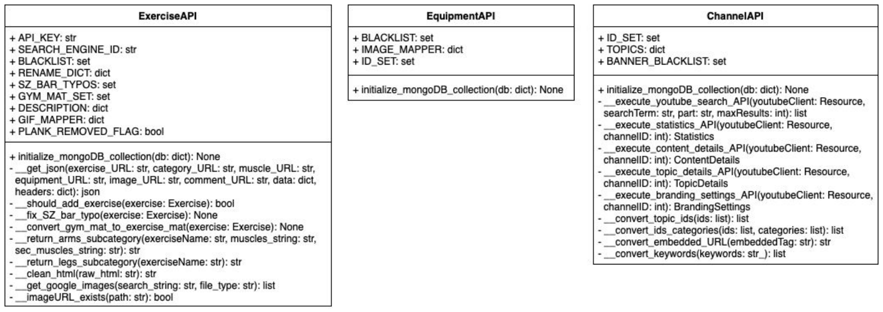

ExerciseAPI
+ API_KEY: str
+ SEARCH_ENGINE_ID: str
+ BLACKLIST: set
+ RENAME_DICT: dict
+ SZ_BAR_TYPOS: set
+ GYM_MAT_SET: set
+ DESCRIPTION: dict
+ GIF_MAPPER: dict
+ PLANK_REMOVED_FLAG: bool
+ initialize_mongoDB_collection(db: dict): None
- __get_json(exercise_URL: str, category_URL: str, muscle_URL: str, equipment_URL: str, image_URL: str, comment_URL: str, data: dict, headers: dict): json
- __should_add_exercise(exercise: Exercise): bool
- __fix_SZ_bar_typo(exercise: Exercise): None
- __convert_gym_mat_to_exercise_mat(exercise: Exercise): None
- __return_arms_subcategory(exerciseName: str, muscles_string: str, sec_muscles_string: str): str
- __return_legs_subcategory(exerciseName: str): str
- __clean_html(raw_html: str): str
- __get_google_images(search_string: str, file_type: str): list
- __imageURL_exists(path: str): bool
EquipmentAPI
+ BLACKLIST: set
+ IMAGE_MAPPER: dict
+ ID_SET: set
+ initialize_mongoDB_collection(db: dict): None
ChannelAPI
+ ID_SET: set
+ TOPICS: dict
+ BANNER_BLACKLIST: set
+ initialize_mongoDB_collection(db: dict): None
- __execute_youtube_search_API(youtubeClient: Resource, searchTerm: str, part: str, maxResults: int): list
- __execute_statistics_API(youtubeClient: Resource, channelID: int): Statistics
- __execute_content_details_API(youtubeClient: Resource, channelID: int): ContentDetails
- __execute_topic_details_API(youtubeClient: Resource, channelID: int): TopicDetails
- __execute_branding_settings_API(youtubeClient: Resource, channelID: int): BrandingSettings
- __convert_topic_ids(ids: list): list
- __convert_ids_categories(ids: list, categories: list): list
- __convert_embedded_URL(embeddedTag: str): str
- __convert_keywords(keywords: str_): list

Fig. 3-1: UML class diagrams for the three model's API classes after refactoring

B. Refactoring Backend

The second refactoring we did was refactoring our backend code which handled the communication and requests between the client and server. Here we added both an interface and a superclass that implements this interface since all three backend models implemented the same 5 methods declared in the ModelInterface interface and also re-used the same 4 methods declared in the ModelBackend superclass. This refactoring can be seen in Figure 3-3 below. In the ModelBackend class it can be seen that methods that apply to multiple classes can be condensed into one file. This greatly reduced the length and complexity of our code, and in turn decreased the dependencies between instances and the methods performed on them.

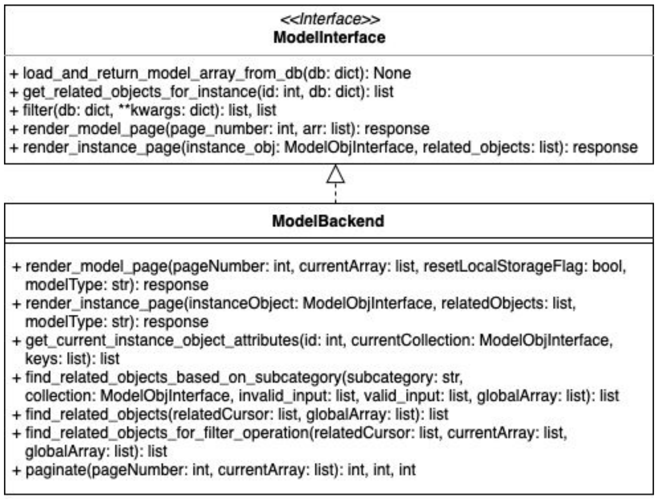

<<Interface>>
ModelInterface
+ load_and_return_model_array_from_db(db: dict): None
+ get_related_objects_for_instance(id: int, db: dict): list
+ filter(db: dict, **kwargs: dict): list, list
+ render_model_page(page_number: int, arr: list): response
+ render_instance_page(instance_obj: ModelObjInterface, related_objects: list): response
ModelBackend
+ render_model_page(pageNumber: int, currentArray: list, resetLocalStorageFlag: bool, modelType: str): response
+ render_instance_page(instanceObject: ModelObjInterface, relatedObjects: list, modelType: str): response
+ get_current_instance_object_attributes(id: int, currentCollection: ModelObjInterface, keys: list): list
+ find_related_objects_based_on_subcategory(subcategory: str, collection: ModelObjInterface, invalid_input: list, valid_input: list, globalArray: list): list
+ find_related_objects(relatedCursor: list, globalArray: list): list
+ find_related_objects_for_filter_operation(relatedCursor: list, currentArray: list, globalArray: list): list
+ paginate(pageNumber: int, currentArray: list): int, int, int

Fig. 3-2: UML class diagram for the backend class and the interface it implements after refactoring

Note that both the API code and the backend code were further refactored by using multiple instances of Channel, Equipment, and Exercise which implement the ModelObjInterface. Originally, these dictionary initializations were in two different places. Realizing this during refactoring, we defined classes as shown in Figure 3-3.

