Patient Scheduling Subsystem for a Small Hospital: Analysis and Design of an Object-Oriented Platform

Blake Costello
Computer Science Department
California State Polytechnic University, Pomona
bpcostello@cpp.edu

Joshua Goodman
Computer Science Department
California State Polytechnic University, Pomona
jjgoodman@cpp.edu

Keita Katsumi
Computer Science Department
California State Polytechnic University, Pomona
kkatsumi@cpp.edu

Dean Mah
Computer Science Department
California State Polytechnic University, Pomona
drmah@cpp.edu

Allan Manangan
Computer Science Department and Geological Sciences Department
California State Polytechnic University, Pomona
amanangan@cpp.edu

Gian David Marquez
Computer Science Department
California State Polytechnic University, Pomona
gdmarquez@cpp.edu

***Abstract* — Appointment Booking software is the modern online connectivity between doctors and patients. The Appointment booking software allows users to select their doctor to create an in-person appointment and view their appointments in their dashboard. The software allows for multiple accounts to be used at once and is not limited to one account on the server. For the purpose of this project, we limited our software to follow the proposed outline and maintain efficiency. We did work on adding some small detailed features like an info board showing the logged-in users' information and a front home page. The program allows for both doctors and patients to log in from the same portal and will only display the dashboard for the user type upon a returned authentication success. The dashboard, although displays nothing without a secured JWT token, displays and renders no data without the JWT token matching the browser and server, before loading data from the API. Our program extends beyond the requirements and features an additional API that houses our SQL commands that are passed to our database. Because of this, our express backend only has 1 get and post request, that checks for JWT, and passes all the requests to the API. This secures the database from unnecessary direct traffic from the front end and limits how the database can be accessed from the outside.**

Keywords* — *database, hospital, javascript, MySQL, object-oriented design, object-oriented programming, scheduling subsystem, schema, portal.

Introduction

The authors describe a patient scheduling subsystem for a hypothetical rural town. The development of this subsystem involves both systems analysis and design. We present (1) a problem to solve, (2) a solution that is the patient scheduling subsystem, (3) an analysis of what is required for the subsystem to solve the problem, and (4) a design for how the subsystem will operate to solve the problem.

To ground a discussion of the subsystem, we first identify the problem. The narrative is as follows: a rural town is building a new hospital that will be their only facility to provide care to residents. The hospital will have several care units where doctors meet with and diagnose their patients. A single patient may have multiple appointments with different doctors and a single doctor may have multiple appointments in their schedule. The rural town planners are forward-thinking and find that having a paper-based scheduling system is inefficient and inconvenient to both doctors and patients. The rural town planners want a modern system that empowers patients, specifically one where they can schedule appointments with doctors for care. The ideal solution would be an accessible platform where patients can book, reschedule, or cancel their appointments.

Analysis

In order to solve the problem, the authors discover and understand the rural town's needs. By careful comprehension of the problem statement, we find three needs. First, the hospital is the only one for the entire population, all of whom could presumably be patients, the solution should be a centralized platform that all the town's residents can access. Second, the hospital is modern and will not use a paper-based system. Third, the solution should be comprehensive and allow patients to manage three aspects of appointments: creating them, updating them, and removing them from the schedule.

The ideal solution would be convenient, modern, and unified for patients. An electronic patient scheduling subsystem allows for ease of information input, storage, and retrieval of not only their appointment information but also the information related to the hospital's pre-existing roster of doctors. This solution lets patients search for a doctor who meets their care criteria and then make an appointment without having to go through an intermediary, like an in-person receptionist. This expedites the appointment process, improves the hospital's responsiveness to patients by streamlining care, and reduces any possible miscommunication about when and where their appointments are by providing a consistent platform.

The stakeholders within the subsystem involve the hospital's faculty and those being cared for, the patients. The doctors are under the Internal Operational stakeholders due to their consistent use of the system to look at their schedules. The patients are external operational stakeholders as they receive most of the system's benefits.

The authors' solution is a web-based portal. This is a modern, unified, and widely accessible location where

patients manage their own care. They can sign in, view their current appointments, search through available doctors, choose a doctor that suits their needs, schedule an appointment, update their appointments, and cancel an existing appointment.

The primary actors in such a patient scheduling subsystem are doctors and patients, who are people. Part of the analysis is understanding the relationships among them. We present a domain class diagram in Fig. 1 below.

