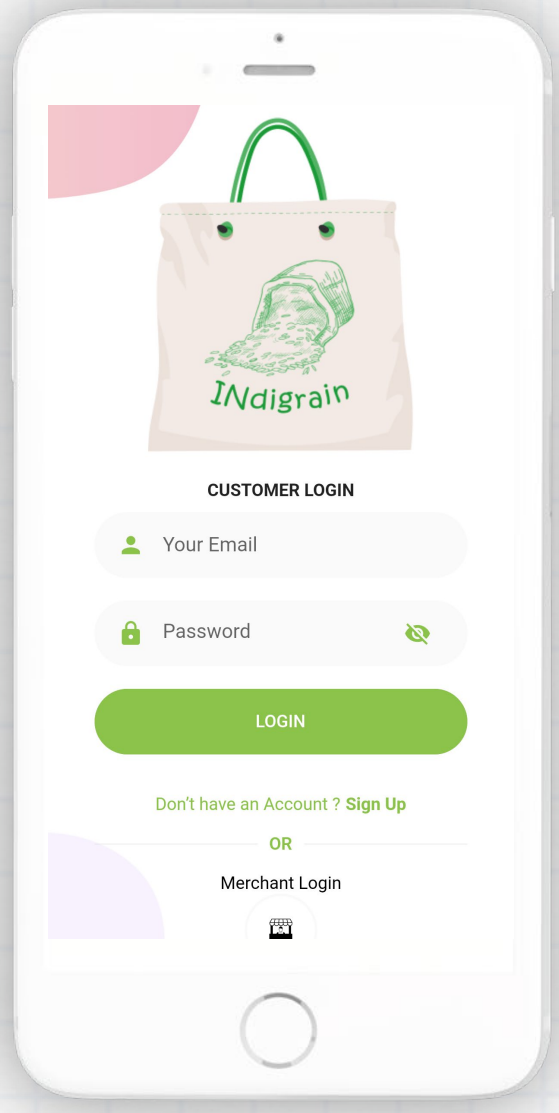
Build an E-marketplace mobile application.



Introduction to the course:	Building customer-facing e-marketplace application from scratch using Flutter framework, Java, and the Spring Boot environment to create a robust backend supporting payments with Razor pay payment gateway.
What does this course aim to achieve?	In this course, you'll build a full-stack e-marketplace application which require full-stack development, involving a backend to handle users, inventory, and payments, and a frontend for customers to view products, manage their cart, and checkout using razor pay. User profiles will also store order history.
What is being built in this course	E-marketplace mobile application integrated with Razor pay payment gateway.
How is it being tested	1 Install the generated .APK file onto an Android device. 2 Open the postman REST client and test the APIs. 3 Make a test payment to verify the Razor pay payment gateway integration
Course Prerequisites	Basic knowledge of Dart language Basic Java programming

Contents

Prerequisites:

TOPIC	LINK
Introduction to Flutter	https://docs.flutter.dev/
Create a Flutter Project from Scratch	Flutter Tutorial Part 1: Build a Flutter App From Scratch - DZone
Add to the app using the Pub spec file	https://docs.flutter.dev/development/tools/pubspec/
Flutter Logo:	https://www.geeksforgeeks.org/flutter-flutterlogo-widget/
Flutter Toaster	https://pub.dev/packages/fluttertoast
Flutter Drawer	https://docs.flutter.dev/cookbook/design/drawer
Shared preferences	https://blog.logrocket.com/using-sharedpreferences-in-flutter-to-store-data-locally/
Widgets	https://docs.flutter.dev/development/ui/widgets-intro/
Container	https://api.flutter.dev/flutter/widgets/Container-class.html
Row and Column	https://www.geeksforgeeks.org/row-and-column-widgets-in-flutter-with-example/
Expanded Widget	https://api.flutter.dev/flutter/widgets/Expanded-class.html
Floating Action Button	https://api.flutter.dev/flutter/material/FloatingActionButton-class.html
List Tile Widgets	https://api.flutter.dev/flutter/material/ListTile-class.html
Card Widget	https://api.flutter.dev/flutter/material/Card-class.html
List view	https://api.flutter.dev/flutter/widgets/ListView-class.html
Grid View	https://api.flutter.dev/flutter/widgets/GridView-class.html
Custom Fonts	https://docs.flutter.dev/cookbook/design/font
Material Icons	https://docs.flutter.dev/development/ui/widgets/material
Making Responsive App	https://docs.flutter.dev/release/breaking-changes/buttons
Stateful Widget	https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html

Stateless Widget	https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html
Text Field	https://docs.flutter.dev/cookbook/forms/text-field-changes/
API	https://www.tutorialspoint.com/flutter/flutter_accessing_rest_api.htm/
JSON	https://docs.flutter.dev/development/data-and-backend/json/ https://blog.logrocket.com/how-parse-json-strings-flutter/ https://medium.com/flutter-community/flutter-part-4-fetch-data-from-the-network-1b5949d84d44/
Cart Feature	https://www.dbestech.com/tutorials/how-to-remove-an-item-from-dart-list-flutter/

Components

Components	Quantity
Window 10 / Linux 64-bit Pc or Laptop • RAM: Min: 8GB Recommended: 16GB • Free Disk Space:	1 No

Min: 10GB Recommended: 30GB • Screen Resolution: Min: 1280x800px Recommended: 1920x1080px	

Software

Software	Download Link
Android Studio IDE and SDK	https://developer.android.com/studio
Flutter SDK	https://docs.flutter.dev/get-started/install
Java SE 13	https://www.oracle.com/java/technologies/javase/jdk13-archive-downloads.html
Apache NetBeans	https://netbeans.apache.org/download/index.html
Resin	www.caucho.com/resin-4.0/admin/starting-resin.xtp
PostgreSQL	https://www.postgresql.org/download/

Part- A Build the frontend of E-marketplace mobile application.

Building the frontend of E-marketplace mobile application using Flutter framework.

Installing Flutter and Android Studio:

1.System requirements:

To install and run Flutter, your development environment must meet these minimum requirements:

Operating Systems: Windows 7 SP1 or later (64-bit), x86-64 based.

Disk Space: 1.64 GB (does not include disk space for IDE/tools).

Tools: Flutter depends on these tools being available in your environment.

Windows PowerShell 5.0or newer (this is pre-installed with Windows 10)

Git for Windows 2.x, with the **Use Git from the Windows Command Prompt** option. If Git for Windows is already installed, make sure you can run git commands fromthe command prompt or PowerShell.

2.Get the Flutter SDK:

1. Download the following installation bundle to get the latest stable release of the
2. FlutterSDK: https://storage.googleapis.com/flutter_infra_release/releases/stable/windows/ flutter_windows_2.10.3-stable.zip
3. Extract the zip file and place the contained flutterin the desired installation location forthe Flutter SDK (for example, C:\Users\<your-username>\Documents).
4. If you don't want to install a fixed version of the installation bundle, you can skip steps 1 and 2. Instead, get the source code from the https://github.com/flutter/flutter on GitHub, andchange branches or tags as needed. For example: git clone https://github.com/flutter/flutter.git -b stable

You are now ready to run Flutter commands in the Flutter Console.

3.Update your path.

If you wish to run Flutter commands in the regular Windows console, take these steps to addFlutter to the PATHenvironment variable:

- From the Start search bar, enter 'env' and select **Edit environment variables for youraccount**.
- Under **User variables** check if there is an entry called **Path**:
- If the entry exists, append the full path to flutter\bin using;as a separator from existing values.
- If the entry doesn't exist, create a new user variable named Pathwith the full path to flutter\binas its value.

You must close and reopen any existing console windows for these changes to take effect.

4.Run flutter doctor.

From a console window that has the Flutter directory in the path (see above), run the followingcommand to see if there are any platform dependencies you need to complete the setup:

C:\src\flutter>flutter doctor

This command checks your environment and displays a report of the status of your Flutter installation. Check the output carefully for other software you might need to install or further tasks to perform.

For example:

[-] Android toolchain - develop for Android devices.

- Android SDK at D:\Android\sdk

✗ **Android SDK is missing command line tools; download from https://goo.gl/XxQghQ**

- Try re-installing or updating your Android SDK, visit https://docs.flutter.dev/setup/#android-setup for detailedinstructions.

The following sections describe how to perform these tasks and finish the setup process. Once youhave installed any missing dependencies, you can run the flutter doctor command again to verify that you've set everything up correctly.

5.Install Android Studio

1. Download and install **Android studio.**
2. Start Android Studio and go through the 'Android Studio Setup Wizard'. This installs thelatest Android SDK, Android SDK Command-line Tools, and Android SDK Build-Tools, which are required by Flutter when developing for Android.
3. Run flutter doctor to confirm that Flutter has located your installation of AndroidStudio. If Flutter cannot locate it, run flutter config – android **https://developer.android.com/studio**-studio-dir.

<directory>to set the directory that Android Studio is installed to.

6.Set up an Android device:

To prepare to run and test your Flutter app on an Android device, you need an Android devicerunning Android 4.1 (API level 16) or higher.

1. Enable **Developer options** and **USB debugging** on your device. Detailed instructions areavailable in the Android Documentation.
2. Windows-only: Install the Google USB driver.
3. Using a USB cable, plug your phone into your computer. If prompted on your device,authorize your computer to access your device.
4. In the terminal, run the flutter devicescommand to verify that Flutter recognizesyour connected Android device. By default, Flutter uses the version of the Android SDKwhere your adb tool is based. If you want Flutter to use a different installation of the Android SDK, you must set the ANDROID_SDK_ROOT environment variable to that installation directory.

7.Set up the Android emulator.

To prepare to run and test your Flutter app on the Android emulator, follow these steps:

- Enable VM acceleration on your machine.
- Launch **Android Studio**, click the **AVD Manager** icon, and select **Create Virtual Device…**
- In older versions of Android Studio, you should instead launch **Android Studio >Tools > Android > AVD Manager** and select **Create Virtual Device….** (The **Android** submenu is only present when inside an Android project.)

- If you do not have a project open, you can choose **Configure > AVD Manager** andselect **Create Virtual Device…**
- Choose a device definition and select **Next**.
- Select one or more system images for the Android versions you want to emulate and select.
  Next. An *x86* or *x86_64* image is recommended.
- Under Emulated Performance, select **Hardware - GLES 2.0** to enable hardware acceleration.
- Verify the AVD configuration is correct and

  select **Finish**.

  For details on the above steps, see Managing

  AVDs.
- In Android Virtual Device Manager, click **Run** in the toolbar. The emulator starts up anddisplays the default canvas for your selected OS version and device.

8.Agree to Android Licenses

Before you can use Flutter, you must agree to the licenses of the Android SDK platform. This stepshould be done after you have installed the tools listed above.

1. Make sure that you have a version of Java 8 installed and that your JAVA_HOME environment variable is set to the JDK's folder.

Android Studio versions 2.2 and higher come with a JDK, so this should already be done.

2. Open an elevated console window and run the following command begin.

3. signing licenses.flutter doctor –android-licenses.
4. Review the terms of each license carefully before agreeing to them.
5. Once you are done agreeing with licenses, run flutter doctoragain to confirm thatyou are ready to use Flutter.

9.Install the Flutter and Dart plugins.

The installation instructions vary by platform.

