

CAPSTONE PROJECT REPORT

Shop From Home (HOMENIQUE)

Submitted by –

Shubhangi	shubhangisinhabgs25@gmail.com	Group 3 C1 batch
Rashi Srivastava	rashisrivastava014@gmail.com	
Chandana Bapathi	bapathichandana2001@gmail.com	
NaveenKumar P	123pnaveenkumar@gmail.com	
Aditya Kumar	adityaumar1398@gmail.com	

Under the Guidance of

Mr. Parth Shukla

(Training mentor Great Learning)



Link to the Demonstration:

<https://drive.google.com/file/d/1DTmJKEA1mFKuffKGsAOv7-Mu8CiiPCft/view?usp=sharing>

ACKNOWLEDGEMENT

We student of Great Learning would like to thank our project mentor **Mr Parth Shukla** for his guidance and support while working on this project. His regular monitoring of the project and correction of various project documents made sure the project was in the right direction. We would also like to thank our **Wipro & Great Learning** for providing such opportunity and their constant efforts to keep our courses going despite all the hardships that the pandemic brought upon us.

Finally, we would also like to thank our family, friends and seniors for their constant support, trust and valuable remarks that directly or indirectly made the project a whole lot better and helped us reach where we are.

TABLE OF CONTENTS

Acknowledgement

1. Introduction

1.1 Overview _____

2. Problem Description

2.1 Problem Description 3 _____

2.2 Solution to the problem _____

3. Technology Required

3.1 Introduction 4 _____

3.2 Existing Software 4 _____

3.3 Data Flow Diagram _____

4. Angular Front End

4.1 Product Definition 7

4.2 Feasibility Analysis 8

4.3 Project Plan _____

5. Spring Boot Backend

5.1 Introduction _____

5.2 Functional

5.3 Requirements

5.4 Non-Functional Requirements

6 Database

6.2 System Design _____

7 Final Result

8 Conclusion

9 Reference _____

INTRODUCTION

Homenique is an Angular, Java Spring Boot and PostgreSQL based application project focused on managing the online shopping platform for the Home-decor and household items. The main purpose for developing this project is to manage home decor item, category, company, order, sales etc. There are two categories of user available first one is admin and second one is user. Admin can add, manage and can also track every single details occurring. Users can order, wishlist and track products according to their requirements.

Problem Description

ShopForHome is a popular Store in the market for shopping the home décor stuff . Due to Covid 19 all the offline shopping stopped. So, the store wants to move to the cloud platforms and wants their own web application.

User-Requirements:

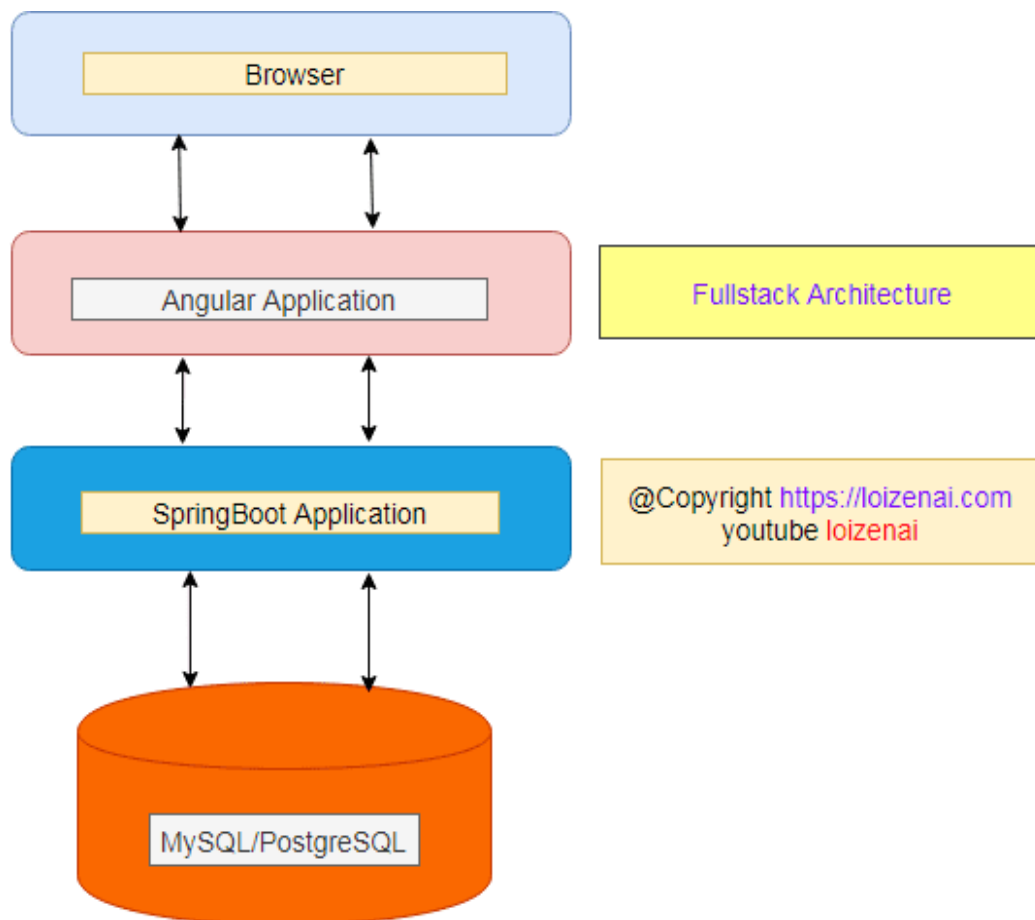
1. As a user I should be able to login, Logout and Register into the application.
2. As a user I should be able to see the products in different categories.
3. As a user I should be able to sort the products.
4. As a user I should be able to add the products into the shopping cart.
5. As a user I should be able to increase or decrease the quantity added in the cart.
6. As a user I should be able to add “n” number of products in the cart.
7. As a user I should be able to get the Wishlist option where I can add those products which I want but don't want to order now
8. As a user I should get different discount coupon

Admin-Requirements:

1. As an Admin I should be able to login, Logout and Register into the application.
2. As an Admin I should be able to perform CRUD on Users.
3. As an Admin I should be able to Perform CRUD on the products.
4. As an Admin I should be able to get bulk upload option to upload a csv for products details
5. As an Admin I should be able to get the stocks.
6. As an Admin I should be able to mail if any stock is less than 10.
7. As an Admin I should be able to get the sales report of a specific duration.
8. As an Admin I should be able to set of users

TECHNOLOGIES REQUIRED

- **HTML:** Page layout has been designed in HTML
- **CSS:** CSS has been used for all the designing part
- **JavaScript:** All the validation task and animations has been developed by JavaScript
- **Java Spring Boot:** All the business and backend API logic has been implemented in Java Spring Boot
- **SQL:** .SQL files has been used as database for the project
- **Angular:** All the frontend logic has been implemented over the Angular and we used angular CLI for it
- **Visual Studio Code-(VSS):** For Angular IDE, we have used Visual Studio Code
- **STS:** We have used STS (Spring Tool Suite) for developing all spring boot API's
- **Tomcat:** Project will be run over the Tomcat server.