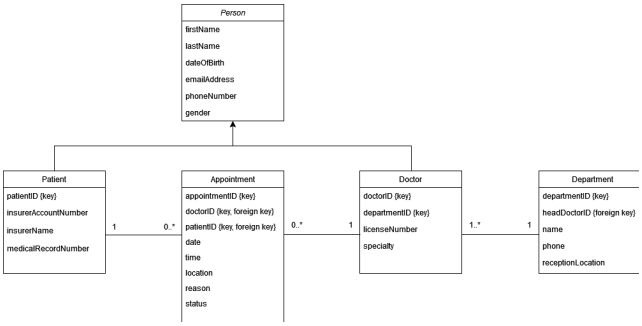

Figure 1. The domain class diagram for a patient scheduling subsystem.

In the domain class diagram, we can lay out the actors in respective classes. The class Person is the superclass with the Doctor and Patient classes being subclasses of Person. This means that both the Doctor and Patient classes will inherit the properties from the Person class along with their individually unique properties respectively. This is because the Person class contains general information that is relevant to every person like their first and last names, date of birth, email address, phone number, and gender.

The appointment class is the most important class as it is the center of the system. This class handles the important information of an appointment such as the date, time, location, reason, and status. The appointment must also be assigned a single patient and doctor to connect to.

Use Cases	Description of Use Case
Search for Doctor	A patient searches for a doctor.
Display Doctors	The system displays a list of available doctors.
View Doctor Information	An patient views a doctor's information.
Select a Doctor	An actor selects a doctor from the doctor search results.
Filter Doctor Availability	The system provides a list of dates and times when the chosen doctor is available.
Display Doctor Availability	The system displays a list of dates and times for an appointment.
Create Appointment	A patient chooses an appointment date and time.
Cancel Appointment	A actor requests that an appointment be canceled.
Reschedule Appointment	A actor requests that an appointment be rescheduled.

Table 1. A set of use cases for the patient scheduling subsystem.

Patients and doctors are the primary users that will be externally interacting with the Patient Scheduling Subsystem. After a user logs into the portal, they can do several things presented in Table 1. Depending on the user's actions, the system will respond. Some of the use cases above require each other. For example, in order to arrive at "Create Appointment" the patient must first "Search for Doctor" and then "Select a Doctor" through the system. The Create Appointment use case will be discussed later in more detail. In regards to the Appointments, the patient can cancel or reschedule appointments. When the patient decides to "Cancel Appointment," the system retrieves that appointment using the unique Appointment ID and cancels that appointment. Along similar lines, when the patient wants to "Reschedule Appointment" the system retrieves that appointment using the unique Appointment ID, filters for a doctor's availability, and displays that new availability to the patient or secretary. Then the patient or secretary can choose a different appointment date and time.

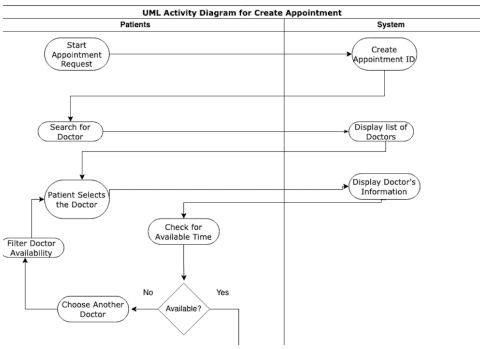

Figure 2. Activity Diagram, Part I for Create Appointment

Create Appointment follows a flow of use cases from Table 1, shown in Figure 2 and Figure 3. Starting with "Search for Doctor," which is when a patient searches for a doctor by using at least one user-inputted search parameter. the system responds with "Display Doctors." According to the given search parameters, the system displays a list of doctors that match the search parameter(s). After seeing the list of doctors a patient "Selects a Doctor" from the list and the system retrieves that specific doctor's information, leading to the "View Doctor Information" use case. The "View Doctor Information" displays all relevant information about the doctor for the patient to see. If the patient chooses that the doctor is appropriate for them, "Filter Doctor Availability" the system verifies that doctor's availability and processes that doctor's schedule for open dates and times. This ensures that this specific doctor will not have overlapping appointments. If there are no available times, the patient is sent back to "Select A Doctor." If there are available times for doctors, we can continue.

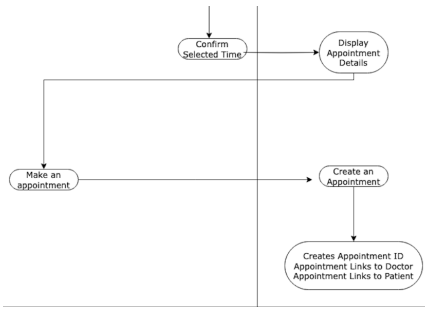

Figure 3. Activity Diagram, Part II for Create Appointment

Figure 3 details what the flow is if there are available times. When there are valid filtered appointment times, the system displays those available dates and times using "Display Doctor Availability." The patient chooses to confirm their appointment time with their selected and is shown their appointment details card. After the patient decides to confirm their appointment details, that is when the "Create Appointment" use case is finally invoked. "Create Appointment" is when the system creates a new appointment with a unique Appointment ID. This appointment is a relationship between the doctor and the patient.