Mac

Use the following instructions for macOS:

1. Start Android Studio.
2. Open plugin preferences (**Preferences > Plugins** as of v3.6.3.0 or later).
3. Select the Flutter plugin and click **Install**.
4. Click **Yes** when prompted to install the Dart plugin.
5. Click **Restart** when prompted.

Linux or Windows

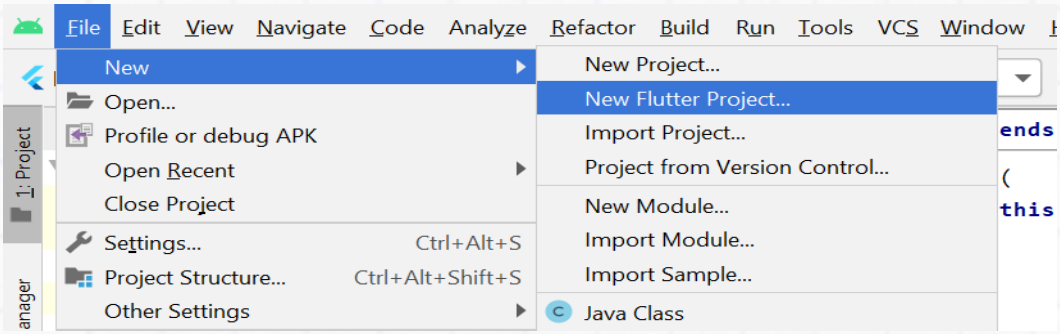
Use the following instructions for Linux or Windows:

1. Open plugin preferences (**File > Settings > Plugins**).
2. Select **Marketplace**, select the Flutter plugin and click **Install**.

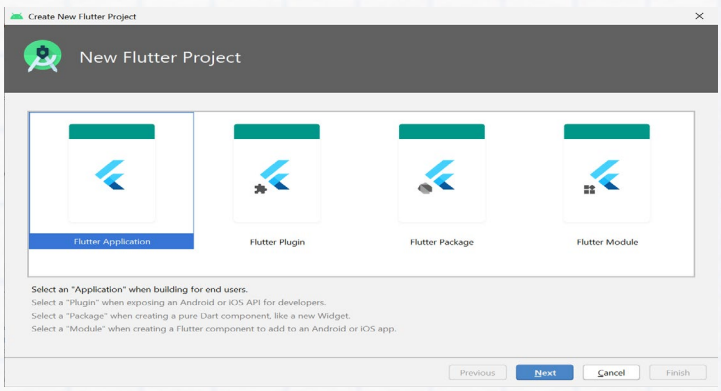
10.Configure Android Studio for Flutter Development:

After installing Dart and Flutter plugins create a flutter app to check if it is workingproperly or not, to do so follow the steps mentioned below:

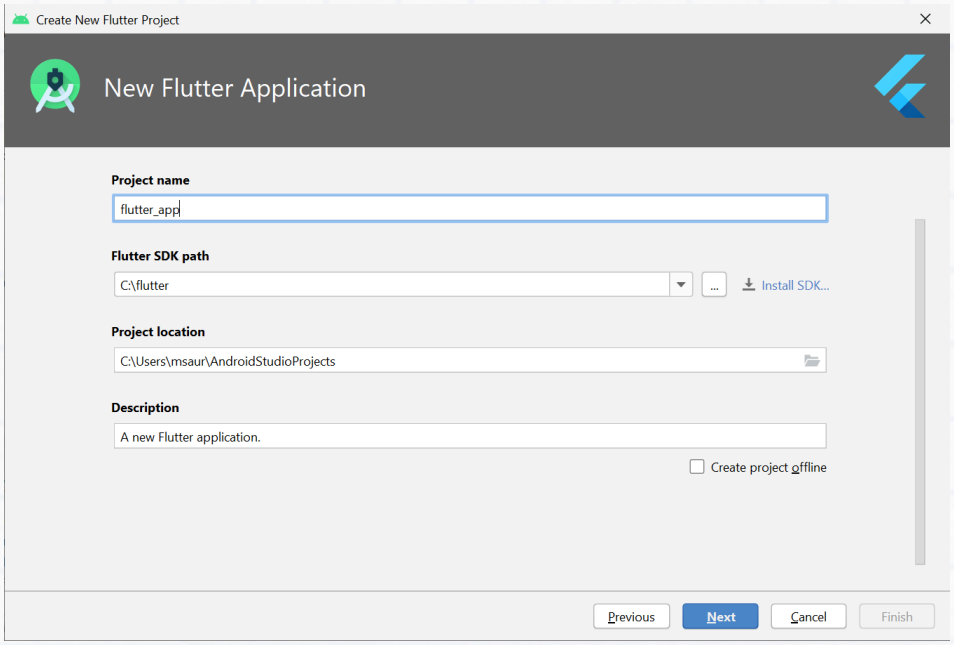
1: Open the **IDE** and select Start a **new Flutter project**



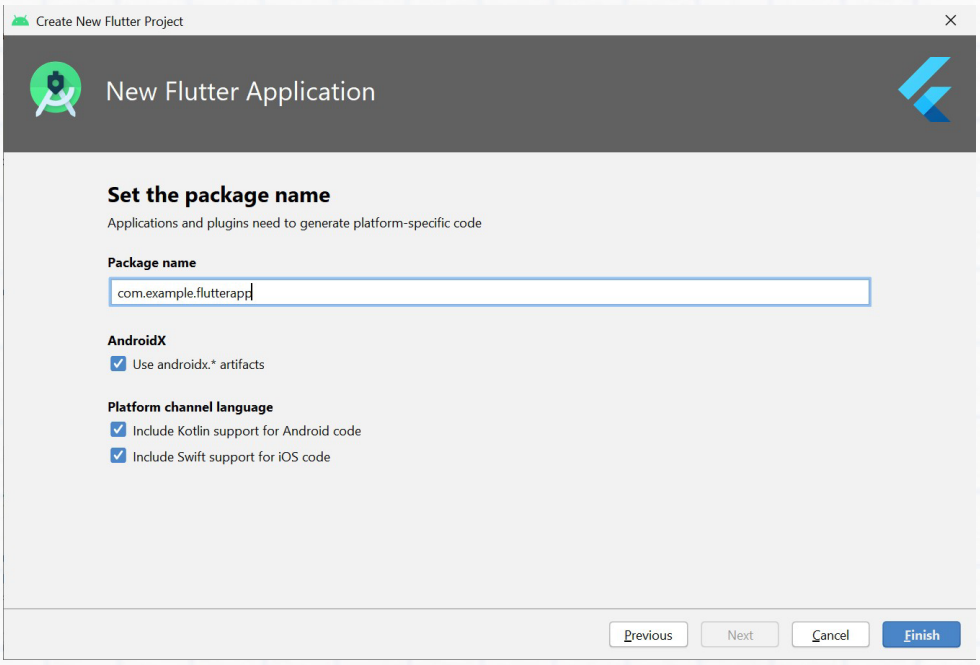
2: Select the Flutter Application as the project type. Then click Next.



3: Verify the **Flutter SDK path** specifies the SDK's location **(select Install SDK... if the textfield is blank)**.



4: Enter a project name (for example, *myapp*). Then click **Next**.



5: Click **Finish**.

6: Wait for Android Studio to install the SDK and create the project.

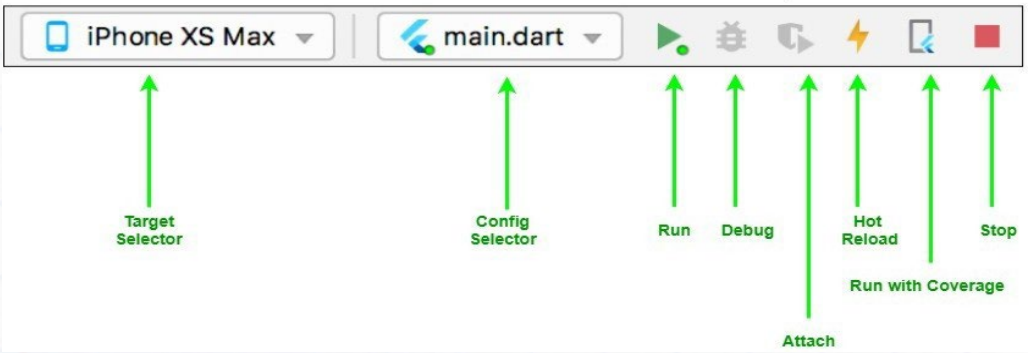
Note: When creating a new Flutter app, some Flutter IDE plugins ask for a company domain namein reverse order, something like co. Example. The company domain name and project name are used together as the package name for Android (the Bundle ID for iOS) when the app is released. If you think that the app might be released, it's better to specify the package name now. The packagename can't be changed once the app is released, so make the name unique.

The above steps create a Flutter project directory called *flutter_app* that contains a simple demo appthat uses Material Components.

11.Running the application:

Follow the below steps to run the flutter application that was structured above:

1: Locate the main Android Studio toolbar:



Step 2: In the **target selector**, select an Android device for running the app. If none are listed as available, select **Tools> Android > AVD Manager** and create one there. For details, see **ManagingAVDs**.

Step 3: Click the run icon in the toolbar or invoke the menu item

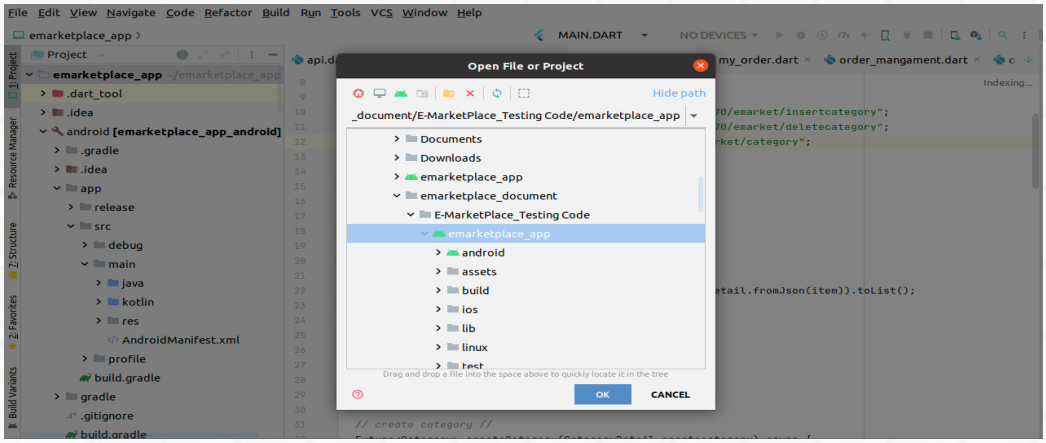
Run > Run.After the app build completes, you'll see the starter

app on your device.