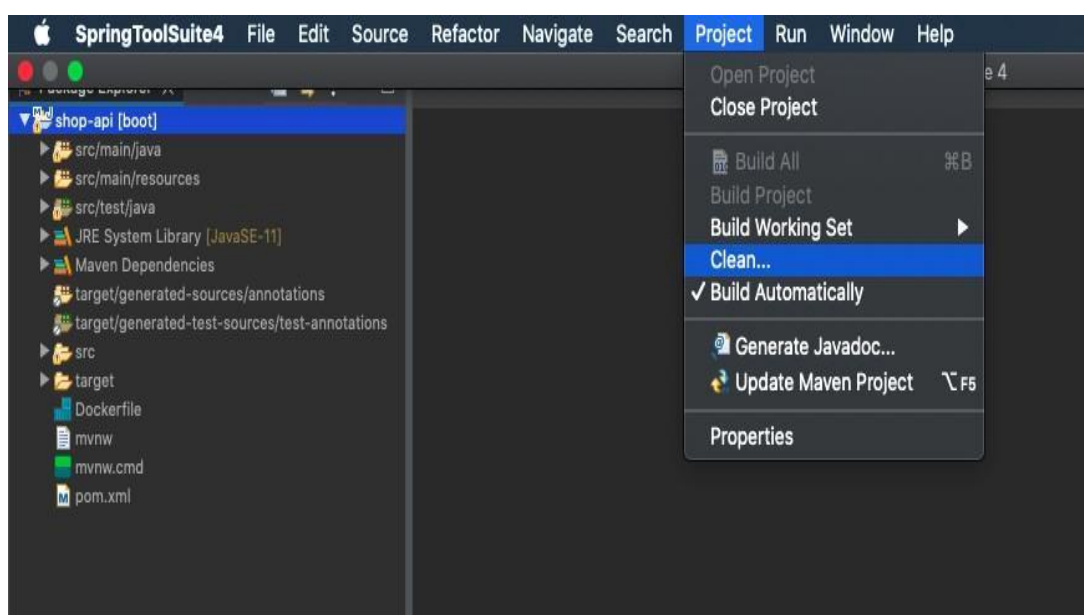


INSTALLATION AND INSTRUCTIONS

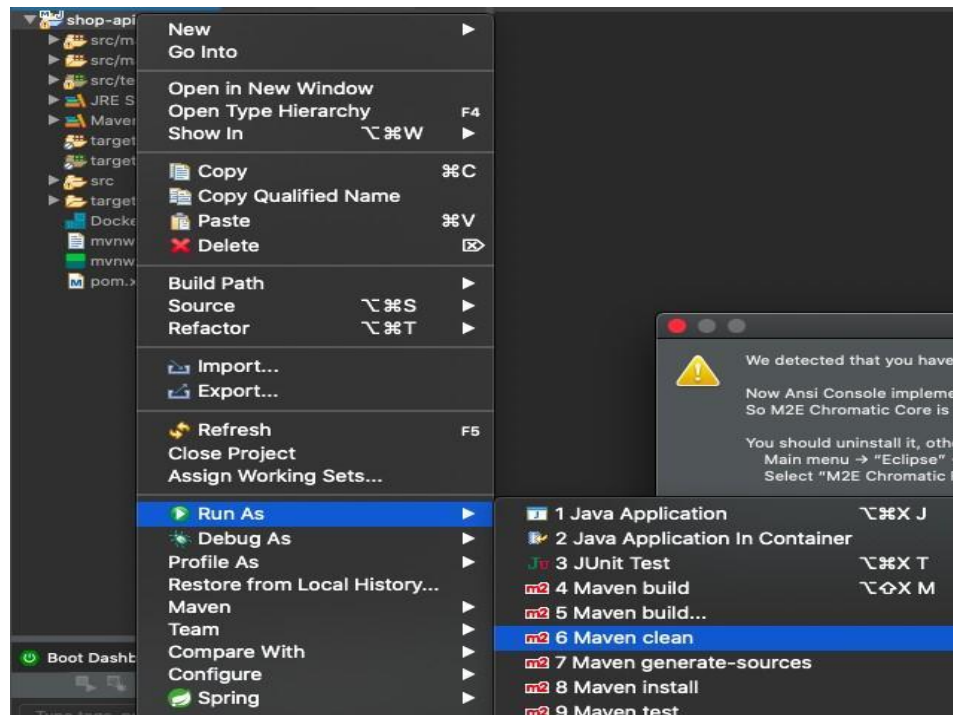
- Open the spring suit tool 4 or Eclipse Ide and Import the project.