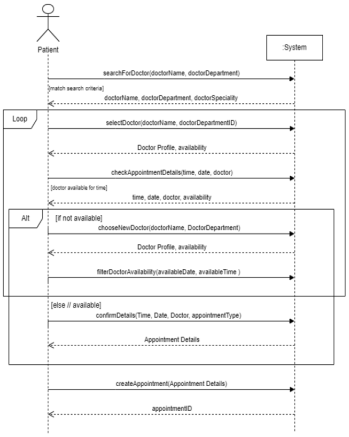

Figure 4. System Sequence Diagram for Create Appointment

In the diagram shown in Figure 4, the patient can search for a doctor and schedule an appointment. The patient can search for a doctor and the system will return the doctor's information. Then the patient can select a doctor and it allows the patient to look at appointment details to see if the doctor is available. Then if the appointment time slot is unavailable then the patient must choose a new doctor within the available times. If the patient chooses an appointment that is available then the system will return the appointment details before the patient finally creates the appointment.

DESIGN

Now that we have completed a foundational analysis of the problem and our patient scheduling subsystem solution, we refine its development. We discuss how the subsystem will operate and allow patients to search for a doctor and then schedule an appointment.

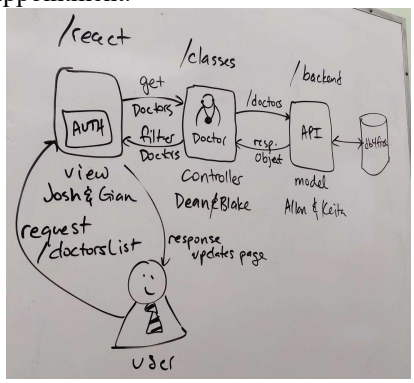

Fig. 5. The original Domain Class Diagram.

The System architecture has been separated into 4 different main sections: React for the front end, Classes and Controller, API, and Server. React acts as our front end ui layer, the controller layer acts as an intermediary for the API which is the intermediary for the server commands. Having the API calls located in the controller or classes section brings major benefits towards decentralization and creating structure. The handlers in the controller layer work to decentralize the calls to the API while the API acts as a protective layer for the server and database.

DATABASE

A database is a centrally managed collection of data that is accessible to many systems and users at the same time. And in the model-view-controller architecture, the model layer represents the data and logic of an application. The model of the patient scheduling subsystem is responsible for performing create, report, update, and delete or archive operations on its data. It does this by interactions with the database.

Since the patient scheduling subsystem is part of a hospital, one of the design considerations included a separation between the application and the database service. Our relational database is MySQL and is hosted by DB4Free. A positive outcome from using this external service is that it allowed team members to access the same data during design and development. However, there were also negative issues with this database service. First, it took a long time to access and fetch data. This slowed the response time of our subsystem. Second, it has restrictions on the number of concurrent database connections within a certain period. This caused several of the model layer's interactions with the database service to fail, which cascaded errors up to the controller layer.

From the analysis phase, we produce a domain class diagram as in Fig. 1, but this does not provide enough detail. The design phase refines the domain class diagram into a database schema by several steps. First, each class becomes a database table. Second the type of each attribute is specified, including any length value. Third, we note which attribute or set of attributes constitute a database table's unique key and whether or not a key attribute is a foreign key that is the primary key of a different table. Fourth we note the relationships between each table using connective lines. The database schema is summarized in Fig. 6 below.

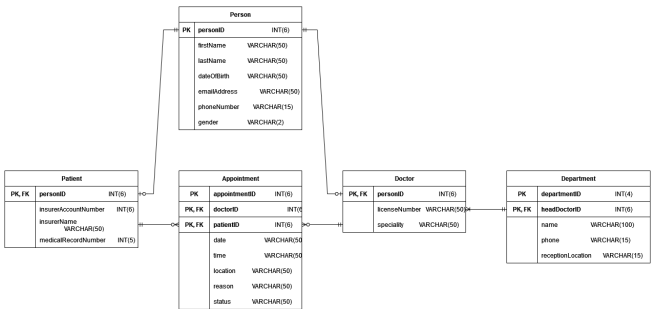

Figure 6. The database schema for the patient scheduling subsystem.