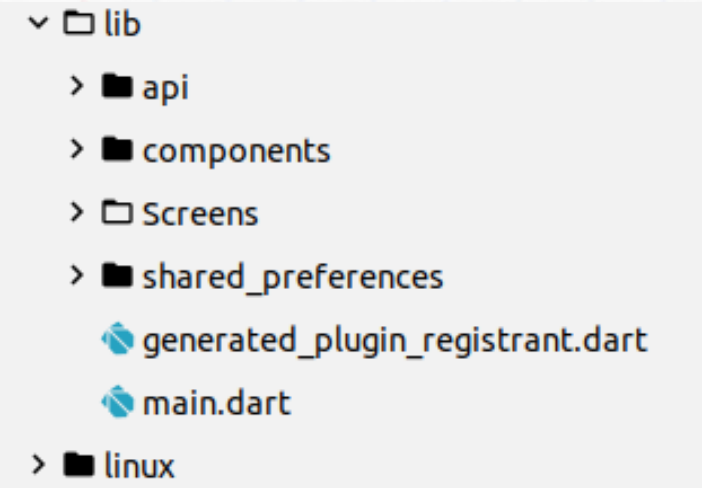
Creating a responsive E-marketplace mobile application with following features for merchant and customer	
Merchant Merchant login page Merchant home page Merchant category list page Merchant category details page Merchant categories add page. Merchant product list page Merchant product details page Merchant products add page. Merchant products edit page. Merchant order received page	**Customer** Customer login page Customer registration page Customer home page My profile page. Product display page Cart page Payment page My order page. Order summary page

Import the Project

1) Open -Android Studio->select unzip emarketplace_app file->open project.



Inside lib Folder



API: Inside api/api.dart it contains api call details all page

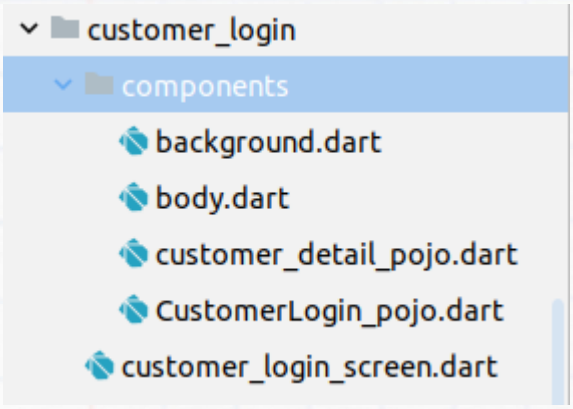
Components: All components (rounded button, text field container) are available here. we can use all the above using the required class.

Shared preferences: In this dart file contain variable store the login details data.

Create Customer login Page.

Here Text field widget used for user input as email id and password. Flat Button widget, to show action. Also, used Image to set logo for login page.

Inside lib/Screen/customer_login



Background. Dart: All page background class are available here

CustomerLogin_pojo.dart: There have a static method called from Json which receives Map object. Then set mail id and password values from our Map object called data. Now can use our function to convert our json to class.

customer_detail_pojo.dart: There have a static method called from Json which receives Map object. Then we set name, emailid, password address and phone number values from our Map object called data. Now use our function to convert our json to class.

customer_login_screen.dart: It contain main function entry of my program execution, and MyApp class which takes an object of Login class as a parameter of the home property.

body.dart:

- Set Scaffold's app Bar property as follows to make heading for our application.
- For this UI, all widgets are placed inside the Column widget, into the Scaffold body. The first child of Column is the Container widget which holds Image widget as it's child.
- flutter-logo.png file copied into asset/images folder in this flutter application and write into pubspec.yaml file to get it in our code.
- Then, for email id and password use the TextField widget from inside component.RoundedmailLoginfield function for emailid and Roundedpasswardfield password is an input widget that helps you to take input from the user
- For the login button, use from component Roundedbutton 'Login' as a child and onPressed() of this button we can write code for control navigation to another home screen. After pressed login button all customer detail sends

to backend

// API Call from server //

String url = api.customerlogin;

// Write the following code Inside lib/Screen/customer_login/body.dart //

```
var res = await http.post(Uri.parse(url),
headers: {'Content-Type': 'application/json'},
body: json.encode({'email': login.email, 'password': login.password}));
// Connect both frond and end back server //
var data = json.decode(res.body);
// Status command from backend server //
var Response = data["Customerdetails"] as List;
// mapping with POJO code with customer detail //
customer_detail =
Response.map<Customer_detail>((json)=> Customer_detail.fromJson(json)).toList();
```

Response: When customer logins, api call occurs and a list appears which contain customer details and in that all customer details are stored using shared preference**.**

Stringvalue.*email* = sharedPreferences.getString("email");

http.post: Request the specified url through POST method by posting the supplied data and return the response as Future<Response>

http.get: is used to fetch the data from the Internet.

json.encode: The Encodable function is used to convert it to an object that must be directly encodable.

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<customerdetails> using fromMap of the CustomerLogin class.

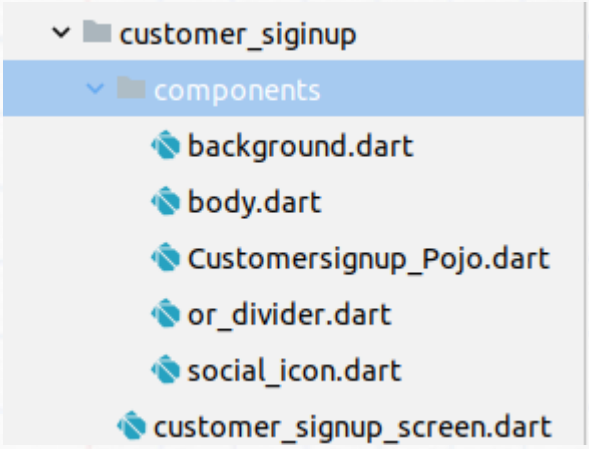
Flutter toast: Once get the response from backend "Login Successfully" add

flutter toaster message.

Create Customer Registration Page

Here Text field widget used for user input as name, emailid, password and phonenumber Flat Button widget, to show action. Also, used Image to set logo for Registration page.

Inside lib/Screen/customer_signup



background. Dart: All page background class are available here

Customersignup_Pojo.dart: There have a static method called from Json which receives Map object. Then set name, emailid, password phone number and address values from our Map object called data. Now use our function to convert our json to class.

or_divider.dart: It contains divider line function

social_icon.dart: It contains the social icon function

customer_signup_screen.dart: It contains main function entry point of my program execution, and MyApp class which takes an object of Login class as a parameter of the home property.

Body.dart:

- Set Scaffold's appBar property as follows to make heading for our application.
- For this UI, all widgets are placed inside the Column widget, into the Scaffold body. The first child of Column is the Container widget which holds Image widget as it's child.
- flutter-logo.png file copied into asset/images folder in this flutter application and write into pubspec.yaml file to get it in our code.
- Then, for emailid and password use the TextField widget from inside component.RoundedmailLoginfield for emailid, name, password and phone number is an input widget that helps you to take input from the user
- For the login button, use from component Roundedbutton 'Signup' as a child and onPressed() of this button we can write code for control navigation to another home screen. After pressed login button all customer detail sends to backend

// API Call from server //

String url = api.customerregister;

http.post: Request the specified url through POST method by posting the supplied data and return the response as Future<Response>

http.get: Is used to fetch the data from the Internet.

json.encode: The Encodable function is used to convert it to an object that must be directly encodable.

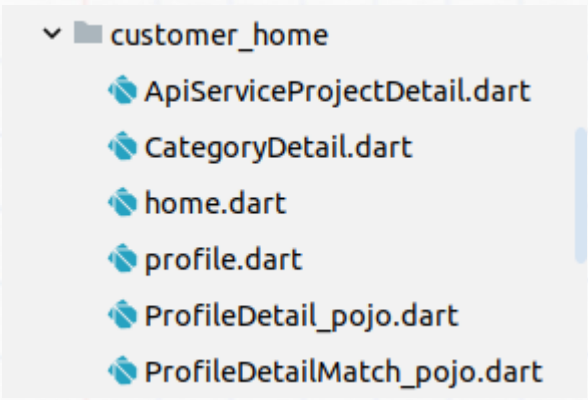
Flutter toast: Once get the response from backend "Register Successfully" add flutter toaster message.

// Write the following code Inside lib/Screen/customer_signup/body.dart //

```
var res = await http.post(Uri.parse(url),
      headers: {'Content-Type': 'application/json'},
      body: json.encode({
        'name': siginup.name,
        'email': siginup.email,
        'password': siginup.password,
        'phone': siginup.phone,
      }));
  // status command get from back end server //
  if (res.body == "Registered Successfully") {
    Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) => LoginScreen(),
        ));

    // flutter toast command //
    Fluttertoast.showToast(
        msg: "Registered Successfully",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.CENTER,
        timeInSecForIosWeb: 2,
        backgroundColor: Colors.black,
        textColor: Colors.white);
  } else {
    Fluttertoast.showToast(
        msg: "Invalid user",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.CENTER,
        timeInSecForIosWeb: 2,
        backgroundColor: Colors.black,
        textColor: Colors.white);
  }
```

Customer Home Page:

Inside lib/Screen/customer_home



CategoryDetail.dart: There have a static method called from Json which receives Map object. Then set category name and category image values from our Map object called data. Now use our function to convert our json to class.

home.dart:

Set Scaffold's app Bar property as follows to make heading for our application.

// API Call from server //

String url = api.category;

// Write the following code Inside lib/Screen/customer_home/home.dart //

```
Future GetCategoryList() async {
var res = await http
.get(Uri.parse(url), headers: {'Content-Type': 'application/json'});
print("category list success${res.body}");
if (res.body != null) {
var data = json.decode(res.body);
// List the category name image in this list //
var Response = data["Category"] as List;
//Map the category detail //
setState(() {
categorydetial =
Response.map<CategoryDetail>((json) => CategoryDetail.fromJson(json))
.toList();
}); }}
```

When category, api call occurs and a list appears which contain category details.

http.get: Is used to fetch the data from the Internet.

json.encode: The Encodable function is used to convert it to an object that must be directly encodable.

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<categorydetail> using from Map of the Categorydetail class pojo code.

- In Flutter, you can encode a local or network image (or another kind of file) to a base64 string like this Ref link:

  https://www.kindacode.com/snippet/flutter-turn-an-image-into-a-base64-string-and-vice-versa/

- Drawer and list view My profile, My cart, My order, logout.
- Display the Category image Base64 is an encoding scheme that can carry data stored in binary formats. The application of base64 string is common in web and mobile app development.
- The Image. Memory constructor helps to display images from bytes. Hence, we must convert the base64 string to bytes using dart convert and display mage list view constructor. The standard List View constructor works well for small lists. To work with lists that contain many items, it's best to use the ListView.builder constructor.
- In contrast to the default List View constructor, which requires creating all items at once, the ListView.builder() constructor creates items as they're scrolled onto the screen.

Create My Profile Page:

ApiServiceProjectDetail.dart: It contains api detail update profile.

String get updateprofile =>"http://121.242.232.216:7070/emarket/updateprofile";

When customer logins, api call occurs and a list appears which contain customer details using shared preferences.

profiledetailpojo.dart: There have a static method called from Json which receives Map object. Then we set name, emailid, phonenumber and address values from our Map object called data. Now use our function to convert our json to class.

Flutter toast: Once get the response from backend server "Update Successfully" add flutter toaster message.

