# 

# 

# 

# 

# 

# 

# **Mentality**

***Therapy Booking Web Application***

**IS213 Enterprise Solution Development**

**G6-T8**

**Assignment**

**Nur Syafiqah Binte Mohamed Raihan**

**Rachel Toh**

**Sim Sheng Qin**

**Tan Zi Rong**

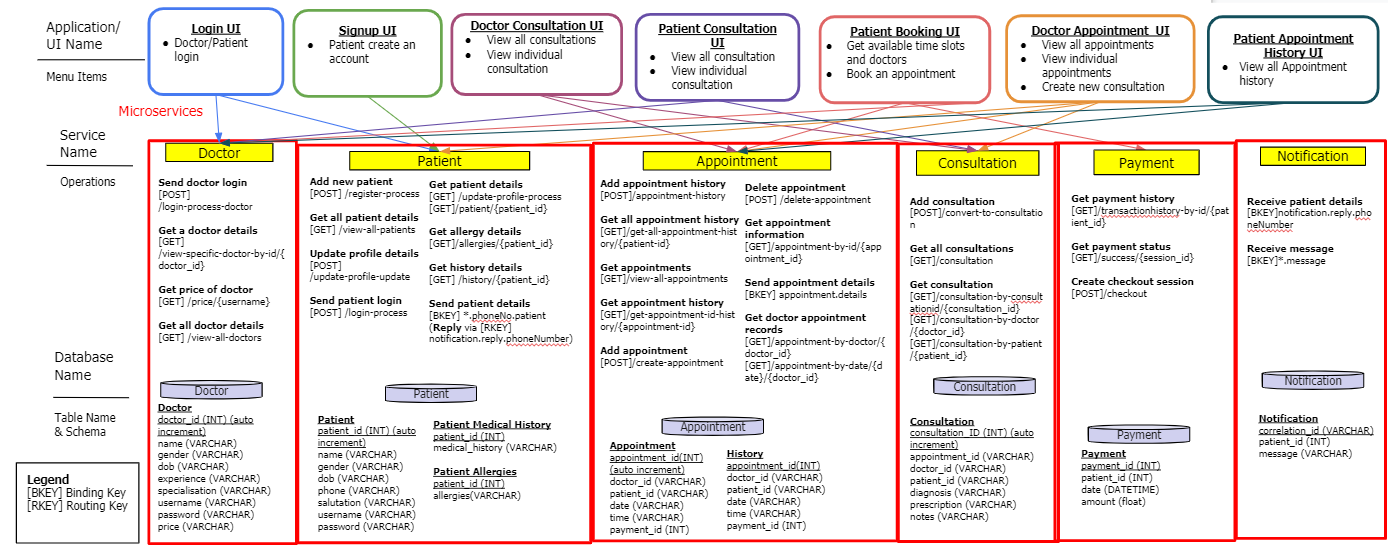
**Vittorio Carrecci Wei Long**

# Introduction

Mentality is a booking enterprise solution where patients can book consultations with the therapists at our clinic. It is an online website that easily and conveniently connects people with our therapists based on their needs, with our doctors’ details such as specialisations and fees explicitly stated on the website. The basis and assumptions of how our clinic runs are 1) we operate from 9am to 6pm, 2) each appointment/consultation is an hour, without breaks in between, and 3) all our doctors are pre-registered in our portal.

|  |  |
| --- | --- |
| **Business Objectives:** | |
| **Patients’ Perspective:** | * New patients can sign up for an account to access the portal * Patients can view all our clinics doctors and their details to choose from * Patients can view doctors’ availability and book accordingly * Patients can review and check their booked appointments and consultation history |
| **Doctors’ Perspective:** | * Doctors can check their appointments/consultations * Doctors can use the website for consultation purposes - to add notes about their patients based on each session * Doctors can review their patients details such as medical history and allergies for prescribing medication (out of the website) |

