1. Course Description

Course / Project Title: **Build a server with an interactive web application.**

Introduction to the course:	Part- A Build a Linux server. • This course takes you through the process of assembling a PC, provides step-by-step guidance on how to assemble a server with the components such as the motherboard, processor, and RAM that make up the system unit, as well as how to install the operating system to complete a fully functioning computer, install necessary software for developing web application. So, if you've ever wondered what it takes to build your own PC, join the course for this adventure. Part- B Build an interactive web application. • This course will also help you to build an interactive web application using React as frontend, Spring boot as backend and MariaDB database. Front-end side is made with React, React Router, Axios. The back-end server uses Spring Boot with Spring Web MVC for REST APIs and Spring Data JPA for interacting with MariaDB database.
What does this course aim to achieve?	Part- A Build a Linux server. • The objective of this course is to provide knowledge about assembling a PC and setting up web server. Part- B Build an interactive web application. • You'll learn the major components of web application architectures, build a fully functional full-stack web application.
What is being built in this course:	Part- A Build a Linux server. • Build a computer from grounds up, and then install the necessary operating system and packages for web application. Part- B Build an interactive web application. • Build a responsive Restaurant table reservation web application with following features: ○ Customer Login & Registration ○ Book a Slot ○ Admin Login

	○ View Bookings ○ Clear Bookings ○ Check Booking Availability
How is it being tested:	Verify the installed java version. Verify with the default resin web page. Verify the installed node and npm version. View the MariaDB test database. Test the web application on the local environment
Course Prerequisites	Knowledge of some JavaScript programming basics. Knowledge of some Java programming basics. Knowledge of some SQL.

2. Component requirements

Prerequisites Components needed	1. Windows PC or Laptop 2. Pendrive (USB) - Minimum Requirement 8 GB

S.no	Components	Quantity	Usage (one time or reusable?)	Cost
1	Motherboard 310 MH	1 No	reusable	5,550/-
2	Intel I3 8th Gen CPU	1 No	reusable	5,050/-
3	8GB DDR RAM	1 No	reusable	1,800/-
4	1 TB HDD Drive	1 No	reusable	3,050/-
5	ZEBRONICS 18.5” Monitor	1 No	reusable	4499/-
6	DELL Mouse	1 No	reusable	250/-
7	DELL Keyboard	1 No	reusable	450/-
8	450W Power supply SMPS	1 No	reusable	650/-
9	Atx Cabinet with SMPS 450W Zeb	1 No	reusable	1,850/-
			Total	23,149/-

Software needed	Download Links
1. Java SE 8	https://www.oracle.com/in/java/technologies/javase/javase8-archive-downloads.html

2. Apache NetBeans	https://netbeans.apache.org/download/index.html
3. Visual Studio Code	https://code.visualstudio.com/
4. Resin	https://www.caucho.com/resin-4.0/admin/starting-resin.xtp
5. Moba Xterm	https://mobaxterm.mobatek.net/download.html

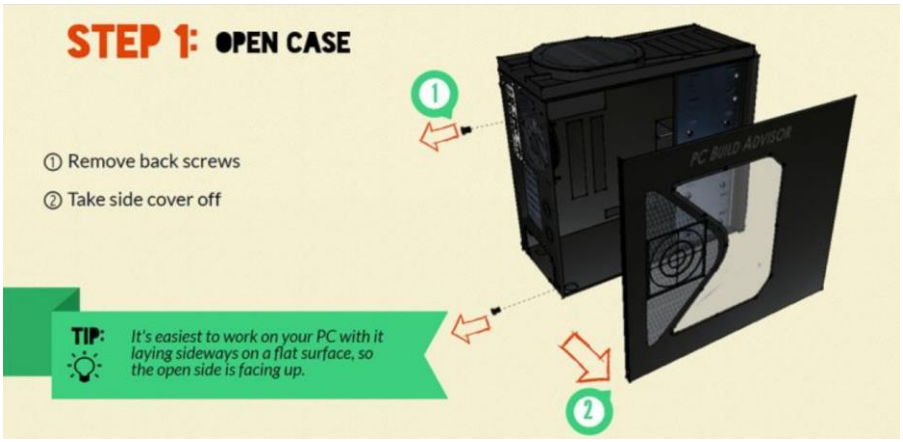
Space required for the Build	__3__ Sq.ft

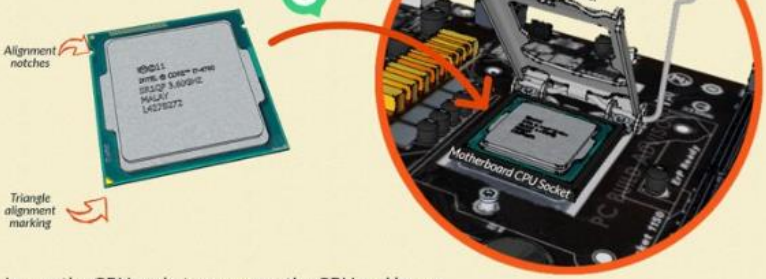
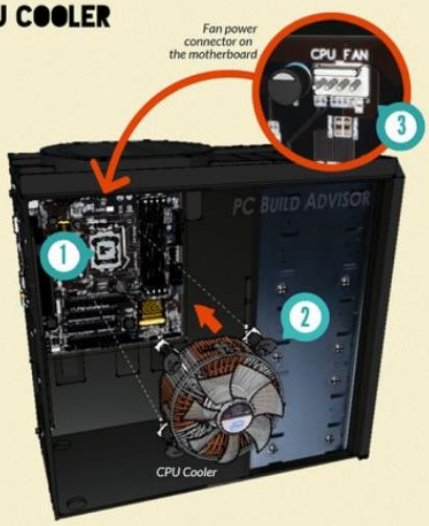
3. Pre-requisites

List of additional resources	Links
ReactJS Axios GET, POST, PUT and DELETE Tutorial	https://www.javaguides.net/2020/08/reactjs-axios-get-post-put-and-delete-example-tutorial.html
Spring Boot Tutorial	https://www.tutorialspoint.com/spring_boot/spring_boot_tutorial.pdf
MariaDB Basic Tutorial	https://linuxhint.com/mariadb-tutorial/

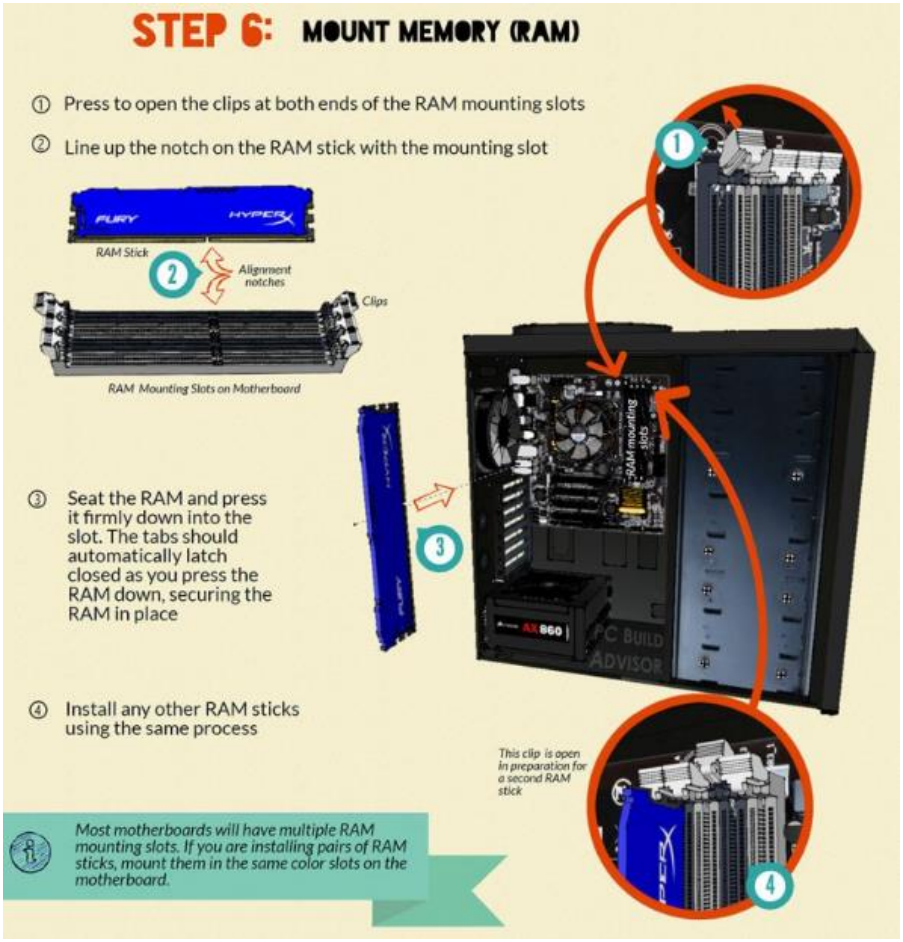
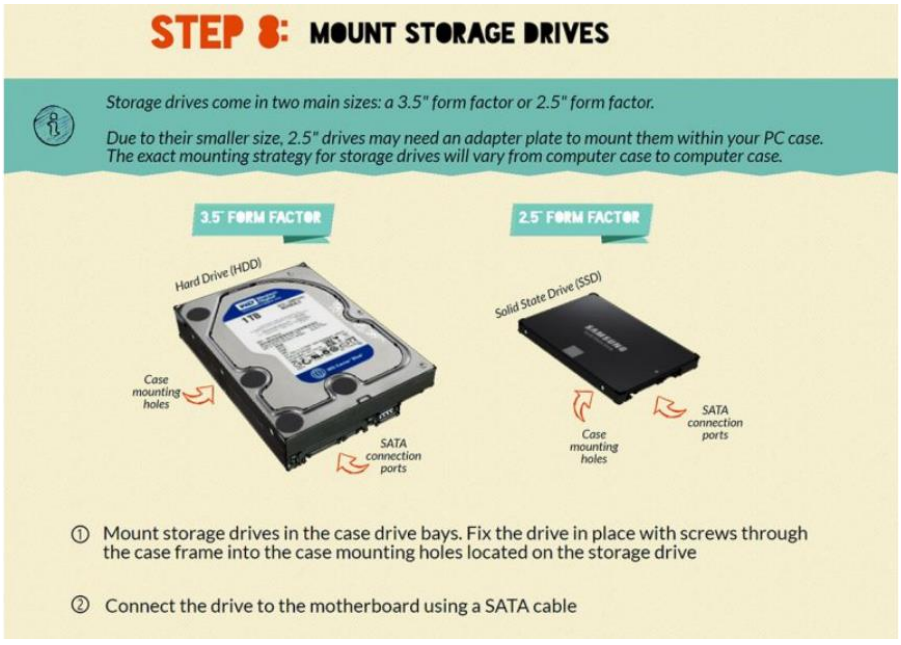
S.no	Additional Tasks/ Assignment
1.	Allocating Restaurant Table & Time slots Multiple restaurants listing Multiple restaurants management

4. Step-by-step Procedure to Build & Test

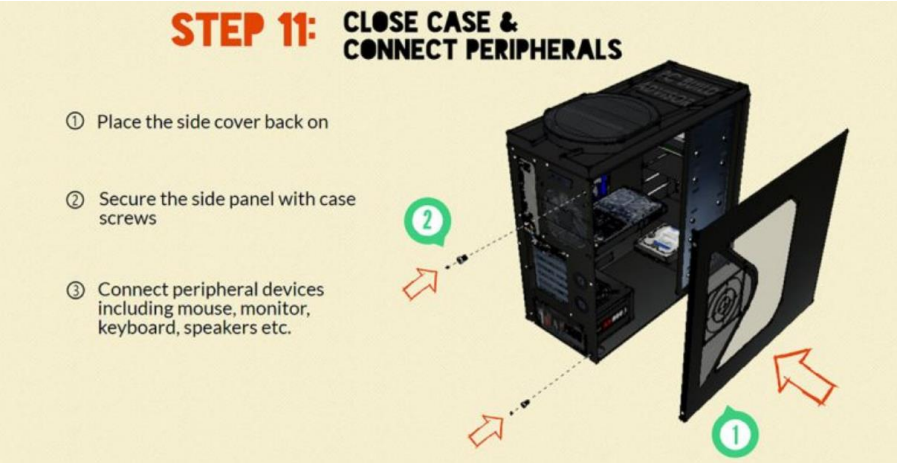
S.no	Step-by-Step Procedure	Time taken
	Part- A Build a Linux server	
	Assemble the PC	4 hours
1.	 	

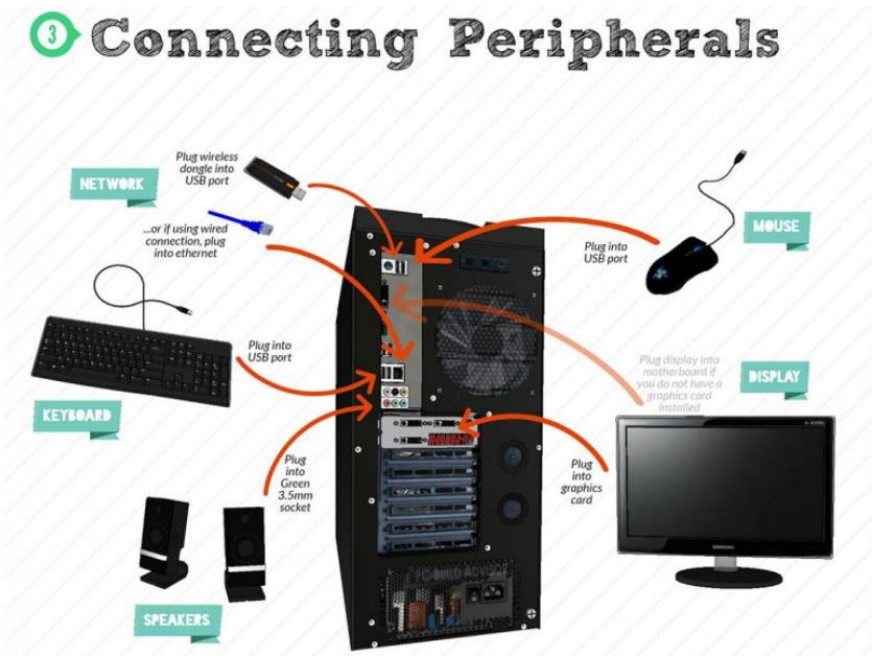
2.	**STEP 3: MOUNT PROCESSOR (CPU)**  ① Locate the CPU socket holder on the motherboard ② Lift up the latch lever to release and hinge open the CPU socket cover ③ Holding the CPU by its sides, line up any alignment notches or the triangle on the corner of the CPU to the triangle marked on the motherboard to ensure the correct orientation Gently place it straight down into the motherboard socket to seat the CPU  ④ Lower the CPU socket cover over the CPU and lower the latch lever closed again to secure the CPU socket holder closed **TIP:** *Don't apply force to seat the CPU. Avoid touching or pressing down on the back of the CPU with your fingers, as any residue from your hands can destroy the heat transfer surface for the cooler which will be mounted next.*	
3.	**STEP 4: INSTALL CPU COOLER** ① If required*, apply thermal paste to the back of CPU  ② Seat CPU heatsink/cooler and fix in position ③ Plug the power cable attached to the cooler fan into the motherboard connector  * Some CPU coolers do come with a thermal pad already applied, in which case you can skip step ①. If yours doesn't, you will need to apply thermal paste to the CPU surface before seating the CPU cooler in position.	