There are several items to note in this database schema that contains five tables: Person, Doctor, Patient, Appointment, and Department. Specifically, we must notice:

- Person is a superclass for Doctor and Patient. To realize this relationship in the database, the Person

primary key personID must either exist in Doctor or Patient with the same personID. Thus personID is a foreign key to Doctor and Patient.

- When the new Patient or Doctor record is created, we must first fetch the corresponding personID. So the Person record precedes the Patient or Doctor record. The model must get the personID and create a new Patient or Doctor with the same personID to keep the inheritance relationship from the domain class diagram.
- The cascaded personID value in Patient and Doctor will be foreign keys in Appointment, so they are renamed to patientID and doctorID respectively.

What is not shown in the database schema above are views. We combine Person data and either Patient or Doctor data through an inner join to produce a Patient view or Doctor view respectively. When the model requests either patient or doctor information the view is returned. The same idea is applied to create an Appointment view. The Person and Patient table records produce a Patient view. The Person and Doctor table records produce a Doctor view. Then a consolidation of the Patient view, Doctor view, and Appointment table record to generate the Appointment view.

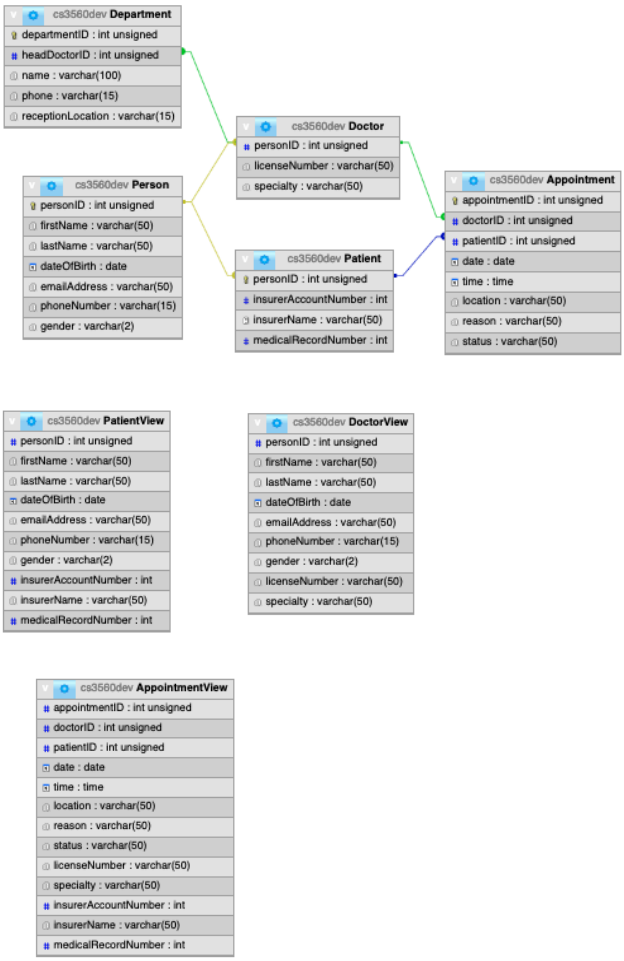

Figure 7. The database schema for the patient scheduling subsystem is directly from the MySQL administrative interface. This shows the views as described for patients, doctors and appointments.

The model of the patient scheduling subsystem interacts with the external database and provides its operations through an Application Programming Interface (API) that effectively encapsulates the data. As such a RESTful API provides reading data through GET requests, inserting data through POST requests, updating data through PUT requests, and deleting or archiving data through DELETE requests. In the following section, we detail the controller layer that places these requests to the model.

CONTROLLER

When formulating the overall structure of the system it was decided to include an intermediary layer between the front end and the back end. This layer was named the Controller and it is a separate layer from the API. The Controller was designed to call the API and format the returned data to be easier to integrate into the front end. When designing the Controller the ease of use and security of it was taken into account resulting in a controller file and handler files. The handler files contained the calls to the API and worked to decentralize access to the API while the controller worked to connect to the handlers and centralize the calls to a single importable file. This was done in an attempt to decentralize the functional code while allowing the front end to call a single centralized file that would contain access to the handler files.