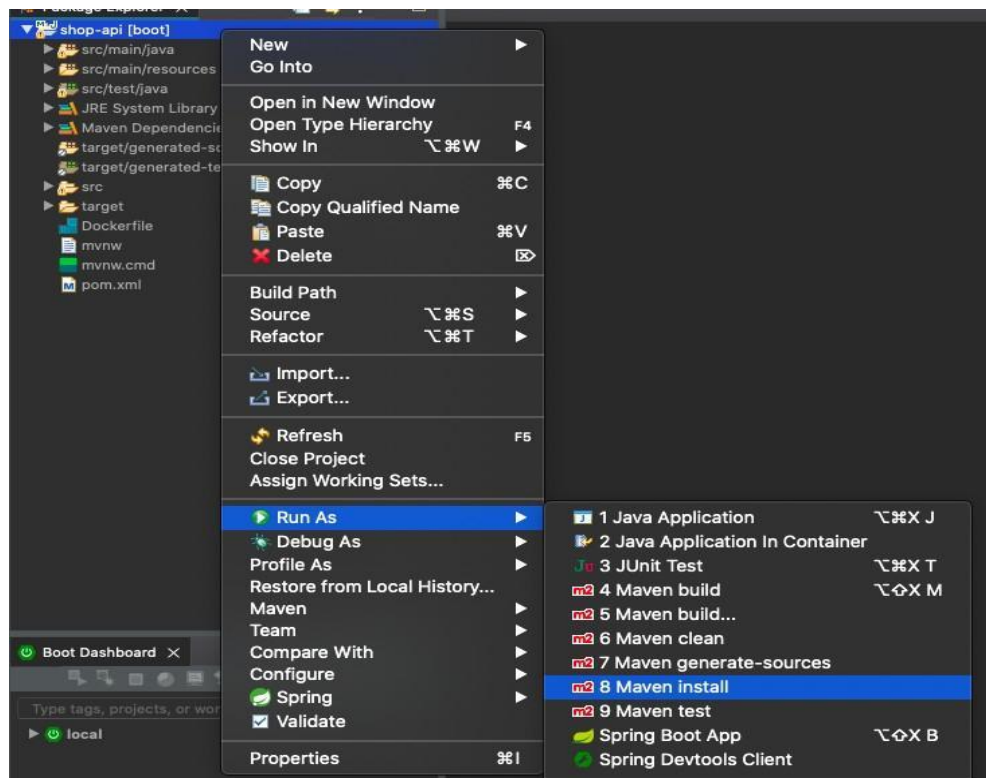
- Do Project clean and Build



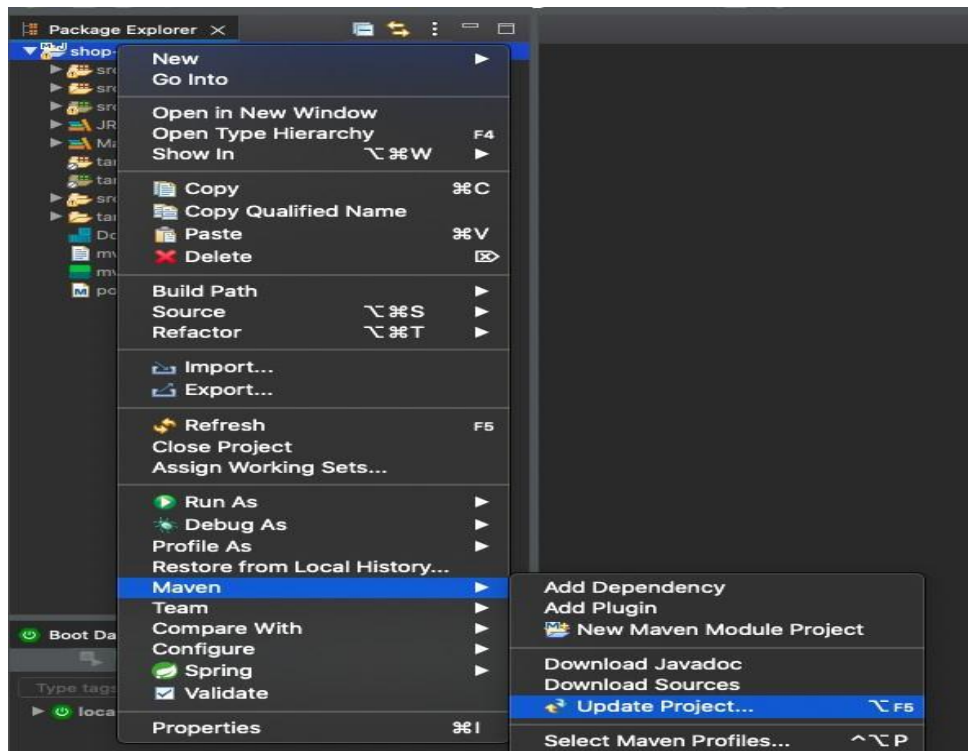
- Then do maven clean



- Maven install



- Maven Update



Now create Database and add required tables

- Install postgres SQL
- After installation serach PgAdmin in your computer.
 - Open that PgAdmin and create the database as ecommerce or any other but make sure to declare the same database name in application file
 - Open this query editor and You can run the query from eshop.sql file. Provided along with report and ppt.
- Run the query.

```

ecommerce/postgres@PostgreSQL 14*
ecommerce/postgres@PostgreSQL 14
Query History
259 INSERT INTO "public"."product_info" VALUES ('PA004', 2, '2022-06-23 23:03:26', 'Sun & Moon Wardrobe with Drawer Storage in I
260 INSERT INTO "public"."product_info" VALUES ('PA005', 2, '2022-06-23 23:03:26', 'Wakefit Eliot Engineered Wood Bookshelf, Ma
261
262 INSERT INTO "public"."product_info" VALUES ('AF001', 3, '2022-06-23 23:03:26', 'Samsung Refrigerator', 'https://iil.pepperfry.com/m
263 INSERT INTO "public"."product_info" VALUES ('AF002', 3, '2022-06-23 23:03:26', 'Robotic vacuum cleaner', 'https://iil.pepperfry.com/m
264 INSERT INTO "public"."product_info" VALUES ('AF003', 3, '2022-06-23 23:03:26', 'Crompton ceiling Fans', 'https://iil.pepperfry.com/m
265 INSERT INTO "public"."product_info" VALUES ('AF004', 3, '2022-06-23 23:03:26', 'Air Conditioner', 'https://iil.pepperfry.com/m
266 INSERT INTO "public"."product_info" VALUES ('AF005', 3, '2022-06-23 23:03:26', 'Air Purifier', 'https://iil.pepperfry.com/m
267
268
269
270
271
272 --Users
273
274 INSERT INTO "public"."users" VALUES (2147483645, true, 'Plot 2, Shivaji Nagar, Benagluru', 'admin@eshop.com', 'Admin', '$2a
275
276 CREATE SEQUENCE IF NOT EXISTS public.hibernate_sequence
277 INCREMENT 1
278 START 1

```

Now, Open Visual Studio Code

Files -> Open project -> locate the directory where you extracted the project and select the frontend folder.