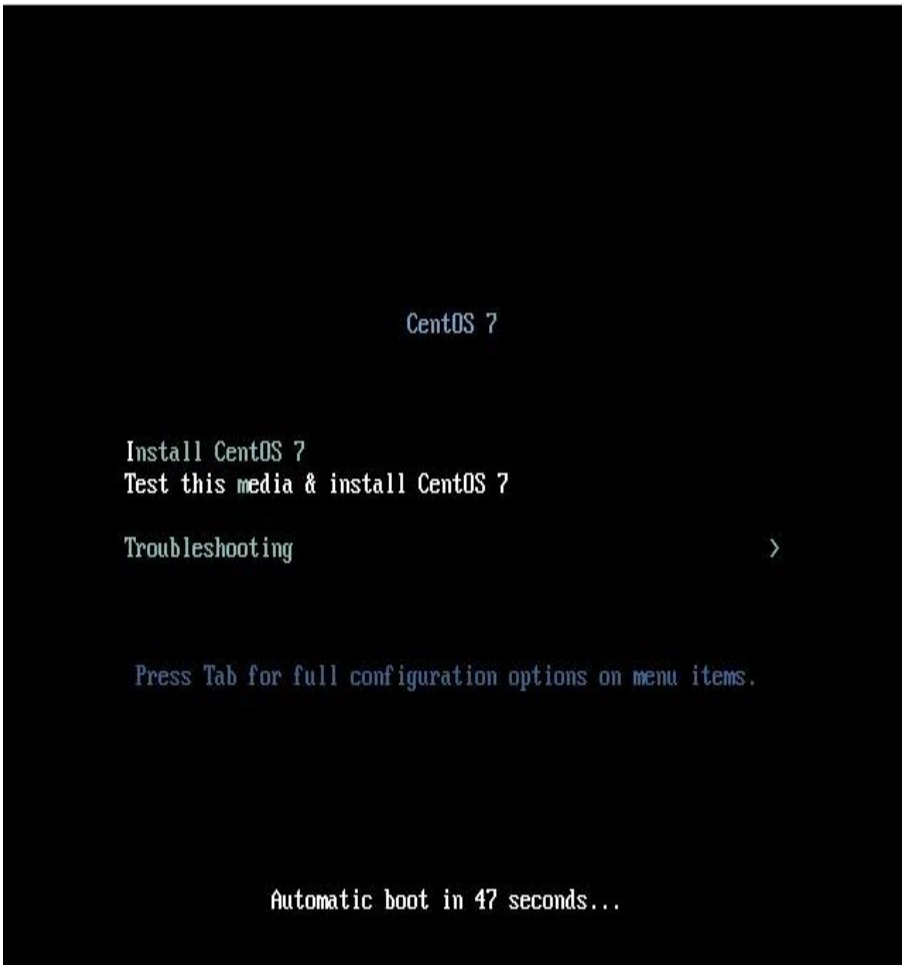
4.	 STEP 5: INSTALL POWER SUPPLY (PSU) ① Mount the power supply and fasten with screws to the case mounting points ② Plug the largest cabling connector from the power supply cabling into the motherboard power connector ③ Plug the 8-pin cabling connector from the power supply cabling into the CPU power connector CPU power connector Motherboard power connector 8-pin connector from power supply cabling PSU mounting screws Power supply Largest connector from power supply cabling PC BUILD ADVISOR AX860 1 2 3	

5.	 STEP 6: MOUNT MEMORY (RAM) ① Press to open the clips at both ends of the RAM mounting slots ② Line up the notch on the RAM stick with the mounting slot RAM Stick Alignment notches Clips RAM Mounting Slots on Motherboard ③ Seat the RAM and press it firmly down into the slot. The tabs should automatically latch closed as you press the RAM down, securing the RAM in place ④ Install any other RAM sticks using the same process This clip is open in preparation for a second RAM stick Most motherboards will have multiple RAM mounting slots. If you are installing pairs of RAM sticks, mount them in the same color slots on the motherboard.	
6.	 STEP 8: MOUNT STORAGE DRIVES Storage drives come in two main sizes: a 3.5" form factor or 2.5" form factor. Due to their smaller size, 2.5" drives may need an adapter plate to mount them within your PC case. The exact mounting strategy for storage drives will vary from computer case to computer case. 3.5" FORM FACTOR 2.5" FORM FACTOR Hard Drive (HDD) Case mounting holes SATA connection ports Solid State Drive (SSD) SATA connection ports Case mounting holes ① Mount storage drives in the case drive bays. Fix the drive in place with screws through the case frame into the case mounting holes located on the storage drive ② Connect the drive to the motherboard using a SATA cable	

	 SATA Ports on Motherboard PC BUILD ADVISOR Drive Bays AX860 1 2 SATA Cable SATA Connector from Power Supply Cabling 3 Hard Drive SATA Ports 2 ③ Plug in power cabling to the storage drive ④ Mount any other storage drives in the same way	
7.	 STEP 10: CONNECT CASE FANS & FRONT PANEL CONNECTORS Some computer cases come with case fans already installed/mounted within the case. In other cases you might need to mount your own case fans, or you may even choose to run your computer without any case fans at all. ① Mount any case fans within your case as required using the supplied screws or clips ② Connect any case fan power connectors to the multiple fan headers located at various places on the motherboard 1 Computer case fan Fan mounting holes Fan power connector	

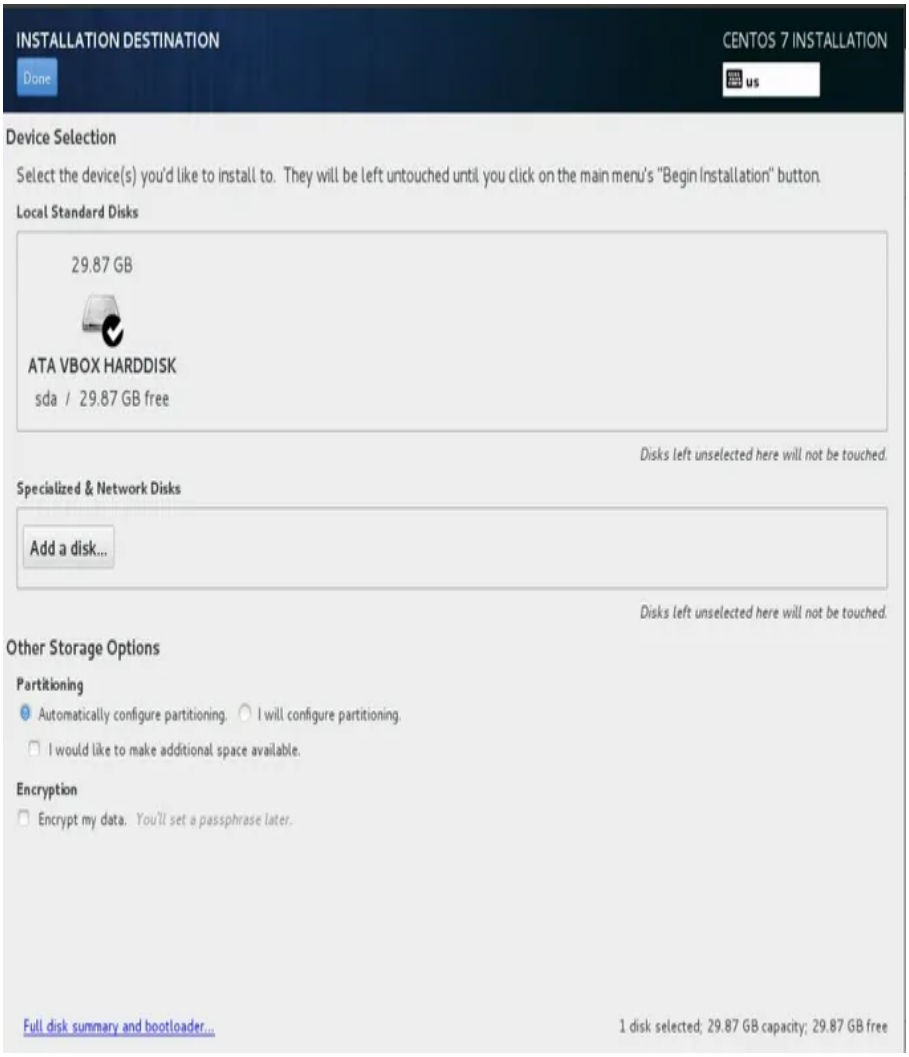
	 ③ Identify the cabling from the front panel ports of your PC* These front panel connectors will need to be plugged into the motherboard so that buttons and inputs/outputs (I/O) on your case front panel will work Fan Header on Motherboard SYS_FAN2 Front Panel Ports Front Panel The front panel ports will be connected to loose cabling inside the case Case fan power connector F_AUDIO SPDIF Front Audio Header on Motherboard USB Case Fan Front Panel I/O Connection Headers on Motherboard Microphone/audio connector from case front panel USB connector from case front panel Case fan power connector Case front panel connectors ④ Connect any front panel audio connectors to the motherboard front audio header ⑤ Connect any front panel USB connectors to the motherboard USB headers ⑥ Connect the front panel case connectors to the motherboard front panel I/O headers * Different computer cases may have slightly different I/O connections, but generally both the connectors and motherboard headers are labelled, so use these to your advantage when working out where to plug each cabling connector!	
8.	 STEP 11: CLOSE CASE & CONNECT PERIPHERALS ① Place the side cover back on ② Secure the side panel with case screws ③ Connect peripheral devices including mouse, monitor, keyboard, speakers etc.	

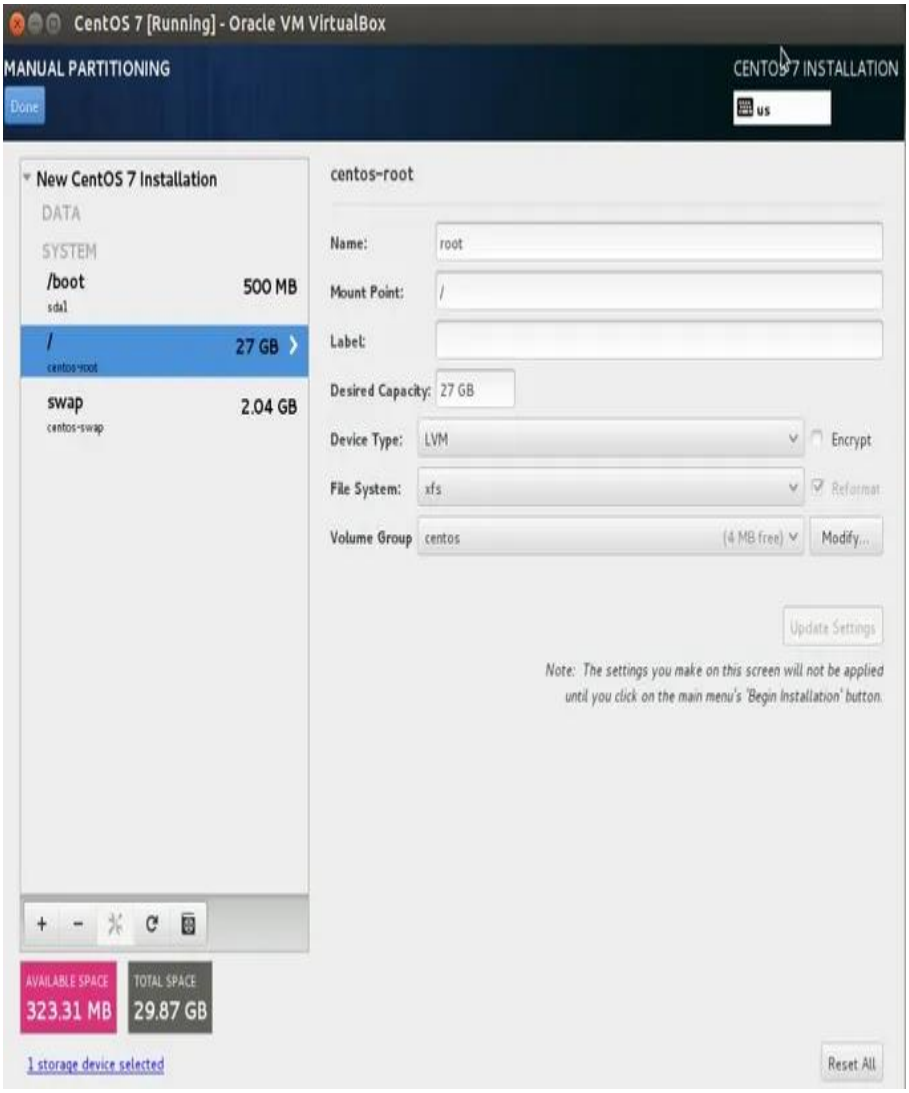
		
	Creating Bootable CentOS USB Stick	1 hour
9.	Note: Download CentOS in a windows PC and follow the process Download CentOS 7 IOS file from here.	
10.	Note: Install PowerISO in a windows PC and follow the process Download PowerISO v8.1(64- bit) from here.	
11.	Flash CentOS ISO file to the USB Stick 1. Start PowerISO. 2. Insert the USB drive you intend to make bootable. 3. Choose the menu "Tools > Create Bootable USB Drive...".	

