

ONLINE OXYGEN MANGEMENT SYSTEM

Submitted as a part of Final Year Project Report.

By:

Name:	Roll No:
Soumyadeep Das	11900318014
Jaydeep Sarkar	11900318042
Enakshi Pal	11900318044

Department of Electronics and Communication Engineering

SILIGURI INSTITUTE OF TECHNOLOGY

2021-2022

Project Code – EC881

Group ID – 04

Project Supervisor:

Assistant Professor Ms. Jayati Routh.

Project Field: Android based system development

Date: 04-06-2022(Saturday)

Subject Descriptors:

Inventory Management

E-commerce

Keywords:

Oxygen Cylinder and Other Medical supply,

Online Ordering,

User Satisfaction and Fulfilment.

Implementation Environment:

Windows 10

HTML, CSS, JavaScript

VS Code

SILIGURI INSTITUTE OF TECHNOLOGY
P.O. SUKNA, SILIGURI, PIN 734 009, WEST BENGAL

2021 – 2022

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

Certified that the project work entitled '**Online Oxygen cylinder management system**' is a bonafide work carried out by:

Jaydeep Sarkar	11900318042
Enakshi Pal	11900318044
Soumyadeep Das	11900318014

In partial fulfilment for the award for degree of **BACHELOR OF TECHNOLOGY** in **ELECTRONICS & COMMUNICATION ENGINEERING** of the **MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, KOLKATA** during the year 2021-2022. It is certified that all correction/suggestion indicated for internal Assessment has been incorporated in the report deposited in the Department. The project report has been approved as it satisfies the academics requirement in respect of Project Work prescribed for Bachelor of Engineering Degree.

Ms. Jayati Routh
(Assistant Professor)

Mr. Debajyoti Misra
(H.O.D. ECE Dept.)

ABSTRACT

Oxygen Cylinder Management System is a type of E-Commerce Website where customers can buy and sell Oxygen Cylinder and Pandemic related medical supply Products through Internet. This type of E-commerce Website is commonly available on internet, particularly as E-commerce Website. However, there is no existing website that could help supplier and consumer to easily buy and sell through one common website, our software provides industrial oxygen distributors an effective and integrated way to handle cylinder inventory.

During the second wave of COVID-19 world has experienced the devastating effects of sudden spike in COVID-19 cases leading to overburdened health systems and many times inadequate health care. Medical oxygen is the key factor in COVID-19 management and hence medical oxygen management system is one of the important key steps to accelerate the pace of managing the oxygen supply at facilities. India has observed increasing trend case load in the year 2021 which has led to sudden spike in patients requiring oxygen care. Therefore, medical oxygen management system is an important public health concern, especially in developing countries during the COVID-19 pandemic situation.

The purpose of this research is to construct a methodology that enables Consumers and Supplier to easily buy and sell Oxygen Cylinder through Online E-commerce Website and to ease the oxygen delivery through oxygen management system. The proposed methodology has four main steps: Domain and Platform for Ecommerce website; Deciding Pricing, Website design, adding products and set up payments; proposed model to measure performance of ecommerce website using SEO Analytics and lastly, customer Satisfaction and great checkout experience. The significance of the research is that we have enabled our E-commerce website to manage a customer's fulfilment towards proper supply of oxygen cylinder and other medical needs. We created an approach to segment customers supply based on the product availability and demand using aggregated transactional dataset. Our Ecommerce website will not only be able to obtain the optimal Inventory Management system, they will also obtain insights on how customers satisfaction towards the different location & delivering on time with best quality of oxygen and medical supply.

ACKNOWLEDGEMENT

Our work in this Engineering thesis is only a small part of a greater project in understanding the methods used to answer questions in Inventory management System through E-commerce website. We have been benefited greatly from the weekly discussions with our Final Year Project (FYP) supervisor, Asst Prof Ms. Jayati Routh. We were fortunate to be able to learn more about JavaScript, HTML, CSS, MongoDB and React technologies through some of the intellectual discussions at our meetings.

We owe a great many favours to a great number of people who have always been there to encourage and support us throughout the trying year of our FYP in Siliguri Institute of Technology. Our deepest thanks to Asst Prof Ms. Jayati Routh, the advisor of the FYP project and a mentor who has guided and supervised us patiently throughout the entire process, introducing us to other research opportunities in the field, encouraged and supported us to promote our ideas and direction. She gave us the huge opportunity working on this project proposed by her and access to the dataset through Internet and other sources. Despite the difficulty that we had with meeting the demands of the project, she continued to patiently give us feedback on how to improve and finish the report, particularly in the past two months when our research begun to cover topics in optimization were, we had no prior knowledge and faced difficulty to decide which materials to include in the Engineering thesis. We are especially grateful to her for being so tolerant and supportive.

We would also extend our thanks to our family, friends who have been accommodating and supporting us throughout the year. We have only thanked a small group of people and we ask for forgiveness from those we may have excluded accidentally.

Lastly, we would like to acknowledge the hard work of our group members for their dedication and commitment in preparing this Ecommerce website on Online Oxygen Management System. we hope this will prove to be useful in the field for effective and efficient oxygen management system.

Table of Contents

Title	Page no
List of abbreviation	07
Introduction	08
Problem Statement	09
The Solution	10
System Overview	11
Factors Affecting the System	13
Tools/Frameworks Used	14
System Architecture	15
Application Infrastructure – The MERN stack <ul style="list-style-type: none"> Node.js <ul style="list-style-type: none"> Node Module Node Package Manager Node Event Loop Express.js <ul style="list-style-type: none"> Basic Routing Writing and Using Middleware MongoDB <ul style="list-style-type: none"> MongoDB Atlas Mongoose React.js <ul style="list-style-type: none"> JSX Virtual DOM Component 	16 – 36
Application Implementation <ul style="list-style-type: none"> Basic Setup & Routing Home page <ul style="list-style-type: none"> Navbar Slider and website feature Product category Register page <ul style="list-style-type: none"> Login and data flow Shop page <ul style="list-style-type: none"> Product List page Logic and data flow Product description page Cart page Rent page <ul style="list-style-type: none"> Rent provider Description page Refill page 	37 – 59
Discussion <ul style="list-style-type: none"> Future evolution Future Scope 	60

Conclusion	61
Reference	62

List of Abbreviations

ASP	Active Server Pages
JSON	JavaScript Object Notation
BSON	Binary JavaScript Object Notation
MERN	MongoDB, Express, React.js, Node.js
JSX	JavaScript Syntax Extension
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
REST	Representational State Transfer
API	Application programming interface
URL	Uniform Resource Locator
NPM	Node Package Manager
UI	User Interface
DOM	Document Object Model
JWT	JSON Web Token
NPM	Node Package Manager

INTRODUCTION

Oxygen supplementation is an integral part of the management of various respiratory diseases. Its importance has been highlighted during the COVID-19 pandemic. Both acute and severe manifestations of COVID-19 are managed with oxygen. Supply of oxygen can take various forms: oxygen cylinders, oxygen concentrators, or liquid oxygen. Despite most people experiencing mild or uncomplicated symptoms, approximately 15% of patients diagnosed with COVID-19 require oxygen support.

During the second wave of COVID-19 infection, a huge load of patients needed oxygen therapy. The existing production and supply chain system were inadequate to meet patient needs in various pockets across the country and the state. Monitoring the levels of oxygen saturation is recommended for all the patients with symptomatic COVID-19 disease along with oxygen therapy for all severe and critical COVID-19 patients with low oxygen saturation. In COVID-19 patients demand of oxygen varies, starting about 5th day onwards. Right amount of oxygen at golden hour is lifesaving. Oxygen is the most important and essential of the drugs for saving the lives of Covid-19 patients.

STATEMENT OF THE PROBLEM

Medical oxygen in India has been in severe shortage as the country grapples with a deadly second wave of the pandemic. But bureaucratic hurdles are delaying delivery of the critical resource to those in need.

It has also drained supplies of medical oxygen, which is vital for those who have been infected. The dire shortage has turned out to be a major challenge facing hospitals in many states across the country.

Due to the lack of management and scarcity of the oxygen cylinder supply, people were losing their lives daily during the pandemic.

THE SOLUTION

The purpose of this research is to construct a methodology that enables Seller and Users to easily buy and sell oxygen cylinder through Online Oxygen Management System and to measure the user's fulfilment and proper satisfaction with higher quality of oxygen cylinder and delivering through inventory management system round the clock. The methodology has four main steps: Domain and Platform for the website; Decide Pricing, tracking and availability of oxygen products, adding the products and setting up payments; proposed model to measure performance of the website using SEO Analytics and lastly, user Satisfaction and great checkout experience. We created an approach to segment users supply based on the product availability and demand using aggregated transactional dataset. Further, we also provide tools for proper functioning, maintenance and safety of the oxygen equipment and for conducting oxygen audit. The scope of this research is based on addressing the gaps identified in the continuous and adequate supply of medical oxygen, consisting of <10+ medical categories of products, <100+ different locations services and account status, with information such as the market price, price of the products and total purchase amount.

So, to ease the functioning of the Oxygen Delivery we have come up with our project - Online Oxygen Management System.

This project presents a comprehensive view at the entire oxygen ecosystem including the pricing, storage, supply and distribution of medical oxygen and will be helpful to the various stakeholders in this field.

AIMS & OBJECTIVES

The main objective of the study is to develop an Online Oxygen Management System. The proposed system aims to achieve the following objectives:

- To design an Online Oxygen Management System.
- To provide a solution to reduce and optimize the expenses and rush of the patients through proper availability tracking of the oxygen products.
- To create an avenue where people can order for oxygen products online.
- To develop a database to store information on the oxygen products and services.

SCOPE AND LIMITATIONS

Every project is done to achieve a set of goals with some conditions keeping in mind that it should be easy to use, feasible and user friendly. As the goal of this project is to develop an Online Oxygen Management System, this system will be designed keeping in mind the conditions (easy to use, feasibility and user friendly) stated above in the aims and objectives. It may help in effective and efficient order of the Oxygen Products. In every short time, the collection will be obvious, simple and sensible. It is possible to observe the user potentials and purchase and tracking patterns because all the order history will be stored in the database. The efficient management of all the operations of an Online Oxygen Management System will be performed within a single platform. The proposed project would cover:

User Side

- Users need to login to the website using their registered user id and password.
- User can view/search products without login.
- User can also add/remove product to cart.
- When user tries to purchase oxygen product, then he/she must login to the website.
- After creating account and logging in to the website, he/she can place the order.
- User can check their ordered details by clicking on orders button.
- User can track the availability of the oxygen products.

Admin Side

- Admin can login to the website using their registered admin login id and password.
- After login, there is a dashboard where the admin can see and add the oxygen products according to the availability.
- Admin can add/delete/view/edit the products.
- Admin can view/edit/delete customer details.
- Admin can view/delete orders.

FACTORS WHICH MIGHT AFFECT ONLINE OXYGEN MANAGEMENT SYSTEM

There are some factors which might affect the system, are stated below:

1. Convenience (no traffic, crowd ,24 Hours Access)
2. Delivery Mode

SYSTEM OVERVIEW

- During these volatile times, uninterrupted supply of oxygen to hospitals is extremely important for the smooth functioning of healthcare systems. The platform will closely monitor and provide the user with a smooth access to the oxygen suppliers.
- The proposed system will have a login page where both the user and admin can login using their respective login id and passwords.
- After logging in, the user can search the product using the search bar and can add the desired oxygen product to their cart and checkout accordingly.
- Before adding the product, the user can find the statistics of how many cylinders are available currently at the given location. This will help the suppliers provide oxygen to the hospitals in the shortest possible time as per the demand and requirement.
- For front-end (user interface) part, we used HTML, CSS, JavaScript, React JS.
- For back-end part, which include connections between different modules and database connections we used MongoDB (Database).
- The proposed system will be assigned to ensure an easy supply of oxygen not only to the government and private hospitals, but also to the patients who are at home and in need of oxygen, round the clock.
- The web link will directly connect the needy with the suppliers/workers who are in-charge of the oxygen supply chain in the state. All the details will be regularly updated on the portal by the admins.
- By tracking availability online, to avoid emergencies, our proposed system will save time and fulfill the oxygen demands.

TOOLS/FRAMEWORKS USED

REQUIREMENTS

Code Editor	Vs Code
Front-End	HTML, CSS, JavaScript
Back-End	NodeJS, ExpressJS, MongoDB, Stripe
Database	MongoDB

SYSTEM SPECIFICATION

VS CODE EDITOR: Visual Studio Code, also commonly referred to as VS Code, is a source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

Front-End Language:

HTML: The Hyper Text Markup Language or HTML is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies such as Cascading Style Sheets (CSS) and scripting languages such as JavaScript.