/// Write the following code Inside lib/Screen/customer_home/profile.dart //

```
    Map data = {
   "email": "${Stringvalue.email}",
   "password": "${Stringvalue.password}"
 };
 final loginRequestJson = jsonEncode(data);
 var res = await http.post(Uri.parse(url),
     headers: {'Content-Type': 'application/json'}, body: loginRequestJson);
 if (res.body != null) {
   var data = json.decode(res.body);
   // list of customer detial from server //
   var Response = data["Customerdetails"] as List;
// map profile detail //
   profiledetailmatch =
       Response.map<ProfileDetailMatch>((json) => ProfileDetailMatch.fromJson(json))
           .toList();
}
```

// API Call from server //

String url = api.customerlogin;

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<profiledetailmatch> using from Map of the ProfileDetailMatch class pojo code.

profile.dart: Flutter User Profile Page UI where you can access and edit your user's information within your Flutter app Text field Controller

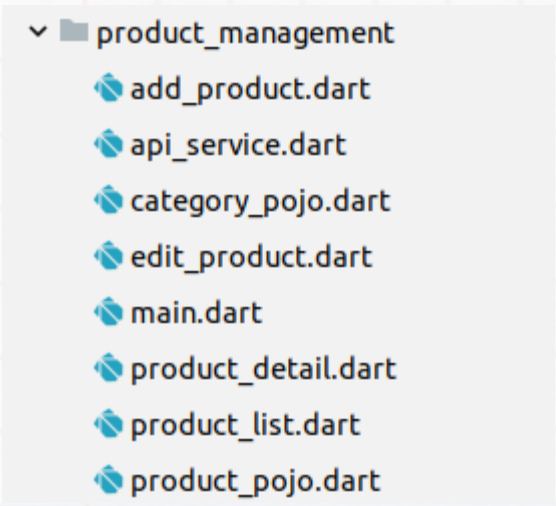
it's useful to run a call back function every time the text in a text field change have edit form where some data in text fields controller from database.

Here is my requirement, when I click the Update button, dynamically new cards with nine Text Fields should be generated,

eg: name, emailid password, doorno, area, city, state and pincode. Once update all value click update button all data send to backend using **customer login api** call

Create Product List and Add to Cart Page:

Inside lib/Screen/product_management



We will be using SQLite and Shared Preferences in our application to store the data locally on the device itself. SQLite and Shared Preferences store data, while Provider manages the application's state.

API CALL PRODUCT LIST

When products, api call occurs and a list appears which contain product details.

// API call product //

String url = api.product;

// Write the following code Inside lib/Screen/product_management/product_list.dart //

```
List<Product_detail> product_detail;
bool isloading = false;
Future<dynamic> productdetailsfuture;
// function get all product list //
Future getallproductlist() async {
var res = await http.post(Uri.parse(url),
headers: {'Content-Type': 'application/json'},
body: json.encode({'category': categoryname}));
if (res.body != null) {
var data = json.decode(res.body);
// get the product detail from database //
var Response = data["Productdetails"] as List;
```

```
product_detail =
Response.map<Product_detail>((json) => Product_detail.fromJson(json))
.toList();
print ("the product descr ${product_detail[0]
.productdetails
.toString()}");
setState(() {
isloading = true;
});
}
}
```

http.post: Request the specified url through POST method by posting the supplied data and return the response as Future<Response>

json.encode: The Encodable function is used to convert it to an object that must be directly encodable.

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<product_detail> using from Map of the product_detail class.

1. Build shopping cart:

The first is a product screen, which displays a list of products along with photos, the name of the product, and the price. Each list item includes a button. that allows you to add it to your shopping basket. The AppBar includes a shopping cart icon with a badge that updates the item count whenever a user presses the Add to Cart button. The second screen, the shopping cart screen, displays a list of the things that the user added to it. If the user decides to remove it from the cart, a delete button removes the item from the cart screen. The entire cost is shown at the bottom of the screen. A button that, for the time being, displays a Snack Bar confirming that the payment has been processed.

2. SETUP:

Next, we are going to start off with creating our model classes named Cart and Item. So, create a new Dart file and name it cart_model, or you can also name it per your requirements **Ref (Cartmodel.dart)**

Create another Dart file and enter product_pojo (**Ref:product_pojo.dart**)

3. Add Sqflite:

As previously stated, we will be utilizing SQFlite, which is essentially SQLite for Flutter, and we will save the data locally within the phone memory. We are not uploading or retrieving data from the cloud because the objective of this post is to learn the fundamental operation of a cart screen. So, using the SQLite package, we're constructing a database class called DB Helper **(Ref:DBHelper.dart)**

4. Add the Provider Class:

The next step will be to develop our Provider class, which will include all our methods and will separate our UI from the logic that will eventually manage our entire application. We use Shared Preferences in addition to SQLite. The reason for using Shared Preferences is because it wraps platform-specific persistence to store simple data such as the item count and total price, so that even if the user exits the application and returns to it, that information will still be available. **(Ref cart_provider.dart)**

5. Create a basic Shopping cart UI:

So, starting from the top that is the AppBar, we have added an Icon Button wrapped with our Badge package that we added to our application. The Icon is of a shopping cart and the badge over it shows how many items have been added to our cart. Please have a look at the image and code below. We have wrapped the Text widget with a Consumer widget because every time a user clicks on the Add

to Cart button, the whole UI does not need to get rebuilt when the Text widget must update the item count. And the Consumer widget does exactly that for us

The Scaffold 's body is a List View builder that returns a Card widget with the information from the lists we created, the name of the product, unit, and price per unit, and a button to add that item to the cart.

We have initialized our Cart Provider class and created a function that will save data to the database when the Add to Cart button is clicked. It also updates the Text widget badge in the AppBar and add total price to the Database that will eventually show up in the Cart screen.

6. Create Cart Screen:

Moving on to the cart screen, the layout is like the product list screen. When the user clicks the Add to Cart button, the entire information is carried onto the cart screen. The implementation is like what we've seen with other ecommerce applications. The primary distinction between the two layouts is that the cart screen includes an increment and decrement button for increasing and decreasing the quantity of the item. When users click the plus sign, the quantity increases, and when they click the minus sign, the quantity decreases. The total price of the cart is added or subtracted when the plus and minus buttons are pressed. The delete button deletes the item from the cart list and subtracts the price from the total price. Again, we have wrapped our ListView builder with the Consumer widget because only parts of the UI need to be rebuilt and updated, not the whole page. **(Ref:cartscreen.dart)**

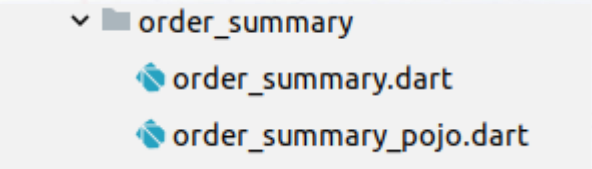
Look towards the end of the code, just before the bottom navigation bar, for a Consumer widget that returns Value Notifier Builder from within the Column widget. It is responsible for updating the quantity for the specific item when the

user clicks either the plus or minus button on the cart screen. There is a bottom navigation bar with a button at the bottom of the screen.

After Pressed the Continue button its navigator to Order summary page

Create Order Summary Page:

Inside lib/Screen/order_summary



order_summary_pojo.dart: There have a static method called from Json which receives Map object. Then set customer detail order date and time order id and product details values from our Map object called data. Now use our function to convert our json to class.

order_summary.dart:

- Previously, store the customer detail id each customer (name emailid phone number, address) data by using Shared Preferences is the way in which one can store and retrieve small amounts of primitive data as key/value pairs to a file on the device storage.
- Display the store all shared preferences top corner of the page.
- Now create a new class named as OrderSummaryScreen( ) this will be going to be a stateful class because our application does change its state at run time. And return MaterialApp( ).

- The Map object is a simple key/value pair. Keys and values in a map may be of any type. Map data list each element.
- **jsonencode**: The Encodable function is used to convert it to an object that must be directly encodable.

// MAP DATA//

```
Map mapData;
list.forEach((element)
{
mapData = {
"productid": element.productId,
"productname": element.productName,
"productquantity": element.initialquantity,
"productprice": element.productPrice,
"productimage": element.image,
"productdetail" : element.productDetails,
};
ls.add(json.encode(mapData));}
```

// API call for order summary //

String get order summary =>
"http://121.242.232.216:7070/emarket/ordersummary"

- Send all detail to order summary api call to backend server using json format

// Write the following code Inside lib/Screen/order_summary/order_summary.dart //

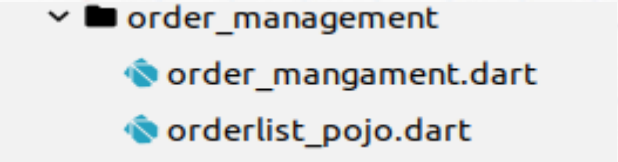
```
var request = json.encode([{"productlist":json.decode("${ls}"),"totalprice":
"${total}",

"customerid":"${Stringvalue.id}","customername": "${Stringvalue.name}","customeremail":
"${Stringvalue.email}",
"customerphoneno": "${Stringvalue.mobilenumber}",
"customeraddressno": "${Stringvalue.addressno}",
```

```
"customerarea": "${Stringvalue.area}",
"customercity": "${Stringvalue.city}",
"customerstate": "${Stringvalue.state}",
"customerpincode": "${Stringvalue.pincode}"
}]);// mapping
var res = await http.post(Uri.parse(ordersummary),
headers: {'Content-Type': 'application/json'},
body: request);
```

- Once Get the response “Inserted Successfully”.
- **Flutter toast**: Once get the response from backend “Order Successfully “add flutter toaster message.
- Follow the same procedure Cart screen page.
- Total price value passed through payment function (**Ref: payment.dart**) We can use Navigator. push () to navigate to a new route and Navigator. pop () to navigate to the previous route.
- After Getting response from backend server customer get invoice mail to customer email id.

Create Order Management Page:

Inside lib/Screen/order_management



//API CALL FOR ORDER PAGE //

String url = api.merchantorder;

orderlist_pojo.dart: There have a static method called from Json which receives

Map object.

Then set orderdetail customer detail and product detail values from our Map object called data. Now we can use our function to convert our json to class.

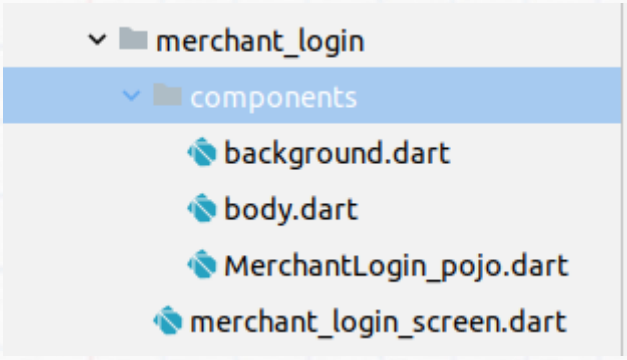
// Write the following code Inside lib/Screen/order_management/order_management.dart //

```
var res = await http.post(Uri.parse(url),
headers: {'Content-Type': 'application/json'});
if (res.body != null) {
var data = json.decode(res.body);
// get the orderdetail from backendd side //
var Response = data["orderdetails"] as List;

setState((){
merchantorder_detail =
Response.map<Merchantorder_detail>((json) => Merchantorder_detail.fromJson(json)).toList();}
);}
```

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<merchantorder_detail> using from Map of the Merchantorder_detail class pojo code.