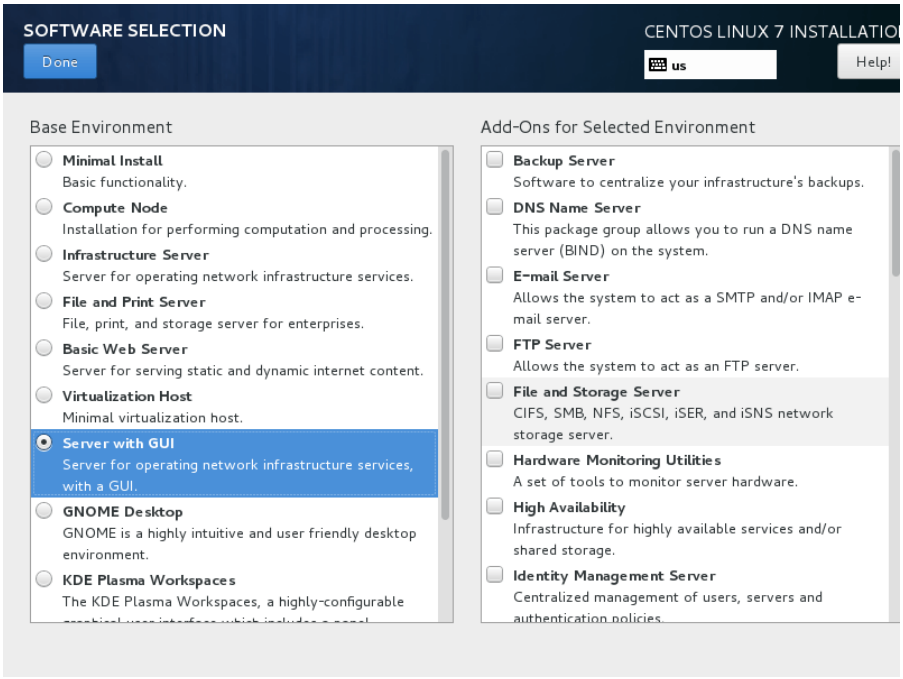
	4. In "Create bootable USB Drive" dialog, click "Browse" button to open the iso file for Linux. 5. Select the USB drive from the "Destination USB drive" list. 6. Choose the proper writing method. "Raw-write" is recommended. 7. Click "Start" button to start creating bootable USB drive for Linux.	
	Installing CentOS	1 hour
12.	Boot the USB, Select Install CentOS 7 from the boot menu. 	
13.	Select the language and continue.	

		
14.	Set Date and Time/ Time zone.	

		
15.	Select the Installation source. • you can specify locally available installation media.	

		
16.	Choose Installation Destination • Select the I will configure partitioning checkbox and choose Done. • If you do not have enough free space, you can reclaim disk space and instruct the system to delete files. • Total space – 250 GiB ○ /Boot part - 2048 MiB size ○ Swap - 16384 MiB ○ / - remaining available space	

		
17.	Software packages selection. • Select the **server with GUI** option.	



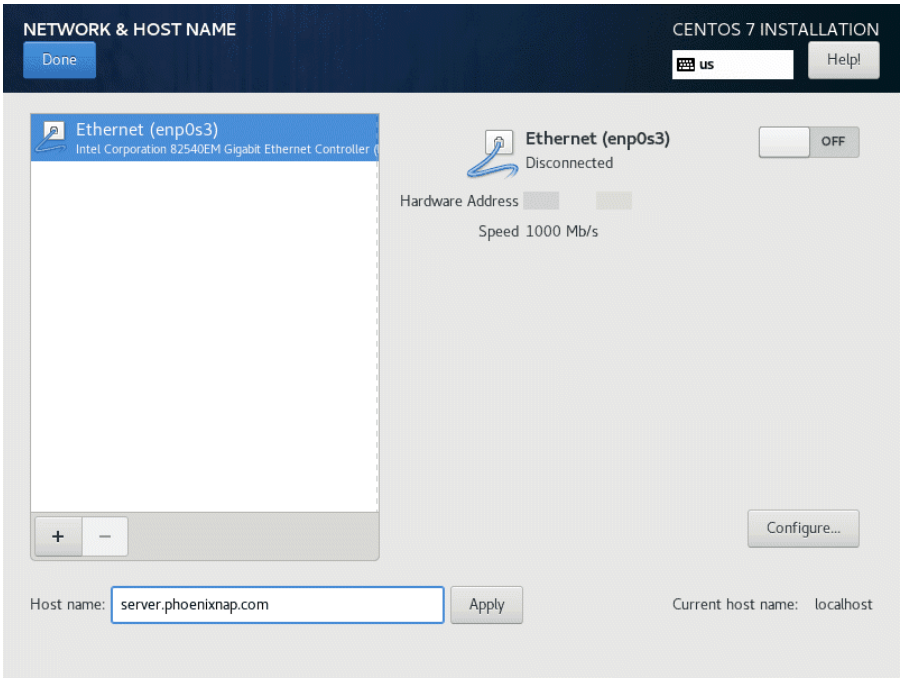
18. Configure Network IP & Host Name.

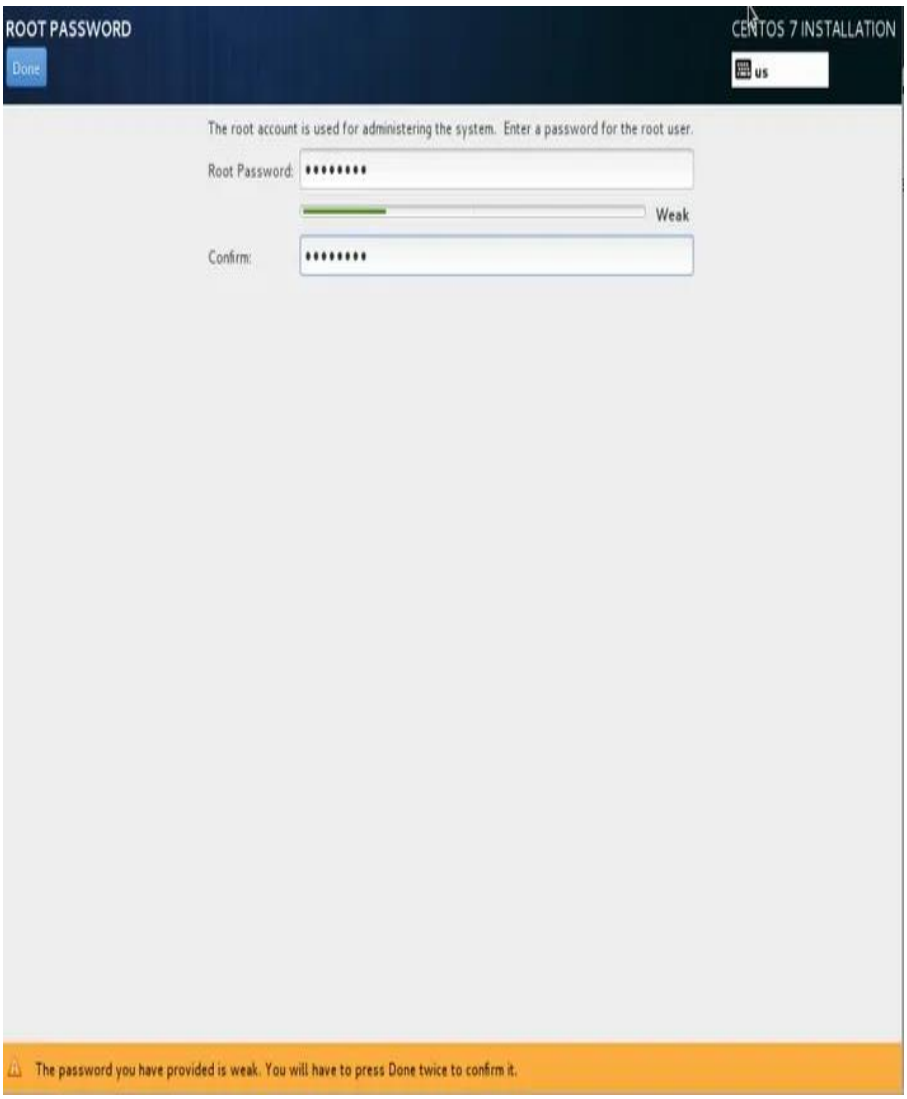
Set the Hostname

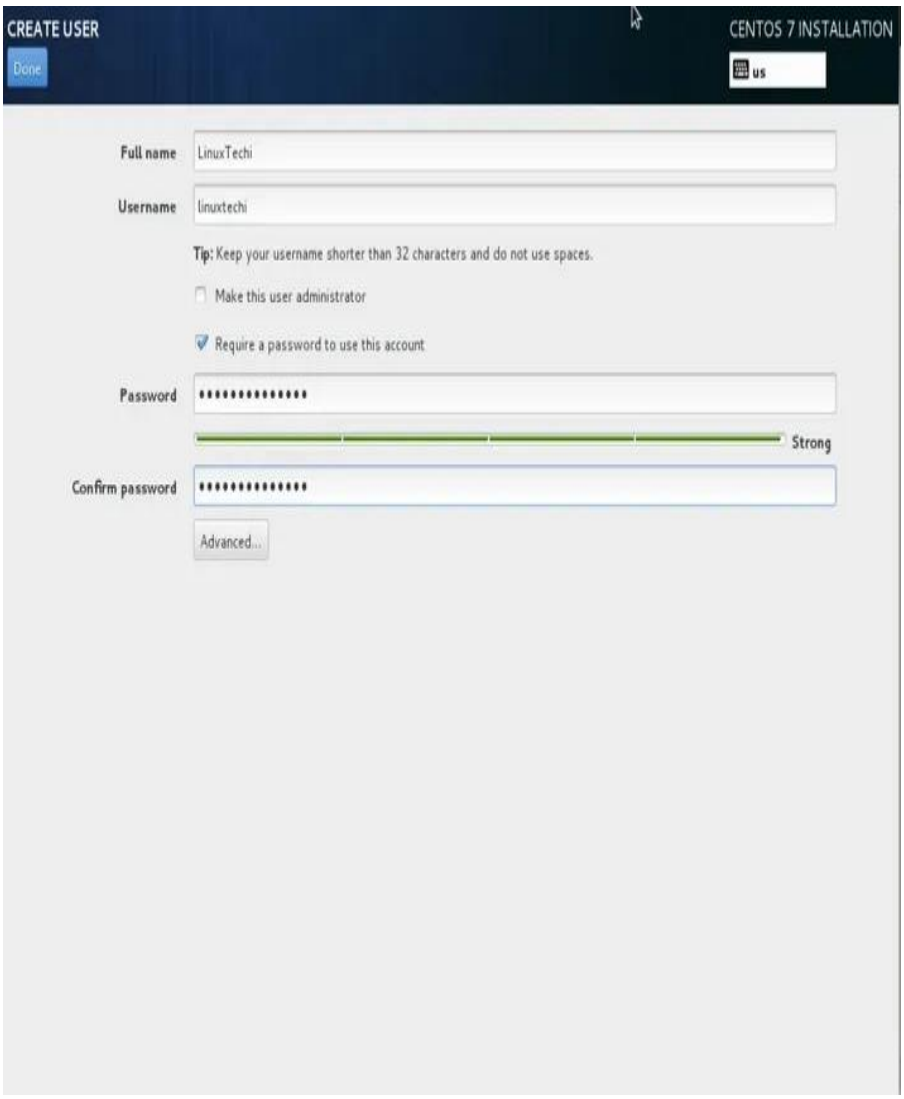
In our example, we will set
the Hostname as webapp.localhost.com, where webapp is the hostname while localhost.com is the domain.

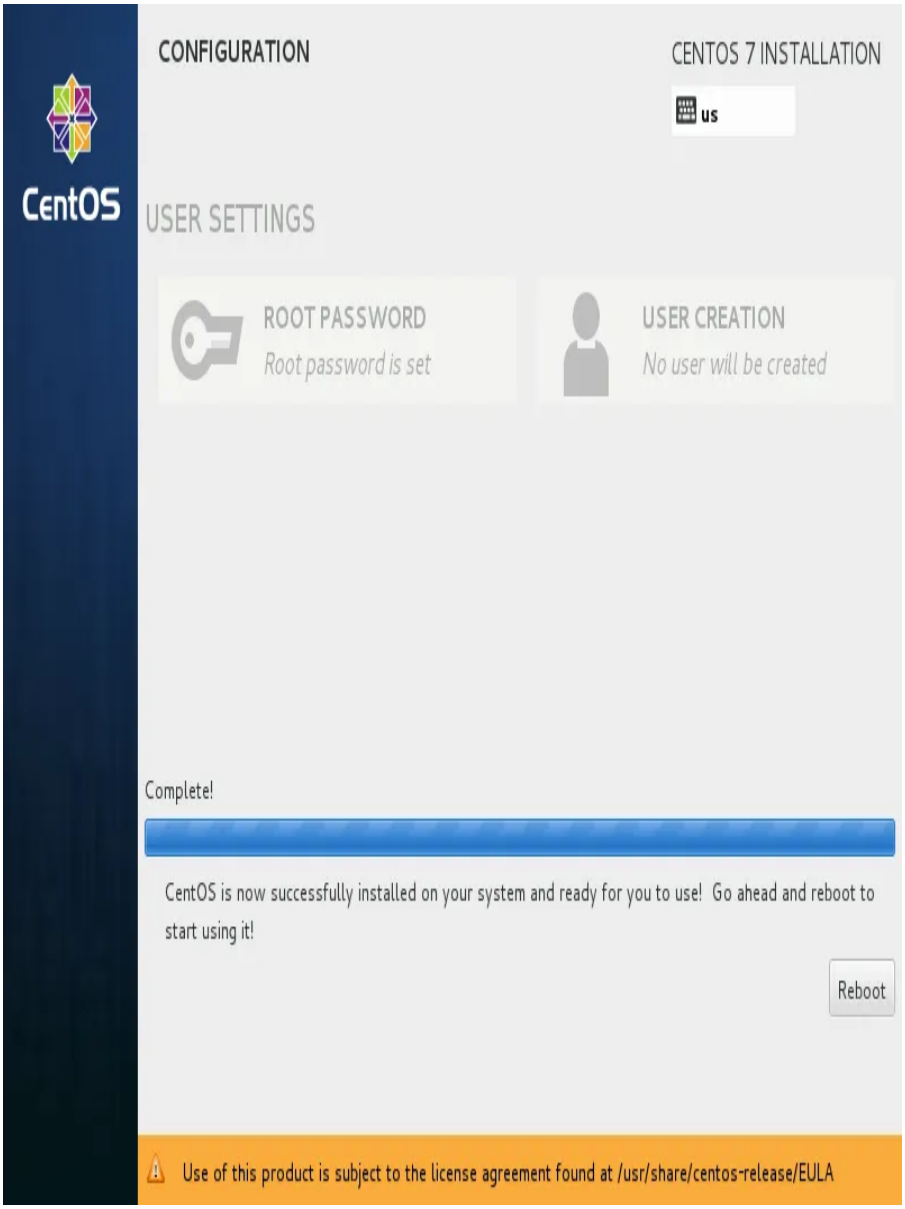
To add a static IPv4 address:

1. Turn ON Ethernet.
2. Select configure.
3. Click the Add button to add a static IP address.
4. Enter the information for your network domain. For example
    - IP Address (192.168.0.10)
    - Netmask Address (255.255.255.0)
    - Gateway Address (192.168.0.254)
    - DNS Servers Address (192.168.0.12)
5. Click Save to confirm your changes.

