UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS301 - Software Engineering

Office Booker

Team name: Kryptos Kode

Name	Student Number	Email
Ying Hao Li*	u20460687	u20460687@tuks.co.za
Arul Agrawal	u18053239	u18053239@tuks.co.za
Grant Bursnall	u15223893	u15223893@tuks.co.za
Damian Vermeulen	u20538945	u20538945@tuks.co.za
Brett du Plessis	u19037717	u19037717@tuks.co.za

* - Team Leader

Contents

1 Architectural Design Strategy **3**
1.1 Decomposition . . . 3

2 Architectural Styles **3**
2.1 Model View Controller . . . 3
2.2 Layered . . . 3

3 Architectural Quality Requirements **4**
3.1 Availability . . . 4
3.2 Responsiveness . . . 4
3.3 Learn-ability . . . 4
3.4 Manageability . . . 4
3.5 Simplicity . . . 4
3.6 Usability . . . 4

4 Architectural Design & Pattern **5**
4.1 Overview . . . 5
4.2 Diagram . . . 5

5 Architectural Constraints **6**
5.1 Free Software . . . 6
5.2 Frontend . . . 6
5.3 Database . . . 6
5.4 Cooperation . . . 6

6 Technology Choices **6**
6.1 Angular . . . 6
6.2 NestJS . . . 6
6.3 PostgresSql . . . 7
6.4 AWS Cognito . . . 7

1 Architectural Design Strategy

1.1 Decomposition

The design strategy that our team decided to use for development was the decomposition strategy as it seemed the most effective way of producing results for the features we had to implement. The decomposition strategy works as solution to the big features we have by taking them and splitting them in to smaller tasks that can be done step-by-step instead of having over complicated implementations of the features. This works well for a specified project as the Office Booker has many parts that work together and such they would need to be modular for a better method of using the systems together. All the features being broken up in to smaller systems allows for the application to be more modular as the smaller systems can be used individually or with other systems without must hassle instead of if we had to work with a large implementation of features forcing them to all be used instead of just the required sections needed.

2 Architectural Styles

2.1 Model View Controller

One of the design Patterns the Office Booker system uses is a model view controller pattern. This system makes sense for the type of work being done in the system. The system is divided into sections of the office the users sees, the database that handles the offices data and the logic behind the office system that creates the office view and allows users to make bookings with visual responses. This design model works well with the system that is to be implemented as it allows for each system to be developed independently and once developed work together to produce the Office Booker experience that users are trying to reach.

2.2 Layered

The Office Booker project employs the use of the Layered architecture to simplify the design and allow different developers to work on different layers and so that small changes in one layer does not affect the whole system which would increase the workload. Our system consists of the visual layer, front-end logic layer, API layer and database layer. Visual layer is all the html/css files which provide the basic layout of the user interface, the front-end logic layer makes the relevant API calls in order to retrieve/store data to and from the backend and also processes the data to display the correct content on the UI, the API-layer is responsible for fetching/sending the requested data to and from the database and the database layer contains the data required for the system to work.

3 Architectural Quality Requirements

3.1 Availability

The Office Booking system must have an availability of 99% so that user may at almost anytime have the system ready in case of creating, deleting or checking a booking. Users should be given the opportunity to use the system whenever they wish to use it.

3.2 Responsiveness

The Office Booking system should be responsive when making a booking to allow users to create or delete a booking whenever they please with no issues. The system must respond when a user is attempting to make a booking to prevent other users from making a booking over the current users current booking.

3.3 Learn-ability

The system must be easy to learn and use. New users should be able to start the system up and use it within minutes of discovering the system. If the system takes too long to learn then it could lead to users not wanting to use the system and trying out alternatives.

3.4 Manageability

Manageability is a standard that must be kept for the office booking system as a user should be able control their bookings both being created and deleted. In other words a user must be able to manage their bookings in the system as well as an admin should be able to manage their offices bookings to prevent issues from arising.

3.5 Simplicity

The Office Booker should aim for a simple solution as this provides the user with a better experience as they will be able to use it faster and more efficiently. Their should be a standard of making the system complete the tasks its required while being easily usable and understandable for all users.

3.6 Usability

The Office Booker must apply usability. The user being able to use the system with all of its features will allows for a better user experience, therefore the system must be implemented with care that the user can make use of the system without restriction to how the core features can be used.

4 Architectural Design & Pattern

4.1 Overview

The Model View Controller and Layered Pattern styles can work hand in hand in our current workspace. With the Controllers implementing logic and connections between the database and the front-end, these controllers and services would link up together in the same layer (front-end logic layer) of the Layered Pattern, or connect directly through another layer (API layer). The Views of the MVC pattern correspond with the Visual Layer of the Layered Pattern, compiling together as the HTML pages showcasing the visuals of the web-page. The Model from the MVC is the section that contains the database and how the system should look, this will correspond with the connections of the Database Layer in the Layered Pattern.

