The Big Result
System Design Document

Team members
Amy Yao
Ananya Poddar
Ava Oveisi
Carlos Fei Huang
Fariha Fyrooz
Noor Nasri
Vishal Sahoo

Table of Contents

CRC Cards

Model CRC Cards

Note: The following classes are simple data structures. SqlAlchemy maps the data in our database into the following data types, passed to our logic classes from the DAO classes.

Class Name: Status (Enum state)	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Stores a booking status as an enum with possible values: Booked, Canceled, In progress, Resolved, and Rescheduled.	**Collaborators:** *None*

Class Name: DayOfWeek (Enum state)	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Stores a booking's day of the week as an enum with possible values: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.	**Collaborators:** *None*

Class Name: IsAvailable (Enum state)	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Stores whether an availability slot is available or not, with values of true or false. This is useful for setting a full day as unavailable.	**Collaborators:** *None*

Class Name: Services	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Knows its name. Knows its description. Knows the associated professionals who offer this service.	**Collaborators:** *Professional*

Class Name: Bookings	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Knows its unique identifier. Knows the unique identifier of its associated customer. Knows the unique identifier of its associated professional. Knows the starting date and time of the booking. Knows the ending date and time of the booking. Knows the status of the booking. Knows the price of the booking. Knows the name of the associated service. Knows their associated review and customer.	**Collaborators:** *Status* *Reviews* *Customer*

Class Name: Reviews	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Knows its unique identifier. Knows the booking id of the associated booking. Knows the customer id for the associated customer. Knows the professional id for the associated professional. Knows its description. Knows its rating.	**Collaborators:** *Customer* *Professional*

Knows their associated customer and professional.	

Class Name: AvailabilitiesRec	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Know their unique identifier. Know the unique identifier of the professional associated with this availability. Know their day of the week. Knows their start time. Knows their end time.	**Collaborators:** *Professional*

Class Name: AvailabilitiesNonRec	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Know their unique identifier. Know the unique identifier of the professional associated with this availability. Know their associated date. Knows their start time. Knows their end time. Knows if they are available (in reference to IsAvailable enum, but stored as an int here).	**Collaborators:** *Professional*

Class Name: User	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Know their unique identifier. Know their first name. Know their last name. Know their email. Know their username. Know their password (hashed) Know their user type.	**Collaborators:** *None*

Class Name: Professional	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Know their unique identifier. Know their rating. Know their description. Know their average cost. Know their location. Knows the services they provide. Knows their reviews.	**Collaborators:** *Services* *Reviews*

Class Name: Customer	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Knows their unique identifier.	**Collaborators:** *None*

Logic CRC Cards

Class Name: gmailAPI	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Verify local token for Gmail API and refresh it when it expires. Retrieve Gmail API token when missing and save it locally. Send a provider an email for successful signup, given their name and email. Send a provider an email for successful approval, given the service provider's unique user id. Send a provider an email for successful booking, given the customer's unique id, provider's unique id, and relevant booking data.	**Collaborators:** *BookingsDAO* *UserDAO*

Send a provider an email for successful rescheduling, given the old booking's id, customer's unique id, provider's unique id, and relevant booking data. Send a provider an email for cancellation, given the old booking's id.	

Class Name: app	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Start the application. Initiate gmailAPI token verification. Register all blueprints to allow other classes to add routes for API calls. Initiate cookies, configurations, and authentication.	**Collaborators:** *gmailAPI* *signup* *listServices* *serviceProviderProfile* *allReviews* *login* *listServiceProviders* *listBookings* *calendar* *book*

Class Name: book	
Parent Class: None **Subclasses:** *None*	
Responsibilities: Add API routes for booking modifications. Create a new booking, given the customerId, service, professionalId, date, start, end, and any special instructions. Reschedules a booking, given the bookingId of the current booking and the information to create the new booking. Cancels a booking, given the bookingId. Resolves a booking, given the bookingId.	**Collaborators:** *gmailAPI* *Status* *BookingsDAO* *ProfessionalsDAO*