		
19.	Define Root Password & User Creation	


ROOT PASSWORD
Done
CENTOS 7 INSTALLATION
us
The root account is used for administering the system. Enter a password for the root user.
Root Password:
Weak
Confirm:
The password you have provided is weak. You will have to press Done twice to confirm it.

		
20.	Once done, remove any installation media and reboot your computer.	

		
	Installing the 64-Bit JDK 11	1 hour
21.	To give sudo access to a user 1. To Open Terminal Using Shortcut (CTRL+ALT+T). 2. First, Switch to the root user if required. `su root`	

	3. Use the visudo command to edit the configuration file: `[root@localhost ~]$ visudo` 4. This will open /etc/sudoers for editing. To add a user and grant full sudo privileges, add the following line: `[webapp] ALL=(ALL:ALL) ALL` 5. Save and exit the file. `:wq`	
22.	First, switch to the root user if required. `[webapp@localhost ~]$ sudo su -`	
23.	Then, download Oracle Java JDK 17 using the wget command in the terminal. `[root@webapp ~]$ wget https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.rpm`	
24.	And then, install Oracle Java JDK 17 using the rpm command. `[root@webapp ~]$ rpm -ivh jdk-17_linux-x64_bin.rpm`	
25.	After the installation of Java, use the below command to verify the version. `[root@webapp ~]$ java –version` Output: `java version "17.0.1" 2021-10-19 LTS` `Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)` `Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)`	
	Steps to start Resin for development	1 hour
26.	Link /usr/java to the Java home or set environment variable JAVA_HOME. `[root@webapp ~]$ vi /etc/bashrc` `or` `[root@webapp ~]$ vim ~/.bashrc` Add the following line at the end: `export JAVA_HOME=/usr/java/jdk-17.0.2`	

	Save and exit the file. `:wq` or `:q!`	
27.	Download the resin package and unzip it `[webapp@localhost ~]$ wget -c http://caucho.com/download/resin-4.0.63.tar.gz` `[webapp@localhost ~]$ tar zxf resin-4.0.63.tar.gz`	
28.	`[webapp@localhost ~]$ mkdir resin` `[webapp@localhost ~]$ cd resin-4.0.63`	
29.	Install openssl-devel package. `[webapp@localhost ~]$ yum install -y openssl-devel`	
29.	Compile and install. Set JAVA_HOME using the syntax export JAVA_HOME=path to JDK. For example, export JAVA_HOME= /usr/java/jdk17.0.2/. `[webapp@localhost resin-4.0.63]$ ./configure --prefix=/home/webapp/resin --with-java-home=/usr/java/jdk-17.0.2 --enable-64bit --enable-64bit-jni --enable-64bit-plugin --enable-debug` `[webapp@localhost resin-4.0.63]$ make` `[webapp@localhost resin-4.0.63]$ make install`	
30.	Use the following method to start the resin installed by compiling. `[webapp@localhost ~]$ /home/webapp/resin/bin/resin.sh start`	
31.	Browse to http://localhost:8080	
32.	Configure firewall. Allow traffic on port 8080.	

	``[root@webapp ~]$ firewall-cmd --zone=public --add-port=8080/tcp --permanent```	
	Enabling HTTPs on your resin server (Additional steps not mandatory from step 33 to 35)	
33.	Generate a self-signed certificate in <resin_dir>/conf/keys/ location ```openssl genrsa -out self-ssl.key openssl req -new -key self-ssl.key -out self-ssl.csr -config csr.conf openssl x509 -req -days 365 -in self-ssl.csr -signkey self-ssl.key -out self-ssl.crt -extensions req_ext -extfile csr.conf```	
34.	Configure the resin with generated certificate and key file. ```<resin_dir>/conf/resin.properties openssl_file : keys/<crt_file> openssl_key : keys/<private_key>```	
35.	Restart resin Use the following method to restart the resin installed by compiling. ```[webapp@localhost ~]$ /home/webapp/resin/bin/resin.sh stop``` ```[webapp@localhost ~]$ /home/webapp/resin/bin/resin.sh start```	
	Install Node.js and npm from Node Source repository	1 hour
36.	Next, add the NodeSource repository to the system with:	

	`[root@webapp ~]$ curl -sL https://rpm.nodesource.com/setup_14.x | bash -`	
37.	The output will indicate you to use the following command if you want to install Node.js and npm: `[root@webapp ~]$ yum install -y nodejs`	
38.	Finally, verify the installed software with the commands: `[root@webapp ~]$ node -v` Your result should be similar to this: `v14.9.2` `[root@webapp ~]$ npm -version` Your result should be similar to this: `6.13.6`	
	Enabling MariaDB	1 hour
39.	We'll start the daemon with the following command: `[webapp@localhost ~]$ sudo systemctl start mariadb`	
40.	systemctl doesn't display the outcome of all service management commands, we'll use the following command: `[webapp@localhost ~]$ sudo systemctl status mariadb`	
	If MariaDB has successfully started, the output should contain "Active: active (running)` and the final line should look something like: `Dec 01 19:06:20 centos-512mb-sfo2-01 systemd[1]: Started MariaDB database server.`	
41.	Next, let's take a moment to ensure that MariaDB starts at boot, using the systemctl enable command, which will create the necessary symlinks.	

	`[webapp@localhost ~]$ sudo systemctl enable mariadb`	
	Output: `Created symlink from /etc/systemd/system/multi-user.target.wants/mariadb.service to /usr/lib/systemd/system/mariadb.service.`	
	## Securing the MariaDB Server	
42.	MariaDB includes a security script to change some of the less secure default options for things like remote root logins and sample users. Use this command to run the security script: `[webapp@localhost ~]$ sudo mysql_secure_installation` The script will prompt you to set up the root user password	
	## Testing the Installation	
43.	We can verify our installation and get information about it by connecting with the `mysqladmin` tool, a client that lets you run administrative commands. Use the following command to connect to MariaDB as root (-u root), prompt for a password (-p), and return the version. `[webapp@localhost ~]$ mysqladmin -u root -p version`	

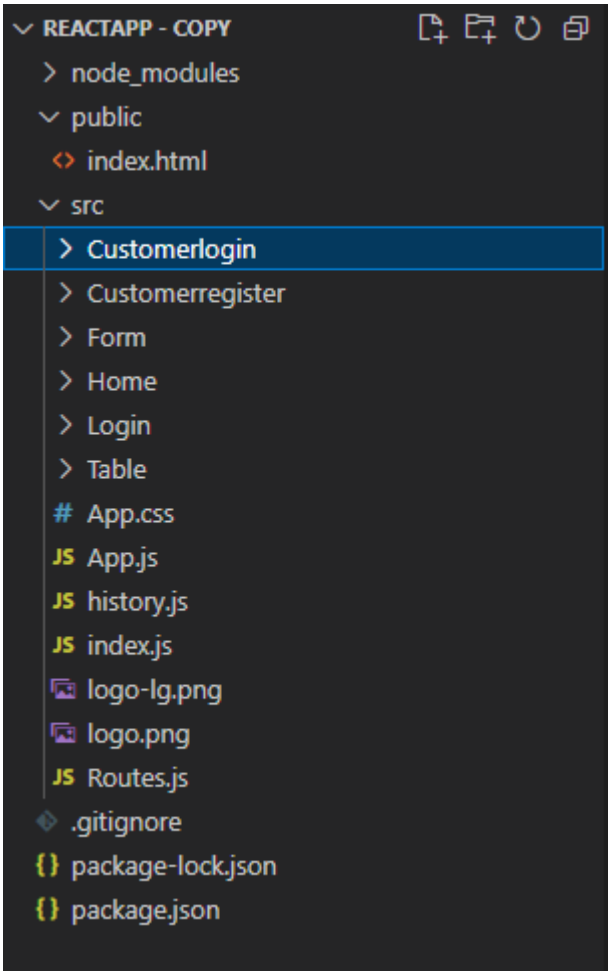
	You should see output similar to this: `Output` `mysqladmin Ver 9.0 Distrib 5.5.50-MariaDB, for Linux on x86_64` `Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab, and others.` `Server version 5.5.50-MariaDB` `Protocol version 10` `Connection Localhost via UNIX socket` `UNIX socket /var/lib/mysql/mysql.sock` `Uptime: 4 min 4 sec` `Threads: 1 Questions: 42 Slow queries: 0 Opens: 1 Flush tables: 2 Open tables: 27 Queries per second avg: 0.172` This indicates the installation has been successful.	
	## Login to MySQL	
44.	First, we'll login to the MySQL server from the command line with the following command: `[webapp@localhost ~]$ mysql -u root -p` `Enter password:`	
	## Show (View) All MySQL Databases	
45.	To view the database, you've created simply issue the following command: `MariaDB [(none)]> SHOW DATABASES;`	
	Your result should be similar to this:	

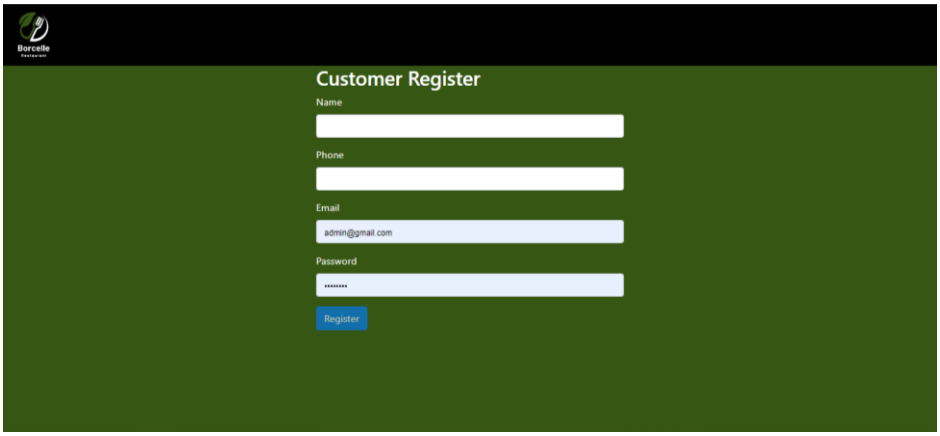
	`` mysql> SHOW DATABASES; +--------------------+ | Database | +--------------------+ | information_schema | | mysql | | test | +--------------------+ 4 rows in set (0.00 sec) ``` To exit, type quit or exit and press [Enter].	
	Automate Services start on reboot (Additional steps not mandatory from step 46 to 50)	
46.	Enable MariaDB Service on Boot `[webapp@localhost ~]$ systemctl enable mysql.service`	
	Start resin on Boot	
47.	Make an entry in `/etc/rc.d/rc.local` with the start command `nano /etc/rc.d/rc.local` `"su -webapp -c '/home/webapp/resin/bin/resin.sh start'"`	
	To make nodejs as service	
48.	Install PM2 `$ npm install pm2@latest -g`	
49.	Start the app `$ pm2 start app.js`	
50.	Managing application state `$ pm2 restart app_name` `$ pm2 reload app_name`	

	``$ pm2 stop app_name``` ```$ pm2 delete app_name```	

Part- B Building an interactive web application		
	Build the frontend of Restaurant table reservation web application using React JS	
51.	Install Visual Studio Code IDE Import the Microsoft GPG key with this command. ```# sudo rpm –import https://packages.microsoft.com/keys/microsoft.asc```	
52.	Create the repo file as below to enable the Visual Studio Code repository. ```# sudo nano /etc/yum.repos.d/vscode.repo```	
53.	Add the below-given content in vscode.repo ```[code]``` ```name=Visual Studio Code``` ```baseurl=https://packages.microsoft.com/yumrepos/vscode``` ```enabled=1``` ```gpgcheck=1``` ```gpgkey=https://packages.microsoft.com/keys/microsoft.asc```	
54.	Save and exit the vscode.repo	
55.	Install the latest version of Visual Studio Code with this command. ```# sudo yum install code```	
56.	Now that VS Code is installed on your CentOS system now you	

	can open it from Applications -> Programming -> Visual Studio Code.	
55.	Create a responsive Restaurant table reservation web application there will be an admin interface and a customer interface. The customer interface is to register, sign in and reserve a table as per the availability of the seats and the admin interface is to sign in and manage table booking.	
56.	App Flow Customers create an account → login → books a slot. Admin login → view bookings → clear bookings	
	Open Visual Studio. (Choose File > Open Folder) and select the react.zip folder that contains the React application.	
56.	Getting Started Inside `public` folder, `index.html` file that will serve as our app's starting point. The index.html file is the root of your application. This is the file the server reads, and it is the file that your browser will display.	
57.	Next, inside `src` folder, `index.js` file is your JavaScript entry point to import dependencies, and it will be run as soon as your app has loaded.	
58.	Building our App There will be a main parent component. Each of the individual "pages" of app will be separate components that feed into the main component.	
59.	Displaying the Initial Frame Inside `src` folder, `App.js` will just be a component that contains UI elements for our navigation header and an empty area for content to load in.	
60.	Adding Some CSS Inside `src` folder, `App.css` is to style the app.	
60.	Creating our Content Pages	

	Our app will have six pages of content.	
	Home Page **Admin** • Admin Login Form • Booking Page to list bookings and clear bookings **Customer** • Customer Register Form • Customer Login Form • Booking Page to book a slot	
	The directory structure of the react project will look like this.  	
61.	Create Customerregister Component	



Customerregister component is for customer register.

Inside `src/Customerregister` folder,

open `Customerregister.js` file and write the following code to create a simple sign-up form with name, phone, email, and password input fields and a submit button that allows for user input and subsequently POSTs the content to an API:

Inside the `handleCustomerRegister` function, you prevent the default action of the form. Then update the `state` to the `data` input.

We have defined states for email, name, phone, and password for holding form data.

Note: The states can only be updated using set methods as shown in the methods.

We're setting email, name, phone, and password to empty strings.

```
// Handling the customer registration form submission
  handleCustomerRegister = e => {
    e.preventDefault();
    const data = {
      email: this.state.email,
      name: this.state.name,
      phone: this.state.phone,
      password: this.state.password,
```

```
  };
    this.setState({
    email: '',
    name: '',
    phone: '',
    password: '',
  });
```

Using `POST` gives you the same response object with information that you can use inside of a then call.

To complete the `POST` request, you first capture the `data` input. Then you add the input along with the `POST` request, which will give you a response. You can then `console.log` the response, which should show the `data` input in the form.