4.2 Diagram

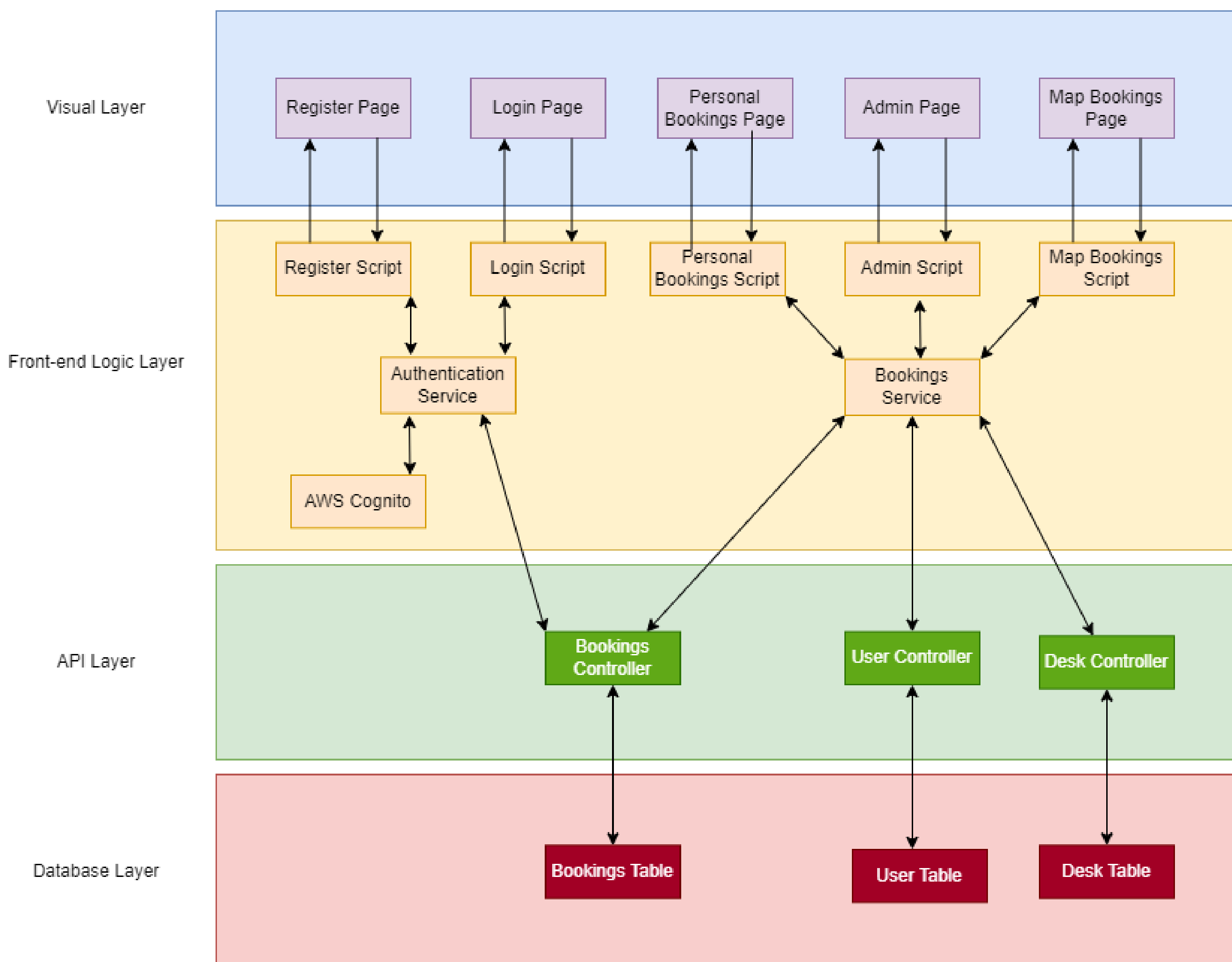

Figure 1: Architecture Pattern Diagram. MVC and Layered Patterns

5 Architectural Constraints

5.1 Free Software

All the different architecture that is made use of must be free or at least have a free option that can be used by the team. Using a paid option would not work as the team can not afford it and that could only lead to the web application being taken down or losing support.

5.2 Frontend

We were limited to 2 options for the frontend of the application. The options were React or Angular. Both of these were given to us by the clients so we had to make a decision on which of the 2 technology's we would want to use for the development of the application.

5.3 Database

For the database we had to decide between using PostgreSQL, Firestore or DynamoDB, as these were the options provided by the client to use.

5.4 Cooperation

If a layer is not working another could potentially suffer preventing progress and the other layer from being used correctly or even being tested. All layers should be properly implemented and taken care of to ensure there are no issues or clashes.

6 Technology Choices

6.1 Angular

Angular was chosen over React, as it is the framework more well known amongst our group. It also provides the necessary structure to meet the architectural pattern needs of Model View Controller. The framework provides a more robust means of styling, with the ability to include Angular Material. This allows for more professional looking designs with the libraries that are provided in Angular Material. Angular also provides easy means to create new components, services and other necessary files with simple commands. These files are then well structured and neatly placed and linked together.

6.2 NestJS

We believed that NestJS would be the best option when it came to creating all the server-side operations required for web application. NestJS was used to implement the API for our system and how the Angular component of the system would communicate with the API.

6.3 PostgresSql

We chose PostgreSQL as we were most familiar with the technology and how to use it for our systems. We had the option of choosing PostgreSQL, Firestone or DynamoDB and of the 3 we believed that PostgreSQL would be the most manageable and would be useful the implementations we were planning.

6.4 AWS Cognito

AWS Cognito was used to provide a secure registration and login process. It provides us with verification codes when a user registers.