Create Merchant Login Page:

Inside lib/Screen/merchant_login



Follow up same procedure as Customer Login Page

body.dart:

// API Call from server //

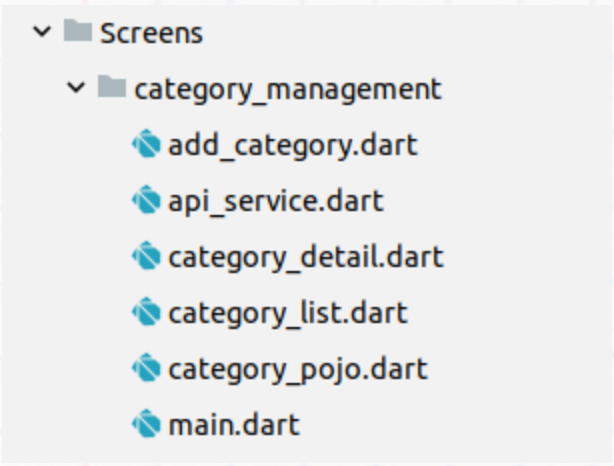
String url = api.merchantlogin;

// Write the following code Inside lib/Screen/merchant_login/body.dart //

```
Future GetMerchantLoginDetail() async {
var res = await http.post(Uri.parse(url),
// json mapping //
headers: {'Content-Type': 'application/json'},
body: json.encode({'email': merchantlogindetail.email, 'password':
merchantlogindetail.password}));
// status command for backend server merchant login page //
if (res.body == "Successfull") {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) => MerchantScreen(),
));
// flutter toast command //
Fluttertoast.showToast(
msg: "Login Successfully",
toastLength: Toast.LENGTH_SHORT,
gravity: ToastGravity.CENTER,
timeInSecForIosWeb: 2,
backgroundColor: Colors.black,
textColor: Colors.white);
} else {
Fluttertoast.showToast(
msg: "Invalid user",
toastLength: Toast.LENGTH_SHORT,
gravity: ToastGravity.CENTER,
timeInSecForIosWeb: 2,
backgroundColor: Colors.black,
textColor: Colors.white); }}
```

Create Category Management:

Inside lib/Screen/category_management



add_category: Newly add category name and image send to through insertcategory api call to backend.

api_service.dart: It contains api detail category.

String get insertcategory => "http://121.242.232.216:7070/emarket/insertcategory";

String get deletecategory => "http://121.242.232.216:7070/emarket/deletecategory";

String get category => "http://121.242.232.216:7070/emarket/category";

category_detail.dart: Displays category details

category_list.dart: Displays List of category name

category_pojo.dart: There is a static method called from Json which receives Map object. Then set category name and image values from our Map object called data. Now use our function to convert our json to class.

main.dart: The main file of the generated project is the entry point of the Flutter application: void main () =>runApp(MyApp()); The main function by itself is the Dart entry point of an application.

As we mention in the first paragraph, we will use the HTTP library package to access the REST full API from the REST API server. For that, install this package by open and edit pubspec.yaml then add this dependency.

category_pojo.dart: That represent the SQLite table. This class is about category detail.

api_service.dart Where we will put all CRUD (POST, GET, PUT, DELETE) methods to the REST API. Fill this class with this CRUD operation of HTTP requests to the REST API.

// Write the following code Inside lib/Screen/category_management/api_service.dart //

```
List<dynamic> categorydetail = [];
var res = await http.get(Uri.parse(category));
if (res.body != null) {
var data = json.decode(res.body);
if (res.statusCode == 200) {
var Response = data["Category"] as List;
categorydetail=Response.map((item)=> CategoryDetail.fromJson(item)).toList();
} else {
throw "Failed to load cases list1";
}
}
return categorydetail;
```

http.get: Is used to fetch the data from the Internet.

Response: When product, api call occurs and a list appears which contain productdetail

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<categorydetail> using from Map of the Categorydetail class pojo code.

// Write the following code Inside lib/Screen/categorymanagement/api_service.dart //

Update Category

```
Map data = {

'name': updatecategory.name,
};
final Response response = await http.put(
Uri.parse('$insertcategory/$id'),
headers: <String, String>{
'Content-Type': 'application/json; charset=UTF-8',
},
body: jsonEncode(data),
);
```

1.CATEGORY LIST:

- We will display the list of data in a separate Dart file that will call from the main. Dart home page body. For that, we need a dart file to view the list of data**. (Ref:category_list.dart)**
- Class name that extends Stateless Widget object. Inside that class, declare these variables that hold Category list that loaded from the main. Dart and create Key for the list. Add an override method after the variables to build the ListView widget for the list of categories. That List View builder contains the Card that has the child of InkWell that use to navigate to the Detail Widget using MaterialPageRoute. The child of the Card is ListTile that contains an Icon (leading), Text (title), and Text(subtitle).
- The Inkwell widget has an on-Tap event with an action to Navigate to the details page. Container, Column, Image, and Text have their own properties to adjust the style or layout.

- Keep in mind, every widget that uses the child only has one widget as its child. If you need to put more than one widget to the parent widget, use children: <Widget> property.
- Next, open and **main.dart** then replace all Dart codes with these lines of codes to display the List View in the main home page. We use the existing floating button as the add-data button with an action to go to Add Category Widget.dart.

2.Category Detail:

We will display data details to another page that opened when tapping on a list item in the list page. For that, create a Dart file in the lib folder detail category.dart. We will use a scrollable Card widget to display a detail to prevent overflow if the Card content is longer. Next, open and edit lib/detailwidget.dart then add these imports of Flutter material, database helper, editdatawidget, and cases object model.

- Add a Detail Widget class that extends Stateful Widget. This class has a constructor with an object field, a field of Category object and _DetailWidgetState that builds the view for data detail.
- Add a _DetailWidgetState class that implementing all required widgets to display data details.
- To handle the delete button, we need to add a method or function after the above method that shows an alert dialog to confirm if data will be deleted.

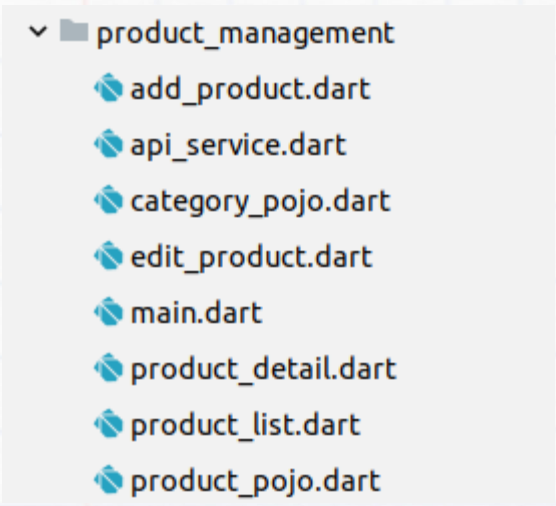
// Write the following code

Inside lib/Screen/categorymanagement/category_detail.dart //

```
return showDialog<void>(
context: context,
barrierDismissible: false, // user must tap button!
builder: (BuildContext context) {
return AlertDialog(
title: Text('Warning!'),
content: SingleChildScrollView(
child: ListBody(
children: <Widget>[
Text('Are you sure want delete this item?'),
],
),
),
actions: <Widget>[
ElevatedButton(
child: Text('Yes'),
onPressed: () {
api.deleteCategory(categoryid);
Navigator.push(
context,
MaterialPageRoute(
builder: (context) {
return CategoryMainPage();
},
), ); },
),
ElevatedButton(
child: const Text('No'),
onPressed: () {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) {
return CategoryMainPage();
},
), ); }, ),],);},);}
```

Create Product Management:

Inside lib/Screen/product_management



add_Product: Newly add product name, description, price, category, quantity, and image send to through insert product api call to back end.

api_service.dart: It contains api detail insert delete product.

String get insertproduct =>
"http://121.242.232.216:7070/emarket/insertproduct";
String get updateproduct =>
"http://121.242.232.216:7070/emarket/updateproduct";
String get deleteproduct =>
"http://121.242.232.216:7070/emarket/deleteproduct";
String get listproduct =>
"http://121.242.232.216:7070/emarket/listproduct";
product_detail.dart: It display product details

product_list.dart: It display List of product details are product name, price, category, description, quantity and image

category_pojo.dart: There is a static method called from Json which receives Map object. Then set category name and image values from our Map object called data. Now use our function to convert our json to class.

main.dart: The main file of the generated project is the entry point of the Flutter application: void main() => runApp(MyApp()); The main function by itself is the Dart entry point of an application.

As we mention in the first paragraph, we will use the HTTP library package to access the REST full API from the REST API server. For that, install this package by open and edit pubspec.yaml then add this dependency.

category_pojo.dart: That represent the SQLite table. This class is about category detail.

api_service.dart where we will put all CRUD (POST, GET, PUT, DELETE) methods to the REST API. Fill this class with this CRUD operation of HTTP requests to the REST API.

// Write the following code Inside lib/Screen/product_management/api_service.dart //

```
List<dynamic> product = [];
var res = await http.get(Uri.parse(listproduct));
if (res.body != null) {
var data = json.decode(res.body);
if (res.statusCode == 200) {
var Response = data["Product"] as List;
product = Response.map((item) => Productdetails.fromJson(item)).toList();
} else {
throw "Failed to load cases list1";
}
}
return product;
```

http.get: Is used to fetch the data from the Internet.

Response: Once fetch category api get data product list.it contain each product name, price, category, description, quantity, and image

json.decode is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<product> using from Map of the Productdetail class pojo code. After getting the Product List it has product name, price, category, decription, quantity and image.

// Write the following code Inside lib/Screen/product_management/api_service.dart //

```
Map data = {
'id': id,
'name': updateproducts.name,
'category': updateproducts.category,
'description': updateproducts.description,
'price': updateproducts.price,
'quantity': updateproducts.quantity,
};
final Response response = await http.post(
Uri.parse(updateproduct),
headers: <String, String>{
'Content-Type': 'application/json; charset=UTF-8',
},
body: jsonEncode(data),
);

if (response.statusCode == 200) {
Fluttertoast.showToast(
msg: "Product Update Succesfully",
toastLength: Toast.LENGTH_SHORT,
gravity: ToastGravity.CENTER,
timeInSecForIosWeb: 2,
backgroundColor: Colors.black,
textColor: Colors.white);
} else {
Fluttertoast.showToast(
msg: "Product Update Faliure",
toastLength: Toast.LENGTH_SHORT,
gravity: ToastGravity.CENTER,
timeInSecForIosWeb: 2,
backgroundColor: Colors.black,
textColor: Colors.white);
}
```

1. PRODUCT LIST:

- We will display the list of data in a separate Dart file that will call from the main.dart home page body. For that, we need a dart file to view the list of data**. (Ref:product_list.dart)**
- Class name that extends Stateless Widget object. Side that class, declare these variables that hold Product list that loaded from the main. Dart and create Key for the list. Add an override method after the variables to build the List View widget for the list of categories. That ListView builder contains the Card that has the child of Inkwell that use to navigate to the Detail Widget using MaterialPageRoute. The child of the Card is List Tile that contains an Icon (leading), Text (title), and Text(subtitle).
- The Inkwell widget has an on-Tap event with an action to Navigate to the details page. Container, Column, Image, and Text have their own properties to adjust the style or layout.
- Keep in mind, every widget that uses the child only has one widget as its child. If you need to put more than one widget to the parent widget, use children: <Widget> property.
- Next, open and edit lib/main.dart then replace all Dart codes with these lines of codes to display the ListView in the main home page. We use the existing floating button as the add-data button with an action to go to AddCategoryWidget.dart.

2. Product Detail:

We will display data details to another page that opened when tapping on a list

item in the list page. For that, create a Dart file in the lib folder detail product Dart. We will use a scrollable Card widget to display a detail to prevent overflow if the Card content is longer. Next, open and edit **detailwidget.dart** then add these imports of Flutter material, database helper, editdatawidget, and cases object model.