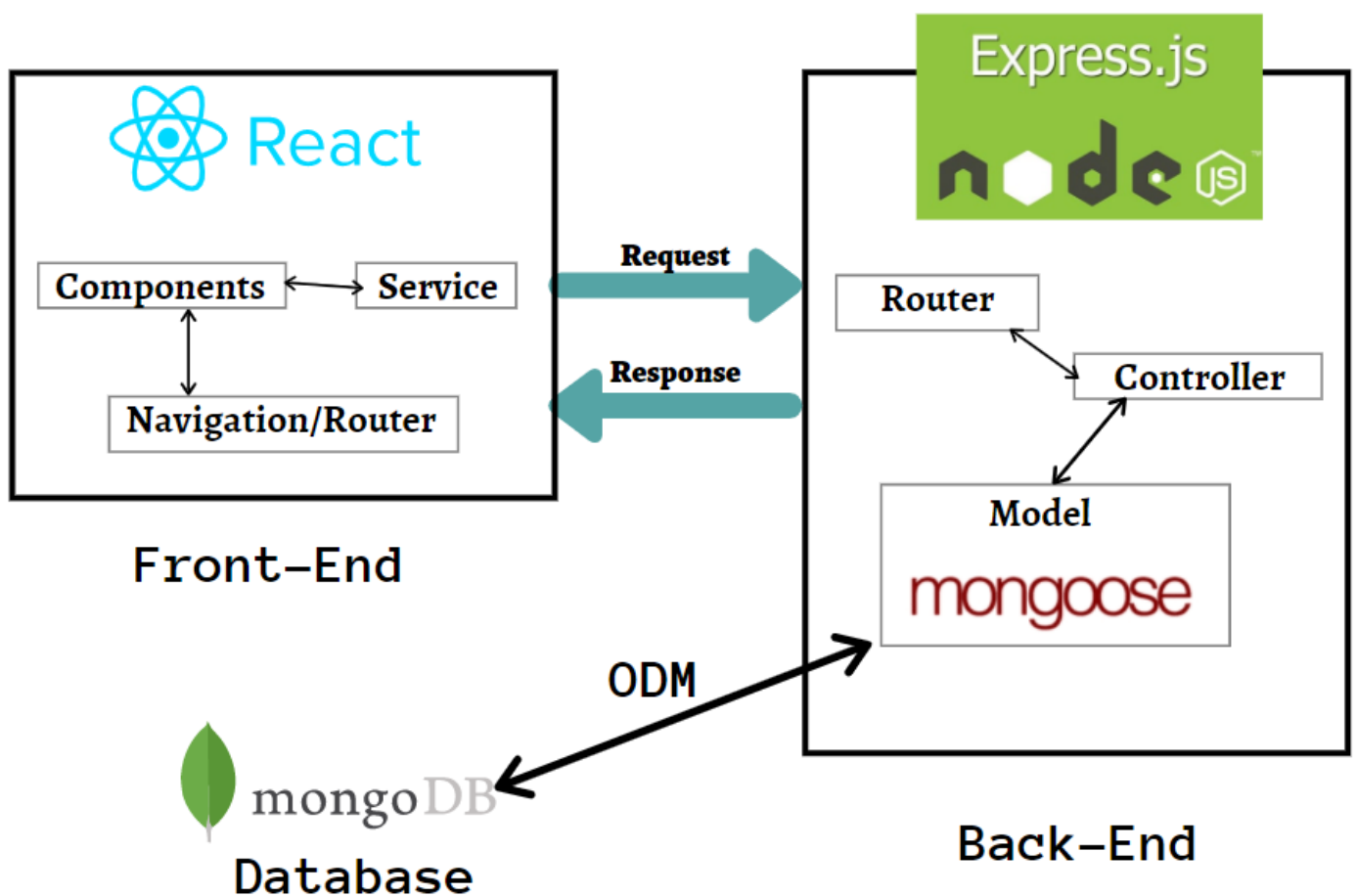
CSS: Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript

JavaScript: JavaScript often abbreviated JS, is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. Over 97% of websites use JavaScript on the client side for web page behaviour, often incorporating third-party libraries. All major web browsers have a dedicated JavaScript engine to execute the code on users' devices.

SYSTEM ARCHITECTURE

• The technologies used for building the Oxygen Management System are MongoDB, React JS, Node JS. This stack is popularly called MERN stack.

- The architectural workflow of MERN stack is as follows:
 - If the client server makes a request, it is processed by React JS. Now, after React JS processes the request, NodeJS takes the control.
 - In NodeJS, the request is processed. Then it requests to the database.
 - Lastly, MongoDB will get the data and will return the data to NodeJS and then the data will be received by React JS, which is responsible for displaying the result.



The architecture control flow is as follows:

Front-end architecture flow:

- New user/admin registers with login id and password. After logging in, the user can search for the desired oxygen products.
- If the user is registered as admin, then he/she can add the oxygen products.
- There is a dashboard, which contains all the products which were ordered, and the user can also track the availability of the products.

Back-end architecture flow:

- On the backend part NodeJS sees that all the data is dynamically updated in MongoDB with the help of Express.js.
- Users are authenticated, and only authenticated users can place the order.
- The users who are authenticated can add items to the cart and checkout.
- NodeJS keeps in contact with MongoDB to update the data dynamically. ExpressJS framework helps NodeJS in achieving this.

APPLICATION INFRASTRUCTURE – THE MERN STACK

The MERN stack is basically a JavaScript-based stack which is created to facilitate the development process. MERN comprises of four open-source elements: MongoDB as the database, Express as server framework, React.js serves as client library and Node.js is an environment to run JavaScript on the server.

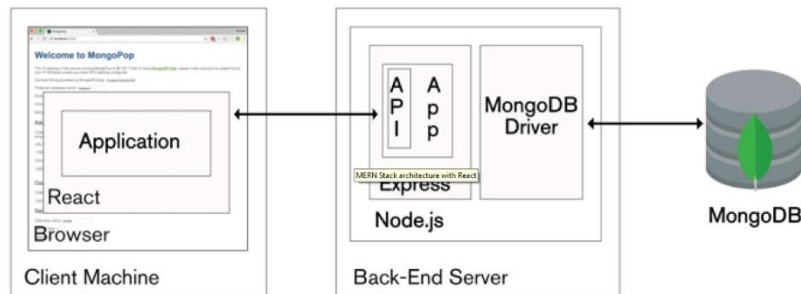


Figure 1. MERN stack architecture

These technologies introduce an end-to-end web stack for developers to utilize in web development.

Figure 1 explains the architecture of MERN stack. Firstly, Express with Node.js create API which is used for logic and to interact with MongoDB. When React client sends a HTTP request to the back-end server. The server analyses the request, retrieves data from the database and responses with that data. React client is updated based on that returned data.

Node.js

Node.js is JavaScript environment provider and the most essential core of the MERN stack. Node.js is now the most widely used free open source web server environment created by Ryan Dahl. It allowed to execute JavaScript code on the server. It is able to run on multiple platforms like windows, Linux and mac OS. Node.js is dependent on Google V8 engine which is the core of Chrome browser. C and C++ run both Node and V8 which is better in performance speed and memory consumption.

A Node app runs in an individual process with event looping, without the need of a new thread for each operation. This is opposed to traditional servers which make use of limited thread to handle requests. Node is a non-blocking, asynchronous and event-driven engine aimed for scalable application development. Generally, Node libraries are created using non-blocking pattern. Application calls a request and then move on to work on next task rather than stalling while waiting for a response. When the request is completed, a call-back function informs the application about the results. This allows multiple connections or requests to a server to be executed simultaneously which is important when scaling applications. MongoDB is also invented to be used asynchronously; there-fore it is compatible with Node.js applications.

Node Module

Node module is similar to JavaScript libraries which consists of a package of functions to be included in application when needed [5]. Node has a variety of modules that offer fundamental features for web applications implementation.

While developers can customize their own modules for personal project, Node has many built in modules that can be used instantly without installation.

One of the most popular built in modules is http module, which can be used to create an HTTP client for a server.

```
var http = require('http');
//create a server object:
http.createServer(function (req, res) {
  res.write('This is shown in client'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

Listing 1. A simple code in Node.js to create a server

Listing 1 demonstrates the usage of http module and ‘create Server’ function to initiate a server running on port 8080. A text string is shown in the client as a response.

Node Package Manager

Node Package Manager (NPM) offers two essential functionalities:

- Command line for performing NPM commands such as package installation and version management of Node packages
- Downloadable repositories for Node package.

In 2017, statistic showed that more than 350 000 packages are found in the npm registry, make it the largest software repository on Earth [7]. Due to being an open source and having a basic structure, the Node ecosystem has prospered dramatically, and currently there are over 1 million open-source free package, which facilitates development process a lot. Beginning with being a method to download and manage Node dependencies, npm has become a powerful means in front-end JavaScript.

There are two types of Node packages installation, globally or locally by the npm install command. The package then is downloaded from the NPM registry and presented in a folder named node_modules. The package is also added to the package.json file under property dependencies. By using require function follow with the package name, it is usable in the project as the example below in listing 2.

```
$ npm install <Module Name>
var name = require('Module Name');
```

Listing 2. Installation and usage of Node package

Express.js

Express is a micro and flexible prebuilt framework based on Node that can provide faster and smarter solution in creating server-side web applications. Express is made of Node so it inherits all Node's features like simplicity, flexibility, scalability and performance as well. In brief, what Express does to Node is the same as Bootstrap does to HTML/CSS and responsive design [11.]. It makes programming in Node a piece of cake and provides developers some additional tools and features to improve their server-side coding. Ex-press is literally the most famous Node framework so that whenever people mention about Node, they usually imply Node combined with Express. TJ Halewyck released Express the first time in 2010 and currently it is maintained by the Node foundation and developers who contribute to the open-source code.

Despite the fact that Express itself is completely minimalist, developers have programmed many compatible middleware packages to solve almost all issues in web development. Express offers a quick and straightforward way to create a robust API with a set of helpful HTTP methods and middleware.

Basic Routing

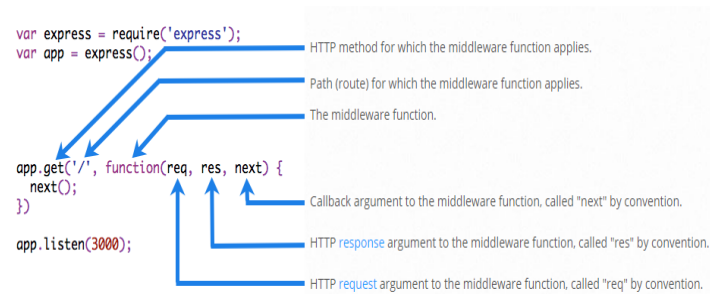
Routing determines the way how an application's endpoints (URIs) interact with client re-quests. The basic syntax consists of an object as instance of Express and the correspondent HTTP request method such as `app.get()`, `app.post()` to handle GET and POST request respectively. [13.]

Routing methods take a call-back function as parameter. The specified function is called when the HTTP requests match the defined routes and methods.

```
var express = require('express')
var app = express() // create instance of express
//respond with 'hello' when a GET request is made to the route :/hello
app.get('/hello', function(req, res) {
  res.send('hello') //response method
})
var callback1 = function(req, res, next) {
  //
  next()
}
var callback2 = function(req, res, next) {
  //
  next()
}
app.get('/multiple', [callback1, callback2])
```

Writing and Using Middleware

Middleware functions have the ability to approach request object, response object and the 'next' function in the application's request-response phase. Middleware execution re-turn an output which could be either the final outcome or could be passed as an argument to the next middleware until the end of the cycle [14.]. The 'next' function belongs to the Express router which starts the following middleware right after the current middleware finished. If a middleware does not finish the req-res cycle, next() must be called to pass control to the next one. Otherwise, the request will be suspended.



MongoDB

MongoDB is a document-based NoSQL database used mainly for scalable high-volume data applications and data involved jobs which does not work well in a relational model. It is among the most popular non-relational database which emerged in the mid-2000s. MongoDB architecture comprises of collections and documents replacing the use of tables and rows from traditional relational databases point of view. One of the most essential functionalities of MongoDB is its ability to store dynamic data in flexible BSON documents, which means documents sharing the same collection can have different fields and key-value pairs, while data structure can be changed without any restrictions at any time. Therefore, this removes the necessity of storing strictly structured data which is obligatory in other relational databases and improves database operation speed significantly. Indexing and aggregation offer robust method to access and work with data easily. Whatever field in a document can be indexed, leading to a much better search performance. MongoDB also provides numerous operations on the documents such as inserting, querying, updating as well as deleting. With the diversity of field value and strong query languages, MongoDB is great for many use cases and can horizontally scale-out to provide storage for larger data volumes, make it stably be the most popular NoSQL database globally.

MongoDB Atlas

MongoDB Atlas, which announced in 2016 by MongoDB creator team, is the cloud ser-vice for storing data globally. It offers everything MongoDB provide without further requirements when building applications, allowing developers to fully focus on what they do best. With Atlas, developers do not have to worry about paying for unnecessary things as they follow a pay-as-you-go standard.

MongoDB Atlas introduces a simple interface to get started and running. Basically, by choosing instance size, region and any customized features needed, Atlas brings out the suitable instance and includes it with the following:

- **Built-in Replication:** Atlas ensures constant data availability by providing multiple servers, even when the primary server is down.
- **Backups and Point-in-time Recovery:** Atlas puts a lot of efforts to prevent data corruption.
- **Automated Update and One-Click Upgrade:** users take advantage of the latest and best features as soon as they are released due to Atlas auto-mated patching.
- **Various options for additional tools:** users can freely decide on which regions, cloud providers and billing options they prefer to use, making them feel like customizing their own instances.
- **Fine-Grained Monitoring:** users are kept up with a diversity of organized information, to let them know the right time to advance things to the next level.

MongoDB Atlas is practical and reliable thanks to its integrated automation mechanisms. With Atlas, it is unnecessary to concern about operational tasks as following:

- **Supply and Configuration:** Atlas gives clear instruction step by step to go through the setup process, so developers do not have to think about what aspect to choose even if they are beginners.
- **Patching and Upgrades:** Atlas is integrated with automatic upgrade and patching, ensures user can reach latest features when they are released as patching process takes just some minutes with no downtime.
- **Monitoring and Alerts:** database and hardware metrics are always visible so users can foresee and prepare for any performance and user experience problems.

Mongoose

Mongoose is an Object Data Modelling (ODM) JavaScript-based library used to connect MongoDB with Node.js. Mongoose provides a straight-forward, schema-based solution to create your application data template. It handles data relationships, provides methods to validate schema, and is used to render and represent between objects in MongoDB.

Mongoose schema is covered inside a mongoose model, and a specific schema creates model. While schema and model are slightly equivalent, there is a primary difference: schema determines the document formation, default values, validator, etc. while model is responsible for document-related operation like creating, querying, updating and deleting. A schema definition is simple, and its structure is usually based on application requirements. Schemas are reusable and can include multiple child-schemas. Schema applies a standard structure to all the documents in a collection. Moreover, Mongoose provides additional features like query

helper functions and business logic in the data. For example, Mongoose can help connect database with the server and perform typical database operations for reading, writing, updating and deleting data. Mongoose removes the need of writing complicated data validations by providing schema validation rules to allow only acceptable data to be saved in MongoDB.

React.js

React is an open-source front-end JavaScript-based library that specializes in building user interfaces. It was originally created by a Facebook software engineer and first implemented in newsfeed feature of Facebook in 2011 and followed by Instagram a year later. Due to the minimal understanding requirement of HTML and JavaScript, React is easy to learn and thanks to the support by Facebook and strong community behind it, React expands its popularity and becomes one of the most used JavaScript library.

React works only on user interfaces of web and mobile applications. This is equivalent to the view layer of MVC template. ReactJS allows developers to create reusable UI components as well as to create scalable web applications that can change data without reloading the page. In addition to providing reusable component code, which means

- JSX (JavaScript Syntax extension)
- Virtual Dom
- Component

JSX

The beginning foundation of any basic website is made by HTML files. Web browsers read and display them as web pages. During this process, browsers create a Document Object Model which is a representational tree of the page structure. Dynamic content can be added by modifying the DOM with JavaScript. JSX is XML/HTML-like syntax used to make it easier to modify the DOM by using simple HTML-like code. Literally, JSX makes it possible to write brief HTML-like structures in the same file with JavaScript code, then transpiler such as Babel will convert these expressions to standard JavaScript code. Contrary to the past when JavaScript is put into HTML, now JSX allows HTML to be mixed in JavaScript.