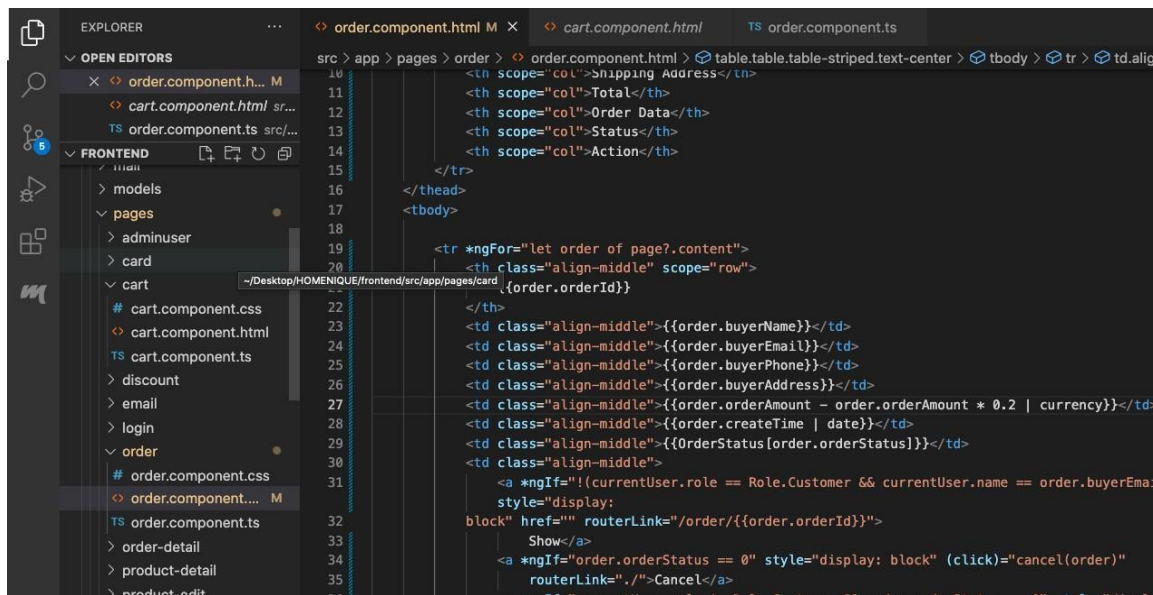
- Open terminal
- Then type **npm install**
- After that **ng serve**
- After the successful compiling you got this link in terminal
http://localhost:4200/ ** • Open this link in any browser

Source Code

```

<?signup.component.html X TS order.component.ts
~/Desktop/HOMENIQUE/frontend/src/app/pages/signup/signup.component.html
1 <h1 align="center" class="display-4 mb-5">New User? Register Here </h1>
2 <hr>
3 <div style="width:40%; margin: 25px auto">
4   <form #form="ngForm" (ngSubmit)="onSubmit()">
5     <div class="form-group">
6       <label><b>Email address</b></label>
7       <input [(ngModel)]="user.email" type="email" class="form-control form-control-lg" id="email"
8         name="email"
9         placeholder="Enter email" email required autofocus #email="ngModel">
10      <div *ngIf="email.invalid && (email.dirty ||email.touched)">
11        <div *ngIf="email.errors.required">
12          Email is required.
13        </div>
14        <div *ngIf="email.errors.email">
15          Invalid Email.
16        </div>
17      </div>
18    <div class="form-group">
19      <label><b>Name</b></label>
20      <input [(ngModel)]="user.name" type="text" class="form-control form-control-lg" id="name"
21        name="name"
22        placeholder="Your name" required #name="ngModel">
23      <div *ngIf="name.invalid && (name.dirty ||name.touched)">
24        <div *ngIf="name.errors.required">
25          Name is required.

```



ANGULAR

What is Angular?

Angular is a web application framework based on TypeScript. It's open-source and developed by Google.

That said, there have been some new developments in the Angular world that are worth a quick mention.

Angular's latest major release is 12, and it boasts loads of bug fixes making it smaller, faster & easier to use. Plus, it now supports TypeScript 4.2.3 and up. It's also starting to drop support for building libraries with View Engine in favour of Angular's compiler/runtime.

Component file in Angular

The Angular component allows us to provide a way for styling the component. This means that we can provide different CSS styling, rules, and other device-specific style configuration for a specific component. For that, the Angular component has metadata properties based on your different needs and requirements.

Module File in Angular

Module in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application. In case you are developing a website, the header, footer, left, centre and the right section become part of a module.

Service File in Angular

Services in Angular are simply typescript classes with the `@injectable` decorator. This decorator tells angular that the class is a service and can be injected into components that need that service. They can also inject other services as dependencies. As mentioned earlier, these services are used to share a single piece of code across multiple components. These services are used to hold business logic. Services are used to interact with the backend. For example, if you wish to make AJAX calls, you can have the methods to those calls in the service and use it as a dependency in files.

There are 2 users in this application:

1. User

2. Admin

n

USER SIGN-UP

Firstly, the user needs to create an account (or) sign-in in to the application

Events- `ngOnInit()` - It will execute if you refresh your browser or first initialize a component but not when other events occur

LOG-IN

Events - `ngOnInit()` - It will execute if you refresh your browser or first initialize a component but not when other events occur

`onSubmit()` - Submit directive specifies a function to run when the form is submitted

CART

Cart is a page where users can pile up what they want to buy from the website and then simply checkout by paying online

Events

`ngOnInit()` - It will execute if you refresh your browser or first initialize a component but not when other events occur
`ngAfterContentChecked()` - It is used to check whether the product is checked or not
`Selected()` - It is used to select the products into cart
`ngOnDestroy()` - gets called when a component is about to be destroyed.

`getCoupon()` - It is used to select a coupon

`remove()` - The remove is used to clear all the products in a cart

`section onchange()` - It is used to change the product checkout()

It is used to check out from product section to payment section

minusOne() - It is used to decrease the quantity of the products in a

cart
addOne() - It is used to increase the quantity of a product in a
cart

Order Details means the details relating to the Order, including without limitation, the description of Products, details of the Seller and the Buyer, date of order placement, total amount payable by the Buyer, delivery date, mode of payment, unique order number (AWB number) etc

Events ngOnInit () - It will execute if you refresh your browser or first initialize a component but not when other events occur. Major part of the development with Angular is done in the components. Components are basically classes that interact with the .html file of the component, which gets displayed on the browser.

The file structure has the app component and it consists of the following files

app.component.css

app.component.html

app.component.spec.ts

app.component.spec.ts

app.module.ts

The above files were created by default when we created new project using the angular-cli command.

If you open up the **app.module.ts** file, it has some libraries which are imported and also a declarative which is assigned the app.component as follows

```
import { BrowserModule }  
from '@angular/platform  
browser'; import { NgModule }  
from '@angular/core';  
import { AppComponent }  
from './app.component';
```

```
@NgModule({  
  declarations:  
  [  
    AppComponent  
  ],
```

```
imports: [
  BrowserModule
],
providers: [],
bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

The declarations include the AppComponent variable, which we have already imported. This becomes the parent component.

Now, angular-cli has a command to create your own component. However, the app component which is created by default will always remain the parent and the next components created will form the child components.

The command used to create the component

```
ng g component new-cmp
```

When you run the above command in the command line, you will receive the following output

```
C:\projectA4\Angular 4-app>ng g component new-cmp installing component create
src\app\newcmp\new-cmp.component.css create src\app\new-cmp\new-cmp.component.html
create
```

```
src\app\new-cmp\new-cmp.component.spec.ts create
```

```
src\app\new-cmp\new cmp.component.ts
```

```
update
```

```
src\app\app.module.t
```

```
s
```

new-cmp.component.css – css file for the new component is created. new

cmp.component.html – html file is created.

new-cmp.component.spec.ts – this can be used for unit testing.