Class Name: calender
Parent Class: None

Subclasses: *None*	
Responsibilities: Add API routes for professionals' availability. Get a one week availability schedule for a professional, accounting for their current bookings, given the professionalId and start date. Get a professional's weekly availability, given the professionalId. Set a professional's weekly availability, given the professionalId and wanted schedule. Add a non recurring availability, given the professionalId, date, and wanted schedule.	**Collaborators:** DayOfWeek IsAvailable AvailabilitiesNonRec Status book AvailabilitiesRecDAO AvailabilitiesNonRecDAO

Class Name: listServiceProviders	
Parent Class: None **Subclasses:** *None*	
Responsibilities: Add API routes for service provider listings. Gets a price filtered list of professionals, given a minimum cost and maximum cost. Get a rating filtered list of professionals, given a rating requirement. Get a location filtered list of professionals, given a location. Get a multi filtered list of professionals, given a combination of cost, rating, and location requirements. Approve a service provider, given their unique id.	**Collaborators:** gmailAPI CustomersDAO ProfessionalsDAO ServicesDAO

Class Name: listServices	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Add an API route for getting a list of services. Get a list of services offered on the app.	**Collaborators:** ServicesDAO ProfessionalServicesDAO

Class Name: login	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Add API routes for login and token authentication. Create access and refresh tokens. Authenticate access and refresh tokens. Clear jwt cookies upon logout. Get the current user, given a token.	**Collaborators:** CustomersDAO ProfessionalsDAO

Class Name: signup	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Add an API route for signups. Create a new customer, given an email, password (hashed), first name, and last name. Create a new professional, given an email, password (hashed), first name, last name, location, description, services provided, and service descriptions.	**Collaborators:** *gmailAPI* *CustomersDAO* *ProfessionalsDAO* *ProfessionalServicesDAO*

Class Name: allReviews	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Add an API route for getting a professional's reviews. Get all reviews on a professional, given the professional's unique identifier.	**Collaborators:** *CustomersDAO* *ProfessionalsDAO* *ProfessionalServicesDAO*

Class Name: listBookings
Parent Class: *None* **Subclasses:** *None*

Responsibilities:	**Collaborators:**
Add API routes for getting lists of bookings. Get all upcoming bookings for a customer, given the customerId. Get all past bookings for a customer, given the customerId. Get all canceled bookings for a customer, given the customerId. Get all upcoming bookings for a professional, given the profId. Get all past bookings for a professional, given the profId. Get all canceled bookings for a professional, given the profId.	*BookingsDAO* *CustomersDAO* *ProfessionalsDAO*

Class Name: *serviceProviderProfile*	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities:	**Collaborators:**
Add API routes for getting and updating service provider information. Get a service provider's name, rating, description, services, profile picture link, location, calendar, reviews, service Descriptions, and hourly rates, given their unique identifier. Update a service provider's information, given their unique identifier and the information to set.	*ProfessionalsDAO* *CustomersDAO* *ProfessionalServicesDAO*

DAO CRC Cards

Note that we have multiple DAO classes rather than a single DAO interface, in order to better organize the tasks related to each class and their database interactions.

Class Name: UserDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities:	**Collaborators:**
Get all users in the database. Get a user, given their unique identifier	User

Class Name: CustomersDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get a customer, given their unique identifier. Get a customer, given their username. Get all customers in the database. Validate a customer login, given their username and password. Add a new customer, given their first name, last name, email, username, password. Verify if a username exists. Verify if an email exists.	**Collaborators:** Customer

Class Name: ProfessionalsDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get a professional, given their unique identifier. Get a professional, given their username. Get all professionals in the database. Validate a professional login, given their username and password. Add a new professional, given their first name, last name, email, username, password, description, rating, averageCost, and location. Verify if a username exists. Verify if an email exists. Get all services for a professional, given their unique identifier. Get all reviews for a professional, given their unique identifier. Get the lowest average cost amongst all professionals. Get the highest average cost amongst all professionals. Update a professional's description, given their unique identifier and wanted description.	**Collaborators:** Professional Services Reviews