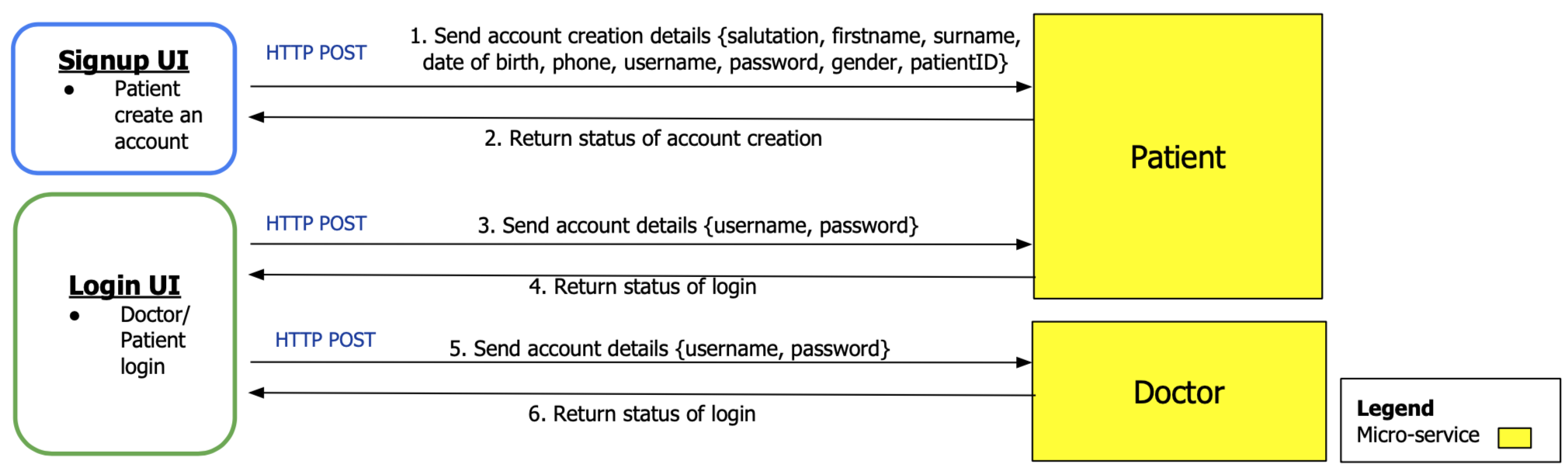
# Technical Overview Diagram



*Technical Overview Diagram*

User Scenarios

## User Scenario 1: *Patient registration and patient/ doctor login*

*  
User Scenario Diagram 1*

Patient creates a new account and Doctor/ Patient log into his account:

1. Patient creates an account via signup UI by filling his details in a form, then signup UI invokes Patient service via HTTP POST to register a new account.
2. Returns the account creation status to signup UI, which displays an alert of successful/failed account creation.
3. Patient logins through the Patient login UI which sends account details to Patient service via HTTP POST.
4. Returns status of login to the Patient login UI.
5. Doctor logins through the Doctor login UI which sends account details to Doctor service via HTTP POST.
6. Returns status of login to the Doctor login UI.

### (Micro)Services

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Service Name** | **Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any)** | **Description of the functionality** | **Input (if any)** | **Output (if any)** |
| **Patient** | Add new patient [POST] /register-process | [No additional description required for this operation as the Operation, Input & Output are clear enough for example] | {salutation, firstname, surname,  date of birth, phone, username,  password, gender} | {status of the account creation} |
| Send patient account details  [POST] /login-process | - | {username, password} | {status of login} |
| **Doctor** | Send doctor account details  [POST] /login-process-doctor | - | {username, password} | {status of login} |

