**Veterinary Appointment Management System TDD**

Author: Rakshith Koravadi Hatwar

Table of Contents

[Veterinary Appointment Management System TDD 1](#_Toc16165474)

[Introduction 3](#_Toc16165475)

[Assumptions 3](#_Toc16165476)

[Proposed Design 4](#_Toc16165477)

[System Architecture 4](#_Toc16165478)

[Data Model 7](#_Toc16165479)

[Interface/API Definitions 9](#_Toc16165480)

[**1)** **/api/user** 9](#_Toc16165481)

[**2)** **/api/services** 9](#_Toc16165482)

[**3)** **/api/users/:userId/pets** 9](#_Toc16165483)

[**4)** **/users/:userId/appointments** 9](#_Toc16165484)

[**5)** **/js/:jsfilename** 9](#_Toc16165485)

[Code Structure 10](#_Toc16165486)

[a) Server 11](#_Toc16165487)

[b) Client 12](#_Toc16165488)

[Optimizations and Best Practices 13](#_Toc16165489)

# Introduction

The intention of the document is to describe the proposed technical solution to build a web application which allows pet owners and doctors manage pet appointments. The application allows the user to add a pet, add a vet, scheduling an appointment for a pet, and displaying a list of appointments for a pet or a vet.

# Assumptions

The Application is based on **3 user roles**

1. **Admin** – Application administrator who has access to Admin view of the application with the ability to
   1. Add/Remove Users
   2. Add/Remove Pets
   3. Search for Pet information
   4. Schedule/Cancel Appoints
2. **Pet Owner** – The user who owns a Pet and want to avail the services of the Doctor. They have the ability to add pets and make/cancel appointments.
3. **Doctor** – The user who provides services to the Pet Owners. They have the ability to fulfil/cancel appointments.

**Appointment –** The time slot reserved by the pet owner to avail the services of the Doctor.

Below are some rules about scheduling and Appointment:

* Two pets cannot schedule for the same vet at the same time.
* Vets are available M-F 8am-5pm
* An appointment can be canceled any time before it has taken place

# Proposed Design

The proposed technical solution for system is Full stack web application, with a Rest backend implemented in Nodejs powering the ReactJS Front End Application.

## System Architecture

**A screenshot of a cell phone

Description automatically generated**The Diagram shown below shows the proposed system architecture for the backend. The Application is built in a way to facilitate to easy deployment to the cloud. The Application is currently deployed on the Heroku cloud instance. The brief explanation for the backend component follows:

1. **Load Balancer** – Routes the Traffic to the Cluster of the Application Instances.
2. **Clustered Application** – Multiple instances of the Backend application used to serve the client request. Each instance has access to same database and session store to ensure that each browser request can be stateless served by any instance of the clustered application.
3. **Express Session** - The Session Store to store the request context and handle sterilization and deserialization for the requests.
4. **Passport Session** – The Authentication Middleware to serialize and deserialize the Auth Session.
5. **Unsecured Web Router** – The application Router tasked to handle the web page request which are not authenticated (e.g. Login, register). The Routes don’t require user to have logged in with their credentials.
6. **Unsecured API Router –** The application Router tasked to handle the API request which are un authenticated (e.g. Login, register). The API calls don’t require user to have logged in with their credentials to be called.
7. **Authentication –** The Middleware that is responsible to check if the current request context has an active user login session. If the Request is not authenticated:
   1. Web Page Get Request is sent a status of 302 and redirected to Login Page.
   2. POST API Request is denied with 401 Unauthorized.

If the Request has a Login Session, then it is allowed to pass thru and hit the Secured Router.

1. **Secured Web Router** – The application Router tasked to handle the web page request which require the user to be authenticated (e.g. Dashboard, Make Appointment). The Routes require user to have logged in with their credentials.
2. **Secured API Router –** The application Router tasked to handle the API request which are authenticated (e.g. Get Pet Information, Get Appointment). The API calls require user to have logged in with their credentials to be called.
3. **Error handler –** The Generic Application error handler which intercepts and centralizes API and Web Request Error from the entire application.

## Data Model

A close up of text on a white background

Description automatically generatedThe database schema diagram shows various tables used in managing the data in the application. A brief description of the tables is given below.