```

1  const name = 'Josh Perez';
2
3  //JS expression inside curly brace in JSX
4  const element = <h1>Hello, {name}</h1>;
5
6  //use quotes for string values
7  const element = <div tabIndex="0"></div>;
8
9  //use curly braces to embed a JavaScript expression in an attribute
10 const element1 = <img src={user.avatarUrl} />;
11
12 ReactDOM.render(
13   element,
14   document.getElementById('root')
15 );

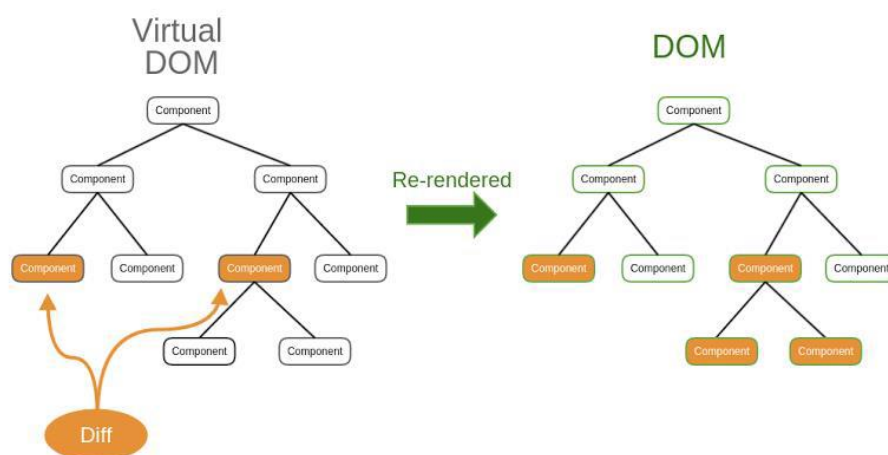
```

Virtual DOM

Considered as the next important breakthrough in web development after AJAX, the virtual DOM (Document Object Model) is the main reason why React is able to create fast and scalable web application.

Normally without react, the web application uses HTML to update its DOM. This works well for uncomplicated and static sites, but it can be a huge performance issue for dynamic websites especially the one that involve high user interaction and view update because the whole DOM needs to reload every time the user triggers a page refresh. Despite of JavaScript engines nowadays which are fast enough to carry out such complicated applications, DOM manipulations are still not that advanced. Another factor is that DOM is tree-structured so that even a change in a layer can cause extreme changes. Updating DOM is always the shortcoming when it comes to performance. Re-act optimized this by introducing Virtual DOM.

Virtual DOM is a representation of the DOM object, where it executes all updates re-quired before rendering the final page to the browser. Whenever a user action happens or by any means, there is a change to the DOM, react will generate a copy of the Virtual DOM. At that point, a diff algorithm compares the previous and current states of the Virtual DOM and figures out the optimal way with the least amounts of updates needed to apply these changes. If there are any differences, react updates that single element only rather than update a ton of elements, while if there is no change, react make browser render nothing. As a result, this behaviour spends less computing power and less loading time, leading to better user experience and performance.



Component

React is all about component, which reflects the primary unit of every application. React allow developers to divide the user interface into simple, independent and reusable components. A developer can start with common tiny components that can be used repeatedly on several pages like navigation bar, button, input form etc. This is what reusability means, which is a perfect way to reduce development time, so that developers can concentrate on more crucial features and business logic. Then comes the turn of wrapper components that include children's components with internal logic. Each component determines its own way how it should be rendered. A component can also be combined with other components or nested inside higher order components. The development process keeps going on like that until it reaches the root component which is also the application. As a result, this approach guarantees the application consistency and uniformity which make it easier for maintenance as well as further codebase growth.

Application Implementation

A prototype version of the e-commerce application was created to apply the study of MERN stack as well as to understand how they contribute to the whole entity in web development.

Application requirements

- As a user, user want to create an account for himself
- As a user, user want to update his profile
- As a user, user want to surf through all the products
- As a user, user want to see product information such as category, price, name, review, picture, etc.
- As a user, user want to add many products to the shopping cart and is able to view the cart
- As a user, user want to delete products from the cart
- As a user, user want to modify the quantity of products inside the cart
- As a user, user want to pay for the products in the cart
- As an admin, admin want to add product to the database
- As an admin, admin want to remove product from the database
- As an admin, admin want to update product to the database
- As an admin, admin want to add category to the database
- As an admin, admin want to manage user orders

Application development

This section is dedicated to demonstrate the functionalities development process from back-end to front-end of the e-commerce application. Due to the limited scope of this thesis, it is not able to mention all the files or describe every step in the project into details, but it aims to discuss precisely about all fundamental and important parts that needed to implement the MERN application. Basic concepts of any third-party libraries or module are also explained along the way. The project structure is divided into 2 folder, ecommerce and ecommerce-front which contain the source code of back-end and front-end respectively.

Back-end development

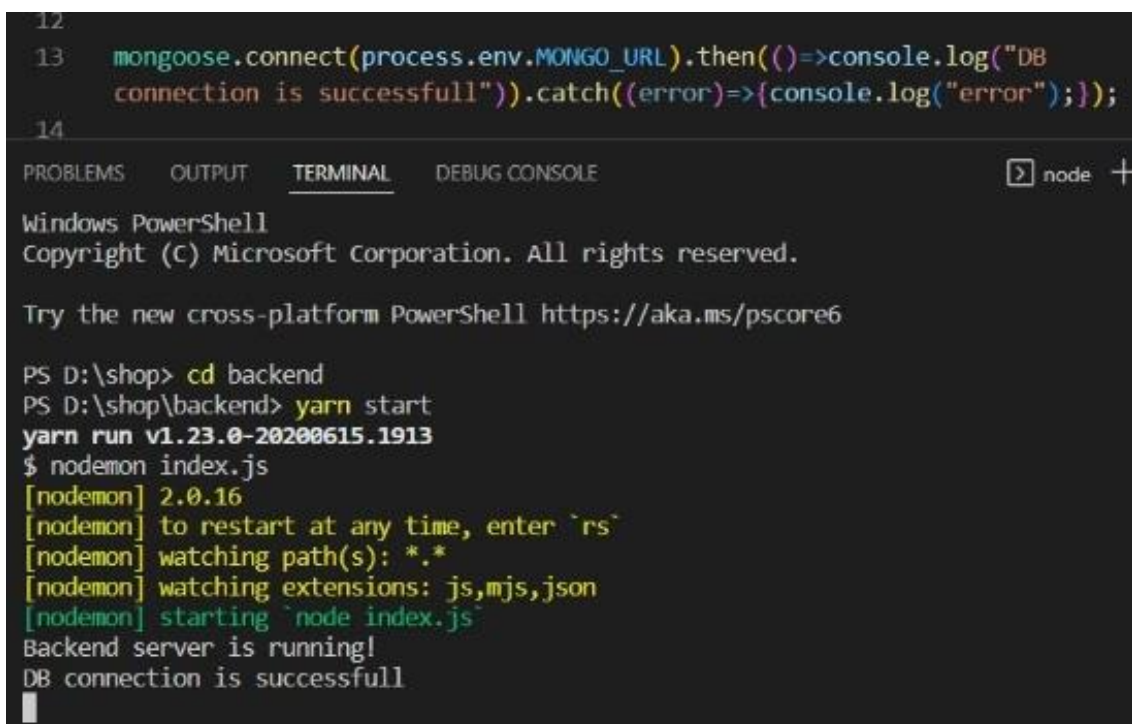
Basic setup

The very first thing to do is to set up an Express application by creating an index.js file, which is the entrance file of the project, where some of the necessary middleware's and node packages are stored.

```
const express = require("express") ;
const app = express();
const mongoose = require ("mongoose");
const dotenv = require("dotenv");
const userRoute = require("./routes/user");
const authRoute = require("./routes/auth");
const productRoute = require("./routes/product");
const cartRoute = require("./routes/cart");
const orderRoute = require("./routes/order");
```

The following step is setting up database connection. MongoDB atlas is used due to various advantages it brings in for the project as chapter 2.3.3 summarizes above. Since Atlas is an online database service, no installation is required. By accessing MongoDB Atlas official site and following instruction there, a cluster is created with the author's personal choice of cloud provider and region followed by a connection string which is saved as MONGO_URI variable in the environment file. It is then used by mongoose to connect to the database. When the connection is successful, 'DB Connected' is displayed in the terminal. Otherwise, a message telling the connection error is printed. The back-end folder is structured by 2 main sub-folders: models, routes and controllers. While models contain all the mongoose schemas that are based on application requirements, routes define all the API routes.

```
12
13 mongoose.connect(process.env.MONGO_URL).then(()=>console.log("DB
    connection is successfull")).catch((error)=>{console.log("error");});
14
```



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE node +

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell <https://aka.ms/pscore6>

```
PS D:\shop> cd backend
PS D:\shop\backend> yarn start
yarn run v1.23.0-20200615.1913
$ nodemon index.js
[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Backend server is running!
DB connection is successfull
```

Mongoose Schema Creation

There is a total of 4 schemas in this application: cart, order, product and user. Each mongoose schema represents for a document structure User schema is written first be-cause user model is needed to handle authentication and authorization.

User schema

As can be seen from figure, a user schema has 3 required properties called name, email and password. Role property is set to 0 by default as it means this is a normal user. By manually changing the role value to other (in this application: 1) the user becomes an admin. The user model only accepts defined properties in the schema to be added to the database. While history shows an array of purchases that user used to make, timestamps display the time of creation and modification of a user object.

```
const mongoose = require("mongoose")

const UserSchema = new mongoose.Schema(
  {
    username:{type: String , required:true, unique:true},
    email:{type: String , required:true, unique:true},
    password:{type: String , required:true},
    isAdmin:{
      type:Boolean,
      default:false,
    },
  },
  {timestamps:true}
);

module.exports = mongoose.model("User", UserSchema);
```

For security reasons, real password is not saved in the database but an encrypted one, therefore the author used Crypto, which is a Node module to hash the password while combined with a tool called CryptoJs.

```
//REGISTER
router.post("/register", async (req, res) => {
  const newUser = new User({
    username: req.body.username,
    email: req.body.email,
    password: CryptoJS.AES.encrypt(
      req.body.password,
      process.env.PASS_SEC
    ).toString(),
  });

  try {
    const savedUser = await newUser.save();
    res.status(201).json(savedUser);
  } catch (err) {
    res.status(500).json(err);
  }
});
```

```
//LOGIN
router.post("/login", async (req, res) => {
  try {
    const user = await User.findOne({ username: req.body.username });
    !user && res.status(401).json("Wrong credentials!");

    const hashedPassword = CryptoJS.AES.decrypt(
      user.password,
      process.env.PASS_SEC
    );
    const OriginalPassword = hashedPassword.toString(CryptoJS.enc.Utf8);

    OriginalPassword !== req.body.password &&
      res.status(401).json("Wrong credentials!");

    const accessToken = jwt.sign(
      {
        id: user._id,
        isAdmin: user.isAdmin,
      },
      process.env.JWT_SEC,
      {expiresIn: "7d"}
    );

    const { password, ...others } = user._doc;

    res.status(200).json({...others, accessToken});
  } catch (err) {
    res.status(500).json(err);
  }
});
```

It demonstrates how 'hashed_password' is created based on CryptJs. Firstly, CryptJs creates a property named password for user object. When a new user is signed up, the value of 'password' property from the rebody is set to the virtual property and is passed to setter function as an argument. Then 'password' is encrypted and set to 'hashed_password' property. A function is also assigned to user Schema method object to create an authenticate method which compare the encrypted input password with the hashed one to verify the user.

Product schema

A product schema is structured as figure below. It has all the properties that any e-commerce product may need, like name, description, price, quantity, category, etc. Sold field is set to 0 as default value and is incremented after each user purchase. One differ-ent with the user schema above is that category field has a type of ObjectId while the 'ref' option indicates which model to refer to during population (in this case category model). This illustrates the relationship between product model and category model and by using populate method, category data is no longer its original _id but replaced with mongoose document retrieved from the database.