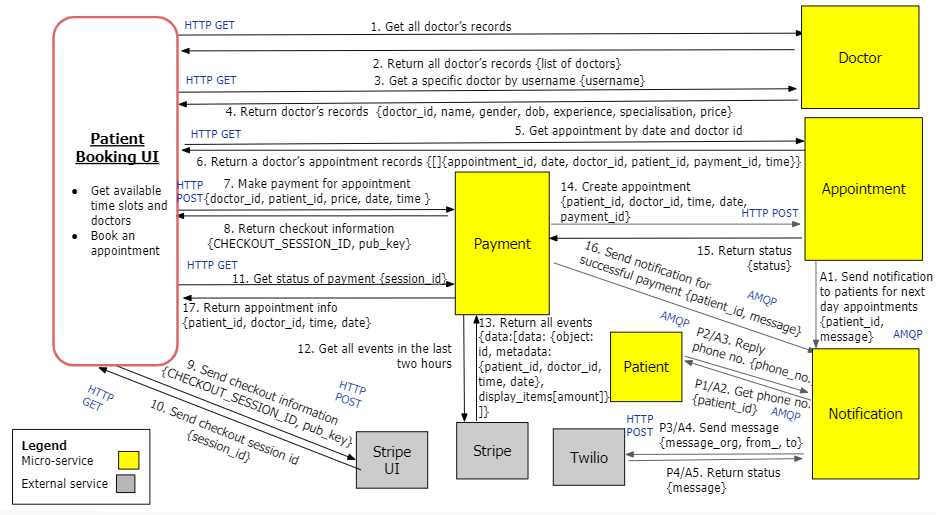
### 

### Beyond the Labs

The below business logic error-handling checks were implemented to prevent any accidental human errors:

1. Checks for duplicate usernames in Signup UI during patient registration. This was done by checking for any existing patient record in the patient database with the username the user wants to register an account with.
2. UI validation to check for any missing compulsory values. This was done by inserting “required” in each compulsory <input> field for patient registration. This will ensure that all compulsory fields will be completely filled before the patient is able to submit the form.

## User Scenario 2: *Booking an appointment*

**

*User Scenario Diagram 2*

Patient books an appointment:

1. Patient clicks the “Book” button on the navigation bar on the Patient Booking UI to invoke the Doctor service, via HTTP GET, to get a list of all doctors.
2. Return record of all doctors, and the Patient Booking UI displays all the doctors.
3. Patient selects a doctor via the “Book Now” button, triggering the Patient Booking UI to invoke Doctor service, via HTTP GET, to get a record of the selected doctor.
4. Return record of the specific doctor selected.
5. Patient clicks on a specific date on the Patient Booking UI to invoke the Appointment service, via HTTP GET, to get the available appointment time slots for the specific doctor on a selected date.
6. Return the specific doctor’s appointments on the selected date.
7. Patient selects a specific time slot on the Patient Booking UI, invoking the Payment service via HTTP POST, to make payment for the selected appointment time slot.
8. Returns the checkout information, which is appended to a hidden input in the Patient Booking UI.
9. The Patient Booking UI sends the checkout information to the Stripe UI.
10. The Stripe UI sends the session ID to the Patient Booking UI.
11. The Patient Booking UI invokes Payment to get payment status.
12. Payment retrieves all transaction events in the last two hours.
13. Stripe external API returns all transaction events in the last two hours.
14. Upon successful verification of payment, the Payment service invokes Appointment service to create a new appointment in the database via HTTP POST.
15. The Appointment service returns the appointment creation status to the Payment service.
16. The Payment service sends notification for successful payment to Notification service.
    * P1/A2. Notification service sends a message to Patient service to get the phone number of the patient.
    * P2/A3. Patient service receives the message and replies with the patient’s phone number from the Patient database.
    * P3/A4. Notification service invokes external Twilio API to send the reminder notification to the phone number.
    * P4/A5. Returns delivery status of reminder notification.
    * A1. Appointment service sends a message to Notification service containing the appointment details. Message is delayed to 6pm the day before the appointment. **Repeat A2 to A5.**
17. Payment service returns details of the booked appointment to Patient Booking UI which displays the information.