```
import { NewCmpComponent } from './new
cmp/newcmp.component'; // includes the new-cmp
component we created
```

```
@NgModule({
```

```
declarations: [
```

```
AppComponent,
```

```
NewCmpComponent // here it is added in declarations and will behave as a child
component
```

```
],
```

```
imports: [
```

BrowserModule

```
],  
  providers: [], bootstrap: [AppComponent] //for bootstrap the AppComponent the  
  main app component is given. })  
  
export class AppModule { }
```

Now, if you go and cheque the same structure, we will get the new-cmp new folder created under the src/app folder.

The **new-cmp.component.ts** file is generated as follows import { Component, OnInit } from '@angular/core'; // here angular/core is imported .

```
@Component({  
  // this is a declarator which starts with @ sign. The component word marked in bold needs  
  to be the same.  
  selector: 'app-new-cmp', //  
  templateUrl: './new-cmp.component.html', //  
  reference to the html file created in the new component.  
  styleUrls: ['./new-cmp.component.css'] // reference to  
  the style file. })  
  export class NewCmpComponent implements OnInit  
  { constructor() { }  
  ngOnInit() {}  
}
```

If you see the above new-cmp.component.ts file, it creates a new class called NewCmpComponent, which implements OnInit. In, which has a constructor and a method called ngOnInit(). ngOnInit is called by default when the class is executed. Let us check how the flow works. Now, the app component, which is created by default becomes the parent component. Any component added later becomes the child component.

When we hit the URL in the **http://localhost:4200/** browser, it first executes the index.html file which is shown below –

```
<!doctype html>  
<html lang = "en">  
<head>  
<meta charset = "utf-8">  
<title>Angular 4App</title>  
<base href = "/">  
<meta name="viewport" content="width = device-width, initial-scale = 1">  
<link rel = "icon" type = "image/x-icon" href = "favicon.ico">  
</head>
```

```
<body>
<app-root></app-root>
</body>
</html>
```

The above is the normal html file and we do not see anything that is printed in the browser. Take a look at the tag in the body section

```
<app-root></app-root>
```

This is the root tag created by the Angular by default. This tag has the reference in the **main.ts** file.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform
browserdynamic'; import { AppModule } from
'./app/app.module'; import { environment } from
'./environments/environment';
```

```
if (environment.production) {
  enableProdMode();
}
```

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

AppModule is imported from the app of the main parent module, and the same is given to the bootstrap Module, which makes the appmodule load.

```
import { BrowserModule } from
'@angular/platform-browser'; import {
NgModule
} from '@angular/core'; import { AppComponent }
from './app.component';
import { NewCmpComponent } from './new-cmp/new-cmp.component';

@NgModule({
  declarations: [
    AppComponent,
    NewCmpComponent
  ],
  imports: [
    BrowserModule
```



```

],
providers: [],
bootstrap: [AppComponent]
})

```

```

export class AppModule { }

```

Here, the AppComponent is the name given, i.e., the variable to store the reference of the **app. Component.ts** and the same is given to the bootstrap.

```

import { Component } from '@angular/core';

```

```

@Component({
  selector: 'app-root',
  templateUrl:
    './app.component.html',
  styleUrls:
    ['./app.component.css']
})

```

```

export class AppComponent
{ title = 'Angular Project!'; }

```

Angular core is imported and referred as the Component and the same is used in the Declarator as

```

@Component({
  selector: 'app-root',
  templateUrl:
    './app.component.html',
  styleUrls:
    ['./app.component.css'] })

```

In the declarator reference to the selector, **templateUrl** and **styleUrl** are given. The selector here is nothing but the tag which is placed in the index.html file that we saw above.

The class AppComponent has a variable called title, which is displayed in the browser.

The **@Component** uses the templateUrl called

```
app.component.html which is as follows<!--The content
below is only a placeholder and can be replaced.-->
<div style="text-align:center">
<h1>
Welcome to {{title}}.
</h1>
</div>
```

It has just the html code and the variable title in curly brackets. It gets replaced with the value, which is present in the **app.component.ts** file. This is called binding. We will discuss the concept of binding in a subsequent chapter.

Now that we have created a new component called **new-cmp**. The same gets included in the **app.module.ts** file, when the command is run for creating a new component.

app.module.ts has a reference to the new component created.

new-cmp.component.ts

```
import { Component, OnInit } from
'@angular/core'; @Component({
selector: 'app-new-cmp',
templateUrl:
'./newcmp.component.html',
styleUrls:
['./newcmp.component.css']
})
```

```
export class NewCmpComponent
implements OnInit { constructor() {}
ngOnInit() {} }
```

Here, we have to import the core too. The reference of the component is used in the declarator.

The declarator has the selector called **app-new-cmp** and the **templateUrl** and **styleUrl**.

The .html called **new-cmp.component.html** is as follows

```
<p> new-comp works! <p>
```

we have the html code, i.e., the p tag. The style file is empty as we do not need any styling at present. But when we run the project, we do not see anything related to the new component getting displayed in the browser. Let us now add something and the same can be seen in the

browser later. The selector, i.e., **app-new-cmp** needs to be added in the **app.component.html** file as follows

```
<!--The content below is only a placeholder and can  
be  
replaced.--> <div  
style="text-align:center">  
<h1>  
Welcome to {{title}}.  
</h1>  
</div>  
  
<app-new-cmp></app-new-cmp>
```

When the **<app-new-cmp></app-new-cmp>** tag is added, all that is present in the .html file of the new component created will get displayed on the browser along with the parent component data. Let us see the **new component.html** file and the **new-cmp.component.ts** file.