```
const ProductSchema = new mongoose.Schema({
  {
    title:{type: String , required:true},
    desc:{type: String , required:true},
    img:{type: String , required:true},
    brand:{type: String , required:true},
    categories:{type: Array},
    capacity:{type: Array , required:true},
    price:{type: Number , required:true},
    inStock:{type: Boolean,default:true},
  },
  {timestamps:true}
});
```

While user schema and product schema look complex, category schema is pretty simple and straightforward with only name and timestamps field.

Order schema

A order schema is structured as figure below. It has all the properties that any e-commerce product may need for ordering , like productId , quantity , amount ,address ,payment status etc. its job is to collect and provide necessary info for uor payment api,and to send a slip to the admin dashboard.

```
const OrderSchema = new mongoose.Schema({
  {
    userId:{type: String , required:true},
    products:[
      {
        productId:{
          type:String
        },
        quantity:{
          type:Number,
          default:1,
        },
      },
    ],
    amount:{type:Number, required:true},
    address:{type:Object, required:true},
    status: {type:Object, default:"pending"},
  },
  {timestamps:true}
});
```


these are the 3 main functions we add using order schema, this an admin dashboard side functionality.

```
// DELETE
router.delete("/:id", verifyTokenAndAdmin, async (req, res) => {
  try {
    await Order.findByIdAndDelete(req.params.id)
    res.status(200).json("Order has been deleted....")
  } catch (err) {
    res.status(500).json(err)
  }
})

// GET USER ORDERS
router.get("/find/:userId", verifyTokenAndAuthorization, async (req, res) => {
  try {
    const orders = await Order.find({userId: req.params.userId});
    res.status(200).json(orders);
  } catch (err) {
    res.status(500).json(err)
  }
});

// CREATE
router.post("/", verifyToken, async (req, res) => {
  const newOrder = new Order(req.body)
  try {
    const savedOrder = await newOrder.save();
    res.status(200).json(savedOrder);
  } catch (err) {
    res.status(500).json(err)
  }
})

// UPDATE
router.put("/:id", verifyTokenAndAdmin, async (req, res) => {
  try {
    const updatedOrder = await Order.findByIdAndUpdate(req.params.id, {
      $set: req.body
    }, {new: true});
    res.status(200).json(updatedOrder);
  } catch (err) {
    res.status(500).json(err);
  }
});
```

Cart schema

A cart schema is structured as figure below. It has all the properties that any e-commerce product may need for adding product to cart, like productId, quantity, amount etc. its job is to collect and provide necessary info for payment api, and to send a slip to the admin dashboard.

```
const CartSchema = new mongoose.Schema({
  {
    userId: {type: String, required: true},
    products: [
      {
        productId: {
          type: String
        },
        quantity: {
          type: Number,
          default: 1,
        },
      },
    ],
  },
  {timestamps: true}
});

router.post("/", verifyToken, async (req, res) => {
  const newCart = new Cart(req.body)

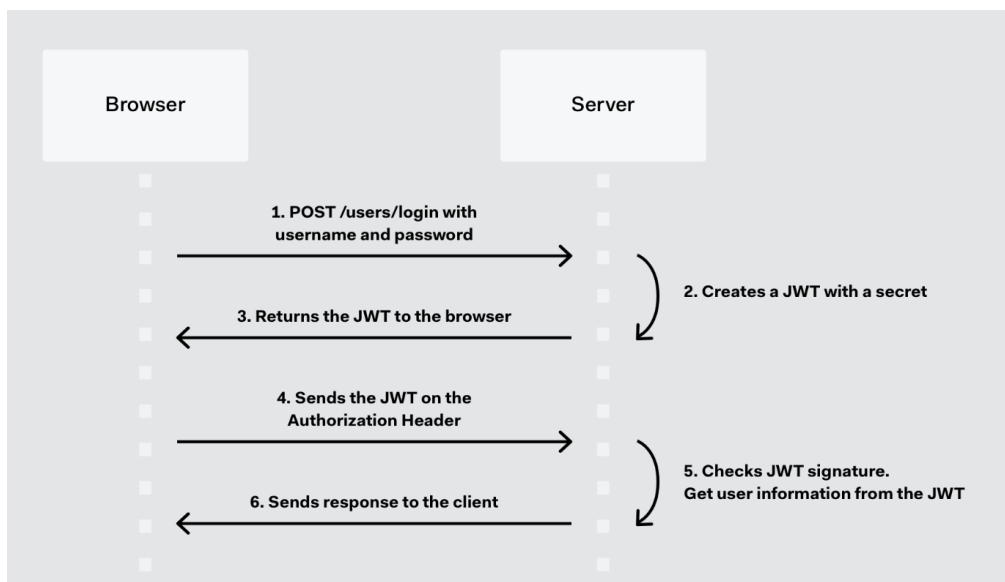
  try {
    const savedCart = await newProduct.Cart
    res.status(200).json(savedCart);
  } catch (err) {
    res.status(500).json(err)
  }
});
```

we also used jwt to verify user to give an security service, so user needed to login first, so that it also allows us to save user cart history.

Authentication with JWT

JSON Web Token (JWT) is a standard that defines a compact and self-contained way to transmit data securely between client and server as a JSON object. Tokens can be transferred through an URL or a HTTP request header easily thanks to its compact size while self-contained feature let JWT comprise all the necessary user information. Data inside a JWT is verified and reliable because it is signed with a secret key or a public private key pair. Moreover, the signature also confirms that only the party holding the private key have signed it.

Authentication is the most popular use case of JWT. After a user signs in, the authorization is granted and a JWT is created by the application and sent back to the user. By that time, all following requests made by user will contain the token in the authorization header with Bearer schema. The server will check for a valid token, and if it is available, the user is allowed to access protected routes, services and resources that require that token.



As JWT is self-contained, all needed data stay there, leading to less works toward the database [36.]. This whole process is analyzed in figure above.

```

const verifyToken = (req,res,next) =>{
  const authHeader = req.headers.token;
  if(authHeader){
    const token = authHeader.split(" ")[1];
    jwt.verify(token, process.env.JWT_SEC,(err,user)=>{
      if(err) res.status(403).json("Token is not Valid!");
      req.user = user;
      next();
    });
  }else{
    return res.status(401).json("you are not authenticated!");
  }
};

const verifyTokenAndAuthorization = (req,res,next)=>{
  verifyToken(req,res, ()=>{
    if(req.user.id === req.params.id || req.user.isAdmin){
      next();
    } else{
      res.status(403).json("you are not allowed");
    }
  });
};

const verifyTokenAndAdmin = (req,res,next)=>{
  verifyToken(req,res, ()=>{
    if(req.user.isAdmin){
      next();
    } else{
      res.status(403).json("you are not allowed!");
    }
  });
};

```


Figure demonstrates how JWT is implemented in the 'login' controller method. This method is executed when there is a POST request to the route '/api/login'. Firstly, user credentials are retrieved from the request body and set to the corresponding variable. Then database looks for the user with that email and checks if the password is matched with hashed password. If yes, a signed token is generated with user ID and a secret random string which is saved in environment file. This secret string key is needed for security reasons since tokens are user credentials. Then the token is attached in cookie with an expiry time and is returned with user data as json format to the client.

Routing and API Documentations

Data distribution and sharing between two or more systems has always been an essential aspect of software development. Taking into account a case when a user search for some books of a specific category, an API is called and the request is sent to the server. The server analyses that request, performs necessary actions and finally, user gets a list of books as a response from the server. This is the way how REST API works. An API stands for Application Programming Interface, which is a set of rules that allows two applications or programs to interact with each other.

REST determines how the API looks like. It stands for "Representational State Transfer" -an abstract definition of the most popular web service technology. REST is a way for two computer systems to communicate over HTTP in a similar way to web browsers and servers. It is a set of rules that developers follow when they create their API.

Product related API

Method	Route path	Description
GET	api/product/:productId	Return a product with the given ID under json format
POST	api/product/create/:userId	Create a new product and save it to database, return a json object of that product. Required to be admin and be authenticated
DELETE	api/product/:productId/:userId	Remove a product with the given ID and return a json message object. Required to be admin and be authenticated

PUT	api/product/:productId:userId	Update a product with the given ID and return updated product under json ob-ject. Required to be admin and be authenticated
GET	api/products	Return a list of products under json for-mat
GET	api/products/search	Return a list of products based on search query under json format
GET	api/products/related/:productId	Return all the products with the same category as the given ID product

```

//Create
router.post("/", verifyTokenAndAdmin, async(req,res)=>{
  const newProduct = new Product(req.body)

  try{
    const savedProduct = await newProduct.save();
    res.status(200).json(savedProduct);
  }catch(err){
    res.status(500).json(err)
  }
})