HTTP POST request to the server and add the data to the database.

```
  axios
    .post("http://localhost:8080/app/customerregister", data)
    .then(res => {
      if (res.data === 1) {
        alert("Registered Successfully");
      }
    }
    )
    .catch(err => console.log(err));


};
```

Inside the `checkUser` function, you prevent the default action of the form.

The user enters their email. If a user with the provided email already exists in the database, an alert message is displayed right away.

```
// Handling the user already exists

checkUser = e => {

    e.preventDefault();

const value = e.target.value;

console.log(value)

var data = '{"email":"'+value+'"}';

console.log(data);
```

Using `POST` gives you the same response object with information that you can use inside of a then call.

To complete the `POST` request, you first capture the `data` input. Then you add the input along with the `POST` request, which will give you a response. You can then `console.log` the response, which should show the `data` input in the form.

HTTP POST request to the server.

```
  axios

  .post("http://localhost:8080/app/checkuser",JSON.parse(data))

      .then(res => {

        if (res.data === 1) {

          alert("User Already Exists");

        }

        else{

          this.setState({email: value});

          this.validateField("email",value);

        }

      })
```

```
      .catch(err => console.log(err));


};
```

Now, we'll call a validation after the user types in the field.

The `setState` method takes a callback function as a second argument, so let's pass a validation function to it.

```
// Handling the name change

  handleUserInput = (e) => {

    const name = e.target.name;

    const value = e.target.value;

    this.setState({ [name]: value },

      () => { this.validateField(name, value) });

  }
```

We do two different checks for the input fields. For the email field, we check it against a regular expression to see if it's an email.

For the phone field, we check if the length is exactly of 10 characters or not.

For the password field, we check if the length is a minimum of 8 characters or not.

When the field doesn't pass the check, we set an error message for it and set its validity to false.

Then we call `setState` to update the `formErrors` and the field validity.

```
// Validating the field

  validateField(fieldName, value) {

    let fieldValidationErrors = this.state.formErrors;
```

```
    let emailValid = this.state.emailValid;
    let phoneValid = this.state.phoneValid;
    let passwordValid = this.state.passwordValid;
    switch (fieldName) {

      case 'email':
        emailValid = value.match(/^[A-Z0-9._%+-]+@[A-Z0-9.-
]+\.[A-Z]{2,4}$/i);
        fieldValidationErrors.email = emailValid ? '' : ' is
invalid';
        break;
       case 'phone':
        phoneValid = value.length === 10;
        fieldValidationErrors.phone = phoneValid ? '' : ' is
invalid';
        break;
      case 'password':
        passwordValid = value.length >= 8;
        fieldValidationErrors.password = passwordValid ? '' : '
must be atleast 8 characters';
        break;
      default:
        break;
    }
    this.setState({
      formErrors: fieldValidationErrors,
      emailValid: emailValid,
      phoneValid: phoneValid,
```

```
      passwordValid: passwordValid,
    }, this.validateForm);
  }
```

we pass the `validateForm` callback to set the value of `formValid`.

```
// Validating the form
 validateForm() {
    this.setState({ formValid: this.state.emailValid &&
this.state.phoneValid && this.state.passwordValid });
  }
```

`errorClass` is a method we can define as:

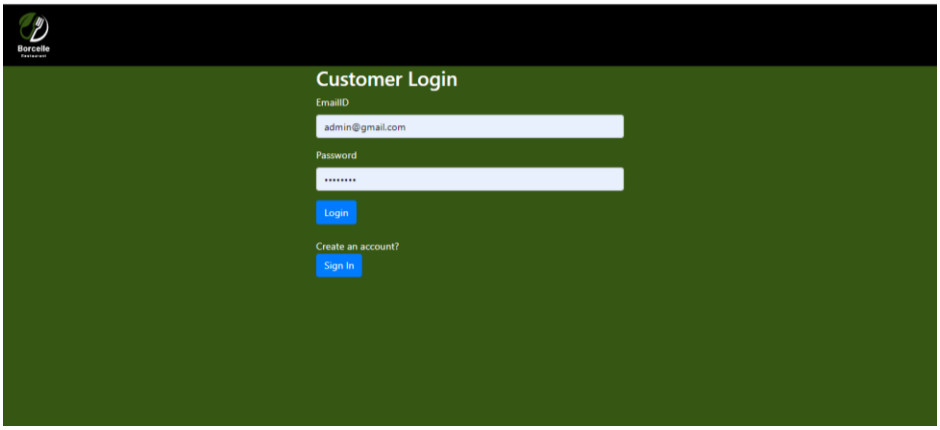
```
  errorClass(error) {
    return (error.length === 0 ? '' : 'has-error');
  }
```

Now when a field has an error, it has a red border around it.

Inside `src/Customerregister` folder, `Customerregistererrors.js` file is a stateless functional component (or presentational component) which simply iterates through all the form validation errors and displays them.

Inside `src/Customerregister` folder, `Customerregister.css` file is to style the form

62.	Create Customerlogin Component	



Customerlogin component is for customer login.

Inside `src/Customerlogin` folder, open `Customerlogin.js` file and write the following code to create a simple sign-in form with email and password input fields and a submit button that allows for user input and subsequently POSTs the content to an API:

Inside the `handleCustomerLogin` function, you prevent the default action of the form. Then update the `state` to the `data` input.

We have defined states for email, and password for holding form data.

Note: The states can only be updated using set methods as shown in the methods.

We're setting email and password to empty strings.

```
// Handling the customer login form submission
  handleCustomerLogin = e => {
    e.preventDefault();
    const data = {
      email: this.state.email,
      password: this.state.password,
    };
      this.setState({
      email: '',
```

```
    password: '',
  });
```

Using `POST` gives you the same response object with information that you can use inside of a then call.

To complete the `POST` request, you first capture the `data` input. Then you add the input along with the `POST` request, which will give you a response. You can then `console.log` the response, which should show the `data` input in the form.

HTTP POST request to the server.

```
  axios
    .post("http://localhost:8080/app/customerlogin", data)
    .then(res => {
      if (res.data !== null)
      {
       console.log(res)
       this.props.history.push({pathname:'/Form',state :{
       customerid:res.data.CustomerID[0].id
        }});
      }
      else
      alert("EmailID or Password Incorrect")
    }
    )
    .catch(err => console.log(err));
};
```

Now, we'll call a validation after the user types in the field.

The `setState` method takes a callback function as a second argument, so let's pass a validation function to it.

```
// Handling the name change
  handleUserInput = (e) => {
    const name = e.target.name;
    const value = e.target.value;
    this.setState({ [name]: value },
      () => { this.validateField(name, value) });
  }
```

We do two different checks for the input fields. For the email field, we check it against a regular expression to see if it's an email.

For the password field, we check if the length is a minimum of 8 characters or not.

When the field doesn't pass the check, we set an error message for it and set its validity to false.

Then we call `setState` to update the `formErrors` and the field validity.

```
// Validating the field
  validateField(fieldName, value) {
    let fieldValidationErrors = this.state.formErrors;
    let emailValid = this.state.emailValid;
    let passwordValid = this.state.passwordValid;

    switch (fieldName) {
      case 'email':
```

```
        emailValid = value.match(/^([\w.%+-]+)@([\w-
]+\.)+([\w]{2,})$/i);
        fieldValidationErrors.email = emailValid ? '' : ' is
invalid';
        break;
      case 'password':
        passwordValid = value.length >= 8;
        fieldValidationErrors.password = passwordValid ? '' :
'must be alteast 8 characters';
        break;
      default:
        break;
    }
    this.setState({
      formErrors: fieldValidationErrors,
      emailValid: emailValid,
      passwordValid: passwordValid,
    }, this.validateForm);
  }
```

we pass the `validateForm` callback to set the value of `formValid`.

```
// Validating the form
  validateForm() {
    this.setState({ formValid: this.state.emailValid &&
this.state.passwordValid });
  }
```

`errorClass` is a method we can define as:

```
  errorClass(error) {
```

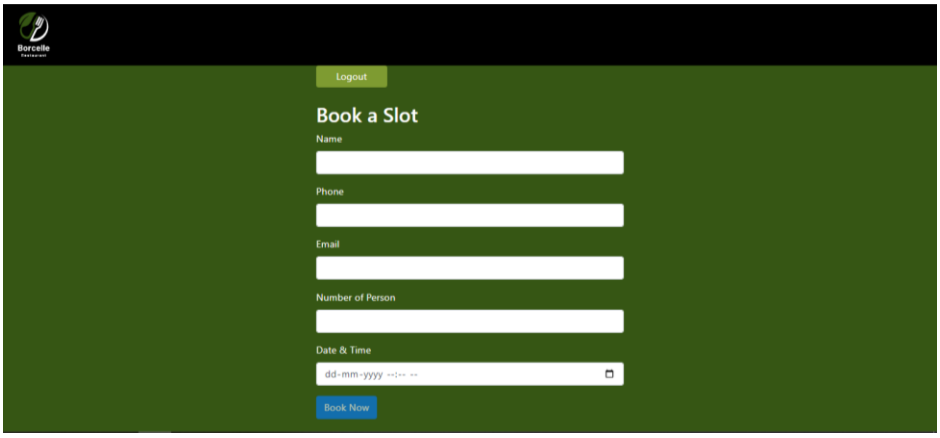
```
    return (error.length === 0 ? '' : 'has-error');
  }
```

Now when a field has an error, it has a red border around it.

Inside `src/Customerlogin` folder, `Customerloginerrors.js` file is a stateless functional component (or presentational component) which simply iterates through all the form validation errors and displays them.

Inside `src/Customerlogin` folder, `Customerlogin.css` file is to style the form

63. Create Form Component



Form component is for booking a slot.

Inside `src/Form` folder, open `Form.js` file and write the following code to create a form with name, phone, number of person and date & time input fields and a submit button that allows for user input and subsequently POSTs the content to an API:

Inside the `checkAvailability` function, you prevent the default action of the form.

The user enters the number of persons. an event checks if booking slot is available or not in the database, an alert message is displayed right away.

```
// Handling the booking slot availability

checkAvailability = e => {
```

```
    e.preventDefault();
const value = e.target.value;
console.log(value)
var data = '{"threshold":"'+value+'"}';
console.log(data);
```

Using `POST` gives you the same response object with information that you can use inside of a then call.

To complete the `POST` request, you first capture the `data` input. Then you add the input along with the `POST` request, which will give you a response. You can then `console.log` the response, which should show the `data` input in the form.

HTTP POST request to the server.

```
axios
.post("http://localhost:8080/app/checkavailability",JSON.parse(
data))
      .then(res => {
        if (res.data === 1) {
          alert("Slot not available");
        }
        else{
          this.setState({person: value});
          this.validateField("person",value);
        }
      })
      .catch(err => console.log(err));
};
```

Inside the `handleBookings` function, you prevent the default action of the form. Then update the `state` to the `data` input.

We have defined states for email, name, phone, customerid, person, and date & time for holding form data.

Note: The states can only be updated using set methods as shown in the methods.

We're setting email, name, person, date & time, and phone to empty strings.

```
// Handling the booking form submission

  handleBookings = e => {

    e.preventDefault();

    const data = {

      email: this.state.email,

      name: this.state.name,

      person: this.state.person,

      datetime: this.state.datetime,

      phone: this.state.phone,

      customerid:this.props.location.state.customerid

    };


    console.log(data);

      this.setState({

      email: '',

      name: '',

      person: '',

      datetime: '',

      phone: '',

    });
```

Using `POST` gives you the same response object with information that you can use inside of a then call.

To complete the `POST` request, you first capture the `data` input. Then you add the input along with the `POST` request, which will give you a response. You can then `console.log` the response, which should show the `data` input in the form.

HTTP POST request to the server and add the data to the database.

```
    axios
      .post("http://localhost:8080/app/createbooking", data)
      .then(res => {
        if (res.data === 1) {
          alert("Booked Successfully");
        }
      }
      )
      .catch(err => console.log(err));


  };
```

Now, we'll call a validation after the user types in the field.

The `setState` method takes a callback function as a second argument, so let's pass a validation function to it.

```
// Handling the name change

  handleUserInput = (e) => {
    const name = e.target.name;
    const value = e.target.value;
    this.setState({ [name]: value },
      () => { this.validateField(name, value) });
  }
```

We do two different checks for the input fields. For the email field, we check it against a regular expression to see if it's an email.

For the phone field, we check if the length is an exactly of 10 characters or not.

When the field doesn't pass the check, we set an error message for it and set its validity to false.

```
// Validating the field

  validateField(fieldName, value) {
    let fieldValidationErrors = this.state.formErrors;
    let emailValid = this.state.emailValid;
    let phoneValid = this.state.phoneValid;


    switch (fieldName) {
      case 'email':
        emailValid = value.match(/^([\w.%+-]+)@([\w-
]+\.)+([\w]{2,})$/i);
        fieldValidationErrors.email = emailValid ? '' : ' is
invalid';
        break;
      case 'phone':
        phoneValid = value.length === 10;
        fieldValidationErrors.phone = phoneValid ? '' : ' is
invalid';
        break;
      default:
        break;
    }
    this.setState({
```

```
      formErrors: fieldValidationErrors,
      emailValid: emailValid,
      phoneValid: phoneValid,
    }, this.validateForm);
  }
```

Then we call `setState` to update the `formErrors` and the field validity.
we pass the `validateForm` callback to set the value of `formValid`.

```
// Validating the form
  validateForm() {
    this.setState({ formValid: this.state.emailValid &&
this.state.phoneValid });
  }
```

`errorClass` is a method we can define as:

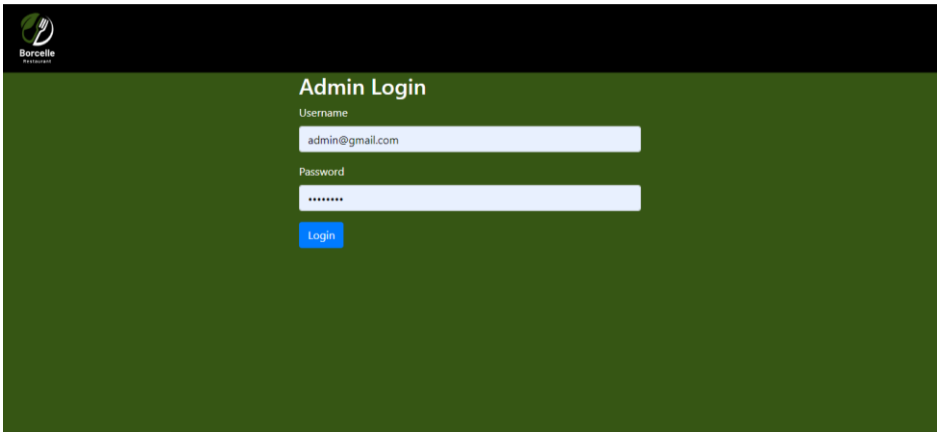
```
  errorClass(error) {
    return (error.length === 0 ? '' : 'has-error');
  }
```

Now when a field has an error, it has a red border around it.