- Add a Detail Widget class that extends Stateful Widget. This class has a constructor with an object field, a field of Product object and _DetailWidgetState that builds the view for data detail.

Add a _DetailWidgetState class that implementing all required widgets to display data details.

- To handle the delete button, we need to add a method or function after the above method that shows an alert dialog to confirm if data will be deleted.

// Write the following code Inside lib/Screen/product_management/product_detail.dart //

```
return showDialog<void>(
context: context,
barrierDismissible: false, // user must tap button!
builder: (BuildContext context) {
return AlertDialog(
title: Text('Warning!'),
content: SingleChildScrollView(
child: ListBody(
children: <Widget>[
Text('Are you sure want delete this item?'),
],
),
),
actions: <Widget>[
ElevatedButton(
child: Text('Yes'),
onPressed: () {
api.deleteProducts(id);
Navigator.push(
```

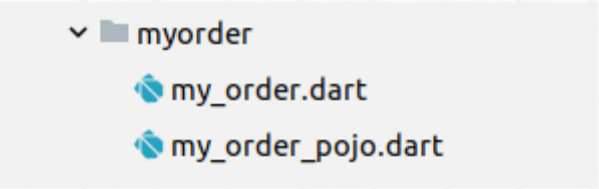
```
context,
MaterialPageRoute(
builder: (context) {
return MyApp_edit_product();
},
),
);
},
),
ElevatedButton(
child: const Text('No'),
onPressed: () {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) {
return MyApp_edit_product(); }, ), ); },),], ); },);
```

3. Edit Product Detail

That codes build widgets combination of Container, Card, Column, Image, Text, and Raised Button. The Raised Buttons has on Pressed event that action to navigate to the EditDataWidget and trigger delete confirm dialog. Next, before the closing of _DetailWidgetState class body add this method or function to navigate to the EditDataWidget with cases object params. The layout for edit data is the same as the add data view with additional object params that get from the details page. This object will fill the default value of the TextFormField and Submit Button. On the submit it will update the data based on the ID then redirect to the list view. First, create a new dart file in the lib folder lib/**editdatawidget.dart**. Open and edit that file then add these lines of the dart codes to build the edit form and function to submit this form to the REST API.

CREATE MY ORDER PAGE:

Inside lib/Screen/myorder



//API CALL FOR ORDER PAGE //

String url = api.customerorder;

my_order_pojo.dart: There have a static method called from Json which receives Map object. Then set order detail customer detail and product detail values from our Map object called data. Now use our function to convert our json to class.

// Write the following code Inside lib/Screen/myorder/my_order.dart //

```
Map data = {
"customerid": "${Stringvalue.id}",
};
// json script //
final loginRequestJson = jsonEncode(data);
var res = await http.post(Uri.parse(url),
headers: {'Content-Type': 'application/json'}, body: loginRequestJson);
if (res.body != null) {
var data = json.decode(res.body);
// list of customer order detail in server //
var Response = data["orderdetails"] as List;


setState((){
customerorder_detail =
Response.map<Customerorder_detail>((json) => Customerorder_detail.fromJson(json))
.toList();
});
}
```

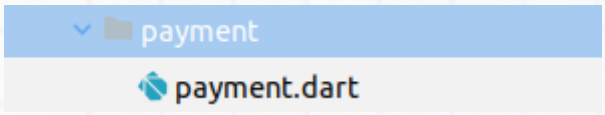
Map data with each customer id

json.decode: Is used to decode the JSON data into the Dart Map object. Once JSON data is decoded, it will be converted into List<customerorder_detail> using from Map of the CustomerOrder class pojo code.

my_order.dart: Just display Customer detail and Product detail **(Ref:productlist.dart)**

PAYMENT GATEWAY INTEGRATION:

Inside lib/Screen/payment



1. Razorpay Payment:

Razor pay Payments provide a range of products to accept payments and make payouts.

2.Create a Razor pay account and log in to the dashboard:

You must sign up for a Razor pay account to use the Razorpay Payments products and access the Razor pay Dashboard.

Sign Up

To create a Razorpay Account, go to the Razorpay website and click Get Started.

Follow these steps for a smooth sign-up process:

1. Contact Details

2. Platform Details

3. Business Type and PAN Details

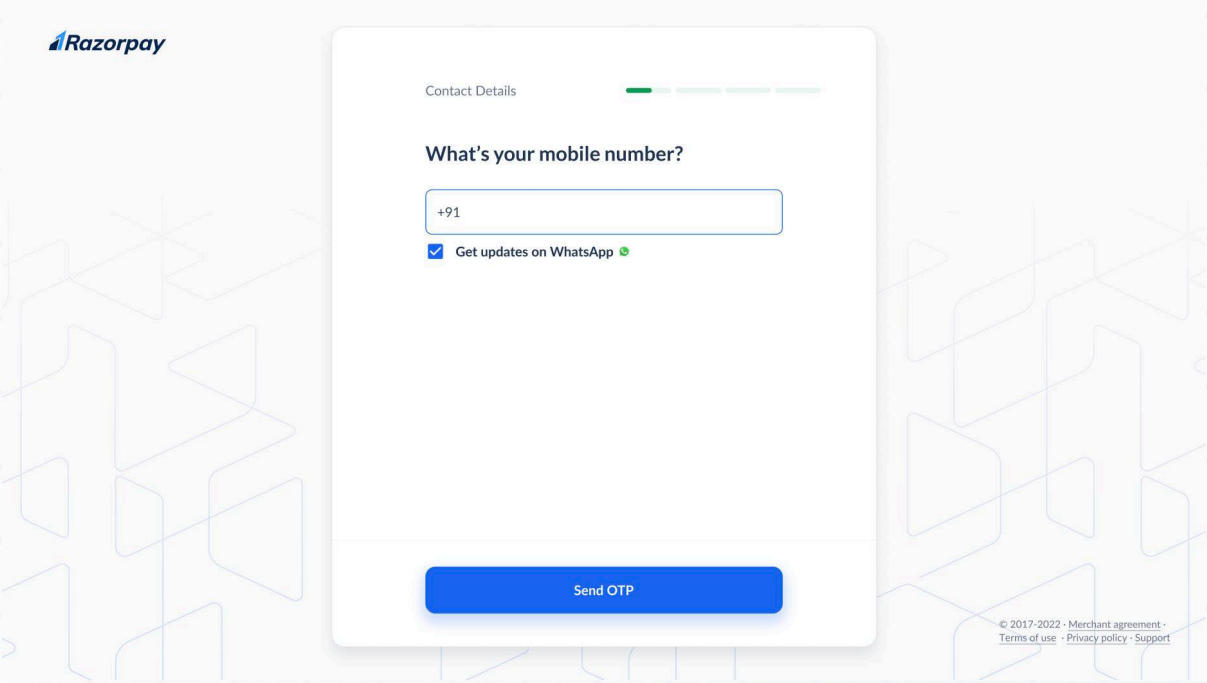
4. Communication Details

Contact Details

Provide your contact details to get started.

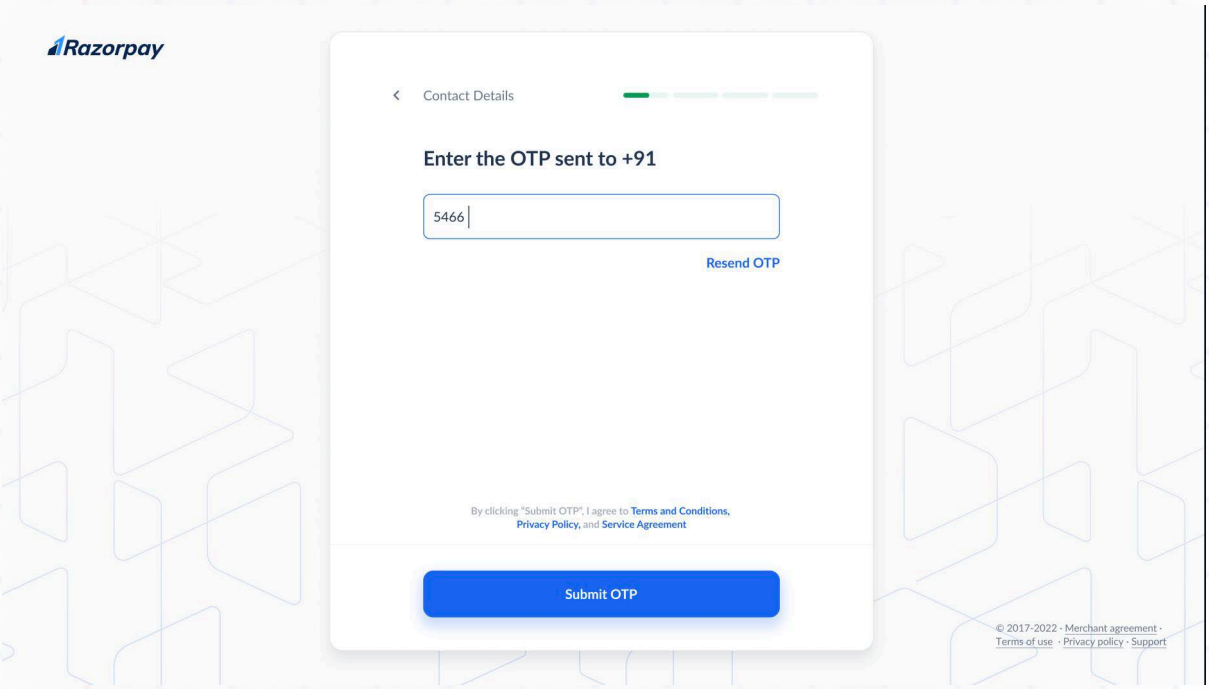
1. Enter your 10-digit Mobile Number.

2. Select the check box below to receive updates on WhatsApp. Click Send OTP.

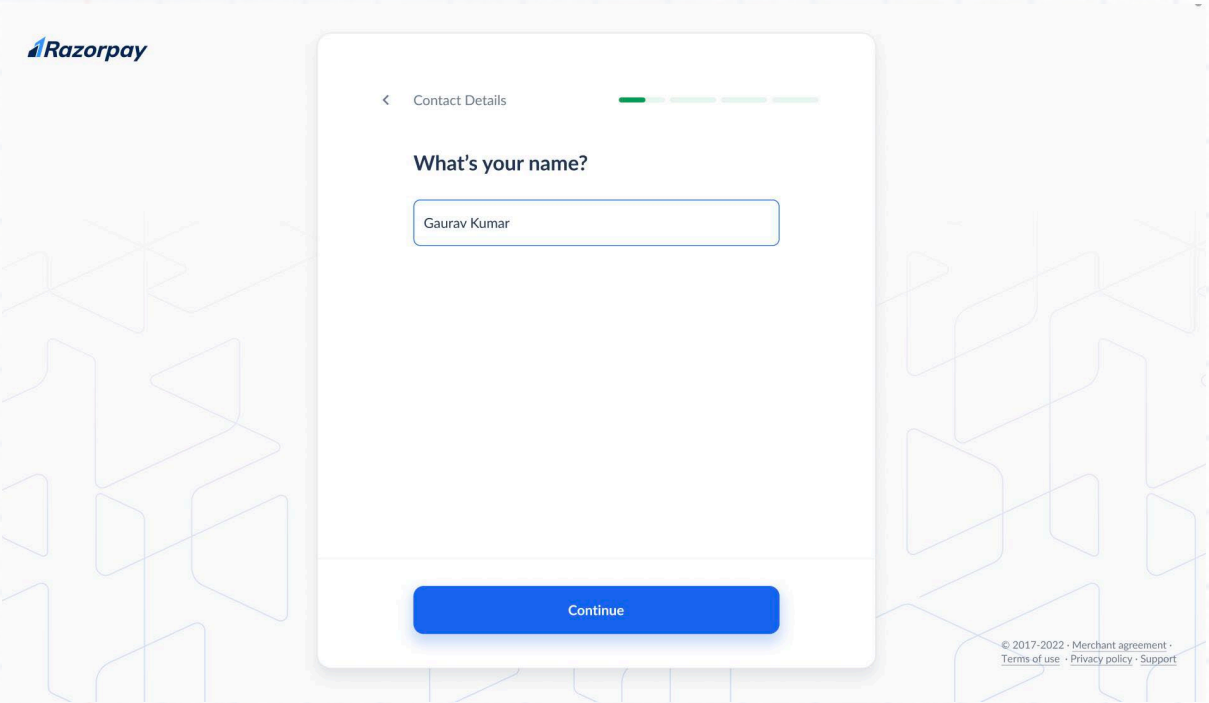


3. Enter the OTP sent to your mobile number. If you did not receive the OTP, click Resend OTP.

4. Click Submit OTP.



5.Enter your Name and click Continue.

