**Design Document**

**Group 1**

Design Pattern: Model View Controller

In our code we used the django framework and implemented the MVC design pattern to ensure maintainable and scalable code.

Model: We have several different models within our codebase: models for the LLM, models for users, and models for workouts. The models represent our data layer, and are used to define the structure of our data and for managing the data.

View: complimenting the models we also have the presentation layer. The views handle UI logic and interact with the models to retrieve or modify data. The views in our project process http requests and return http responses, enabling the frontend to interface with the database and perform operations like creating, updating, or deleting records.

Controller: In django the views also act as the controller in the MVC design pattern. The controller manages the transfer of data between the models and views. It does this by processing the incoming requests and retrieves or modifies the data within the model, and returns the output in the form of a view. Using a controller allows for the separation between data, logic and the user interface allowing each to be maintained easier.

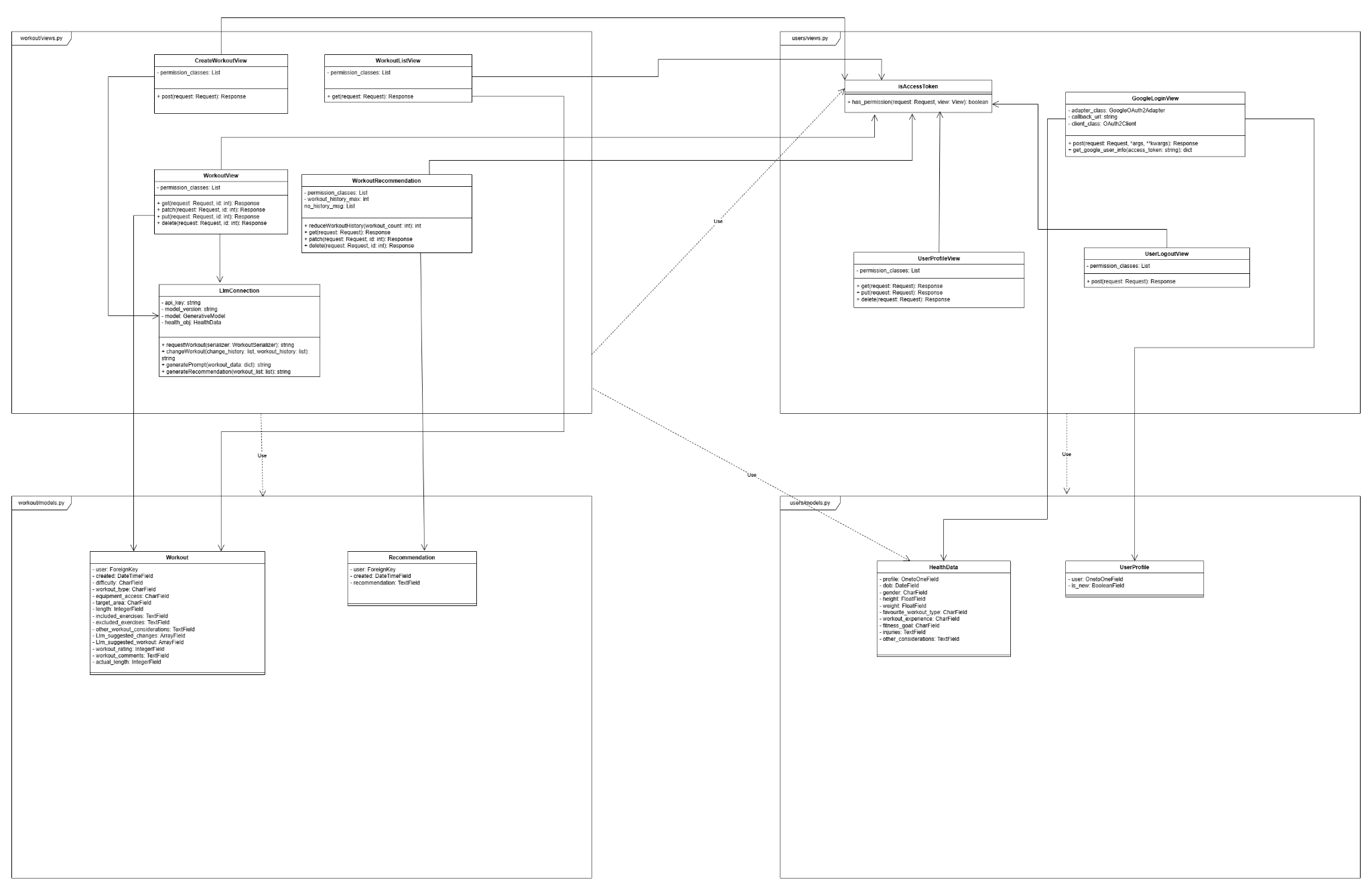
Some of our views and models are shown in figure 1.

We have applied the SOLID design principles to ensure a well maintained code base.

Single Responsibility: by following the MVC design pattern, many of our classes are responsible for a single and well defined task. For example, the CreateWorkoutView is only responsible for creating the initial/new workout, and as such is not able to edit the workout, ensuring one responsibility. Similarly, our class IsAccessToken is only responsible for ensuring other classes have permission to access particular data.

Open Closed: We have tried to design our classes in such a way where a class can be extended, but is unable to be modified. This is emphasized by our IsAccessToken class having the ability to be extended to add additional logic, without changing its functionality. Additionally it would be simple enough for us to extend our GoogleLoginView to include additional login providers if we had a desire to do so.

Interface Segregation: Our IsAccessToken class serves as an interface for handling token-based authentication, allowing our other classes that need to handle authentication to not depend on unnecessary methods.



*Figure 1 UML Class Diagram for (workout & users) views and models*