Inside `src/Form` folder, `FormErrors.js` file is a stateless functional component (or presentational component) which simply iterates through all the form validation errors and displays them.

Inside `src/Form` folder, `Form.css` file is to style the form

64.	Create Login Component	



Login component is for admin login.

Inside `src/Login` folder, open `Login.js` file and write the following code to create a simple sign-in form with email and password input fields and a submit button that allows for user input and subsequently POSTs the content to an API:

Inside the `handleLogin` function, you prevent the default action of the form. Then update the `state` to the `data` input.

We have defined states for username, and password for holding form data.

Note: The states can only be updated using set methods as shown in the methods.

We're setting username and password to empty strings.

```
// Handling the admin login form submission
  handleLogin = e => {
    e.preventDefault();
    const data = {
      username: this.state.username,
      password: this.state.password,
    };
      this.setState({
```

```
      username: '',
      password: '',
    });
```

Using `POST` gives you the same response object with information that you can use inside of a then call.

To complete the `POST` request, you first capture the `data` input. Then you add the input along with the `POST` request, which will give you a response. You can then `console.log` the response, which should show the `data` input in the form.

HTTP POST request to the server.

```
    axios
      .post("http://localhost:8080/app/login", data)
      .then(res => {
        if (res.data === 1)
        {
         console.log(res)
         this.props.history.push('/Table');
        }
        else
        alert("Username or Password Incorrect")
      }
      )
      .catch(err => console.log(err));
  };
```

Now, we'll call a validation after the user types in the field.

The `setState` method takes a callback function as a second argument, so let's pass a validation function to it.

```
// Handling the name change
```

```
  handleUserInput = (e) => {
    const name = e.target.name;
    const value = e.target.value;
    this.setState({ [name]: value },
      () => { this.validateField(name, value) });
  }
```

We do two different checks for the input fields. For the email field, we check it against a regular expression to see if it's an email.

For the password field, we check if the length is a minimum of 8 characters or not.

When the field doesn't pass the check, we set an error message for it and set its validity to false.

```
// Validating the field
```

```
  validateField(fieldName, value) {
    let fieldValidationErrors = this.state.formErrors;
    let usernameValid = this.state.usernameValid;
    let passwordValid = this.state.passwordValid;

    switch (fieldName) {
      case 'username':
        usernameValid = value.match(/^([\w.%+-]+)@([\w-
]+\.)+([\w]{2,})$/i);
        fieldValidationErrors.username = usernameValid ? '' : '
is invalid';
        break;
      case 'password':
        passwordValid = value.length >= 8;
```

```
        fieldValidationErrors.password = passwordValid ? '' : '
is invalid';
        break;
      default:
        break;
    }
    this.setState({
      formErrors: fieldValidationErrors,
      usernameValid: usernameValid,
      passwordValid: passwordValid,
    }, this.validateForm);
  }
```

Then we call `setState` to update the `formErrors` and the field validity.
we pass the `validateForm` callback to set the value of `formValid`.

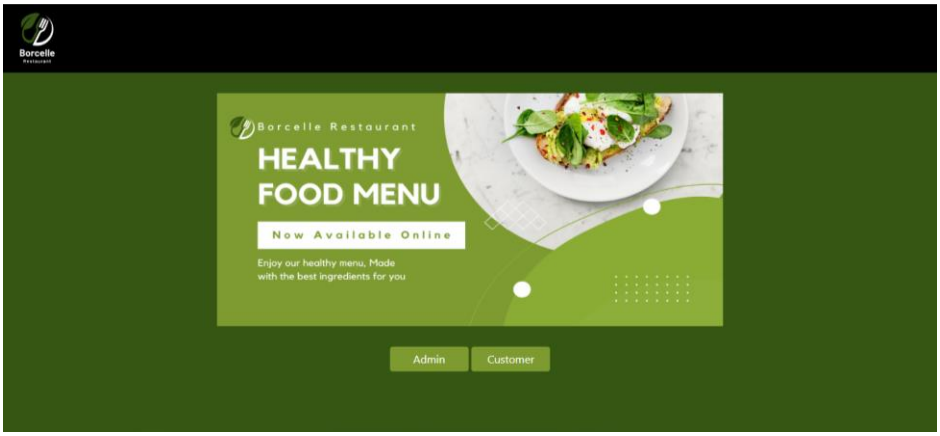
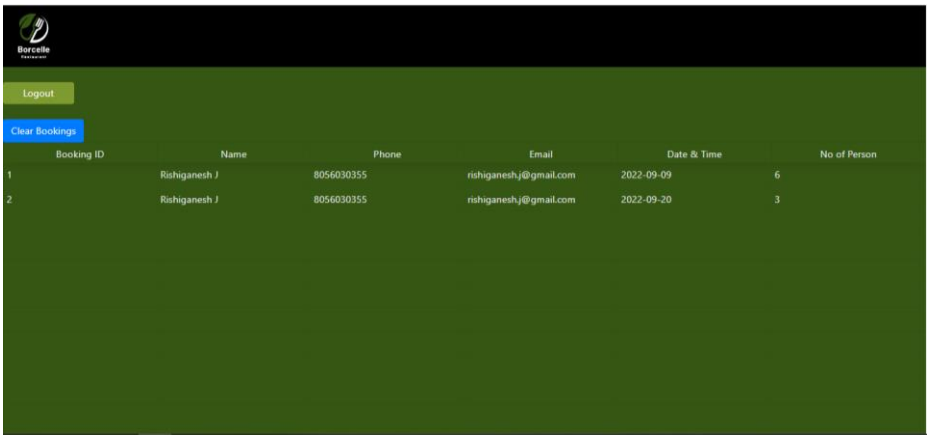
```
// Validating the form
  validateForm() {
    this.setState({ formValid: this.state.usernameValid &&
this.state.passwordValid });
  }
```

`errorClass` is a method we can define as:

```
  errorClass(error) {
    return (error.length === 0 ? '' : 'has-error');
  }
```

Now when a field has an error, it has a red border around it.

Inside `src/Login` folder, `LoginErrors.js` file is a stateless functional component (or presentational component) which simply iterates through all the form validation errors and displays them.

	Inside `src/Login` folder, `Login.css` file is to style the form	
65.	Create Home Component  Home component is for home page. Inside `src/Home` folder, `Home.js` file Inside `src/Home` folder, `Home.css` file is to style the form	
66.	Create Table Component  Table component is for listing bookings and clear bookings. Inside `src/Table` folder, open `Table.js` file and add write the following code to list bookings, subsequently GET the content from an API and to clear bookings that allows for user input, subsequently POSTs the content to an API:	

You use `axios.get(url)` with a URL from an API endpoint to get a promise which returns a response object.

```
// Handling the booking list

  async getUsersData(){

    const res = await

axios.get("http://localhost:8080/app/viewbooking")

    console.log(res.data)

    this.setState({loading:false, users: res.data})

  }
```

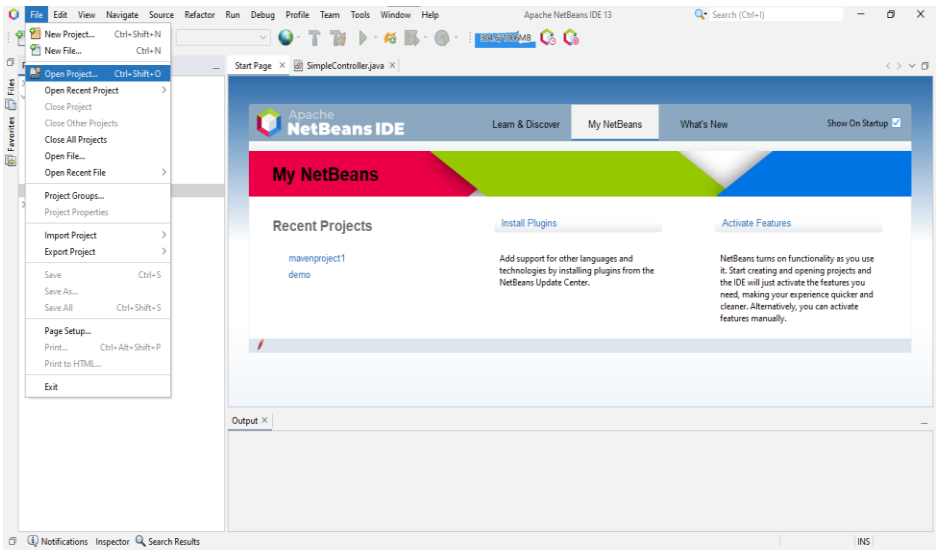
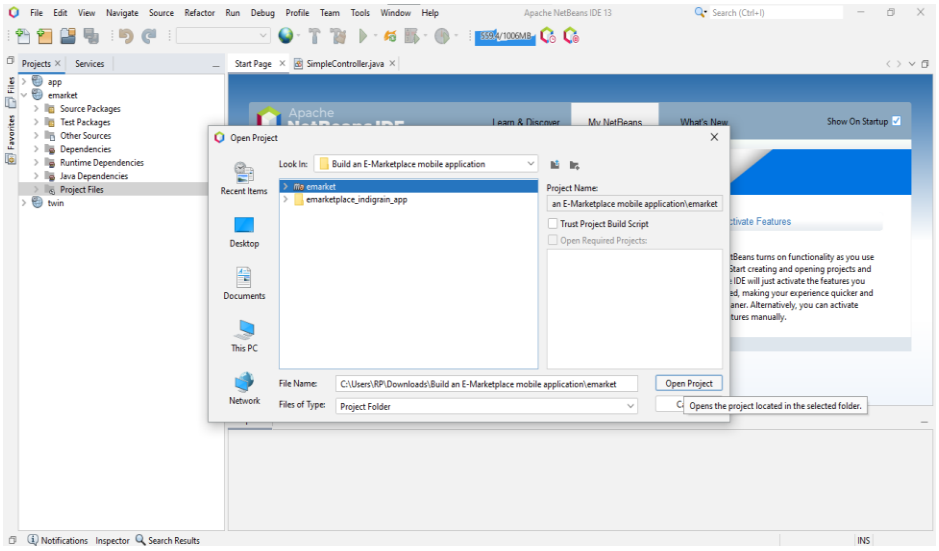
Inside the `clearBookings` function, you prevent the default action of the form.

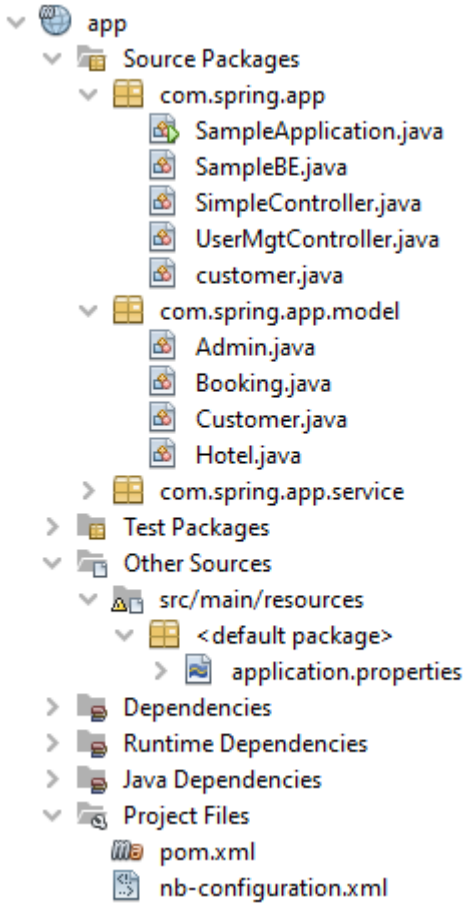
Using `POST` gives you the same response object with information that you can use inside of a then call.

HTTP POST request to the server.

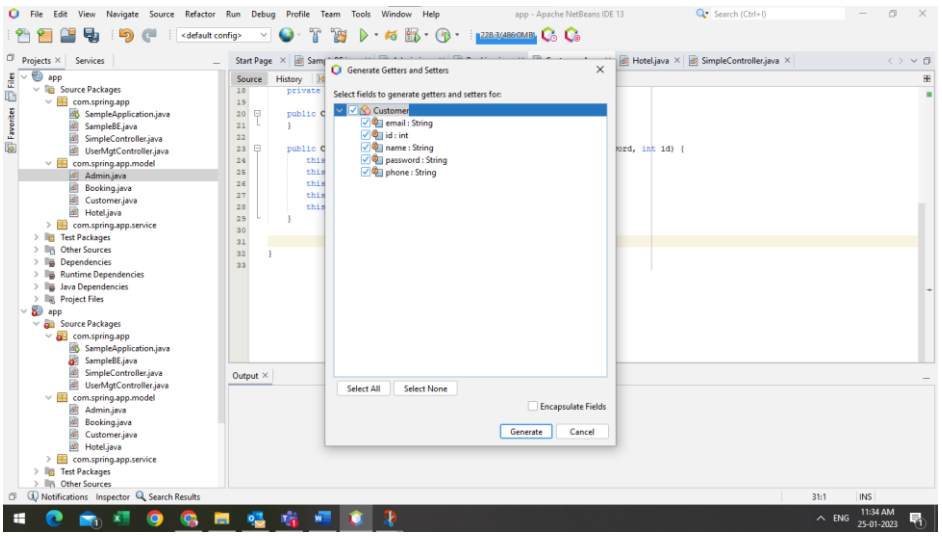
```
// Handling the clear booking

  clearBookings = e => {

    e.preventDefault();


  axios

      .post("http://localhost:8080/app/clearbookings")

      .then(res => {

        if (res.data === 1) {

          alert("Cleared Bookings");

         this.getUsersData()

        }

      }

      )

      .catch(err => console.log(err));
```