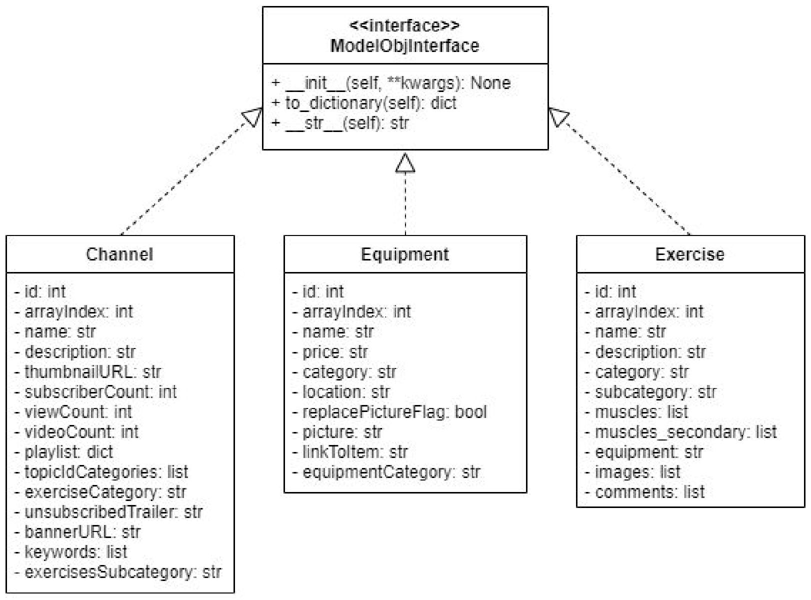


Figure 3-3: UML class diagram for the Channel, Equipment, and Exercise classes

C. Refactoring Search, Sort, and Filter

Finally, we refactored the way that filtering, sorting, and searching were implemented so that the site can better handle multi-client users on the site. Prior to this refactoring, the site was sometimes unstable. We were maintaining a modified array in the server that was updated based on the client's choices of filtering, searching, or sorting; however, this meant that all clients would be passed the same modified array, even though not all clients may have chosen the exact same operations. Yet, after the refactoring, the logic on the backend is much more concise and clear, and the modified array specific to the client is passed from the frontend to the backend. This ensures that no modified array is stored on the server and in turn ensures our site can support multiple clients properly. Before refactoring, you can see in Figure 3-5, that the Flask route only took in page number as a parameter. However, after refactoring, as shown in Figure 3-6, the Flask route now takes in the current array which the operations are being performed on. We also passed an additional parameter to specify which operation is being used.

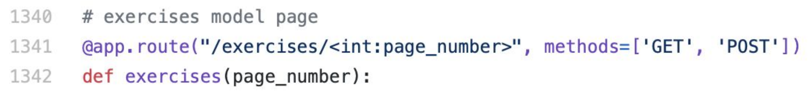
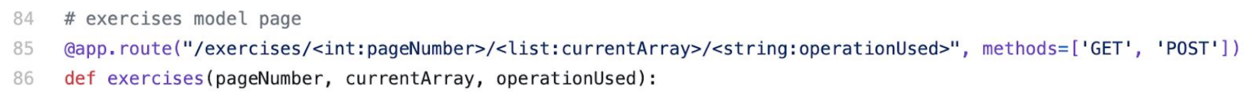

```
# exercises model page
@app.route("/exercises/<int:page_number>", methods=['GET', 'POST'])
def exercises(page_number):
```

Fig. 3-4: Flask route declaration before refactoring



```python
# exercises model page
@app.route("/exercises/<int:pageNumber>/<list:currentArray>/<string:operationUsed>", methods=['GET', 'POST'])
def exercises(pageNumber, currentArray, operationUsed):
```

Fig. 3-5: Flask route declaration after refactoring, with additional parameters of currentArray and operationUsed

With each of these refactorings, the code was greatly modularized by dividing much of the modules and allowing them to implement from very descriptive interfaces. This could especially be seen in the main file which originally held the most of our backend code, but now is a very short and concise file that handles the Flask operations needed to navigate to different pages on the site. It can also be seen that we implemented procedural abstraction in each of these refactorings. Many of our methods were used to increase code reusability; this means much of our code can be reused and can decide how to act based on other factors such as parameters passed into our methods. Finally, we weakened the coupling. Originally, all objects in our code could easily be accessed by other objects; however, our implementation of a design pattern has ensured that only the ModelFacade class can be accessed by main.py and can relate to other modules. Furthermore, our separation of classes and interfaces has further reduced the connections between modules. With these large refactorings and other refactorings we completed, we greatly reduced the complexity of our code and made it significantly easier for any future changes we decide to implement. For example, if we decided to add another model viewable on the site, this would be fairly simple. These refactorings are largely evident in our codebase and have greatly improved our code.