New-cmp.component.ts

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-new  
cmp',  
  templateUrl:  
    './newcmp.component.  
html', styleUrls:  
    ['./newcmp.component.  
css']  
})  
  
export class NewCmpComponent  
implements OnInit { new component =  
  "Entered in new component created";  
  constructor() {} ngOnInit() {}  
}
```

In the class, we have added one variable called new component and the value is “**Entered in new component created**”.

The above variable is bound in the **.new-cmp.component.html** file as follows –

<p>

{{newcomponent}}

</p>

<p> new-cmp works! </p>

Now since we have included the <app-new-cmp></app-new-cmp> selector in the **app.component**

.html which is the .html of the parent component, the content present in the new component .html file (new-cmp.component.html) gets displayed on the browser.

Homenique

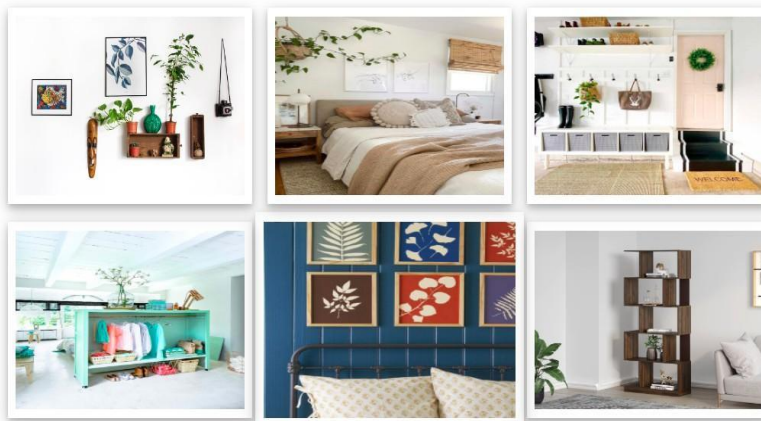
FURNITURE BED&MATTRESSES STORAGE APPLIANCES

[CART](#) [LOGIN](#) [SIGNUP](#)

Furniture



What are you looking for?



Spring boot – Backend

Dependency added in POM –

- Lombok
- spring security
- spring Jpa
- spring -web
- Common csv
- Jwt – (disable – for now)
- Spring-boot starter mail

LomBok

Project Lombok is a Java library tool that generates code for minimizing boilerplate code. The library replaces boilerplate code with easy-to-use annotations.

- @Getter,
- @Setter
- @NoArgsConstructor,
- @AllArgsConstructor
- @Data
- @NotNull

Spring security

Spring Security Web dependencies are used for providing features related to the web security of an application like:

Restricted access to URLs in a Servlet Container environment • Managing the Spring Security filters related to web

Common csv- Apache Commons CSV is a java library that reads and writes files in Comma

Separated Value (CSV) format. This library can write in all supported variations of CSV files like MSEXcel, Informix, Oracle, MySQL and TSV(Tab-separated values). You can also create custom CSV formats using its fluent API.

Order-main

Entities

- Entities Classes are used to initialize the variables and make a model format for table •
- User
- Product-Info
- Wishlist
- Product order
- Product category
- discount
- cart

User Entity – This class entity used for collecting data from the user for registration purpose.

public class User implements a seriliazeble - Serializability of a class is enabled by the class implementing the java.io.Serializable interface.

Annotation used –

- @Entity – for make class to table
- @Data – it contains getter setter to string method itself
- @noArgsConstructor – for no argument constructor
- @ID – make a primary key for table
- @NotEmpty – The annotated element must not be null nor empty.

Supported types are: Private Variable -ex

private Long id;

- ID
- Name
- Email Password
- Phone
- Address
- Active status
- Role - For_Customer
- Cart

ProductInfo – Class entity used for the product information

Annotation used

- @Entity @Id
- @DynamicUpdate - For updating, should this entity use dynamic sql generation where only changed columns get referenced in the prepared sql statement?

@min - The annotated element must be a number whose value must be higher or equal to the specified minimum.

- @Column Default - Identifies the DEFAULT value to apply to the associated column via DDL.
- @creation TimeStamp - Marks a property as the creation timestamp of the containing entity. The property value will be set to the current VM date exactly once when saving the owning entity for the first time.
- @Update TimeStamp - Marks a property as the creation timestamp of the containing entity. The property value will be set to the current VM date exactly once when saving the owning entity for the first time.

Variables

- productid
- productDescription
- productPrice
- productStock
- productStatus
- productIcon
- createTime
- updateTime
- categoryType

WishList Entity class -

Class use for add product into wishlist table

Annotation

used- @id

-

@GeneratedValue – GenerationType.AUTO - used for autogenerated values o @Column – Specifies the mapped column for a persistent property or field

@ManyToOne - Specifies a single-valued association to another entity class that has many-to-one multiplicity . The ManyToOne annotation used within an embeddable class to specify a relationship from the embeddable class to an entity class.

@JoinColumn - Specifies a column for joining an entity association or element collection. If the JoinColumn annotation itself is defaulted, a single join column is

assumed and the default values apply.

@OneToOne - Specifies a single-valued association to another entity that has one-to-one multiplicity. I.e. one entity interlink with another entity

The FetchType EAGER –

Eager strategy is a requirement on the persistence provider runtime that data must be eagerly fetched.

Variables-

- ID
- User user (entity) – One to one Relationship
- ProductInfo productInfo (Entity) – manytoOne relationship

Product Order - Entity class used for the placed Product order it has manytoOne relationship with

Cart entity and Order main entity

Annotation -

- @Entity
- @Data
- @Id
- @GeneratedValue
- @ManyToOne
- @JoinColumn
- @NotNull
- @NotEmpty
- @Min
- @JsonIgnore - Annotation is usually used just a like a marker annotation, that is, without explicitly defining 'value' argument (which defaults to true): but argument can be explicitly defined. This can be done to override an existing `JsonIgnore` by explicitly defining one with 'false' argument: either in a sub-class, or by using "mix-in annotations".

Variable

- Id
- Cart cart(Entity with manytoOne relationship)
- OrderMain(entity with
@many to one Relationship)
- productid

- prooductName
- productDescription
- productIcon
- CategoryType
- productPrice
- productStock
- count

Method

To string method for data display. Getter and setter methods

Product -Category

this entity class used for distinguish the product into different categories.

Annotation –

- @Entity
- @Data
- @Dynamic Update
- @Id
- @Generated value
- @naturalID - This specifies that a property is part of the natural id of the entity.

Methods -

Getter setter methods and no Argument Constructor and Required Argument Constructor **Variables**

- CategoryId
- categoryName
- CategoryName
- categoryType
- createTime(Date as DataType)
- updateTime **Order-Main –**

this entity class used for the placed the order with productINorder checkout and deatials of the user **Annotation**