// //DELETE
router.delete("/:id",verifyTokenAndAdmin, async (req,res)=>{
  try{
    await Product.findByIdAndDelete(req.params.id)
    res.status(200).json("Product has been deleted....")
  }catch(err){
    res.status(500).json(err)
  }
})

```

Table above illustrates all the product related API routes with corresponding method and description in brief. As can be seen, only user with admin role can perform operation like creating, deleting and updating to a product. Other public routes can be accessed wherever and by whatever kind of user even without a signed account.

We also make use of router. Param method as most of the product API routes above have a parameter (productId, userId) inside. Router. Param finds the product with that ID and populate its information to request object under product property.

With the help of jwt, after verifying authentication we are allowed to add or delete products.

API Testing with Postman

At this moment of the development process, the front-end is not available yet so a third-party tool called Postman is used to handle API testing.

Registering Users & fetching all user's data from Database

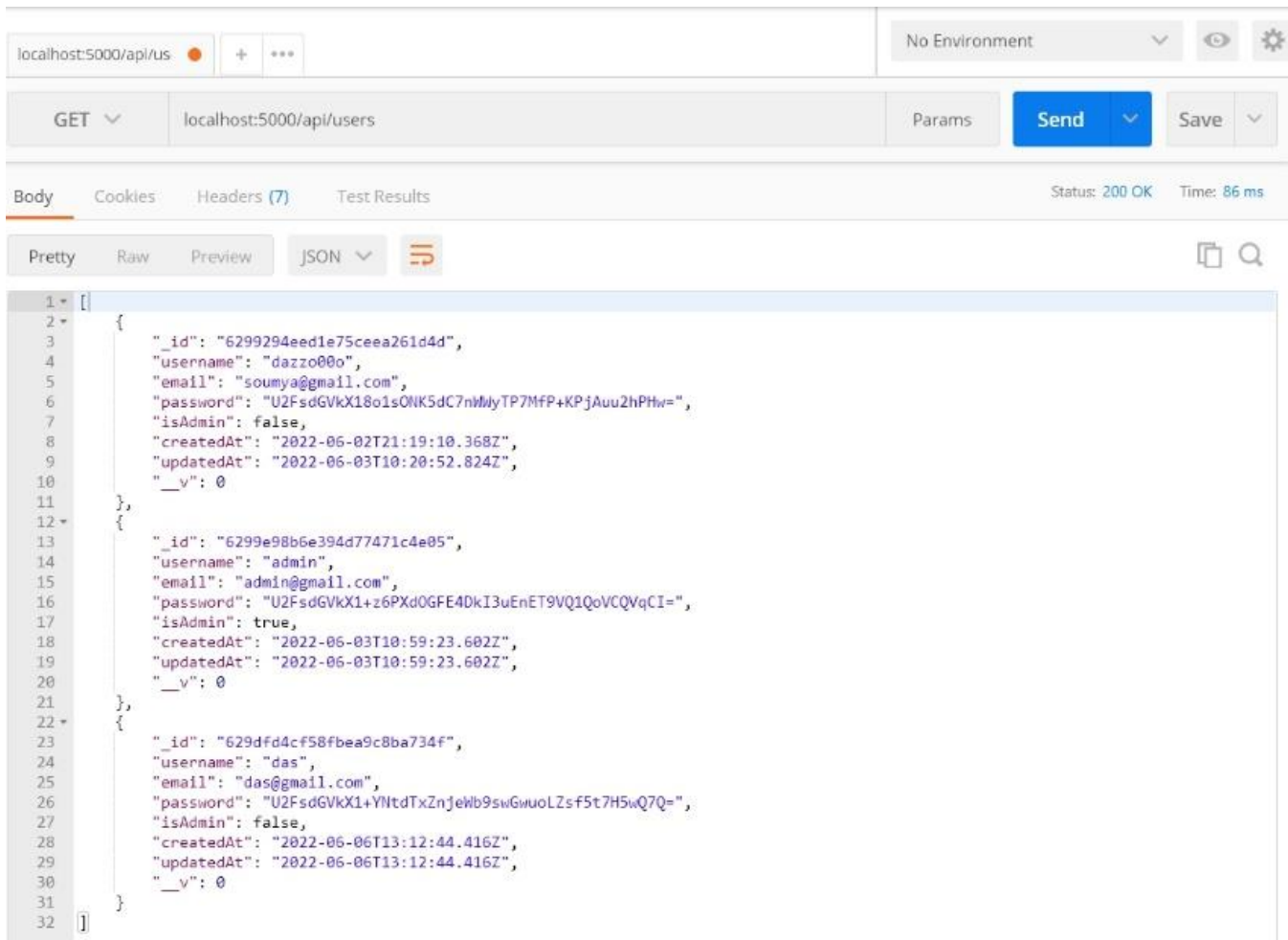
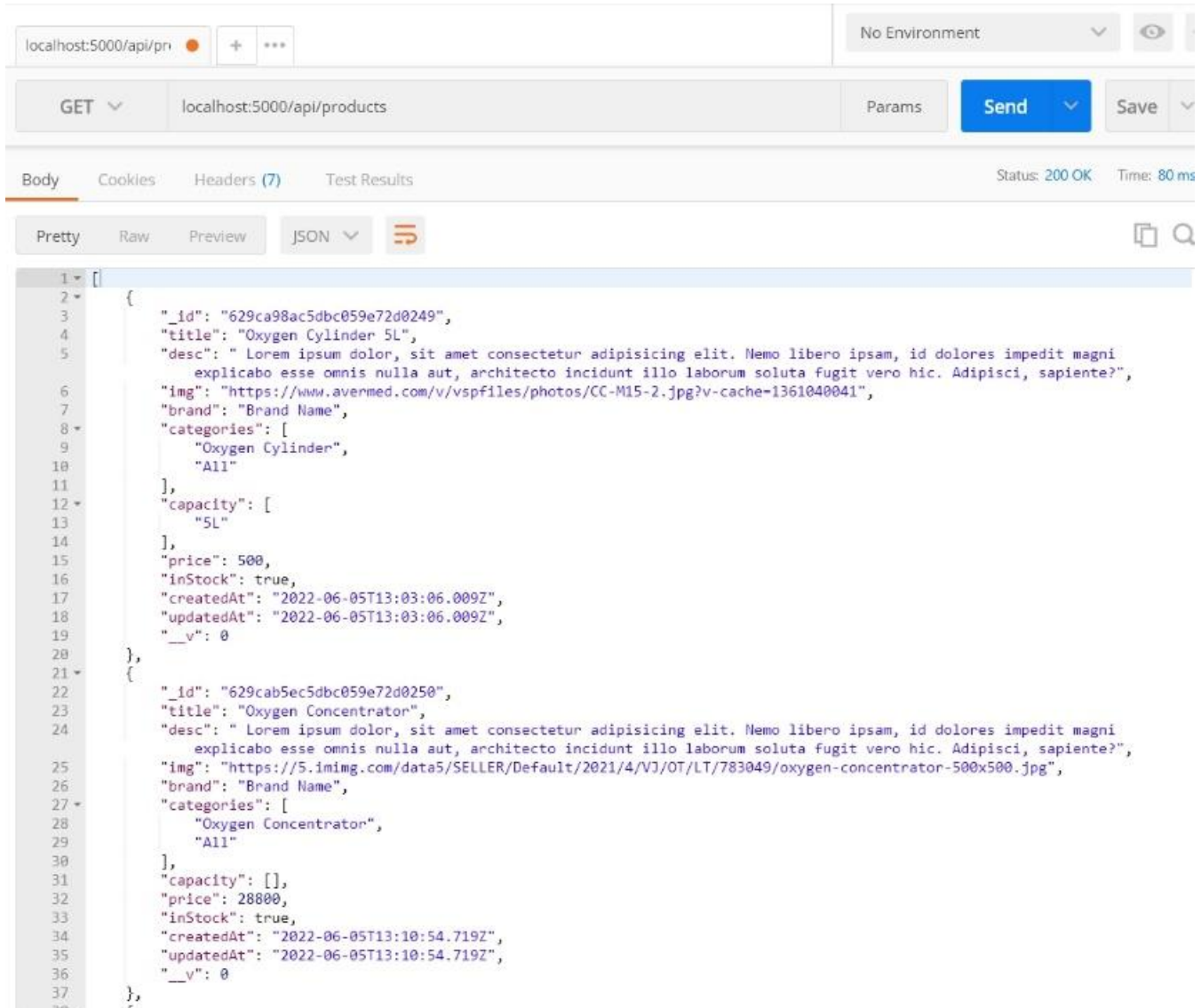


Fig: A screenshot of Testing with Postman

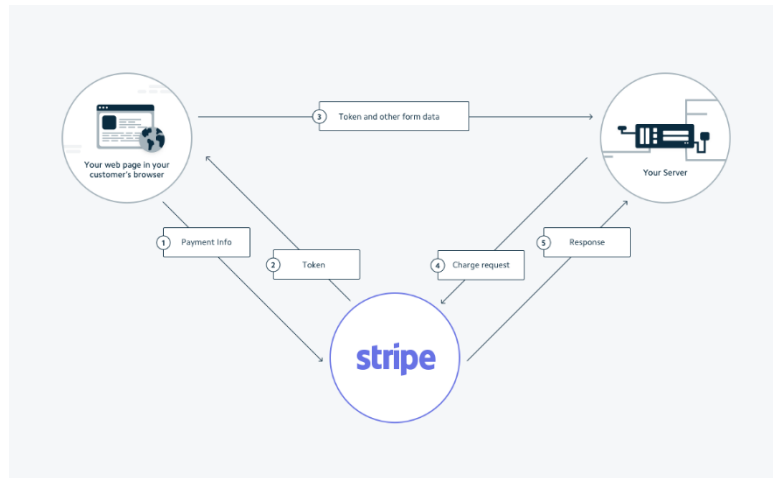
Adding products & fetching all product's data from Database



As can be seen from figure, a GET request has been called to the route 'api/users' and 'api/products' been successfully handled. Since this is a public route and categories is not protected resource, no authorization token in the header is needed. As a result, an array of json object containing all the categories is returned. Each category object includes all the properties defined in Mongoose schema like name, timestamps and auto generated ID.

Stripe Payment API

The Stripe Node library provides convenient access to the Stripe API from applications written in server-side JavaScript. For collecting customer and payment information in the browser, we used **Stripe.js**.



Stripe Payments handles the steps between a customer providing their card information and learning that their payment has been accepted.

Here's how it works:

- First, the customer provides their card information, either online or in person.
- Those card details enter Stripe's payment gateway, which encrypts the data.
- Stripe sends that data to the acquirer, a bank that will process the transaction on the merchant's behalf. In this step, Stripe serves as the merchant (with the business owner as a sub-merchant). This means Stripe users don't have to set up a merchant account, which can be cumbersome.
- The payment passes through a credit card network, like Visa or Mastercard, to the cardholder's issuing bank.
- The issuing bank approves or denies the transaction.
- That signal travels from the issuing bank through the card network to the acquirer, then through the gateway to the customer — who sees a message telling them the payment has been accepted or declined.

```

router.post('any yment', (req,res) =>{
  stripe.charges.create({
    source: req.body.tokenId,
    amount: req.body.amount,
    currency: "inr",
  },(stripeErr,stripeRes)=>{
    if(stripeErr){
      res.status(500).json(stripeErr);
    }else{
      res.status(200).json(stripeRes);
    }
  });
});

```

Dotenv

Dotenv is a zero-dependency module that loads environment variables from a `.env` file into `process.env`. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.

we have saved all our important secret keys like stripe keys, database access keys here and processed it when needed.

```
const stripe = require("stripe")(process.env.STRIPE_KEY);
```

Front-end development

Basic setup and Routing

Unlike back-end is performed with multiple technologies, front-end development is handled by only React.js. It is not impossible to set up a React application from scratch, but this process includes of many intimidating and time-consuming steps, such as setting up Babel to convert JSX to compatible JavaScript that old browser can understand and configuring Webpack to bundle all the modules. Fortunately, Facebook has developed Create React App node module for quickly scaffolding a React template, which saves developers from all those complicated configurations. Create React App not only generates a boilerplate with core structure for a React application but also takes care of some build script and development server.

After running command ‘npx create-react-app’ to generate a template, bootstrap – the most popular CSS framework is added to minimize styling tasks from scratch since the author want to focus on the development part using MERN stack.

Then Routes.js file is created with the sole purpose of including all the route paths with their corresponding components by using a package called react-router-dom. Route component is then rendered by React Dom as the root component of the application.

```
const App = () => {
  const user = useSelector((state) => state.user.currentUser);
  return (
    <Router>
      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/Shop"><Shop/></Route>
        <Route path="/products/:category"><Shop/></Route>
        <Route path="/product/:id"><Product /> </Route>
        <Route path="/Rent"><Rent/></Route>
        <Route path="/RefillCenter"><RefillCenter/></Route>
        <Route path="/VendorDtIs"><VendorDtIs /></Route>
        <Route path="/cart"><Cart /></Route>
        <Route path="/success"><Success /></Route>
        <Route path="/login">{user ? <Redirect to="/" /> : <Login />}</Route>
        <Route path="/register">{user ? <Redirect to="/" /> : <Register />}</Route>
      </Switch>
    </Router>
  );
};
```

Figure summarizes all the route paths and the component to be render when that route is accessed. For example, any kind of user can go to public route like ‘/shop’ to access Shop component, but only authenticated user can see Dashboard or Profile com-ponent as they are protected routes.

There is a total of three main types of components in React Router:

- routers, like <BrowserRouter> and <HashRouter>
- route matchers, like <Route> and <Switch>

- navigation, like `<Link>`, `<NavLink>`, and `<Redirect>`

Router component is the wrapper of React Router application. `BrowserRouter` is mainly used for web-based projects. Router component must be rendered as the root element.

`Switch` and `Route` are route matchers. When a `<Switch>` is rendered, it finds the `<Route>` element whose path is similar to the current URL. When a `<Route>` is found, it renders that `<Route>` and ignores the rest. This means `<Route>` with more specific (usu-ally longer) paths should be written before less-specific ones [38.].

One essential thing to consider when setting up `Route` is that a `Route` path matches the beginning of the URL, not the whole URL. Therefore a `<Route path="/">` will match every URL. That is why `<Route>` should be located last in the `<Switch>`, otherwise `<Route exact path="/">` can be used to match the whole URL.

Home Page

Oxygen+

EN

Shop


Rent

Refill

Register

Login

Logout



96% PURE OXYGEN

Remove Breathlessness and let's Fight Covid-19 Together.

VIEW NOW



Free Shipping



Online Order



Save Life



Pure Oxygen



Rental Service



24/7 Support

Product Catagory

Let's Fight Covid-19 Together



Oxygen Cylinder



Oxygen Concentrator



Oxygen Mask



Regulator+Flowmeter

Sign Up For Newsletter

Get timely updates on your product's Availability.

Enter Your Email Address



Oxygen+

Contact

52 Abcd Road, Street 44, Siliguri
+91 9609912804
10:00 - 18:00, Mon - Sat

Follow Us

10:00 - 18:00, Mon - Sat

Follow Us



About

About Us
Delivery Information
Privacy Policy
Terms & Conditions
Contact Us

CONTACT US

My Account

Sign In
View Cart
My Wishlist
Order Details
Help

MY ACCOUNT

Install App

From App Stores or Google Play



Secure Payment Gateways



Secure Payment Gateways



On the above figure can see the home page of the app. It includes the indispensable components of an online oxygen cylinder website like the function bar, the search bar and a list of products. Customers who visit the store can search for the products categories that they want to buy and they can customize the filter on the toolbar to be able to buy the best component for medical oxygen supply.

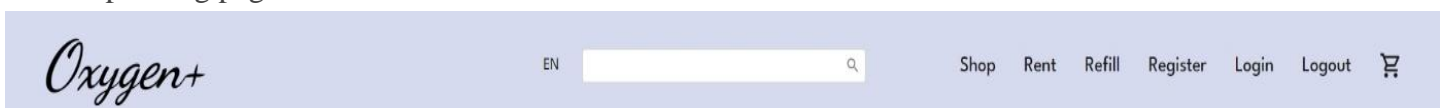
From navbar we can go to the respective destinations, here are the components of the home page,

```
const Home = () => {
  return (
    <div>
      <Navbar/>
      <Slider/>
      <Features/>
      <Categories/>
      <Newsletter/>
      <Footer/>
    </div>
  );
};
```

1.Navbar

A navigation bar is a link to appropriate sections/pages in a website that helps readers in traversing the online document. Considered a traditional method of navigation, a navigation bar can be implemented in a number of ways, namely, horizontally or vertically, or fixed or dynamic. A navigation bar implementation is considered one of the key points of Web design and usability.

In our navbar, first we have our beautiful logo of our site followed by a search bar and the links of our corresponding pages.



2.Slider and Website's features

Next what we have in our website is a slider followed by the 6 main features of what we are offering in the site as you can see on our home page screenshot in above page.

3.Product Category

In product category section we have displayed the main 4 types of products we are trying to offer. we have used a js shorting method to redirect users to the shopping page with products filtered with that category.

```
useEffect(() => {
  const getProducts = async () => {
    try {
      const res = await axios.get(
```

```

categories
?`http://localhost:5000/api/products?categories=${categories}`
: "http://localhost:5000/api/products"
);
setProducts(res.data);
} catch (err) {}
};
getProducts()
}, [categories]);

```

Register Page

Although some main pages of this application like Home page, Shop page do not require the user to sign up in order to enhance user experience, if users want to make a purchase, they have to sign up and log in to their account first. Below is a simple user interface to sign up new users.