### (Micro)Services

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Service Name** | **Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any)** | **Description of the functionality** | **Input (if any)** | **Output (if any)** |
| **Doctor** | [GET] /view-all-doctors | Get a list of doctors | - | {list of doctors} where a doctor is {doctor\_id, name, gender, dob, experience, specialisation, price} |
| [GET] /view-specific-doctor/{username} | Get a specific doctor | {username} | {doctor\_id, name, gender, dob, experience, specialisation, price} |
| **Appointment** | [GET] /appointment-by-date/{date}/{doctor\_id} | Get available time slots on a particular date for a specific doctor | {date}  {doctor\_id} | {list of appointments} where an appointment is {appointment\_id, patient\_id, doctor\_id, date, time, payment\_id} |
| [POST] /create-appointment | Create new appointment | {patient\_id, doctor\_id, time, date, payment\_id} | {status of the appointment creation} |
| Topic Exchange with  [RKEY] appointment.message | Send appointment details notification using [RKEY] appointment.message  \*message is delayed to 6pm the day before the appointment | {patient\_id, message} |  |
| Topic Exchange with  [RKEY] paymentSuccess.message | Send the notification on successful payment using [RKEY] paymentSuccess.message | {patient\_id, message} |  |
| **Payment** | [POST] /checkout | Gets appointment details, creates a checkout session and returns the checkout session id | {patient-id, doctor\_id, time, date, price} | {CHECKOUT\_SESSION\_ID, pub\_key} |
| [GET] /success/{session\_id} | Checks the session\_id of transaction and return status of payment. If successful, return appointment details. | {session\_id} | {patient\_id, doctor\_id, time, date} |
| **Notification** | Topic Exchange with  [BKEY] notification.reply.phoneNumber  [RKEY] notification.phoneNo.patient | Send a request using [RKEY] notification.phoneNo.patient to get patient phone number where a reply will be sent back via [BKEY] notification.reply.phoneNumber | {patient\_id} | {phone} of patient with the specified patient\_id |
| Topic Exchange with  [BKEY] \*.message | Receive message from appointment and notification microservice using [BKEY] \*.message | {patient\_id, message} |  |
| **Patient** | Topic Exchange with  [BKEY] \*.phoneNo.patient  [RKEY] notification.reply.phoneNumber | Retrieve patient id via [BKEY] \*.phoneNo.patient and reply phone number back via [RKEY] notification.reply.phoneNumber | {patient\_id} | {phone} of patient with the specified patient\_id |
| **Twilio API** | <https://www.twilio.com/docs/usage/api>  [POST] /{date}/Accounts/{AccountSid} | Send the notification message to the patient's phone. | {body, from\_, to} | {From, MessageServiceSid}  From refers to the queued status or error |
| **Stripe API** | <https://stripe.com/docs/api/events/list>  [GET] /v1/events | Make payment for the appointment booked. | {type, created} | {amount  {data:[data:  {object: id,  metadata: {patient\_id, doctor\_id, time, date}, display\_items  [amount]}]} |