Class Name: ServicesDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get all services. Verify if a service exists, given its name. Get a service, given its name. Add a service, given its name and description. Get all professionals offering a service, given the service name.	**Collaborators:** Services Professional

Class Name: ProfessionalServicesDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get price filtered professionals, given a service price range. Verify that a professional offers a service, given their unique identifier. Add a service to a professional, given the professiona's unique identifier, the service's name, wanted price and sent description. Remove a service from a professional, given their unique identifier and the name of the service.	**Collaborators:** Professional

Class Name: AvailabilitiesRecDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get all recurring availability for a professional, given their unique identifier. Get recurring availability for a specific day, given that day and the professional's unique identifier. Add a recurring availability, given the professional's unique identifier, day of the week, start, and end. Delete all recurring availability for a given day,	**Collaborators:** AvailabilitiesRec

given that day of the week and the professional's unique identifier. Delete a professional's weekly schedule, given their unique identifier.	

Class Name: AvailabilitiesNonRecDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get all non-recurring availability for a professional, given their unique identifier. Get non-recurring availability for a specific day, given that day and the professional's unique identifier. Add a non-recurring availability, given the professional's unique identifier, date, start, and end. Delete all non-recurring availabilities for a given day, given the date and the professional's unique identifier. Delete a professional's edited schedule, given their unique identifier.	**Collaborators:** AvailabilitiesNonRec

Class Name: BookingsDAO	
Parent Class: *None* **Subclasses:** *None*	
Responsibilities: Get all bookings for a professional/customer, given their unique identifier. Get all bookings with a specific status for a professional/customer, given their unique identifier and the status they want. Add a booking, given its customer id, professional id, start, end, location, status, price, service, and special instructions. Resolve a booking, given its unique identifier. Reschedule a booking, given its unique identifier.	**Collaborators:** Bookings

Software Architecture

This project uses the **Three-Tier Architecture** - https://www.ibm.com/cloud/learn/three-tier-architecture. The user accesses the presentation tier, which is made with React, by opening the web app on a browser. The app will make HTTP requests to a flask server, which is the Application tier. This handles all logic, and acts as an internal firewall before accessing the Data tier, which uses Azure SQL. We also used AWS S3 Bucket for storing uploaded images, which is in the Data tier and acts as a separate data source. Since billing is involved in our application, the security of that data is very important. The project uses this design due to the disconnect between the client tier and the data tier, as opposed to the MVC structure.