```

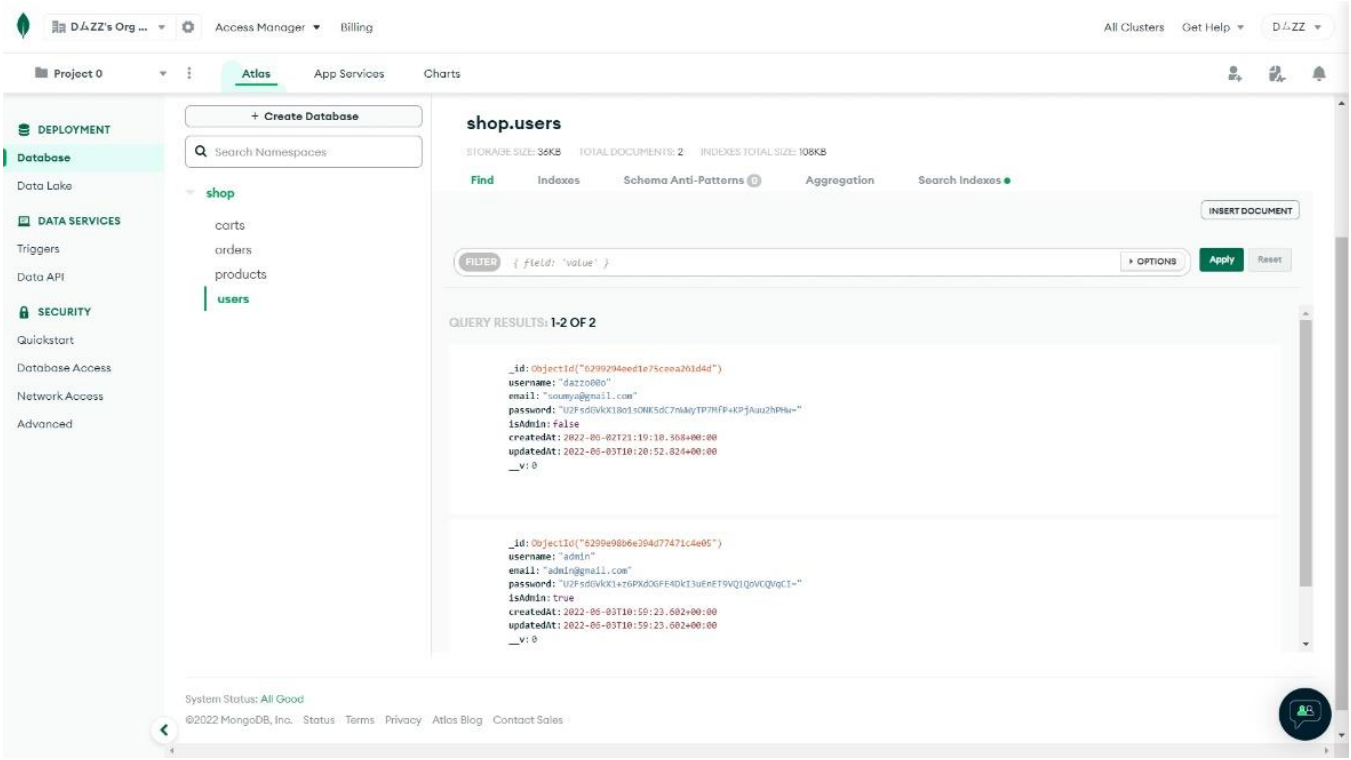
//REGISTER
router.post("/register", async (req, res) => {
  const newUser = new User({
    username: req.body.username,
    email: req.body.email,
    password: CryptoJS.AES.encrypt(
      req.body.password,
      process.env.PASS_SEC
    ).toString(),
  });

  try {
    const savedUser = await newUser.save();
    res.status(201).json(savedUser);
  } catch (err) {
    res.status(500).json(err);
  }
});

```

A web page with shared navigation bar and layout. Right below is the signup form which includes some inputs for name, email and password with a submit button. Bootstrap class 'form-group' is applied to facilitate these styling tasks. In this component, beside the state of each input field there are error and success state which determine if some alerts should be shown up when user fail or succeed to sign up.

Given the user put in correct data and press submit, fetch method is used to send a POST request with user data under json format in the body to the back-end API 'api/sign-up'. After validation process, a new instance of user model is created based on req. body and is saved into the database. The server then returns a json file containing user data without hashed password back to the client. Finally, a success alert is shown up and is redirecting the user to the sign in page.



we have used CryptoJs to encrypt password before saving it to the database for security enhancement.
above fig. shows the successful registration of an user.

Login Page



Logic and Data Flow

The sign in component is quite similar to the sign up one except the data flow when the user press submit button. after the server handled the POST request from the client, a token is signed and sent back to the client along with user information. Those data then are passed to authenticate function and stored in the local storage as jwt.

```
const Login = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const dispatch = useDispatch();
  const { isFetching, error } = useSelector((state) => state.user);

  const handleClick = (e) => {
    e.preventDefault();
    login(dispatch, { username, password });
  };

  return (
    <Container>
      <Navbar/>
      <Wrapper>
        <Page>
          <Box><Title>Log In</Title></Box>
          <Form>
            <Input placeholder="username" onChange={(e) => setUsername(e.target.value)} />
            <Input type="password" placeholder="Enter password" onChange={(e) => setPassword(e.target.value)} />
            <Button onClick={handleClick} disabled={isFetching}>LOGIN</Button>
            {error && <Error>Something went wrong...</Error>}
            <Link>DO NOT YOU REMEMBER THE PASSWORD?</Link>
            <Link>CREATE A NEW ACCOUNT</Link></Form>
          </Page>
        </Wrapper>
      </Container>
    )
  );
}
```

```

</Container>
)
}

```

we have used react-redux in the login page, When using Redux with React, states will no longer need to be lifted up. This makes it easier for us to trace which action causes any change.

As you can see i, the component does not need to provide any state or method for its children components to share data among themselves. Everything is handled by Redux. This greatly simplifies the app and makes it easier to maintain.

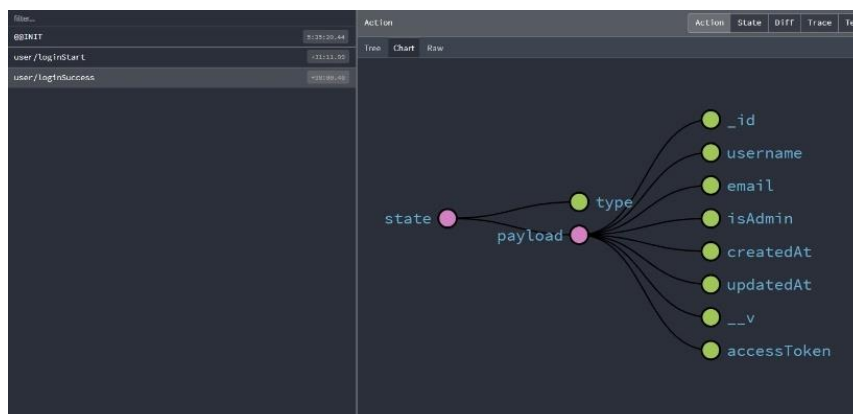
This is the primary reason why you should use Redux, but it's not the only benefit. Take a look at the list below for a summary of what you stand to gain by using Redux for state management.

```

reducers: {
  loginStart: (state) => {
    state.isFetching = true;
  },
  loginSuccess: (state, action) => {
    state.isFetching = false;
    state.currentUser = action.payload;
  },
  loginFailure: (state) => {
    state.isFetching = false;
    state.error = true;
  },
},
});

```

Reducers are handy for creating small pools of isolated data contexts. You can use it in conjunction with React Context API to keep your state where it belongs.



the fig. above shows a successful login and post login state.


we have also used `redux persist`, cause `Redux-persist` allows you to transform your store depending on the version you want for the app: you can control the version of your store. Migrations are applied on your state before replacing your store data in `REHYDRATE` step. This version 1 of migration will put my previous number of change where I want.


```
const rootReducer = combineReducers({ user: userReducer, cart: cartReducer });

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }),
});
```


SHOP PAGE


EN


Shop Rent Refill Register Login Logout 

#staysafe


Let's Fight Covid-19 Together




Filter Products : All
Sort Products : Newest




Brand Name
Oxygen Cylinder 5L
₹500




Brand Name
Oxygen Concentrator
₹28800




Brand Name
Oxygen Cylinder 10L
₹600




Brand Name
Oxygen Mask
₹200




Brand Name
Oxygen Cylinder 15L
₹700




Brand Name
Regulator+Flowmeter
₹150




Brand Name
Oxygen Cylinder 20L
₹800



Brand Name
Oxygen Cylinder 10L
₹650




Brand Name
Oxygen Cylinder 15L
₹750



Brand Name
Oxygen Concentrator
₹32600

Sign Up For Newsletter

Get timely updates on your product's **Availability**.



Contact






52 Abcd Road, Street 44, Siliguri

+91 9609912804

10:00 - 18:00, Mon - Sat

10:00 - 18:00, Mon - Sat

Follow Us

About

About Us

Delivery Information

Privacy Policy

Terms & Conditions

Contact Us

My Account

Sign In

View Cart



My Wishlist

Order Details





Help





Install App

From App Stores or Google Play

Secure Payment Gateways

The main shop page that customers can select the products that they want to add to cart. User can scroll down the mouse to browse all the products, or filter by categories or filter by price range like Figure.

the components we used in shop page: Navbar, Banner, products etc

Products

we have fetched products from our database and have them sorted.

Logic and Data Flow

fetching product from database

```
useEffect(() => {
  const getProducts = async () => {
    try {
      const res = await axios.get(
        categories
        ? `http://localhost:5000/api/products?categories=${categories}`
        : "http://localhost:5000/api/products"
      );
      setProducts(res.data);
    } catch (err) {}
  };
  getProducts()
}, [categories]);
return (
  <Container>
    {categories
      ? filteredProducts.map((item) => <Product item={item} key={item.id} />)
      : products

      .map((item) => <Product item={item} key={item.id} />)}
    </Container>
  )
}
```

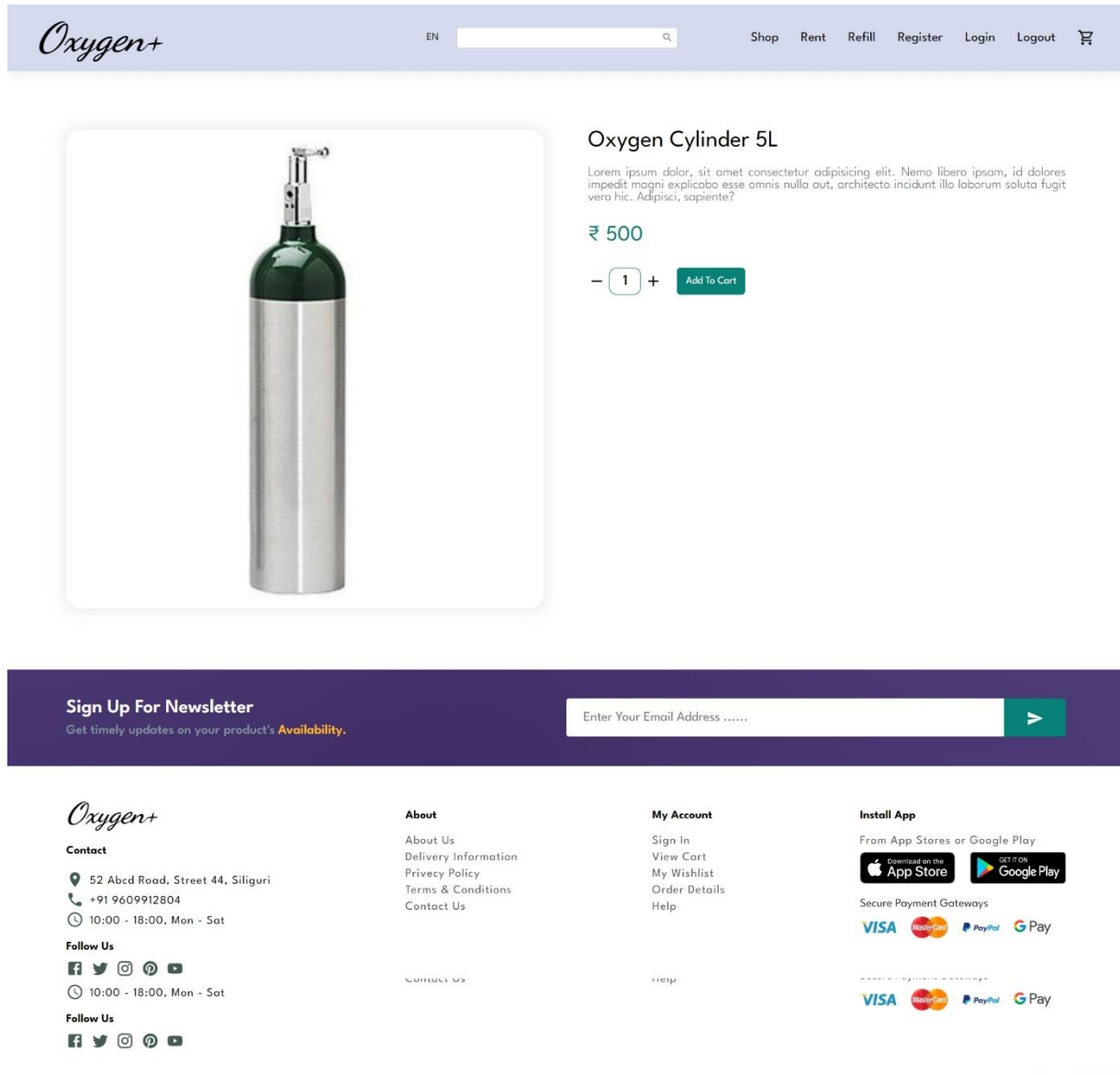
For user experience we have used a filter statement and a sort statement. In filter statement we have the 4 main product titles and we have used a shorting method to short products by Newest, Price asc and Price desc..

```
useEffect(() => {
  categories &&
  setFilteredProducts(
    products.filter((item) =>
      Object.entries(filters).every(([key, value]) =>
        item[key].includes(value)
      )
    )
  );
}, [products, categories, filters]);

useEffect(() => {
  if (sort === "newest") {
    setFilteredProducts((prev) =>
      [...prev].sort((a, b) => a.createdAt - b.createdAt)
    );
  }
}
```

```
    } else if (sort === "asc") {  
      setFilteredProducts((prev) =>  
        [...prev].sort((a, b) => a.price - b.price)  
      );  
    } else {  
      setFilteredProducts((prev) =>  
        [...prev].sort((a, b) => b.price - a.price)  
      );  
    }  
  }, [sort]);
```

Product Description Page



In Product description page, first we fetch product related data by product id, like name, disc, price etc.