### 

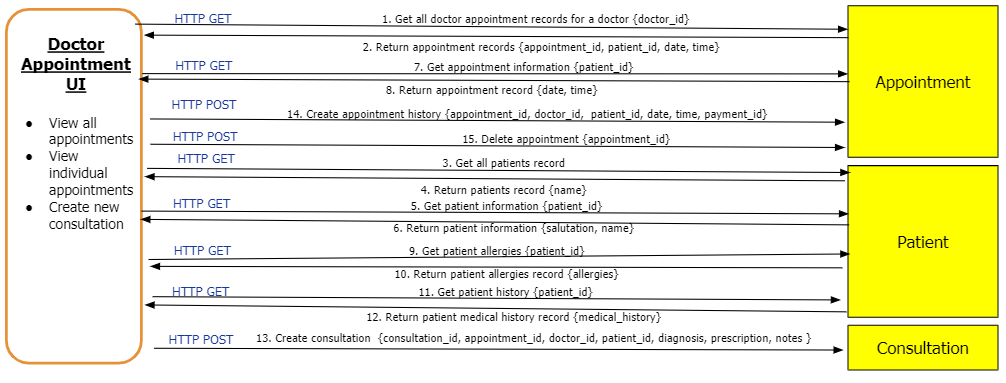
### Beyond the Labs

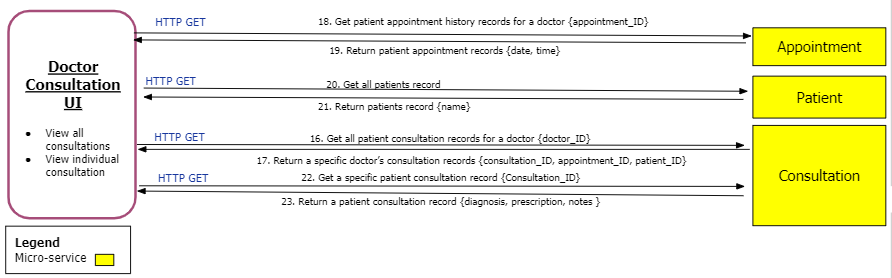
1. Stripe: External API Stripe is used to provide a secure payment gateway for our patients to pay for their appointment bookings

Stripe API is invoked by the Payment service to create a checkout instance. Users will then be redirected to Stripe’s UI page which will display the payment page. Payment service will then confirm the payment status by invoking Stripe API for the list of events.

1. Twilio: External API Twilio is used to provide an automated notification service for our patients to be notified of their successful payments via a text to their mobile phones. Twilio API is invoked by Notification service, which in turn creates a new instance of the Message resource and sends a HTTP POST request to the Message resource URI. This will send the message to the patient’s phone number retrieved from the Patient service.
2. Business logic error-handling: The min attribute, which specifies the minimum value (date) for the date field, is set to today’s date. This will disable past dates for appointment bookings, thus preventing invalid appointment bookings accidentally.
3. Business logic error-handling: If all appointment time slots for a specific doctor are fully booked, the UI will display an error message “No more available time slots for this date for this therapist.”. This was done by checking if the timeslot table is empty.
4. Business logic error-handling: If time slots have been booked by other patients, they will not be displayed in the timeslots table available for appointment booking. This is done by having a if-statement which checks whether a time slot is already added to the database.

## User Scenario 3: *Doctor in a consultation session (Doctor)*





*User Scenario Diagram 3*

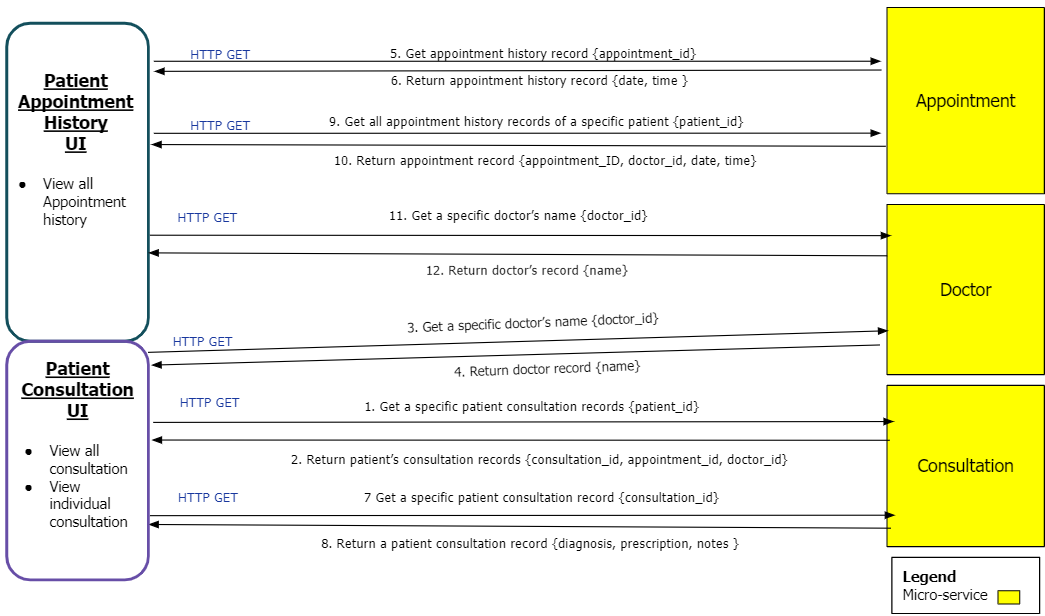
Doctor creates a consultation session:

1. The doctor clicks the “Appointments” button on the navigation bar on the Doctor Appointment UI to invoke the Appointment service, via HTTP GET, to get all his/her appointment records.
2. Return list of appointment records.
3. For each element in the appointment list record, it will invoke the Patient service, via HTTP GET, to get an individual patient record.
4. Return individual patient records to get their names.
5. The doctor selects the individual patient by clicking the link “view patient” as the appointment is about to begin, to invoke the patient service,via HTTP GET, to get the individual patient record.
6. Return patient information.
7. Doctor appointment UI automatically retrieves appointment information via Appointment service, via HTTP GET, to get appointment records.
8. Return appointment record to get the date and time of the appointment.
9. Doctor appointment UI also automatically retrieves patient allergies from Patient service, via HTTP GET, to get patient allergies records.
10. Return patient allergies.
11. Doctor appointment UI also automatically retrieves patient medical history from Patient service, via HTTP GET, to get patient medical history records.
12. Return patient history.
13. The appointment session ends by clicking the “Submit” button and the Doctor Consultation UI wIll invoke the Consultation service, via HTTP POST, to create consultation records.
14. The UI will automatically create an appointment history record by invoking the Appointment service, via HTTP POST.
15. The UI will also automatically delete the appointment record by invoking the Appointment service, via HTTP POST.
16. The doctor would be redirected to consultation UI, allowing her to view all consultation. The Doctor Consultation UI invokes the consultation via HTTP GET, to get all his/her consultation records.
17. Return list of doctor consultation records.
18. For each consultation in the doctor’s consultations list, the UI will invoke the appointment service, via HTTP GET, to get the date and time for that consultation.
19. Return individual patient appointment history record.
20. For each consultation in the doctor’s consultation list, the UI will invoke the Patient service, via HTTP GET, to get an individual patient record, or more specifically, his/her name.
21. Return individual patient records.
22. As the doctor wants to view the newly created consultation done, The doctor selects the individual patient by clicking the link “View Consultation” to invoke the Consultation service,via HTTP GET, to get individual consultation records.
23. Return a patient consultation record.

### (Micro)Services

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Service Name** | **Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any)** | **Description of the functionality** | **Input (if any)** | **Output (To UI)** |
| **Appointment** | [GET] /appointment-by-doctor/{doctor\_id} | Get a list of appointment based on doctor id | {doctor\_id} | {list of appointments} |
| [GET] /appointment-by-id/{appointment\_id} | Get current appointment based on appointment id | {appointment\_id} | {date , time} |
| [POST] /appointment-history | Create a appointment history based upon appointment data | {Appointment\_id, doctor\_id,  patient\_id, date, time, payment\_id} |  |
| [POST] /delete-appointment | Delete current appointment based on appointment id | {appointment\_id} |  |
| **Patient** | [GET] /patient/{patient\_id} | Get a patient’s name and salutation based on patient id | {patient\_id} | {salutation, name} |
| [GET] /allergies/{patient\_id} | Get a patient allergies based on patient id | {patient\_id} | {allergies} |
| [GET] /history/{patient\_id} | Get a patient medical history based on patient id | {patient\_id} | {medical\_history} |
| [GET] /view-all-patients | Get all patients |  | {name} |
| **Consultation** | [POST] /convert-to-consultation | Create consultation based on consultation information | {consultation\_id, appointment\_id, doctor\_id, patient\_id, diagnosis, prescription, notes } |  |
| [GET] /consultation-by-doctor/{doctor\_id} | Get all consultation via doctor id | {doctor\_id} | {consultation\_id, appointment\_id, patient\_id} |
| [GET]/consultation-by-consultationid/{consultation\_id} | Get current consultation via consultation id | {consultation\_id} | {diagnosis, prescription, notes } |