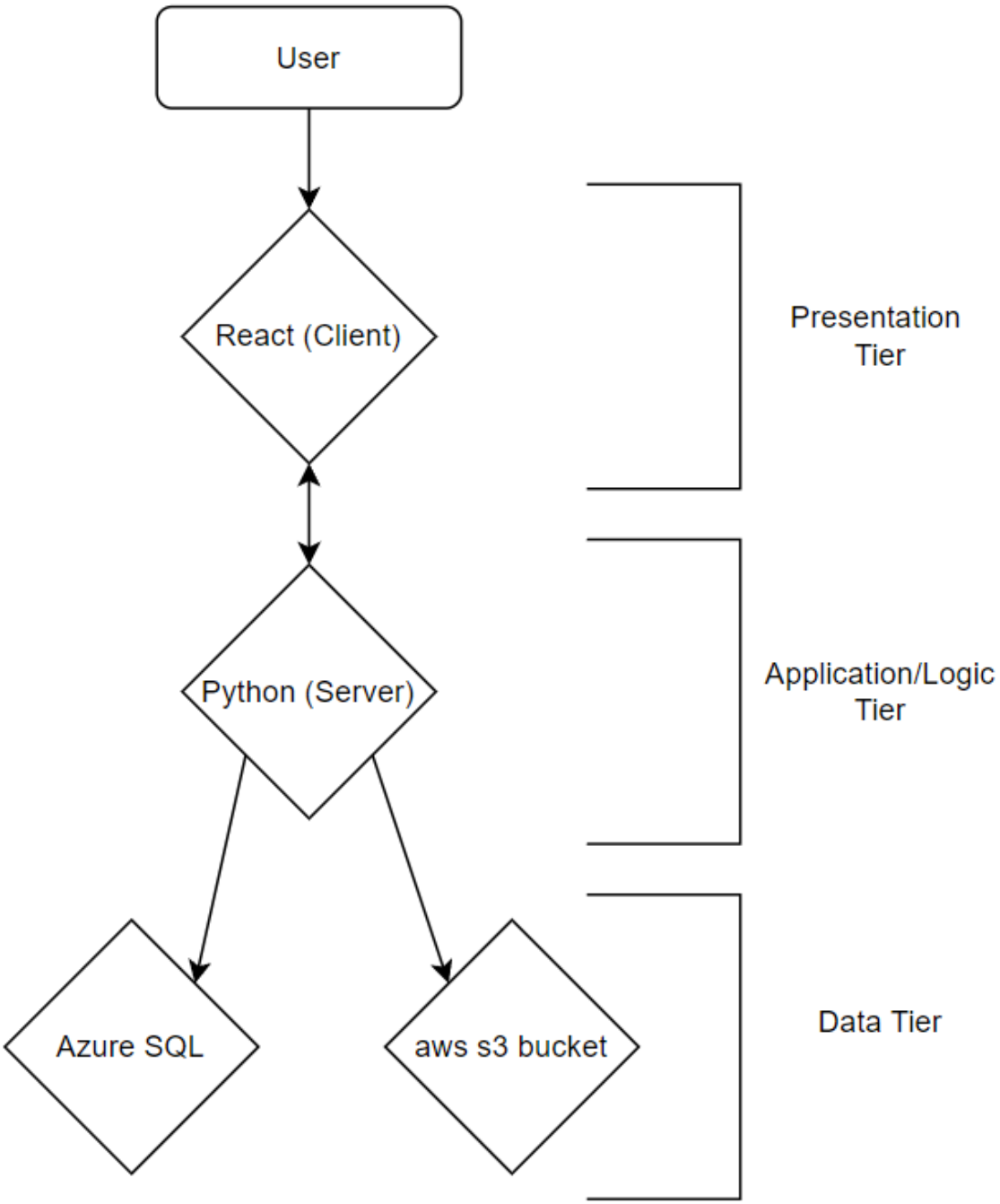
Extra Layer of Security
To better understand the purpose behind this architecture, it is relevant to understand the implemented authentication system. Our application uses industry standard jwt authentication, authentication requests with a short lived access token.

Upon signing up on the frontend, we use bcrypt to one-way salt and hash the password and make the account in the database. Upon login, we authenticate the password before granting two tokens: a refresh and an access token. The access token is a short lived (few minutes) token sent to the API calls to identify the user, stored in memory on the frontend to avoid outside parties. When the frontend needs to access a page after the access token is expired, it will have to re authenticate using the refresh token in order to get a new access token.

Through this system, we can verify that a user is asking for information relevant to them, enabling an internal firewall system. This is distinct from a high level MVC setup, which relies more on real-time updates from the database going directly to the model, which is displayed on the clients. This provides vulnerabilities that can be exploited. While it is possible to set up security measures in such systems, the Three Tier Architecture is much better defined for this purpose, allowing the security to be well compartmentalized as a middle man.

Better Compartmentalization of Tasks
An added benefit of this architecture is that it eases development for the team by decoupling tasks in separate tiers. For example, consider the image uploading feature added in sprint 4. The user uploads the image locally on their device, the presentation tier. It retrieves or updates the image by calling an endpoint, and that's the only communication required between the presentation and logic tiers, which allow it to be easily mocked. This allows one individual to work strictly in the presentation tier, focusing on the frontend aspects, and another individual to work on the logic tier, the flask backend. One could also be assigned to only setup the aws s3 bucket for the data tier, and it would still be easy to connect the three tiers at the end, as the method of communications are limited and easy to see. While it is possible to work separately in an MVC architecture, it is not as simple, as each level is connected to the other two levels.

Diagram



Three-Tier Architecture - https://www.ibm.com/cloud/learn/three-tier-architecture