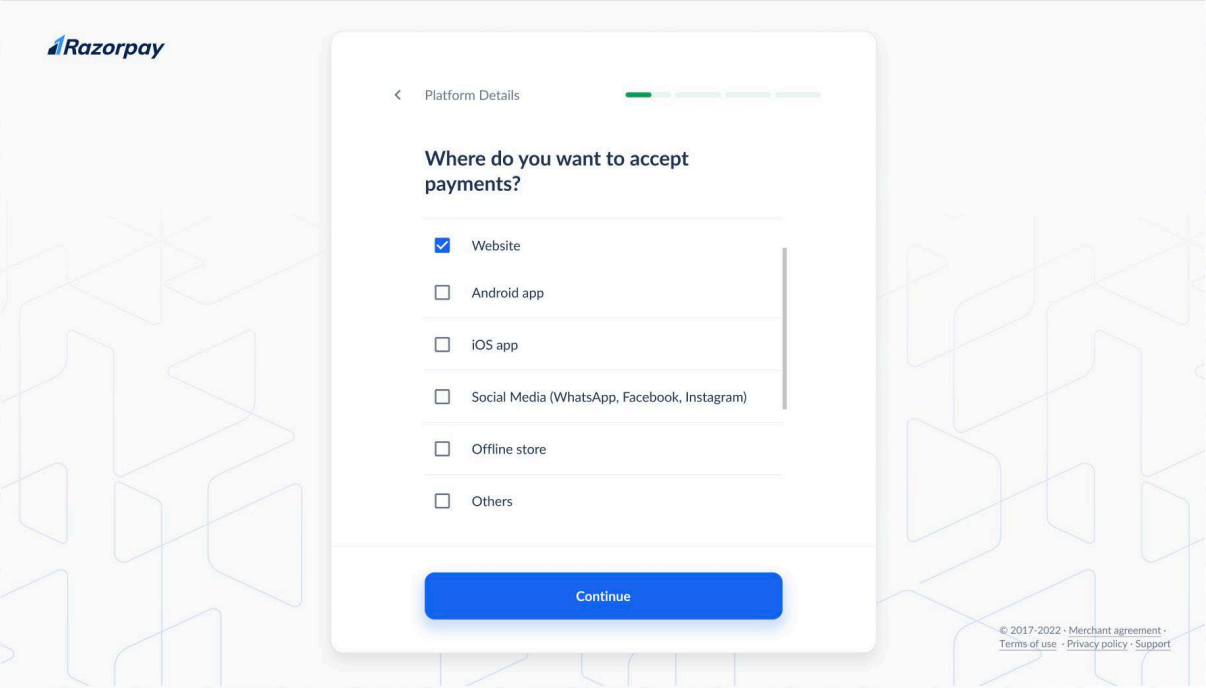


Platform Details

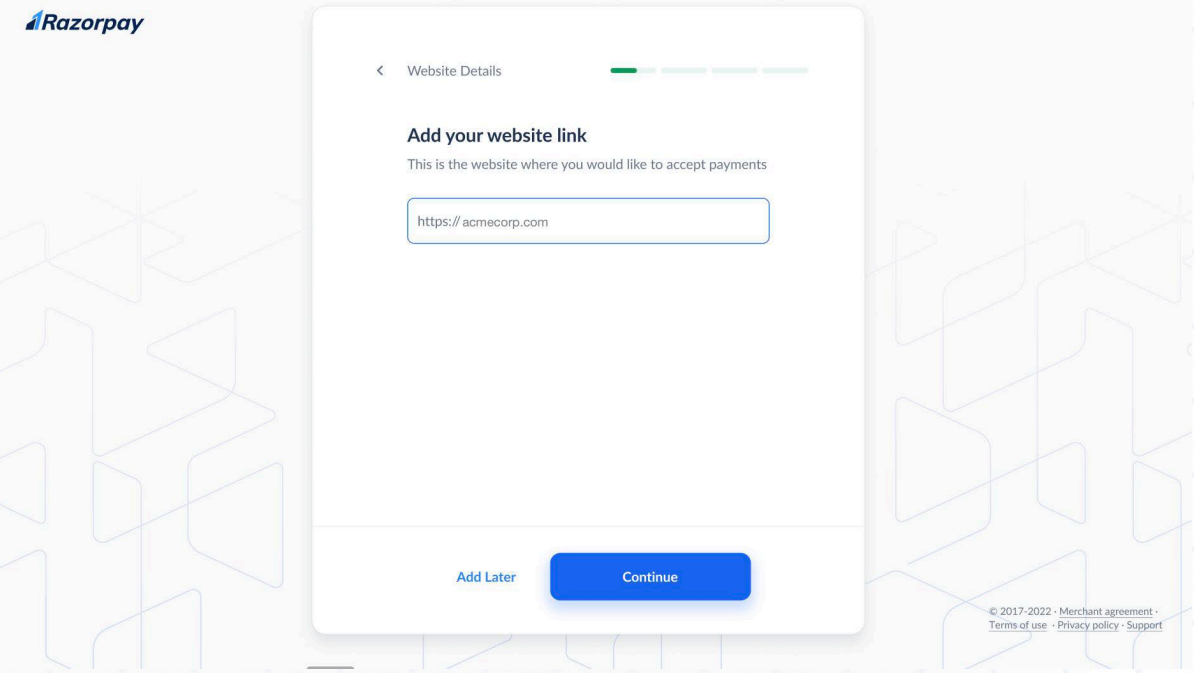
Tell us more about where you want to accept payments.

1.Select where you want to accept payments from the given options. You can also select multiple options if relevant.

2.In case you select Others as an option, add a description. Click Continue.



3.If you select Website/App, add your website/app link and click Continue or click Add Later to add the details afterwards.

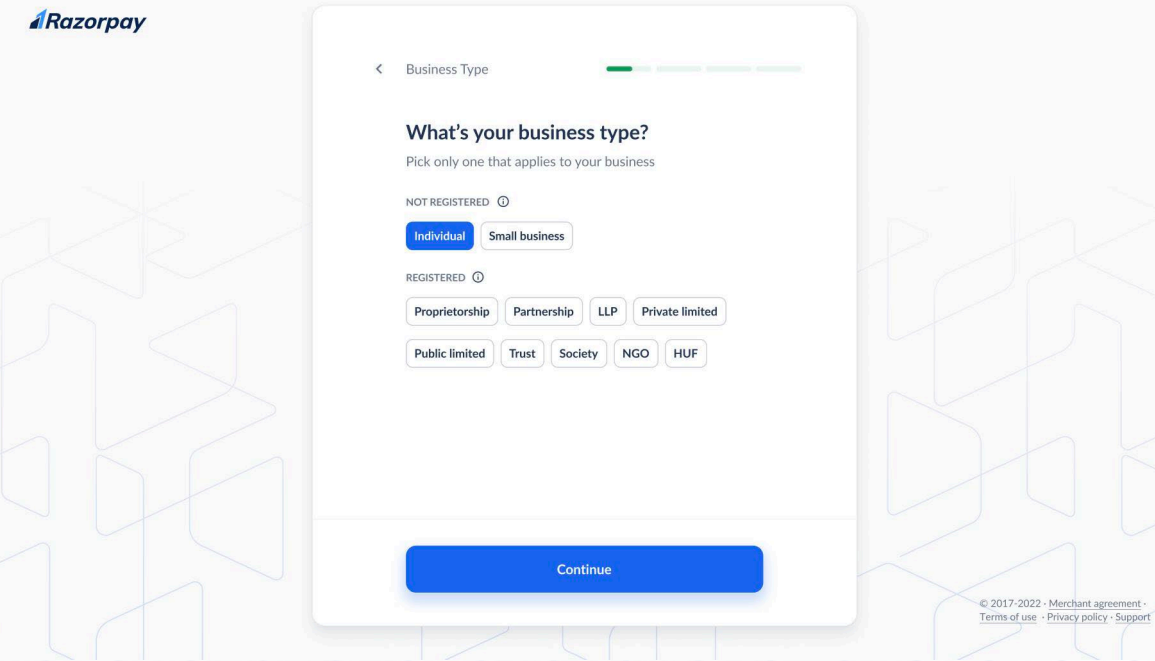


Business Type and PAN Details

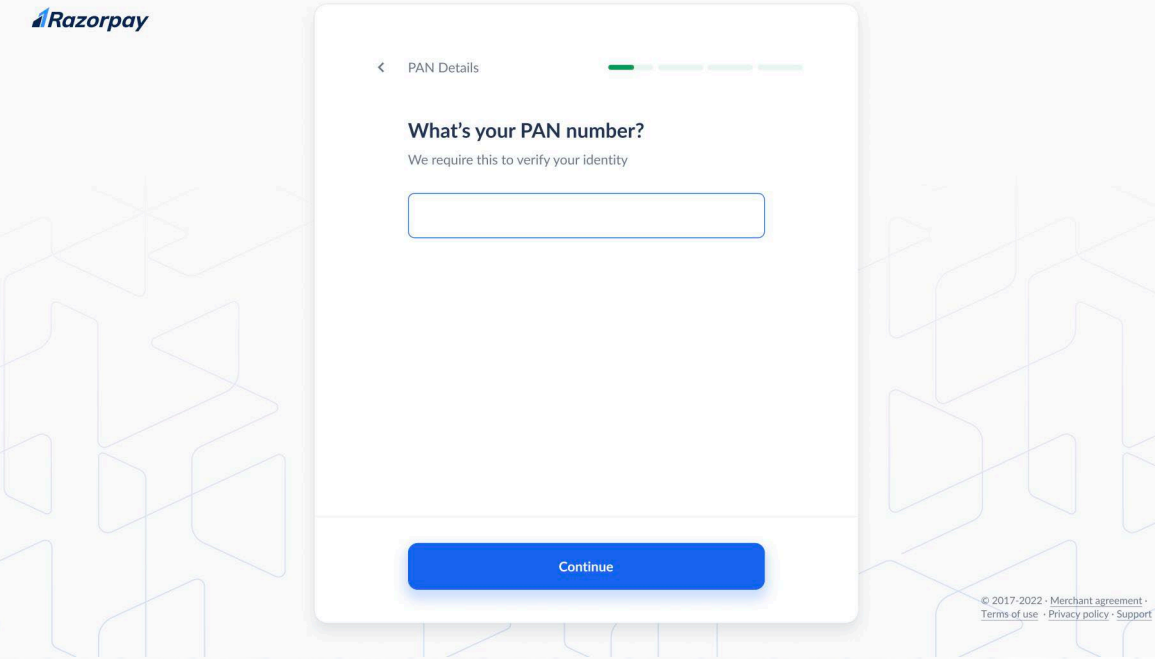
Provide the following business and PAN details:

1.Select your business type from the list. Know more about different business types.