FRONT END

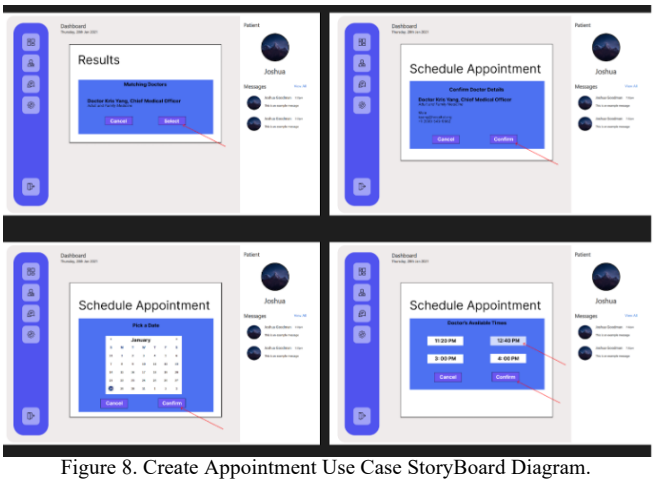

Figure 8. Create Appointment Use Case StoryBoard Diagram.

Although Figure 8 doesn't show the final graphic representation, it does show how the database and front end interact. The interaction point of the user picking an appointment time shows how the database would store the appointment data. It was useful to have the database schema in place because the attribute types informed the user interface design.

We present designs for the user interface that doctors and patients will manipulate. First, we have the login page in Figure 9 below.

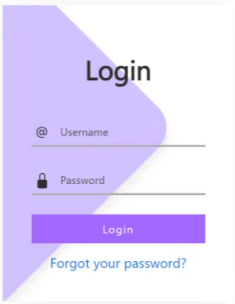

Figure 9. The login user interface for patients and doctors.

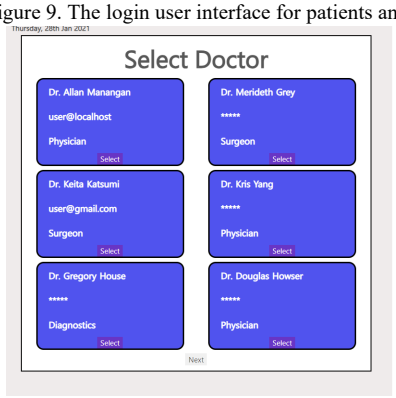

Figure 10. The user interface where patients may select a doctor.

The frontend system was designed using a helper tool called Codux. This allowed other team members who lacked the experience and knowledge about react front ends, to participate in the design phase. The functions that load data from the front-end design are structured inside the classes section allowing for the API calls to sit here, rather than the front end. This allows us to re-use front-end components and keep structure when sending and receiving API calls which also allows us to load the data before attempting to render the widget when blank.

Suggestions for Future Improvement

There is room for improvement for this patient scheduling subsystem. We present criticisms and possible upgrades for each of the application layers.

1. Model and Database: One update to be made to the application programming interface that is presented for the controller. It lacks the flexibility to specify multiple data attributes and be able to specify them as a structured set of parameters in the API request. A second update to the model layer that would improve efficiency and reduce the number of server errors is to add database connection pooling. Since the authors are not in control of an external service, there is no way to increase the maximum number of database connections unilaterally. The solution is to create a pool of database connections that are maintained by the model, distributed when needed, and then kept alive so that connections are not wasted. This will keep data throughput high and provide a faster experience in the user interface because data storage and retrieval is the slowest operation, especially in a web-based environment.

2. Controller: The controller layer was not tested for compatibility with the front end due to its lack of development and confusion during development. It would be beneficial to research compatibility before implementation.

3. Front End: Better usage of CSS styling and integrations. Better naming of components to ensure that we don't get lost during development.

4. Development Cycle: The subsystem described in this paper did not have multiple development cycles. Although the authors were able to produce a viable proof of concept, further development would need to first revisit our development process. We may need to do further testing to reveal bugs and address the stability of the system prior to adding new features.

Conclusion

The analysis and design of a patient scheduling subsystem is a complex task. It requires first understanding the needs of the client and other stakeholders before being able to draft an initial set of diagrams that lightly describe the subsystem. Once the stakeholders are on board, it is possible to go further and refine the crude diagrams into ones that convey how the subsystem will operate to bring about what the client needs.

Ultimately, it will take several rounds of software development lifecycles to produce more than just a proof of concept. However, considering the amount of time and resources the authors had at their disposal, it was inevitable that analysis, design and development reached the levels they did.

REFERENCES

[1] Blaha and J. Rumbaugh, *Object-Oriented Modeling and Design with UML*, 2nd ed., New Jersey, United States: Prentice Hall, 2005.

[2] Administrators of DB4Free. “Free MySQL Database for Testing” *DB4Free*, 2023. https://db4free.net/