## User Scenario 4: *Patient in a consultation session (Patient)*

**

*User Scenario Diagram 4*

Patient views a specific consultation and appointment history:

1. After consulting with the doctor at the scheduled appointment, the patient can view the consultation or previous consultation whenever the patient wants. The patient uses the Patient Consultation UI to invoke the Consultation service, via HTTP GET, to get the consultation\_id, appointment\_id and doctor\_id of all his/her consultation records.
2. Return list of consultation records.
3. For each consultation in this patient’s consultation list, Patient Consultation UI will invoke the Doctor service, via HTTP GET, to get the doctor name based on the doctor id, given that patients are allowed to have different doctors.
4. Return individual doctor record.
5. For each consultation in this patient’s consultation list, Patient Consultation UI will also invoke the Appointment service, via HTTP GET, to get the date and time of an appointment based on the appointment id.
6. Return individual appointment history record.
7. The patient decides to view the current consultation record and by clicking “View Consultation” to invoke the Consultation service, via HTTP GET, to get the individual consultation record.
8. Return the consultation record, which the Doctor Consultation UI displays.
9. Assuming that the patient needs to view past appointment records, perhaps to check a discrepancy on an appointment that was never conducted, the patient can view all appointment history records, using the Patient Appointment History UI, which invokes the Appointment service, via HTTP GET, to get all his/her appointment history records.
10. Return list of appointment history records.
11. For each appointment in the patient’s appointment history list, Patient Appointment History UI will invoke the Doctor service, via HTTP GET, to get the doctor name based on doctor\_id.
12. Return the doctor record.

### (Micro)Services

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Service Name** | **Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any)** | **Description of the functionality** | **Input (if any)** | **Output (To UI)** |
| **Appointment** | [GET] /get-appointment-id-history/{appointment\_id} | Get a specific appointment history record’s date and time | {appointment\_id} | {date, time} |
| [GET] /get-all-appointment-history/{patient\_id} | Get all appointment history records of a specific patient | {appointment\_id} | {appointment\_id, doctor\_id, date, time} |
| **Doctor** | [GET] /view-specific-doctor-by-id/{doctor\_id} | Get a specific doctor’s name | {doctor\_id} | {name} |
| **Consultation** | [GET] /consultation-by-patient/  {patient\_id} | Get all consultations of a specific patient | {patient\_id} | {consultation\_id, appointment\_id, doctor\_id} |
| [GET] /consultation-by-consultationid/{consultation\_id} | Get the diagnosis, prescription and notes of a specific consultation | {consultation\_id} | {diagnosis, prescription, notes} |

# Remaining Beyond the Labs not covered above

With the usage of cloud, we are taking advantage of its efficiency and robustness, and its abilities to maintain and scale. These are brief descriptions of our cloud implementation. The in-depth technical configurations can be found in the README file.

1. Hosted MySQL database on cloud (Amazon Relational Database Services - AWS RDS).
2. Implemented rabbitMQ on cloud using cloudAMQP.
3. Deploy docker containers for our microservices on AWS cloud running constantly (24/7). They interact with each other through Cloud RabbitMQ and retrieve data from AWS RDS. Using docker containers will make our solution more decoupled.