- @Entity
- @Data
- @Dynamic update
- @Generated value
- @OneToMany
- @NotEmpty

- @CreationTimeStamp
- @UpdateTimeStamp
- @ColumnDefault – if the value is not given then its takes default column value

Methods – Getter setter , no- Argument Constructor, Required-Argument

Constructor **Variables**

- Order
- ID
- ProductOrder set(entity with OneToMany Relationship)
- BuyersEmail(current User Email)
- BuyersPhone
- BuyersAddress
- OrderAmount
- OrderStatus
- CreatTimeStamp
- updateTimeStamp

Discount - this Entity class used for the making discount coupon

Annotation

- @Entity
- @Data
- @ID
- @Table
- @NoArgsData

Methods – Getter Setter

Cart – this class entity used for the add product into cart by each user

Annotation -

- @Entity
- @Data
- @Dynamic Update
- @Id
- @Generated value
- @OneToMany
- @JsonIgnore
- @MapsId - The value element specifies the attribute within a composite key to which the relationship attribute corresponds. If the entity's primary key is of the same Java type as the primary key of the entity referenced by the relationship, the value attribute is not specified.

Variables

- Id
- User user(Entity with OnetoOne Relationship) mainly for is user has its separate cart reference column is email)
- Set ProductOrder (entity with OneToMany mapping)

Methods

- @Data
- @toString Method
- @NoArgsConstrutor.

Repository

Repository - A repository is **a mechanism for encapsulating storage, retrieval, and search behaviour which emulates a collection of objects**. It is a specialization of the

- @Component annotation allowing for implementation classes to be autotected through classpath scanning.

Annotation Used in All classes -

- @Repository
- @Repository Annotation is a specialization of
- @Component annotation which is used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects. Though it is a specialization of
- @Component annotation, so Spring Repository classes are autotected by spring framework through classpath scanning. This annotation is a general-purpose stereotype annotation which very close to the DAO pattern where DAO classes are responsible for providing CRUD operations on database tables.

All classes Extends a JpaRepository

JpaRepository is a **JPA (Java Persistence API) specific extension of Repository**. It contains the full API of CrudRepository and PagingAndSortingRepository. So, it contains API for basic CRUD operations and also API for pagination and sorting.

Repository Classes -

- CartRepository – used for performing CRUD Operation on Cart Entity
- DiscountRepository – used for performing CRUD Operation on

DiscountEntity.

- OrderRepository - used for performing CRUD Operation on OderEntity
- ProductCategoryRepository - used for performing CRUD Operation on ProductCategory entity
- ProductInOrderRepository - used for performing CRUD Operation on ProductOrder
- Entity
- UserRepository- used for performing CRUD Operation on User Entity
- WishListCustomRepository- used for performing CRUD Operation on • WishListCustom Entity
- WishListRepository - used for performing CRUD Operation on WishList Entity.

Controller class - In Spring Boot, the controller class is responsible for processing incoming REST API requests, preparing a model, and returning the view to be rendered as a response.

Annotation -

- @Controller annotation is a specialization of the generic stereotype
- @Component annotation, which allows a class to be recognized as a Springmanaged component.
- @Controller annotation extends the use-case of
- @Component and marks the annotated class as a business or presentation layer
- @RestController – The @RestController annotation in Spring is essentially just a combination of @Controller and @ResponseBody. This annotation was added during Spring 4.0 to remove the redundancy of declaring the @ResponseBody annotation in your controller
- @RequestMapping -Annotation for mapping web requests onto methods in requesthandling classes with flexible method signatures.
- @CrossOrigin - Annotation for permitting cross-origin requests on specific handler classes and/or handler methods. Processed if an appropriate HandlerMapping is configured.
- @AutoWired - annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.
- @GetMapping - Annotation for mapping HTTP GET requests onto specific handler methods. Specifically, @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

- `@PostMapping` - Annotation for mapping HTTP POST requests onto specific handler methods. Specifically, `@PostMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.POST)`.
- `@PutMapping` - Annotation for mapping HTTP PUT requests onto specific handler methods. Specifically, `@PutMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.PUT)`.
- `@DeleteMapping` - Annotation for mapping HTTP DELETE requests onto specific handler methods. Specifically, `@DeleteMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.DELETE)`.
- `@PathVariable` - Annotation which indicates that a method parameter should be bound to a URI template variable. Supported for `RequestMapping` annotated handler methods.
- `@RequestBody` - Annotation indicating a method parameter should be bound to the body of the web request. The body of the request is passed through an `HttpMessageConverter` to resolve the method argument depending on the content type of the request. Optionally, automatic validation can be applied by annotating the argument with `@Valid`.
- `@RequestParam` - Annotation which indicates that a method parameter should be bound to a web request parameter.

Controller Classes -

- `CartController`
- `CategoryController`
- `CSVController`
- `DISCountController`
- `EmailController`
- `OrderController`
- `ProductController`
- `ProductController`
- `USerController`
- `WishListController`

Service Class –

A service interface is a published interface used to invoke a service. An interface can be implemented using any number of technologies.

In service Interface We Just Write a Method and thir logic code will be Write in Service Impl.

Service Interface –

- CartService
- CategoryService
- DiscountService
- OrderService
- ProductService
- UserService

SERVICE IMPL CLASS

The service implementation should be a Spring bean (it either has to have a `@Component` or `@Service` annotation, or should be defined in a Spring XML configuration file), so that Spring will find it and register it in the Spring application context.

