University of Leeds **School of Computer Science**

COMP3011, 2024-2025

Web Services and Web Data

A RESTful API for Rating Professors

By

Toby Hutchinson

201530569

sc21t2hh@leeds.ac.uk

Date: 10/03/25

1. The Database

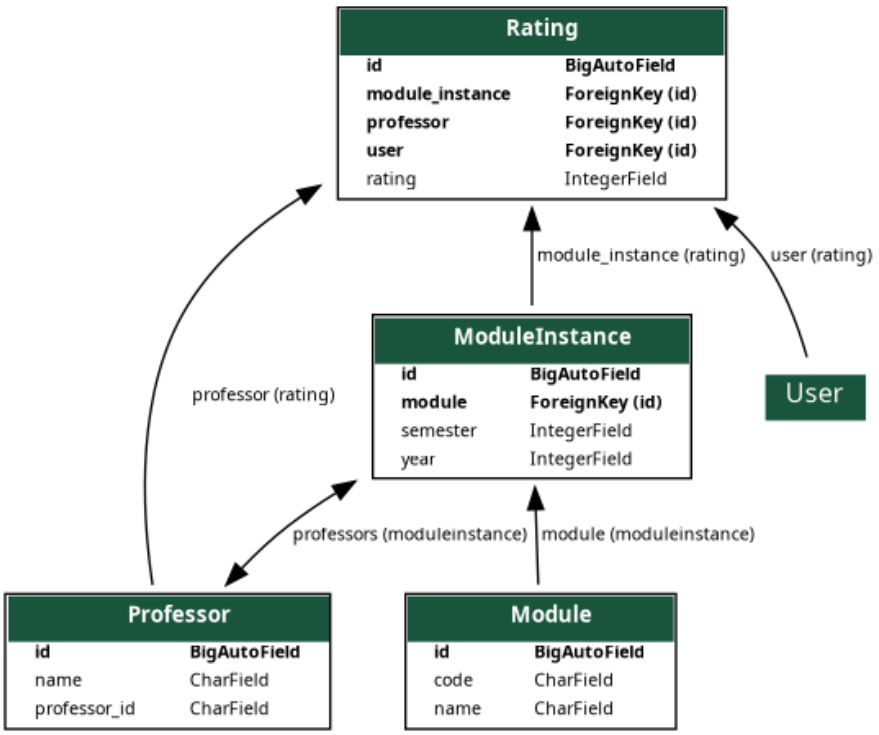

Figure 1 - The models used, their fields and the relationship between them

In *figure 1* you can see the 4 main models and their fields used in the implementation of the service. The *ModuleInstance* model also contains a *professors* field which isn't shown in the graph. The *id* fields of each model are the primary keys and they're automatically generated by Django. The *User* model is automatically generated by Django and stores usernames, emails and passwords, among other fields and implements validation checks.

The fields *professor_id* in *Professor*, *code* in *Module* must be unique. The database enforces *module, year* and *semester* in *ModuleInstance* to all be unique together. This means no two ModuleInstances can have the same *Module*, *year* and *semester*. This is also true for *user, professor* and *module_instance* in *Rating*.

The *module* field in *ModuleInstance* is a foreign key which links to a *Module* as a one-to-one relationship. The *professors* field is a foreign key to one or more *Professor* instances. This is implemented as a many-to-many field, since many professors may teach a *ModuleInstance*, and a *Professor* may teach many *ModuleInstances*.

In the *Rating* model, *user*, *professor* and *module_instance* are all foreign keys. These are all one-to-one relationships. A *Rating* relates to one *Professor* and one *ModuleInstance*.

All foreign keys refer to the *id* field of the respective model. All foreign keys' *on_delete* setting is set to *CASCADE*, so that if a *Module* is deleted, all *ModuleInstances* are also deleted. The same goes for *Ratings* when a *User* is deleted. This helps prevent possible errors that occur when values are null and makes the service more robust.

When designing the database I debated whether it was better to use the *code* field in the *Module* model, and *professor_id* in the *Professor* model as the primary keys. This would have been valid since both fields are unique, however leaving Django's built-in *id* field allows for flexibility and should the situation arise where the module code and professor ID needs to be changed, they can.

2. The API

api/csrf-token/

This endpoint takes *GET* requests. It allows users to get a CSRF token which they include in further requests. It is assumed that all requests include the CSRF token.

api/register/

This endpoint allows users to register a new account. The endpoint accepts *POST* requests. The request must contain a username, email and password. The data in the request's body should be valid JSON. If it isn't, the server responds with an "Invalid JSON" message, and a status code of 400, bad request. If there are any issues with the data in the request, the server sends all errors back to the client with a status code of 400. Errors are created using the *UserSerializer*, and they could be missing fields, non-unique username or email, invalid email and so on. If all data is correct, the server responds with a "Registration Successful." message, and a status code of 201, resource created.

api/login/

The login endpoint allows users to authenticate. It accepts *POST* requests and expects valid JSON. In the absence of correct JSON, it responds in the same way as **api/register/**. The body of the request should contain the username and password of the user. If either of these fields are missing the server responds "Please provide a username and a password", and a status of 400. If the credentials are present, but incorrect the server responds with "Invalid credentials" and a status of 401, unauthorized. If the credentials are correct, the server responds with "Login Successful", a status code of 200, and includes an authentication token for the user, which can be included in future requests in order to authenticate.