```
useEffect(() => {
  const getProduct = async () => {
    try {
      const res = await publicRequest.get("/products/find/" + id);
      setProduct(res.data);
    } catch {}
  };
  getProduct();
}, [id]);
```

In this page the 2 main functions are quantity manager and add to cart option. For determining quantity, increasing and decreasing value use State method.

```
const handleQuantity = (type) => {
```

```
if (type === "dec") {  
  quantity > 1 && setQuantity(quantity - 1);  
} else {  
  setQuantity(quantity + 1);  
}  
};
```

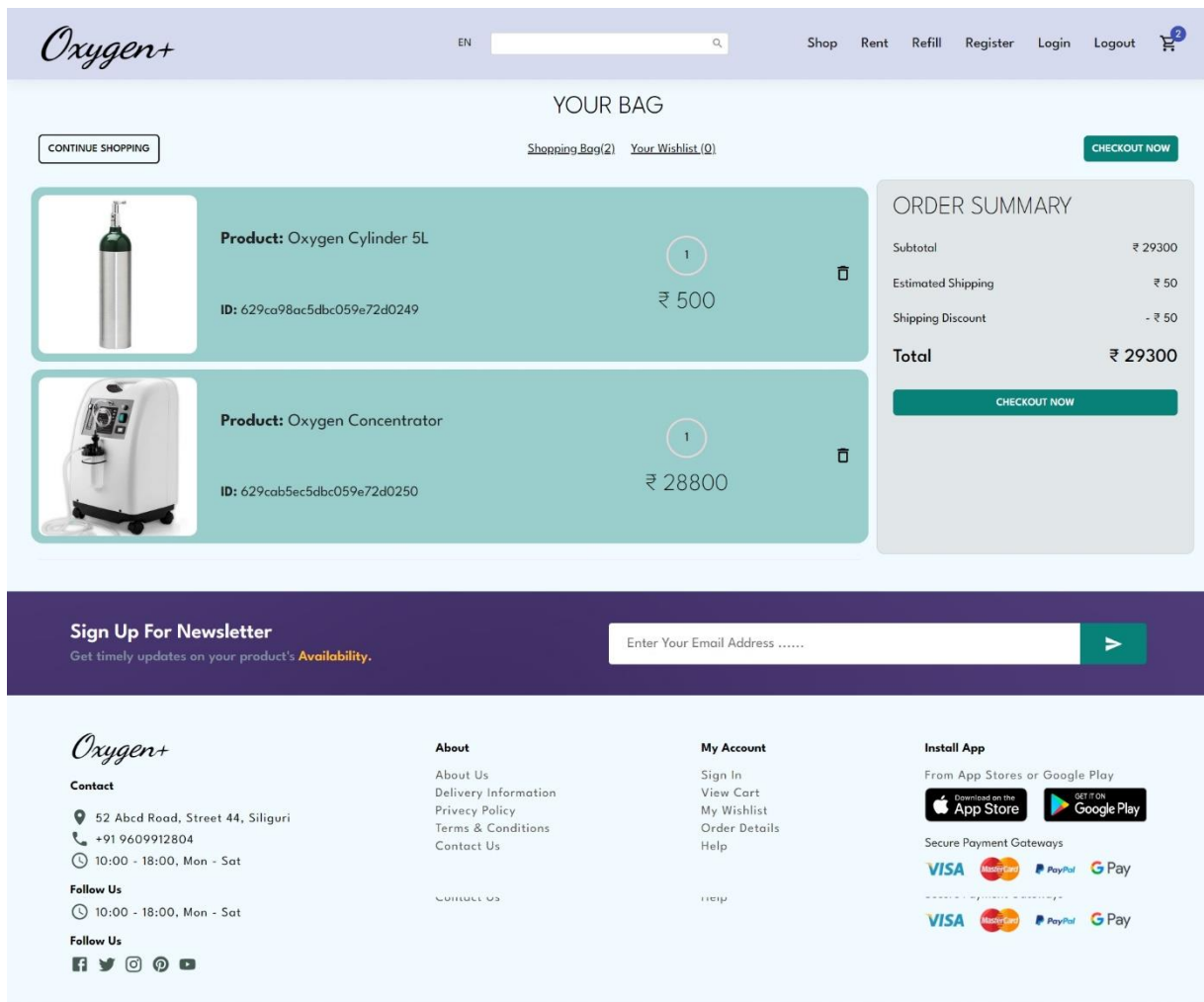
Finally, to add product to the user cart we used react-redux to manipulate data and used persist to contain data in store.

```
const handleClick = () => {  
  dispatch(  
    addProduct({ ...product, quantity })  
  );  
};
```

using persist to hold user data even after page refresh.

```
const rootReducer = combineReducers({ user: userReducer, cart: cartReducer });  
const persistedReducer = persistReducer(persistConfig, rootReducer);  
export const store = configureStore({  
  reducer: persistedReducer,  
  middleware: (getDefaultMiddleware) =>  
    getDefaultMiddleware({  
      serializableCheck: {  
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],  
      },  
    }),  
});
```

Cart page



Cart page is the last page users have to go through to get the purchase done. Beside some common shared components, there are a Card component which renders the product that users want to buy and a check-out component that demonstrates the total amount, delivery address and a form to fill in with payment information. This Card component inside Cart is slightly different than the one in Home or Shop component. It has a remove button to delete the product from the cart and an input field where users can increase or decrease the quantity of product in the end.

Logic and Data Flow

When a user clicks Add to cart button on a specific product, that product item will be added to the local storage. Then the user is redirected to this Cart page. Local storage is utilized to store cart data so that the data will not be lost every time the user refreshes the page. Cart data stays in local storage until it is removed or after the purchase is handled successfully.

```
useEffect(() => {
  const makeRequest = async () => {
    try {
```

```
const res = await userRequest.post("/checkout/payment", {
  tokenId: stripeToken.id,
  amount: cart.total * 100,
});
history.push("/success", {
  stripeData: res.data,
  products: cart, });
} catch {}
};
stripeToken && makeRequest();
}, [stripeToken, cart.total, history]);
```

To implement and test payment gateway, a test environment which is almost equivalent to production environment – Braintree sandbox is installed. Braintree supports credit card and PayPal, two of the most popular and emerging payment methods. Firstly, Braintree module is installed in the back-end server so that it can generate a token which is necessary to perform Braintree in the client as described in backend section.


Stripe payment process


Stripe collects necessary data to send back data to admin.


The screenshot shows a mobile app checkout interface for 'Oxygen+'. At the top, the app logo and name 'Oxygen+' are displayed, along with the text 'Your total is ₹29300'. Below this, there is a form with the following fields: an email address 'dazz@gmail.com', a checkbox for 'Same billing & shipping info', a name field with 'Dazz', an address field with 'asvdsa,sadfaefaf,aafa', a phone number field with '735111', a city field with 'Kolkata', and a country dropdown menu currently showing 'India'. At the bottom of the form is a blue button labeled 'Payment Info' with a plus icon.

```
<StripeCheckout
  name="Oxygen+"
>
  <Button>CHECKOUT NOW</Button>
image="https://drive.google.com/uc?export=view&id=1Svh3dE2NJ2JDI274W9eQ18rDXCoPs8fS"
  billingAddress
  shippingAddress
  description={`Your total is ₹${cart.total}`}
  amount={cart.total * 100}
  token={onToken}
  stripeKey={KEY}
```

RENT PAGE


EN


Shop Rent Refill Register Login Logout




Shop Name

Product Availability


Oxygen Cylinder 5L : 22 10L : 2 15L : 2 20L : 22	Oxygen Concentrator Available : 2	Oxygen Mask Available : 22	Regulator+Flowmeter Available : 2
--	--------------------------------------	-------------------------------	--------------------------------------



Shop Name

Product Availability

Oxygen Cylinder 5L : 22 10L : 2 15L : 2 20L : 22	Oxygen Concentrator Available : 2	Oxygen Mask Available : 22	Regulator+Flowmeter Available : 2
--	--------------------------------------	-------------------------------	--------------------------------------



Shop Name

Product Availability


Oxygen Cylinder 5L : 22 10L : 2 15L : 2 20L : 22	Oxygen Concentrator Available : 2	Oxygen Mask Available : 22	Regulator+Flowmeter Available : 2
--	--------------------------------------	-------------------------------	--------------------------------------

Sign Up For Newsletter

Get timely updates on your product's **Availability**.

Enter Your Email Address






>



Contact

52 Abcd Road, Street 44, Siliguri
+91 9609912804
10:00 - 18:00, Mon - Sat

Follow Us



About

About Us
Delivery Information
Privacy Policy
Terms & Conditions
Contact Us





My Account

Sign In
View Cart
My Wishlist
Order Details
Help

Install App

From App Stores or Google Play



Secure Payment Gateways

localhost:3000/VendorDtls

Rent page shows the no. of available shop vendors to rent the oxygen cylinder to their customer with contact details, number of product category available, and prices.

Logic and Data Flow

```

<Container><Link style={{ textDecoration:'none',color:'#222'}}to={'`/VendorDtls`'}>
  <Wrapper>
    <Img><Image src = {item.img}/></Img>

    <Data>
      <Name><b>{item.name}</b></Name>
      <Title>Product Availability</Title>
      <Product>
        <Product1><Tag>Oxygen Cylinder</Tag>
        <Capacity>

```


```


        <Liters><b>5L : </b>{item.L5}</Liters>
        <Liters><b>10L : </b>{item.L10}</Liters>
        <Liters><b>15L : </b>{item.L15}</Liters>
        <Liters><b>20L : </b>{item.L20}</Liters>
    </Capacity>
</Product1>
<Product2><Tag>Oxygen Concentrator</Tag>
    <Available> <b>Available : </b>{item.OC} </Available></Product2>
<Product3><Tag>Oxygen Mask</Tag>
    <Available> <b>Available : </b>{item.OM} </Available></Product3>
<Product4><Tag>Regulator+Flowmeter</Tag>
    <Available> <b>Available : </b>{item.RF} </Available></Product4>
</Product>
</Data>
</Wrapper></Link>
</Container>


```

Rent page shows the no. of available shop vendors to rent the oxygen cylinder to their customer with contact details, number of product category available, and prices. These all data are taken in the above code snippet.

Rent Provider's description



EN

Shop Rent Refill Register Login Logout





#staysafe

Let's Fight Covid-19 Together




Oxygen Cylinder

Capacity	Available	Rent	Refill
5 L	22	₹100/day	₹500/refill
10 L	2	₹130/day	₹600/refill
15 L	2	₹170/day	₹700/refill
20 L	22	₹200/day	₹800/refill




Oxygen Concentrator

Available	Rent
2	₹200/day




Oxygen Mask

Available	Rent
22	₹50/day

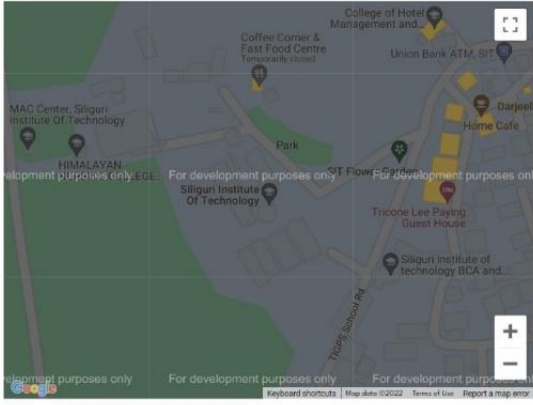


Regulator+Flowmeter

Available	Rent
2	₹20/day




Shop Name	Shop Enterprise
Owner	Owner Roy
Phone No.	+91 9609912805
Email	contact@example.com
Hours	10:00 - 20:00, Mon - Sat



Sign Up For Newsletter


Get timely updates on your product's **Availability**.



Contact

52 Abcd Road, Street 44, Siliguri
 +91 9609912804
 10:00 - 18:00, Mon - Sat
 10:00 - 10:00, Mon - Sat

Follow Us



About



[About Us](#)
[Delivery Information](#)
[Privacy Policy](#)
[Terms & Conditions](#)
[Contact Us](#)

My Account





[Sign In](#)
[View Cart](#)
[My Wishlist](#)
[Order Details](#)
[Help](#)

Install App

From App Stores or Google Play

Secure Payment Gateways

The rental provider page shows the details of products by their category level, with the number of quantity available for purchase and their price level. And the google map provided in the above UI page shows number of shops available in particular location.

Logic and Data Flow

```

<Products>
  <Img><Image src = {item.img1}/></Img>
  <Data>
    <Name><b>Oxygen Cylinder</b></Name>
    <Product1>
      <Capacity>
        <Liters><b>Capacity</b></Liters><Liters><b>Available</b></Liters>
        <Liters><b>Rent</b></Liters><Liters><b>Refill</b></Liters>
      </Capacity>
      <Capacity>
        <Liters>5 L</Liters><Liters>{item.L5}</Liters>
        <Liters>₹{item.L5P}/day</Liters><Liters>₹{item.L5R}/refill</Liters>
      </Capacity>
      <Capacity>
        <Liters>10 L</Liters><Liters>{item.L10}</Liters>
        <Liters>₹{item.L10P}/day</Liters><Liters>₹{item.L10R}/refill</Liters>
      </Capacity>
      <Capacity>
        <Liters>15 L</Liters><Liters>{item.L15}</Liters>
        <Liters>₹{item.L15P}/day</Liters><Liters>₹{item.L15R}/refill</Liters>
      </Capacity>
      <Capacity>
        <Liters>20 L</Liters><Liters>{item.L20}</Liters>
        <Liters>₹{item.L20P}/day</Liters><Liters>₹{item.L20R}/refill</Liters>
      </Capacity>
    </Product1>
  </Data>
</Products>

```

Google map location

Map function will take data from input and locate the owners through api call send to google map.