1. **USER** – This table holds the information about the User of all Roles (Doctor, Pet Owner and Admin). This is the primary source of truth for the User’s Personal Information and Credentials.
2. **PET** – The Table used to store the Pet Information.
3. **PET\_USER** – The Intersection table to store the relation between **PET** and **USER**.
4. **SCHEDULE** – The Table use to associate a Time of Availability to a **USER**. **NOTE**: this is only applicable to the User with Role as Doctor.
5. **APPOINTMENT** – The table to store the instance of appointment booked in the system.
6. **SERVICES** – The Table to store the list of services that the Pet Owners can avail in an appointment. It provides the time taken and cost of the service that can booked in an appointment.
7. **APPOINTMENT\_SERVICES** – The Intersection table to store the services that are to be given as part of the appointment.

## 

## Interface/API Definitions

The Following Endpoints are exposed to client to interact with the Backend Database. API are secured based on the type of data that is returned by the service. Secured services require the user to be logged in to be able to access that data.

### **/api/user**

* 1. GET **(SECURED)**

Get the Users in the System

* 1. POST

Create a User in the System.

### **/api/services**

* 1. GET

Get All the Services that can be added to the appointments.

* 1. POST

Create a new Service in the System.

### **/api/users/:userId/pets**

* 1. GET**(SECURED)**

Get all pets for a User.

* 1. POST**(SECURED)**

Create a new pet for the user.

### **/users/:userId/appointments**

* 1. GET**(SECURED)**

Get all appointments for the user.

* 1. POST**(SECURED)**

Create an appointment in the System for the user.

* 1. PATCH**(SECURED)**

Update the Appointment status and Cancellation Reason.

### **/js/:jsfilename**

* 1. GET

Get a Javascript files for the front end server.

# Code Structure

A screenshot of a cell phone

Description automatically generated

1. **Client** – Contains the code for the Front-end application.
2. **Config** – Contains the application configuration that is used for building and running the application.
3. **Dist** – the folder with the compiled client code to be served by the server.
4. **Docs** – Documentation related to the application.
5. **Server** – Contains the code for the application server.

## Server

A screenshot of a computer

Description automatically generated

1. Controller – Contains controller code for the each of the routes of the application.
2. Dao – Contains code which is used to mutate the database.
3. Helpers – Contains the wrapper code to some common actions.
4. Lib – Code related to extending the libraries.
5. Middlewares – Express middleware
6. Models – The Database tables configurations files.
7. Routers – The Code used for router
8. Utils – Utility code to abstract out common operations.
9. Views – The view template for the application pages.
10. App.js – The main Server applications code.
11. Constants.js – The files containing the application wide constants.
12. Index.js – Server code entry point.

## Client

A screenshot of a cell phone

Description automatically generated

1. Public – The Public Folder containing static resources.
2. Component- Folder containing the components used by the application.
3. Routers – The folder containing the routing code of the application.
4. Screens – The Folder containing the code for the various screens for the application.
5. Utils – The common code Utils functions grouped by the functions.
6. App.js – The main client application.
7. Constants.js – The files containing the application wide constants.
8. Index.js – Client code entry point.

# Optimizations and Best Practices

Describe the potential impacts of the design on overall performance, security, and other aspects of the system.

## Server

1. Code is cloud ready and is already deployed on Heroku Cloud.
2. NodeJs instance is monitored and clustered by PM2, to handle scale and handle application restarts due to failures.
3. Memoized Function to memorize and cache any function (that requires).
4. Connection Pooling
5. Passport Auth Management (extendable to use Facebook login easily).
6. DAO Object for Database management.
7. JSON API Responses are compressed to decrease the amount of the data sent over the wire.
8. ES 6,7 Coding without No Babel on the Server (Babel on server code bloats codes unnecessary)
9. Async Request Handling Middleware Error to centralize the Error handling in the code. (no need to put try catches everywhere).
10. Centralized Handling of application authorizations using Secured Routing, so no need to handle auth checking the at each web/api route.

## Client

1. Differentially served JS files.
2. ES6 Syntax used and Babel preset Env configured to only
3. Full custom configured webpack with code splitting
4. PostCSS-loader to handle the cross-browser CSS differences.
5. Preload, Prefetch and Preconnect critical resources to speed up rendering.
6. Service Worker to preload js and static resources in the background.
7. Manifest.json generated by the webpack for the client.
8. Client library like react are split into its own chunk since they don’t change as often as client code. This facilitates those libraries to be served from browser cache.
9. Mobile First approach while developing. So, it’s designed responsively to handle desktop and mobile.
10. Small Client foot print (shown below).
11. React Hooks used extensive in the application instead of the Class based component.

A screenshot of a cell phone

Description automatically generated