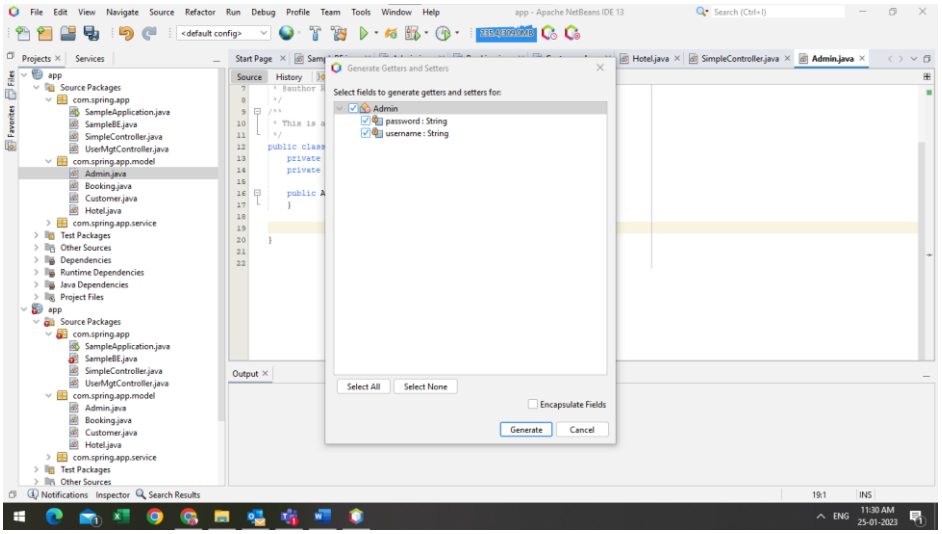
	`};`	
67.	Configure React Router Inside `src` folder, In `Routes.js` file `Route` component contains a `path` prop. The value you specify for the `path` determines when this route is going to be active. When a route is active, the component specified by the `component` prop gets rendered.	
68.	Import the history package Inside `src` folder, `History.js` file.	
69.	Run the React App Open the command prompt, on the root of the project, execute the following command. **Using npm** `npm start` This will run the application on the port, `localhost:3000`	
	Build the backend of Restaurant table reservation web application using Spring Boot framework	
64.	Install Apache NetBeans IDE To install Apache NetBeans, simply use the following command: `sudo yum install epel-release`	
65.	`sudo yum install snapd`	
66.	`sudo systemctl enable --now snapd.socket`	
67.	`sudo ln -s /var/lib/snapd/snap /snap`	

68.	`sudo snap install netbeans --classic`	
69.	A Java Spring project requires a set of libraries and packages that enable the requested features. For our project, we select Maven as the project management tool. Maven helps to build and manage your Java project. It creates a so-called POM (Project-Object-Model) with all the information and configuration details of the project, which is saved in a pom.xml file.	
	Import the Project	
70.	Open Apache NetBeans, select File › Open Project 	
71.	Select the folder containing the Maven project you want to import. 	

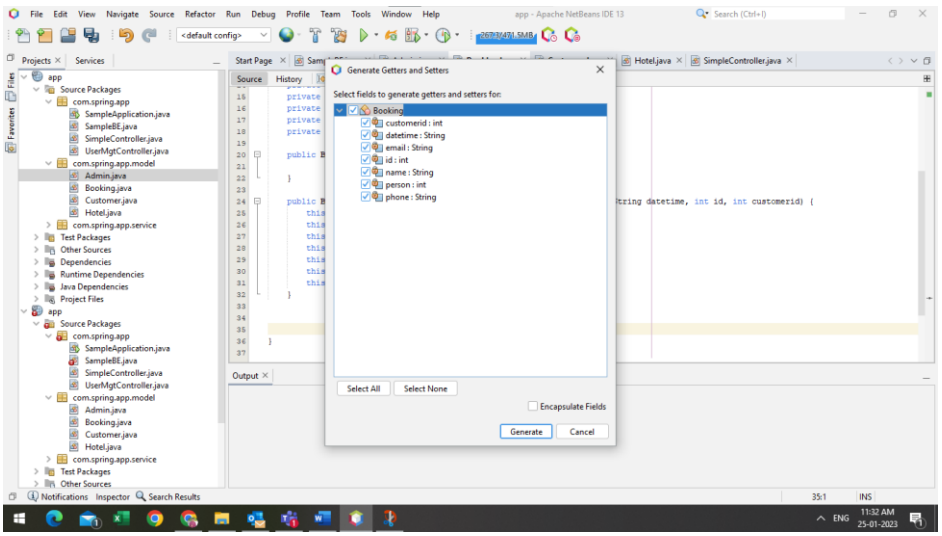
	Click Open Project to complete the process.	
	The directory structure of the spring boot project will look like this. 	
72.	Create POJOs Create .java class for Admin, Customer, Booking and Hotel	
73.	Customer.java In the Projects window, Inside project file > source packages > com.spring.app.model. Open `Customer.java` file and write the following code. Inside Customer class, Create private fields with their data types for id, name, email, phone, and password. `private int id;`	

	`` private String name; private String email; private String phone; private String password; ```	
74.	Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). ``` public Customer() {} ```	
75.	Create a constructor with the arguments name, email, phone, password, and id. Write the following code. ``` public Customer (String name, String email, String phone, String password, int id) { this.id = id; this.name = name; this.email = email; this.phone = phone; this.password = password; } ```	
76.	Create accessor methods (i.e., getter and setter methods) for this field. The IDE can create accessor methods for you. In the editor, right-click on `value` and choose Insert Code (or press Alt-Insert). In the popup menu, choose Getter and Setter.	

	 In the dialog that displays, select all the fields, then click Generate. The getValue() and setValue() methods are added to the Customer class.	
77.	Admin.java In the Projects window, Inside project file > source packages > com.spring.app.model. Open `Admin.java` file and write the following code. Inside Admin class, Create private fields with their data types for username, and password. ``` private String username; private String password; ```	
78.	Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). ``` public Admin () {} ```	
79.	Create a constructor with the arguments name, email, phone, password, and id. Write the following code. ``` public Admin (String username, String password) { this.username = username; this.password = password; ```	

	`` } ```	
80.	Create accessor methods (i.e., getter and setter methods) for this field. The IDE can create accessor methods for you. In the editor, right-click on `value` and choose Insert Code (or press Alt-Insert). In the popup menu, choose Getter and Setter.  In the dialog that displays, select all the fields, then click Generate. The getValue() and setValue() methods are added to the Admin class.	
81.	Booking.java In the Projects window, Inside project file > source packages > com.spring.app.model. Open `Booking.java` file and write the following code. Inside Booking class, create private fields with their data types for email, phone, name, person, datetime, id, and customerid. ``` private String email; private String phone; private String name; private int person; private String datetime; private int id; ```	

	`` private int customerid; ```	
82.	Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). ``` public Booking() {} ```	
83.	Create a constructor with the arguments email, phone, name, person, datetime, id and customerid. Write the following code. ``` public Booking(String email, String phone, String name, int person, String datetime, int id, int customerid) { this.email = email; this.phone = phone; this.name = name; this.person = person; this.datetime = datetime; this.id = id; this.customerid = customerid; } ```	
84.	Create accessor methods (i.e., getter and setter methods) for this field. The IDE can create accessor methods for you. In the editor, right-click on `value` and choose Insert Code (or press Alt-Insert). In the popup menu, choose Getter and Setter.	

	 In the dialog that displays, select all the fields, then click Generate. The getValue() and setValue() methods are added to the Booking class.	
85.	Hotel.java In the Projects window, Inside project file > source packages > com.spring.app.model. Open `Hotel.java` file and write the following code. Inside Hotel class, create private fields with their data types for threshold. ``` private int threshold; ```	
86.	Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). ``` public Hotel() {} ```	
87.	Create a constructor with the argument threshold. Write the following code. ``` public Hotel(int threshold) { this.threshold = threshold; } ```	
88.	Create accessor methods (i.e., getter and setter methods) for this	

field.

The IDE can create accessor methods for you. In the editor, right-click on `value` and choose Insert Code (or press Alt-Insert). In the popup menu, choose Getter and Setter.

In the dialog that displays, select all the fields, then click Generate. The getValue() and setValue() methods are added to the Hotel class.

89. Create Spring Boot API Controller for admin and customer.

`controller` package is used to implement a Spring Boot RestAPI controller to handle all incoming requests (post/get/put/delete) and response to rest-client.

Admin	Customer
• Handling admin login • List bookings • Clear bookings	• Handling customer login • Handling customer register • Book a Slot • Check booking availability • Check user already exists

90. Handling admin login

In the Projects window, Inside project file > source packages > com.spring.app Open `SimpleController.java` file and write the following code.

```
  @CrossOrigin(origins = "*")
   @RequestMapping(value = "/login", method =
RequestMethod.POST)
    public int adminLogin(@RequestBody Admin admin) {
        int result = BEObj.adminLogin(admin);
        return result;
    }
```

`@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service.

	`@RequestMapping("/login")` annotation sets the base path to the resource endpoints in the controller as /login. `@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to send username and password of a merchant. `@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/login" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the adminLogin method.	
91.	List bookings In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleController.java` file and write the following code. `@CrossOrigin(origins = "*")` `@RequestMapping(value = "/viewbooking", method = RequestMethod.GET)` `public List<Booking> viewBooking() {` `List<Booking> result = BEObj.viewBooking();` `return result;` `}` `@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service. `@RequestMapping("/viewbooking")` annotation sets the base path to the resource endpoints in the controller as /login. `@RequestMapping(method = RequestMethod.GET)`, and is used to map HTTP GET requests to the mapped controller methods. We used it to return all the bookings. `@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/viewbookings" URL with a Post JSON body, the	

	HttpMessageConverters converts the JSON request body into a Post object and passes it to the viewBooking method.	
92.	Clear bookings In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleController.java` file and write the following code. `@CrossOrigin(origins = "*")` `@RequestMapping(value = "/clearbookings", method = RequestMethod.POST)` `public int clearBookings() {` `int result = BEObj.clearBookings();` `return result;` `}` `@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service. `@RequestMapping("/clearbookings")` annotation sets the base path to the resource endpoints in the controller as /clearbookings. `@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to clear bookings. `@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/clearbookings" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the clearBookings method.	
	Handling customer login In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleController.java` file and write the following code. `@CrossOrigin(origins = "*")`	

```
    @RequestMapping(value = "/customerlogin", method =
RequestMethod.POST)
    public JSONObject customerLogin(@RequestBody Customer
customer) {
        JSONObject result = BEObj.customerLogin(customer);
        return result;
    }
```

`@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service.

`@RequestMapping("/customerlogin")` annotation sets the base path to the resource endpoints in the controller as /customerlogin.

`@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to send username and password of a merchant.

`@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/customerlogin" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the customerLogin method.

Handling customer register

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleController.java` file and write the following code.

```
    @CrossOrigin(origins = "*")
    @RequestMapping(value = "/customerregister", method =
RequestMethod.POST)
    public int customerRegister(@RequestBody Customer customer)
{
        int result = BEObj.customerRegister(customer);
```

```
        return result;
    }
```

`@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service.

`@RequestMapping("/customerregister")` annotation sets the base path to the resource endpoints in the controller as /customerregister.

`@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to send details of a customer.

`@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/customerregister" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the customerRegister method.

Book a Slot

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleController.java` file and write the following code.

```
  @CrossOrigin(origins = "*")
    @RequestMapping(value = "/createbooking", method =
RequestMethod.POST)
    public int createBooking(@RequestBody Booking booking) {
        int result = BEObj.createBooking(booking);
        return result;
    }
```

`@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service.

`@RequestMapping("/createbooking")` annotation sets the base path to the resource endpoints in the controller as /createbooking.

`@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to send details of bookings.

`@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/createbooking" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the createBooking method.

Check booking availability.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleController.java` file and write the following code.

```
  @CrossOrigin(origins = "*")

    @RequestMapping(value = "/checkavailability", method =

RequestMethod.POST)

    public int checkAvailability(@RequestBody Hotel hotel) {

        int result =

BEObj.checkAvailability(hotel.getThreshold());

        return result;

    }
```

`@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service.

`@RequestMapping("/checkavailability")` annotation sets the base path to the resource endpoints in the controller as /checkavailability.

`@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to send threshold of hotel.

`@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/checkavailability" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post

	object and passes it to the checkAvailability method.	
	Check user already exists. `@CrossOrigin(origins = "*")` `@RequestMapping(value = "/checkuser", method = RequestMethod.POST)` `public int checkUser(@RequestBody String email) {` `int result = BEObj.checkUser(email);` `return result;` `}` `@RestController`: This annotation marks the SimpleController as an HTTP request handler and allows Spring to recognize it as a RESTful service. `@RequestMapping("/checkuser")` annotation sets the base path to the resource endpoints in the controller as /checkuser. `@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the mapped controller methods. We used it to send email of a single customer. `@RequestBody`: This annotation takes care of binding the web request body to the method parameter with the help of the registered HttpMessageConverters. So, when you make a POST request to the "/checkuser" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the checkUser method.	
	Implement a method to handle admin login. In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code. `String s = "select count(*) from admin where admusername=? AND admpassword=?";` `int count = 0;` `try {`	

```
            count = jdbc.queryForObject(s, new
Object[]{admin.getUsername(), admin.getPassword()},
Integer.class);
        } catch (Exception e) {
            System.out.println("Exception" + e);
            count = 0;
        }
        if (count == 1) {
            return SUCCESS;
        } else {
            return FAILURE;
        }
```

Inside adminLogin method is where you create the query to count data values from the admin table.

The SQL SELECT statement can be used along with COUNT (*) function to count of all rows present in the admin table and SQL query that returns a value object like String then you can use the queryForObject() method of JdbcTempalte class. This method takes an argument about what type of class query will return and then convert the result into that object and returns it to the caller.

Implement a method to list bookings.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
        String s = "select bkgid AS id, bkgname AS name,
bkgemail AS email, bkgphone AS phone, bkgfromdatetime AS
datetime, bkgnoofperson AS person from booking";
        List<Booking> bklist;
        try {
```

```
            bklist = jdbc.query(s, new
BeanPropertyRowMapper(Booking.class));
        } catch (Exception e) {
            System.out.println("Exception" + e);
            bklist = null;
        }
        return bklist;
```

Inside viewBookings method is where you create the query to return a list of bookings from the booking table.

The SQL string contains a query to select all the booking details from the booking table and if your SQL query is going to return a List of objects instead of just one object then you need to use the query () method of JdbcTempalte. These methods provide to convert the result to a custom object. For instance, the simplest way to query and handle results is via the query (String, RowMapper) method. This method uses RowMapper to map the returned row to an object.

Implement a method to clear bookings.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
String s = "delete from booking";
        int resultRec = 0;
        try {
            resultRec = jdbc.update(s);
        } catch (Exception e) {
            System.out.println("Exception" + e);
            resultRec = 0;
        }
        if (resultRec == 1) {
            return SUCCESS;
        } else {
```

```
            return FAILURE;
        }
```

Inside clearBookings method is where you create the query to delete bookings from the booking table.