```

<Map
  google={this.props.google}
  style={{ width:"667px", height:"500px" }}
  initialCenter={{
    lat: 26.769374635466658,
    lng: 88.37619704991646
  }}
  zoom={18}
  onClick={this.onMapClicked}
/></Contain

```

we have used google map to locate shopper availability to a specific area.

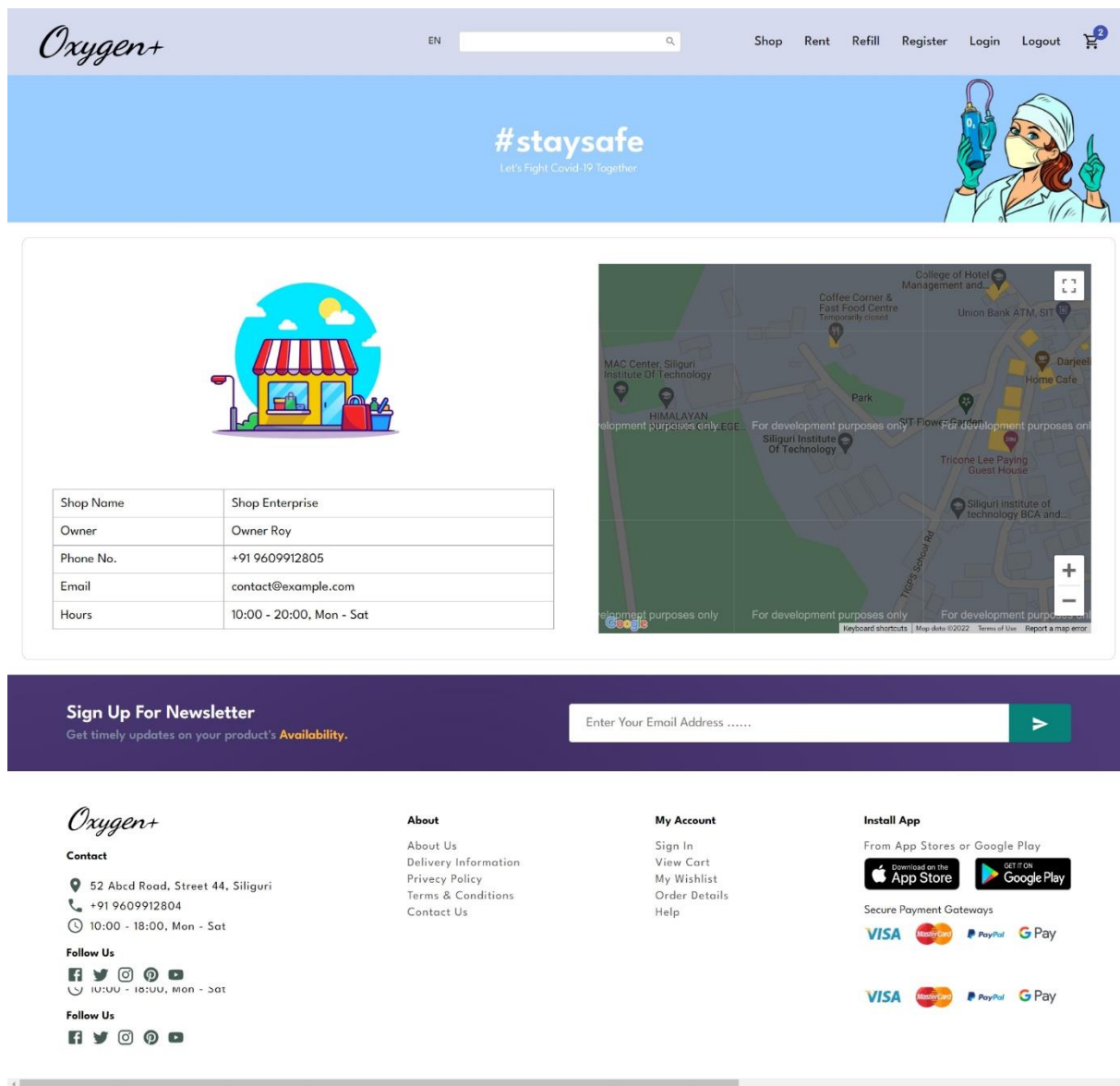
```

<StoreDtls>
  <Img><Image src = {item.img}/></Img>
  <Table>

```

```
<TableRow>
  <TableData>Shop Name</TableData>
  <TableData>{ item.name }</TableData>
</TableRow>
<TableRow>
  <TableData>Owner</TableData>
  <TableData>{ item.owner }</TableData>
</TableRow>
<TableRow>
  <TableData>Phone No.</TableData>
  <TableData>{ item.phone }</TableData>
</TableRow>
<TableRow>
  <TableData>Email</TableData>
  <TableData>{ item.email }</TableData>
</TableRow>
<TableRow>
  <TableData>Hours</TableData>
  <TableData>{ item.hours }</TableData>
</TableRow>
</Table>
```

Refill Centre Details



The above fig shows Refill center UI, where consumer can see shop name, product availability, Owners name, Contact detail, and Service available timing. We have also placed google map to help customers to locate the nearest oxygen refilling center with google map UI.

```
<StoreDtls>
<Img><Image src = {item.img}/></Img>
<Table>
  <TableRow>
    <TableData>Shop Name</TableData>
    <TableData>{item.name}</TableData>
  </TableRow>
  <TableRow>
    <TableData>Owner</TableData>
    <TableData>{item.owner}</TableData>
  </TableRow>
  <TableRow>
```

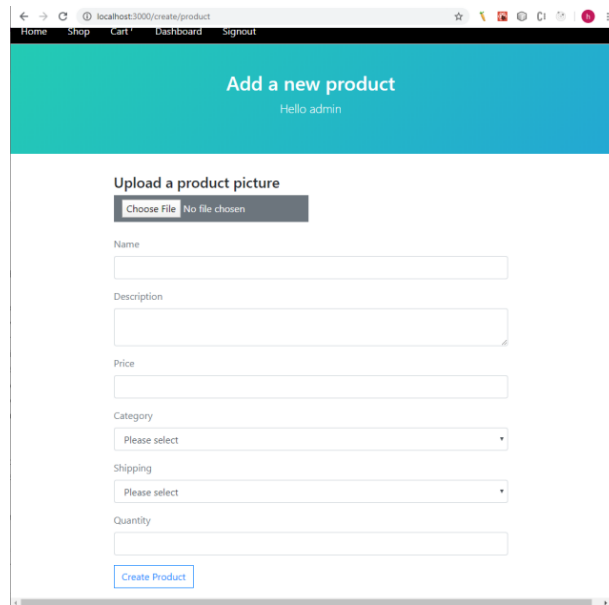
```
<TableData>Phone No.</TableData>
<TableData>{item.phone}</TableData>
</TableRow>
<TableRow>
  <TableData>Email</TableData>
  <TableData>{item.email}</TableData>
</TableRow>
<TableRow>
  <TableData>Hours</TableData>
  <TableData>{item.hours}</TableData>
</TableRow>
</Table>
```

Data has been taken from customer provided in the above UI and the above code snipt will allocate in the information.

DISCUSSION

This section illustrates the evaluations of the e-commerce web application. There is also an in-depth discussion about possible improvements that can be done in the future.

Application Evaluation

A screenshot of a web browser showing a form titled "Add a new product" with the subtitle "Hello admin". The form is for uploading a product picture and includes fields for Name, Description, Price, Category (a dropdown menu), Shipping (a dropdown menu), and Quantity. A "Choose File" button is next to the "Upload a product picture" label. A "Create Product" button is at the bottom of the form. The browser's address bar shows "localhost:3000/create/product".

In the end, a prototype version of the e-commerce application simulating an online bookstore was developed successfully with the aid of 4 main technologies in the MERN stack along with numerous other Node modules. This application is easy to use and user-friendly as it takes only a couple of clicks from the home page to get the purchase done. Users can sign in, edit their profiles, check purchase history and the most important feature – to search for their favorite book and pay for it. This application offers 2 type of product filtering, the first one is based on a search engine which takes user input and the other one is dependent on category and price of the products. Admin users have all the same functionalities, added some more features such as creating product and managing orders. Payment gateways also works well as it is already tested in stripe account.

Interface of Add-product component

Future Scope

There is a scope for further development in our project to a great extent. Several features can be added to this system in future like expanding to various other locations with time, more authenticated and secured transfer of money. We are also looking forward to providing a “Return Empty Cylinder” option, so that the user can return the empty oxygen cylinder after using the oxygen in it in order to reuse the cylinder by refilling the oxygen gas.

CONCLUSION

The project entitled Online Oxygen Management System was completed successfully. The system has been developed with much care and free of errors and at the same time it is efficient and less time consuming. The purpose of this project was to develop a web application to provide the oxygen products to the needy in a hassle-free manner, round the clock. This project has helped us gain valuable information and practical knowledge on several topics like designing web pages using html & CSS, usage of responsive templates, and management of database using MongoDB. The entire system is secured. Also, the project has helped us understanding the development phases of a project and software development life cycle. We have learned how to test different features of a project. This project has given us a sense of satisfaction in designing an application that can be of use to all the users and the suppliers to provide the user with the desired oxygen product, anytime, anywhere, by simple modifications. However, all the challenges helped us learn, explore and implement the new technologies on a real-world scenario. Looking forward to enhancing the features for even more smooth user experience.

REFERENCE

- 1 Hyperion Development (2018). Everything you need to know about the MERN stack. Available at: <https://blog.hyperiondev.com/index.php/2018/09/10/everything-need-know-mern-stack/> (Accessed 12 April 2020)
- 2 Node.js Documentation. The V8 JavaScript Engine. Available at: <https://nodejs.dev/learn/the-v8-javascript-engine> (accessed 12 April 2020)
- 3 Priyesh Patel, Freecodecamp (2018). What exactly is Node.js? Available at: <https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/> (accessed 12 April 2020)
- 4 w3school. Node.js Introduction. Available at: https://www.w3schools.com/nodejs/nodejs_intro.asp (accessed 12 April 2020)
- 5 TutorialsTeacher. Node.js Module. Available at: <https://www.tutorialsteacher.com/nodejs/nodejs-modules> (accessed 12 April 2020)
- 6 Tutorialspoint. Node.js - NPM. Available at: https://www.tutorialspoint.com/nodejs/nodejs_npm.htm (accessed 12 April 2020)
- 7 Node.js Documentation. An introduction to the npm package manager. Available at: <https://nodejs.dev/learn/an-introduction-to-the-npm-package-manager> (accessed 12 April 2020)
- 8 TutorialsTeacher. Node Package Manager. Available at: <https://www.tutorialsteacher.com/nodejs/what-is-node-package-manager> (accessed 12 April 2020)
- 9 Node.js Documentation. Update all the Node.js dependencies to their latest version. Available at: <https://nodejs.dev/learn/update-all-the-nodejs-dependencies> to-their-latest-version (accessed 13 April 2020)
- 10 Node.js Documentation. The Node.js Event Loop. Available at: <https://nodejs.dev/the-nodejs-event-loop> (accessed 13 April 2020)
- 11 Impressico Business Solutions. Advantages of using Express.js. Available at: <https://www.impressico.com/2015/10/06/advantages-of-using-express-js/> (accessed 13 April 2020)
- 12 Mozilla. Express/Node introduction. Available at: https://developer.mozilla.org/enUS/docs/Learn/Server-side/Express_Nodejs/Introduction (accessed 13 April 2020)
- 13 Express Documentation. Routing. Available at: <https://expressjs.com/en/guide/routing.html> (accessed 13 April 2020)
- 14 Express Documentation. Using middleware. Available at: <https://expressjs.com/en/guide/using-middleware.html> (accessed 13 April 2020)
- 15 Derick Bailey (2016) In what order does my Express.js middleware execute? Available at: <https://derickbailey.com/2016/05/09/in-what-order-does-my-expressjs-middleware-execute/> (accessed 13 April 2020)
- 16 Decode Web, Medium (2019) What is MongoDB? Available at: <https://medium.com/@decodeweb/what-is-mongodb-7693e2f2f4f6> (accessed 14 April 2020)

- 17 DB-engines (2020) Complete Ranking. Available at: <https://db-engines.com/en/ranking> (accessed 14 April 2020)
- 18 MongoDB inc. NoSQL Databases Explained. Available at: <https://www.mongodb.com/nosql-explained> (accessed 14 April 2020)
- 19 MongoDB inc. Data Modeling Introduction. Available at: <https://docs.mongodb.com/manual/core/data-modeling-introduction/> (accessed 14 April 2020)
- 20 MongoDB inc. Data Model Design. Available at: <https://docs.mongodb.com/manual/core/data-model-design/> (accessed 14 April 2020)
- 21 javaTpoint. MongoDB Atlas. Available at: <https://www.javatpoint.com/mongodbatlas> (accessed 14 April 2020)
- 22 Nick Parsons, Medium (2017). MongoDB Atlas – Technical Overview & Benefits. Available at: <https://medium.com/@nparsons08/mongodb-atlas-technical-overview-benefits-9e4cff27a75e> (accessed 15 April 2020)
- 23 Nick Karnik, freecodecamp (2018) Introduction to Mongoose for MongoDB. Available at: <https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/> (accessed 15 April 2020)
- 24 Jamie Munro, envatotuts (2017) An Introduction to Mongoose for MongoDB and Node.js. Available at: <https://code.tutsplus.com/articles/an-introduction-to-mongoose-for-mongodb-and-nodejs--cms-29527> (accessed 15 April 2020)