api/logout/

The logout endpoint expects a *POST* request from an authenticated user. This means the request must contain the authorization token in the request's body. If the request is authenticated the server deletes the user's token and responds with "Logout Successful" and a status of 200. If there's no authentication token present, the server responds with "User is not authenticated." and a status code of 401, unauthorized.

api/list/

The list endpoint returns all module instances and their corresponding professors. It accepts *GET* requests. The server fetches all *ModuleInstances* and serializes them into JSON. The server responds with the "Successfully fetched module instances" message and another "module_instances" field which contains all module instances. The response has a status of 200.

api/view/

This endpoint returns a list of professors and their average rating across all module instances. The endpoint accepts GET requests. The endpoint loops through all professors and calculates their average rating. The server responds with the "Successfully fetched module instances." message and the "professor_ratings" field, which is an array of objects which contain the name, ID and average rating as an integer of each professor. The response has a status of 200.

api/average/

This endpoint returns the average rating of a professor for a given module. The endpoint accepts *GET* requests. The professor and module are identified by their *professor_id* and *module_code* which are included in the request as JSON. If the JSON is invalid or it's missing the *professor_id* or *module_code* field, the server responds with "Expected 'professor_id' and 'module_code' in JSON format." and a status code of 400. The server attempts to fetch the module and professor using the code and ID, if they don't exist the server responds with "Error: 'module_code' or 'professor_id' is incorrect. Check values and try again." and a status of 400. The server then checks to see if the professor has taught the module, if not it responds with "Chosen professor has never taught that module." and a status of 400. If the data provided is correct, the server calculates the average across all ratings and responds with "Successfully fetched average rating for <professor_id>, <module_code>" and a status of 200. The response also includes the name of the professor and the module, as well as the professor's ID, module code and average rating.

api/rate/

The rate endpoint allows authenticated users to rate a professor for a module instance. It accepts *POST* requests. The request expects data to be in JSON format and if it's not the server responds with an "Invalid JSON" message and a status of 400. The request should contain the ID of the professor, the code, year and semester of the module instance, and the user's rating. If any of the data is missing the server responds with "Request is missing required keys." and a status of 400. If the ID of the professor is incorrect, the server responds with "Professor could not be retrieved." and a status of 400. If no module instance is matched the server responds with "Module instance could not be retrieved." and a status of 400. If the rating field in the request cannot be made into an integer, the server responds with "Rating must be integer value", and a status of 400. The server then attempts to create a new *Rating* instance using the supplied data. If the rating is not between 1 and 5 the server responds with "Rating must be between 1 and 5" and a status of 400. If the *Rating* is created successfully, the server responds with "Rating created successfully" and a status of 201, resource created.

The endpoint also checks for errors in the database. If multiple *ModuleInstances* are matched, the server responds with "Internal Error: Multiple instances were matched." and a status of 500, internal server error. While the database also enforces uniqueness, this adds another level of security and robustness.

All responses are sent using Django Rest Framework's *Response* class which ensures the responses are formatted into JSON correctly. Responses are kept consistent with Django Rest Framework's responses and all endpoints respond using the "*detail*" key.

3. The Client

The client is implemented as a class. The main driver of the client is an infinite loop which prompts the user for input. In the constructor the client stores the CSRF token using a call the **api/csrf-token/**. Commands are stored as a dictionary of command names and functions. Each function correlates to an API endpoint. Each endpoint function checks the number of arguments and prompts the user for correct usage if incorrect. Data is built up into Python dictionaries which are easily translated into JSON using the Requests library. Each function is responsible for creating the correct headers and data.

The *_clientRequest* function handles sending the requests and any errors that occur with the request or the response. Any responses that have a status code greater than or equal to 400 and less than 600 are flagged as an error. The client then attempts to decode the error. If the server has detected an error, the response will contain the "detail" field, the client can then print out the errors which have been formatted by the server. If the response doesn't contain JSON, this is considered a server error and the message "Server Error. Could not decode error response." is printed. If the "field" key isn't present, "Format of response was incorrect: <error>" is printed. In the rare case that more generic *RequestExceptions* occur, they are printed like so: "An Error Occurred: <error>". If the server responds with a successful status code, but the response isn't in JSON format, the client prints "Server Error: Invalid JSON response from server.". After capturing and displaying error messages, the client returns to the main loop without exiting.

On success, the endpoint functions each handle the data returned to them and display appropriate messages. The Rich library is used to display tables in the terminal.

4. Using the client

Requirements for the client are listed in "*myclient/requirements.txt*" and can be installed using "*pip install -r requirements.txt*". The two dependencies are the Requests and Rich libraries. To run the program call "*python myclient/client.py*". Once running you can run the following commands:

- **'register'** - Prompts you for a username, email and password. The client then attempts to register a new user with these credentials.
- '**login <URL>**' - Prompts you for username and password and attempts to log you in. <URL> is the URL of the API.
- **'logout'** - Attempts to log you out of the web app. Users must be logged-in to logout.
- **'list'** - Returns a list of all module instances and the professors teaching them.
- **'view'** - Returns a list of all professors and their overall rating across all modules.
- '**average <professor_id> <module_code>**' - Returns the average rating of the professor (<professor_id>) for the specified module (<module_code>).
- **'rate <professor_id> <module_code> <year> <semester> <rating>'** - Creates a rating for a specific professor (<professor_id>) for the specified module instance. The user must be logged-in in order to rate a module instance.
- **'help'** - Returns a list of valid commands.
- **'exit'** - Exit the client program.