Create a SQL string to delete all the bookings from booking table. Call the update method of JdbcTemplate and pass the string to be bound to the query.

Implement a method to handle customer login.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
String s = "select cstid AS id from customer where cstemail='"
+ customer.getEmail() + "' AND cstpassword='" +
customer.getPassword() + "'";
        List customerlist;
        try {
            customerlist = jdbc.query(s, new
BeanPropertyRowMapper(Customer.class));
            JSONObject json = new JSONObject();
            if (!customerlist.isEmpty()) {
                json.put("CustomerID", customerlist);
                System.out.println("json = " + json);
                return json;
            }
        } catch (Exception e) {
            System.out.println("Exception" + e);
            customerlist = null;
        }
        return null;
```

Inside customerLogin method is where you create the query to return customer details as list from the customer table.

The SQL s string contains a query to select the customer ID by email and password from the customer table and if your SQL query is going to return a List of objects instead of just one object then you need to use the query () method of JdbcTempalte. These methods provide to convert the result to a custom object. For instance, the simplest way to query and handle results is via the query (String, RowMapper) method. This method uses RowMapper to map the returned row to an object.

Implement a method to handle customer register.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
String s = "insert into

customer(cstname,cstemail,cstphone,cstpassword)values(?,?,?,?)"

;

        int insert = 0;

        try {

            insert = jdbc.update(s, customer.getName(),

customer.getEmail(), customer.getPhone(),

customer.getPassword());

        } catch (Exception e) {

            System.out.println("Exception" + e);

            insert = 0;

        }

        if (insert == 1) {

            return SUCCESS;

        } else {

            return FAILURE;

        }
```

Inside customerRegister method is where you create the query to

create a customer in the customer table.

The update method provided by JdbcTemplate can be used for insert, update, and delete operations.

The SQL string is used to perform a single insert operation. Here '?' means it acts as the parameter which we need to pass while executing the query. Now to execute the query, we have used the JdbcTemplate update() method, which takes the query as an argument, and other than the query there are 4 values that correspond to 4 '?' respectively.

Implement a method to create bookings.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
String s = "insert into

booking(bkgname,bkgemail,bkgphone,bkgnoofperson,bkgfromdatetime

,bkgcstid)values(?,?,?,?,?,?)";

        int insert = 0;

        try {

            insert = jdbc.update(s, booking.getName(),

booking.getEmail(), booking.getPhone(), booking.getPerson(),

booking.getDatetime(), booking.getCustomerid());

        } catch (Exception e) {

            System.out.println("Exception" + e);

            insert = 0;

        }

        if (insert == 1) {

            return SUCCESS;

        } else {

            return FAILURE;

        }
```

Inside createBooking method is where you create the query to create a booking in the booking table.

The update method provided by JdbcTemplate can be used for insert, update, and delete operations.

The SQL string is used to perform a single insert operation. Here '?' means it acts as the parameter which we need to pass while executing the query. Now to execute the query, we have used the JdbcTemplate update() method, which takes the query as an argument, and other than the query there are 6 values that correspond to 6 '?' respectively.

Implement a method to check booking availability.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
        String s = "select htlthreshold AS threshold from
hotel";
        List<Hotel> thresholdlist = jdbc.query(s, new
BeanPropertyRowMapper(Hotel.class
        ));
        int availability = thresholdlist.get(0).getThreshold();
        System.out.println("availability = " + availability);
        String t = "select sum(bkgnoofperson) AS person from
booking";
        long thresholddblist = 0;
        if (jdbc.queryForObject(t, Long.class) != null) {
            thresholddblist = jdbc.queryForObject(t,
Long.class);
        }
int reserved = 0;
        if (thresholddblist > 0) {
            reserved = Integer.parseInt(thresholddblist + "");
            System.out.println("reserved = " + reserved);
        }
```

```
        int totalperson = reserved + threshold;

        System.out.println("totalperson = " + totalperson);

        if (availability >= totalperson) {

            return FAILURE;

        } else {

            return SUCCESS;

        }
```

Inside checkAvailability method is where you create the query to create check availability in the booking table.

The SQL s string contains a query to select the htlthreshold from the hotel table and if your SQL query is going to return a List of objects instead of just one object then you need to use the query () method of JdbcTempalte. These methods provide to convert the result to a custom object. For instance, the simplest way to query and handle results is via the query (String, RowMapper) method. This method uses RowMapper to map the returned row to an object.

The SQL t string contains a query to select the bkgnoofperson from the booking table and SQL query that returns a value object like String then you can use the queryForObject() method of JdbcTempalte class. This method takes an argument about what type of class query will return and then convert the result into that object and returns it to the caller.

Implement a method to check user already exists.

In the Projects window, Inside project file > source packages > com.spring.app. Open `SimpleBE.java` file and write the following code.

```
        JSONParser parser = new JSONParser();

        JSONObject emailObj = null;

        try {

            emailObj = (JSONObject) parser.parse(email);

        } catch (Exception e) {

            e.printStackTrace();

        }
```

```
        String InputEmail = emailObj.get("email").toString();

        String s = "select count(*) from customer WHERE
cstemail=?";

        int count = jdbc.queryForObject(s, new
Object[]{InputEmail}, Integer.class);

System.out.println("count = " + count);

        if (count == 1) {

            return SUCCESS;

        } else {

            return FAILURE;

        }
```

Inside checkUser method is where you create the query to create a check user already exists in the customer table.

The SQL SELECT statement can be used along with COUNT (*) function to count of all rows present in the customer table and SQL query that returns a value object like String then you can use the queryForObject() method of JdbcTempalte class. This method takes an argument about what type of class query will return and then convert the result into that object and returns it to the caller.

Configure pom.xml.

In the Projects window, Inside project file > Project Files. Open `pom.xml` file.

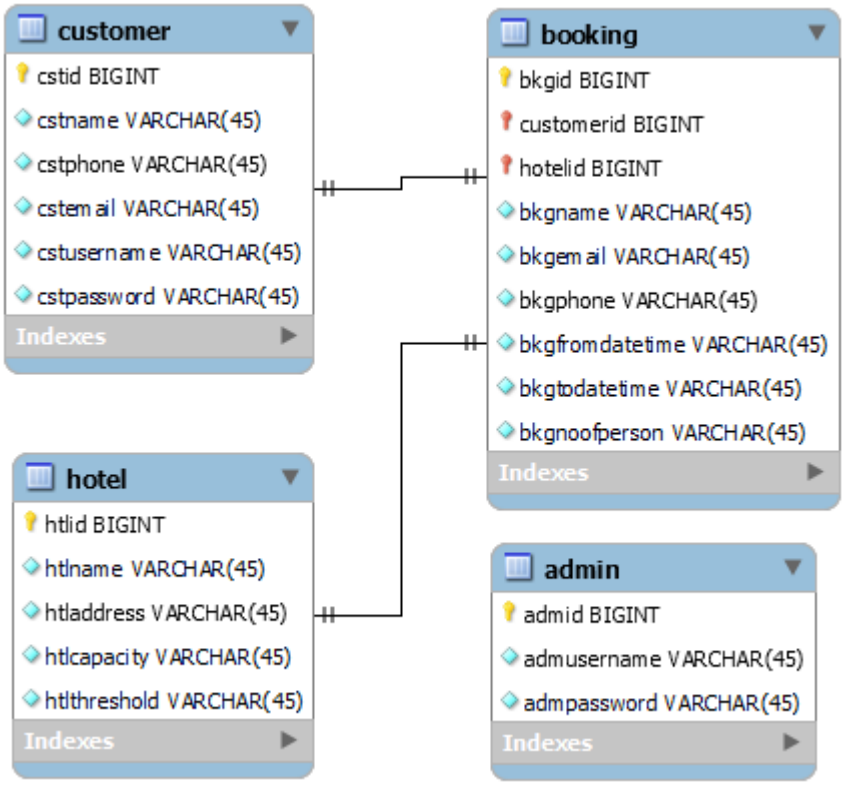
For handling the web-request and doing CRUD operations with MariaDB database, we need the supporting of 3 Spring Boot dependencies: **`spring-boot-starter-web`** and **`spring-boot-starter-data-jdbc`**, **`mariadb`**.

Configure Spring Data source.

`application. Properties` is used to add the Spring Boot application's configurations such as: database configuration.

In the Projects window, Inside project file > other sources > src/main/resources > default package. Open `application. properties`

	file. Since we're using MariaDB as our database, we need to configure the database URL, username, and password so that Spring can establish a connection with the database on startup. `spring.datasource.url=jdbc:mariadb://localhost:3306/<MariaDB database name>` `spring.datasource.username=< MariaDB username>` `spring.datasource.password=< MariaDB password>` `spring.datasource.driver-class-name=org.mariadb.jdbc.Driver`	
72.	Run the Spring Boot Project file. Right-click on the project file and click on "Clean and Build"	
	Deploy war file in the resin	
73.	Go to your spring boot project directory and inside target folder you will get war file. Copy the .war file (E.g.: webapp.war) to `[webapp@localhost ~] $/home/webapp/resin/webapps` Start the resin server. Run `[root@localhost ~] $/home/webapp/resin/bin/resin.sh start` Your .war file will be extracted automatically to a folder that has the same name (without extension) (E.g.: webapp)	
	Create a database for Restaurant table reservation web application in PostgreSQL	
74.	Create a webapp database and Create admin, customer, hotel and booking table, populate the table with data, retrieve and store data for future use, or delete if needed.	
75.	Database Design	



76. Start the MariaDB shell.

At the command prompt, run the following command to launch the MariaDB shell and enter it as the root user:

```
[root@webapp ~]$ /usr/bin/mysql -u root -p
```

When you're prompted for a password, enter the one that you set at installation, or if you haven't set one, press Enter to submit no password.

The following shell prompt should appear:

```
MariaDB [(none)]>
```

77. Create a database called `webapp`:

```
MariaDB [(none)]> CREATE SCHEMA webapp;
```

```
MariaDB [(none)]> USE webapp;
```

78. Create a table called `admin`:

```
MariaDB [webapp]> CREATE TABLE admin (
admid INT NOT NULL AUTO_INCREMENT,
admusername VARCHAR(50) NOT NULL,
admpassword VARCHAR(40) NOT NULL,
PRIMARY KEY (admid)
);
```

In the admin table "admid", "admusername", "admpassword" represents the name of the columns. INT and VARCHAR are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, "admid" is defined as the Primary Key Column. The primary key column is used for distinguishing a unique row in a table. AUTO_INCREMENT to create a column whose value can be set automatically from a simple counter. You can only use AUTO_INCREMENT on a column with an integer type. The column must be a key, and there can only be one AUTO_INCREMENT column in a table.

79. Create a table called `customer`:

```
CREATE TABLE customer (
cstid INT NOT NULL AUTO_INCREMENT,
cstname VARCHAR(50) NOT NULL,
cstemail VARCHAR(40) NOT NULL,
cstphone BIGINT(10) NOT NULL,
```

	cstpassword VARCHAR(40) NOT NULL, PRIMARY KEY (cstid), CONSTRAINT uniqueemail UNIQUE (cstemail)); In the admin table “cstid”, “cstname”, “cstemail”, “cstphone”, “cstpassword” represents the name of the columns. INT, BIGINT and VARCHAR are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “cstid” is defined as the Primary Key Column. The primary key column is used for distinguishing a unique row in a table. AUTO_INCREMENT to create a column whose value can be set automatically from a simple counter. You can only use AUTO_INCREMENT on a column with an integer type. The column must be a key, and there can only be one AUTO_INCREMENT column in a table. UNIQUE to specify that all values in the cstemail column must be distinct from each other. For UNIQUE indexes, you can specify a name for the constraint, using the CONSTRAINT keyword. That name will be used in error messages.	
80.	Create a table called `booking`: CREATE TABLE booking (bkgid INT NOT NULL AUTO_INCREMENT, bkgname VARCHAR(50) NOT NULL, bkgemail VARCHAR(40) NOT NULL, bkgphone BIGINT(10) NOT NULL, bkgfromdatetime DATETIME, bkgnoofperson BIGINT NOT NULL, bkgcstid INT NOT NULL, PRIMARY KEY (bkgid, bkgcstid),	

```
foreign key(bkgcstid) references customer(cstid)

);
```

In the booking table “bkgid”, “bkgname”, “bkgemail”, “bkgphone”, “bkgfromdatetime”, “bkgnoofperson”, “bkgcstid” represents the name of the columns. INT, BIGINT and VARCHAR are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “bkgid”, “bkgcstid” are defined as the Primary Key Column. The primary key column is used for distinguishing a unique row in a table. AUTO_INCREMENT to create a column whose value can be set automatically from a simple counter. You can only use AUTO_INCREMENT on a column with an integer type. The column must be a key, and there can only be one AUTO_INCREMENT column in a table. The bkgcstid column is the foreign key column that references the cstid column of the customer table.

81. Create a table called `hotel`:

```
CREATE TABLE hotel (

htlid INT NOT NULL AUTO_INCREMENT,

htlname VARCHAR(50) NOT NULL,

htladdress VARCHAR(40) NOT NULL,

htlcapacity BIGINT NULL,

htlthreshold BIGINT NOT NULL,

PRIMARY KEY ( htlid )

);
```

In the hotel table “htlid”, “htlname”, “htladdressl”, “htlcapacity”, “htlthreshold” represents the name of the columns. INT, BIGINT and VARCHAR are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here,

	"htlid" is defined as the Primary Key Column. The primary key column is used for distinguishing a unique row in a table. AUTO_INCREMENT to create a column whose value can be set automatically from a simple counter. You can only use AUTO_INCREMENT on a column with an integer type. The column must be a key, and there can only be one AUTO_INCREMENT column in a table.	
82.	Insert a record into the admin table. `INSERT INTO admin (admusername, admpassword) VALUES ('admin@gmail.com', 'admin@123');` The 'admin' is an already created table. Now we are adding a new row of records under the respective columns with the corresponding values: 'admin@gmail.com' and 'admin@123. Verify the insertion, using the SELECT statement. `SELECT * FROM admin;`	
83.	Insert a record into the hotel table. `INSERT INTO hotel (htlname, htladdress, htlcapacity, htlthreshold) VALUES ('Grill box', 'Adyar', 50, 10 );` The 'hotel' is an already created table. Now we are adding a new row of records under the respective columns with the corresponding values: 'Grill Box', 'Adyar' 50 and 10. Verify the insertion, using the SELECT statement. `SELECT * FROM hotel;`	
84.	Test the web application. Run the React App Open the command prompt, on the root of the React JS project, execute the following command. **Using npm** `npm start`	

	Start the resin server. Run `[root@localhost ~] $/home/webapp/resin/bin/resin.sh start`	