Project Backend Report

CSC309

By: Praket Kanaujia, Michele Massa, Aakash Vaithyanathan

- Model Design:

Model	*Description*	*Contents*
accounts	The model that stores the user and each user makes up a row in the database.	- The base user has been implemented but extended using AbstractUser to allow for the usage of the fields from the base user - **avatar:** The avatar of the user that is an image field - **phone_number:** The phone_number of the user
studios	This model keeps track of any studios that are added by the admin.	- **name:** The name of the studio - **address:** The address of the studio - **latitude:** The latitude of the studio (used for finding the distance to user) - **longitude:** The longitude of the studio (used for finding the distance to the user) - **postal_code:** The postal code of the studio - **phone_number:** The phone_number of the studio
images	This model stores the images of the studios that can be inserted by the admin.	**- studio:** The corresponding studio to which the image belongs to **- image:** The image of the studio that is an image field
amenities	This model stores the amenities of the studios, corresponding to each studio.	**- studio:** The corresponding studio to which the amenity belongs to **- type:** The type of amenity that is present at the studio **- quantity:** The quantity of the amenity at the studio
Classes	The model storing the Studio-Class relationship where each studio has information about the class they offer as mentioned in the handout (along with the information on whether wants	-**studio:** The corresponding studio to which the class belongs to -**name:** The name of the class -**description**: The description of the class -**coach:** The name of the coach who is teaching this

	to cancel a class(s), reactivate class(s) and the ability to edit all fields if necessary. The mentioned additional fields are stored as boolean values indicating the user's preference for such operations).	class -**keywords:** A list of keywords associated with the class -**start_time:** The date and time when the class starts -**end_time:** The date and time when the class ends -**end_recursion:** The date and time up to which the admin wants to create recurring classes. -**is_cancelled:** A boolean field indicating whether the class is canceled or not (by default set to False) -**cancel_all:** A boolean field indicating whether the admin want's to cancel future occurrences of the class (by default set to False) -**reactivate_all:** A boolean field indicating whether the admin want's to reactivate future occurrences of canceled classes (by default set to False) -**edit_all:** A boolean field indicating whether the admin want's to edit class field values for future occurrences of the class (by default set to False)
UserClasses	The model storing the User-Class relationship where each user has information about the class they have enrolled in.	-**user:** A foreign key reference to the user model from accounts indicating the user who is enrolled in the class. -**class_info:** A foreign key reference to the class model from Classes indicating the class details for the user. -**modify_future_classes:** A boolean filed indicating whether the user would like to bulk enroll or bulk drop future occurrences of the class.
Subscriptions	The model storing the user-subscription relationship, where each user with an active subscription makes up a row in the database.	- **Sub_type:** the cost of the subscription the user is affiliated to (14.99 or 149.99) - **Related_user:** a link to the accounts app user model, identifying the owner of the subscription. - **Sub_start_date:** the date the current active subscription started at - **Payment_card:** the 16 digit payment card used to pay for the subscription.
Payment History	The model storing the records of payment of all users that have an active subscription. So if a user has an active subscription, they will be able to see past and up to 1 future (the next) payment in line for such subscription.	-**Related_user:** a link to the accounts app user model, who made this payment. -**Payment_card:** the 16 digit payment card used to make this payment. -**Payment_date**: the date the payment for a subscription went through.

	If a user cancels their subscription, we wipe their information from the payment history as they are no longer a valid subscriber.	**-Payment_amount:** the amount paid by the user for the subscription they had during this period.

- **API Endpoints:**

Endpoint (after LocalHost…)	***Description***	***Usability/Methods***	***Payloads***
/accounts/register/	The endpoint where the user can register themselves	Fill the parameters required for registering the user through the POST request	username, password, email, first_name, last_name,avatar, phone_number The username and password fields are compulsory while the other fields are not required.
/accounts/api/token/	The endpoint where the user can get an api access token in order to be authorized to view further files	Fill the parameters username and password in order to get an api token to be authenticated through a POST request	username, password The username and password are both compulsory to be sent through the post request
/accounts/api/token/refresh	The endpoint where the user can refresh the api token when it runs out	Fill the parameter refresh, and send a POST request in order increase the longevity of the token	refresh The refresh payload in the body contains the token as it's value
/accounts/<int:pk>/update/	The endpoint where the user can update it's parameters	Fill the parameters that want to be updated and send a PATCH request in order to update the fields	URL - id of the user(pk) Body - password, email, first_name, last_name,avatar, phone_number The username field cannot be