The main function of the Service Impl is to provide services to the Admins and the Users. Cart Service Impl class: - In this class we will able to add the products to the cart. • `@Service` annotation: These class files are used to write business logic in a different layer, separated from `@RestController` class file.

`@Service`

`@Autowired`

`@Override`

`@Transactional`

Product In Order Service Impl: - In this it will show that the product is ordered or not and we will able to increase the number.

`@Service`

`@Autowired`

`@Override`

`@Transactional`

Product Service Impl class: - In this we will able to add the details of the products.

`@ Service`

`@Autowired`

@Override

@Transactional

User Service Impl class: - In this the user can able to set the password for the application or the account of the website.

@ Service

@DependsOn: We should use this annotation for specifying bean dependencies. Spring guarantees that the defined beans will be initialized before attempting an initialization of the current bean.

@Autowired

@Override

@Transactional

WishList Service class: - In this creation of the wishlist bar or menu, and able to add the products to it.

@ Service

@Transactional

@Autowired

SECURITY JWT AUTHENTICATION

JWT, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. The main function of this is to provide the security User data and the customes data that have stored in the server.

1. Jwt Entry Point class: - taking the entry of the users.

- @Component: is an annotation that allows Spring to automatically detect our custom beans.
- @Override

2. Jwt Filter class: - checks if the incoming request has a valid JSON Web Token (JWT).

- @Component
- @Autowired
- @Override

3. Jwt Provider class: - It will allow to see the user details.

- @Component

- @Value

Other Annotation Used are: -

- @Qualifier annotation: we can eliminate the issue of which bean needs to be injected.
- @Configuration annotation: indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
- @EnableWebSecurity annotation: is annotated at class level with @Configuration annotation to enable web securities in our application defined by Web Security Configured implementations.
- @Bean annotation: is applied on a method to specify that it returns a bean to be managed by Spring context.

REQUEST

Defines a builder that adds a body to the response entity. Defines a builder that adds headers to the request entity. Request Entity initialized with a URI template and variables instead of a URI.

Login Form class: - It is used to create the login details of the customers.

RESPONSE

Response Entity represents the whole HTTP response: status code, headers, and body. As a result, we can use it to fully configure the HTTP response. If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

This is the response class which are used to see the responses of the customers which will stores the data. We need responses of only these categories.

1. Category Page class
2. Jwt Response class
3. Product Info Response class
4. User List Response

class Annotation Used: -

- @Data: It is a convenient shortcut annotation that bundles the features of @ToString.

Application.yml

- `@Data`: It is a convenient shortcut annotation that bundles the features of in this file we defined some method for connecting Database to the spring.

Like databases URL , dialect username password properties mail

etc Properties

`## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)`

`spring: datasource:`

`driver-class-name: org.postgresql.Driver username: postgres password:`

`rashmum20 url: jdbc:postgresql://localhost:5432/homenique`

`platform: postgres initialization-mode: always continue-on-error: true`

`mail:`

`default-encoding: UTF-8`

`host: smtp.gmail.com`

`username:`

`postgres`

`password:`

`vwiqtpokkynwhkkc`

`port: 587`

`properties:`

`mail:`

`smtp:`

`auth: true`

`starttls:`

`enable:`

`true`

`protocol: smtp test`

`connection: false`

jpa:

show-sql: false #

Generate-ddl: false

hibernate:

ddl-auto: none

database: postgresql

properties:

hibernate: temp:

use_jdbc_metadata_defaults: false; database-platform:
org.hibernate.dialect.PostgreSQL9Dialect

queries:

users-query: select email, password, active from users where email=?
roles query: select email, role from users where email=?

server:

servlet:

contextPath: /api

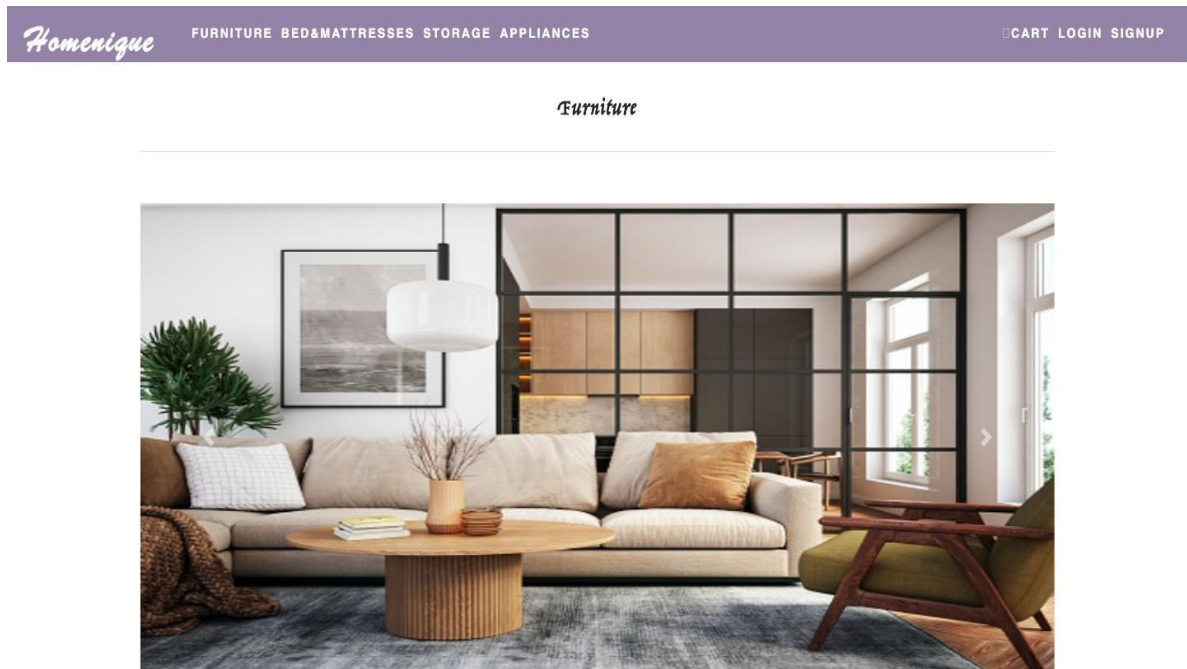
jwtSecret: me.zhulin

jwtExpiration:

86400

FINAL RESULT

- Dashboard Page



- Sign In page

The screenshot shows the Homenique Sign In page. The header is a purple bar with the Homenique logo on the left, navigation links (FURNITURE, BED&MATTRESSES, STORAGE, APPLIANCES) in the center, and a shopping cart icon with links to CART, LOGIN, and SIGNUP on the right. Below the header, the word 'Log In' is centered. The main content area contains a sign-in form with the following elements: a label 'Email address' above a text input field with the placeholder 'Enter email'; a label 'Password' above a password input field with the placeholder 'Password'; a checkbox labeled 'Remember me' and a link 'Sign Up' to the right; and a green 'Sign In' button at the bottom. The form is centered on the page.

- Signup Page

Homenique

FURNITUREBED & MATTRESSESSTORAGEAPPLIANCES

CARTLOGINSIGNUP

New User? Register Here

Email address

Name


Password

Phone

Address

Sign Up

- Cart Page



FURNITURE

BED&MATTRESSES

STORAGE

APPLIANCES



CART WISHLIST

ORDERS

RASHI

SIGNOUT

My Cart

Photo	Name	Price	Quantity	Subtotal	Action
	Petunia Teak Wood Coffee Table With Marble Top In Rustic Teak Finish	\$36.00	<div>- 1 +</div>	\$36.00	Remove
	Corner Wall Shelves	\$100.00	<div>- 1 +</div>	\$100.00	Remove

Select Coupon

Total: \$136.00

Checkout

Conclusion

By performing or following the above-mentioned methodology we can conclude that we can create a web page using the Angular, Spring boot, Node-JS, MySQL, Eclipse etc. In this project, we have created the page of name “**HOMENIQUE**” in which a user can buy the house hold items like decorating things and many other. We are needed updated or latest version software for smooth working of the project. Both frontend and backend needs to run in a frequent manner for to have the page with user interface flexibility. At the end we can say that web page is created by following the path regulations and shown the output as required.

Reference