			updated
/studios/<int:lat>/<int:long>/all/	Lists out all the studios sorted from the closest to the user	Takes the latitude and longitude from the user through the URL and uses GET to show the list of studios based on the distance to the user.	URL - latitude and longitude are the parameters taken through the url Further improvement on the implementation of the latitude and longitude in frontend
/studios/<int:pk>/details	Lists out the details with regard to the studio that has been mentioned	Takes the studio id in the URL(as pk) and shows the details about that studio including images and amenities	URL - Studio id is the parameter taken through the url
/subscriptions/<int:user_id>/subscription_page/	The main page for the user to view and modify their subscription status.	Use GET (along with the <user_id> in the url) to view your subscription status.	N/A
		Fill the POST form to subscribe (if not subscribed yet).	sub_type (14.99 or 149.99), related_user, sub_start_date, payment_card
		Fill the PUT form to modify payment_card or sub_type (if subscribed), other modifications are omitted.	sub_type (14.99 or 149.99), related_user, sub_start_date, payment_card Note: all can be input, but only sub_type and payment_card lead to changes.
		Use DELETE to delete a subscription (if subscribed)	N/A
/subscriptions/<int:user_id>/payment_history/	The page for a subscribed user to see the payments they have and will make in the future for subscriptions.	Use GET (along with the <user_id> in the url) to view your subscription status.	N/A
/classes/<int:studio_id>/find_class/	The page for viewing the classes offered by a studio and filtering of such search.	Use GET along with the required studio_id to fetch all classes for such studio. And use query parameters to specify filtering.	The payload for this GET refers to QUERY PARAMETERS so make sure to use that in postman (these are the ones that go in the url) following:

			url/?<param>=value&... These are: class_name, coach, date, initial_time, ending_time (Note that date, initial_time and ending_time must be used together to work!).
/studios/find_studio/	The page for viewing the studios available with your subscription.	Use GET along with the supported query parameters to fetch all studios with the applied filters.	The payload for this GET refers to QUERY PARAMETERS so make sure to use that in postman (these are the ones that go in the url) following: url/?<param>=value&... These are: name We wanted to include the others but we have to rethink our models before implementing further filters.
classes/ <str:studio_name>/	The page for viewing the classes schedule of a specific studio	Use GET along with the required studio name (after /classes/) for which you want to view the schedule.	studio, name, description, coach, keywords, capacity, start_time, end_time.
classes/ user/<int:user_id>/	The page for viewing the class schedule and history in chronological order.	Use GET along with the required user's id (after /classes/) for which you want to view the user's schedule.	studio, name, description, coach, keywords, capacity, start_time, end_time.
classes/ user/enroll/<int:user_id>/	The page for enrolling the user into classes from different studios	Use GET along with the required user's id for which you want to enroll the user into different class(s)	N/A (We display the user, class_info → id which references to the class object, modify_future_classes for the class the user wished to enroll into as Response object displayed to the user)
classes/ user/drop/<int:user_id>/	The page for dropping the user from classes of different studios	Use GET along with the required user's id for which you want to drop the user from different class(s)	N/A (We display the user,

			class_info → id which references to the class object, modify_future_classes for the class the user wished to drop from into as Response object displayed to the user)

Note #1 : a couple of our models are very payload heavy and any spelling mistake will obviously result in an error, so despite it not being the established way for testing, we found using the django API much nicer for the testing of endpoints and therefore believe its the best way for making requests and viewing some features our models have such as the selection of a subscription type, instead of having to write out the exact value that is assigned for it, and such. Still, we understand the requirements of the assignments and have made sure everything works exactly the same in the django API as in postman.

Note #2: **In order to view the class schedule for a studio from its page** and **view the class history of a user,** please use POSTMAN with the given payload information. For enrolling the user into different classes and dropping them from different classes, please use the DJANGO API

Note #3: As per post @7 on Piazza, we have made the move to using environment variables in our postman requests (mainly for), and therefore, they are present in our postman.postman_collection submitted file (the postman export), so the variables that the TA will be required to set up (through a postman environment) are the following: auth_token (for authentication), user (user id), lat (location latitude), long (location longitude), studio (studio id), and studio_name (name of a studio).

Note #4: When running our postman.postman_collection file, make sure the environment variables are set up (as explained in Note #3), and have their values set (as needed per each request). When the postman.postman_collection is imported, the variables themselves should already be there (such as payload variables), and only their value will need to be registered for the requests to run. For example, we have a variable ‘auth_token’ which is used inside the Headers Auth section of a request to authorize the requests, and so setting up this variable once should work for all requests to be made. For other variables, you might need to change them in order to access different parts of the backend functionality (such as url variables).