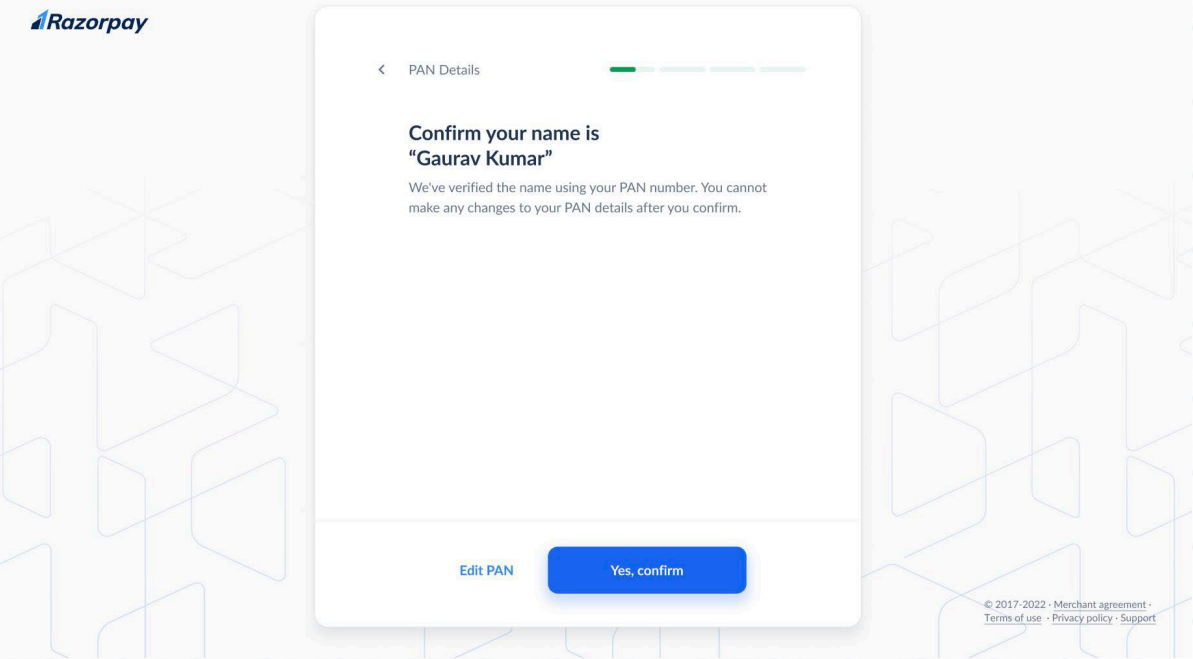
2.Click Continue.



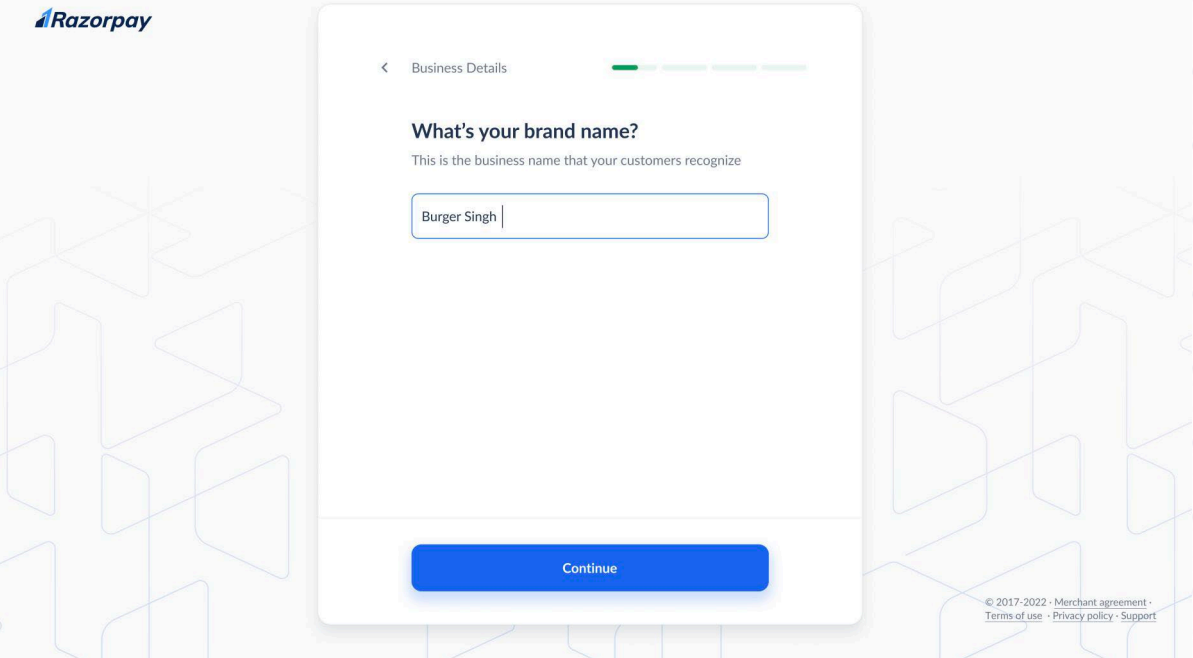
3.Enter your PAN/Business PAN and click Continue. We will verify the details with the Central PAN database.



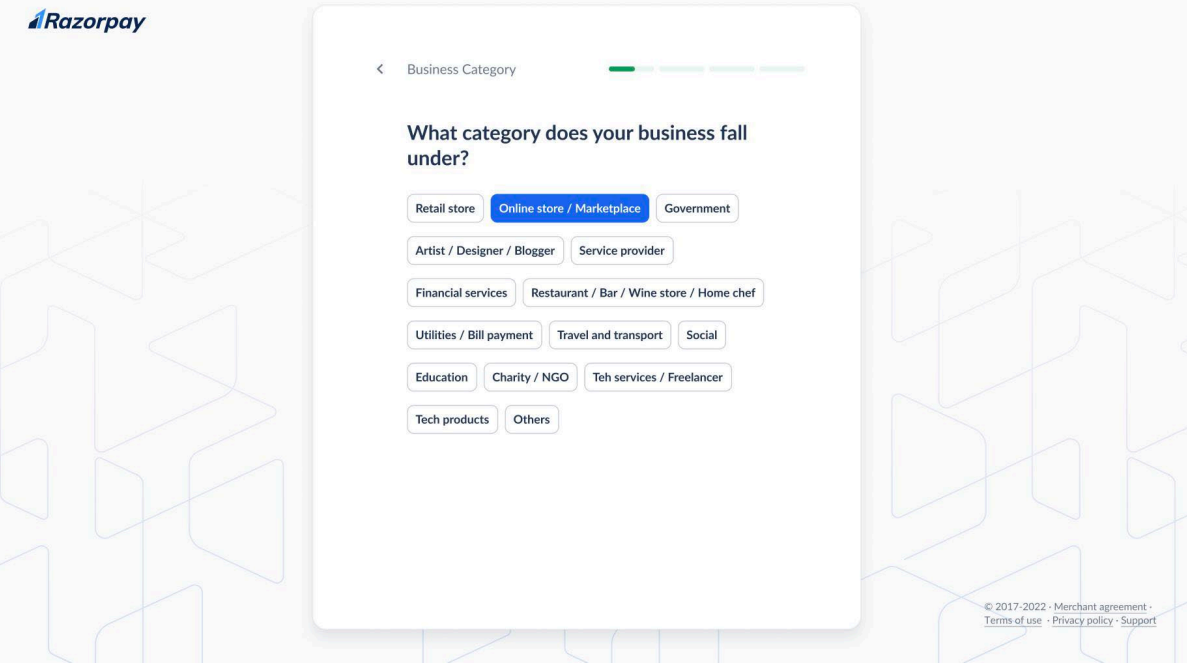
4.Confirm the name associated with the PAN by clicking Yes, Confirm. To edit the PAN details, click Edit PAN.



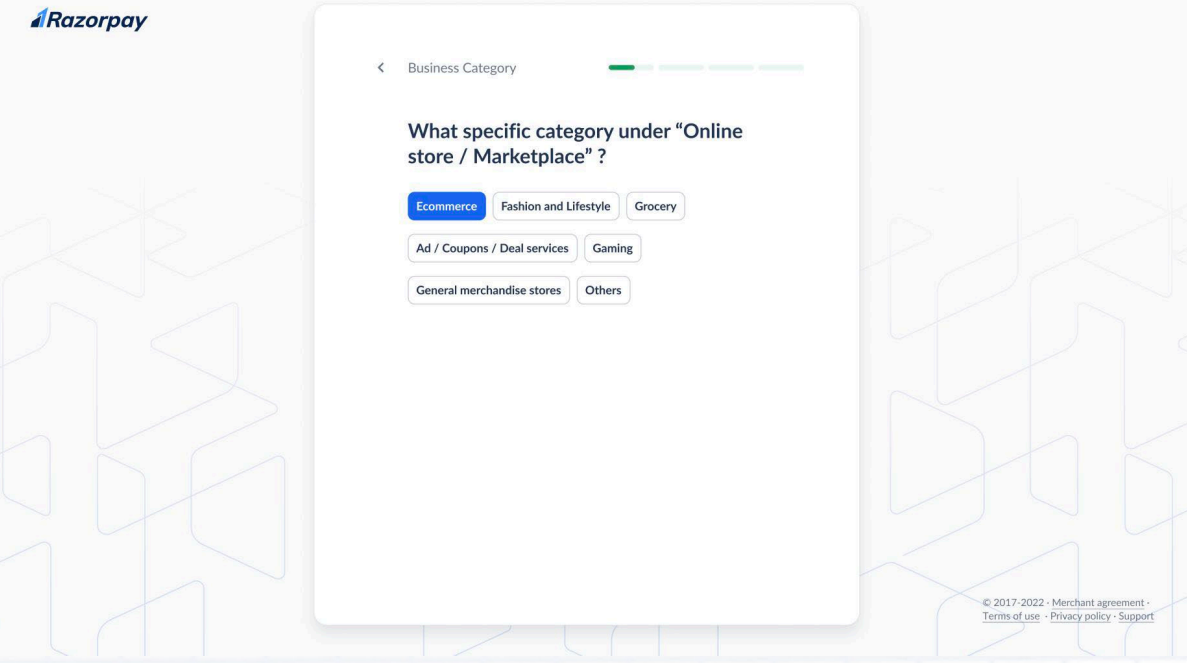
5.Enter your Brand Name. This should be the name of your business that your customers recognise. Click Continue.



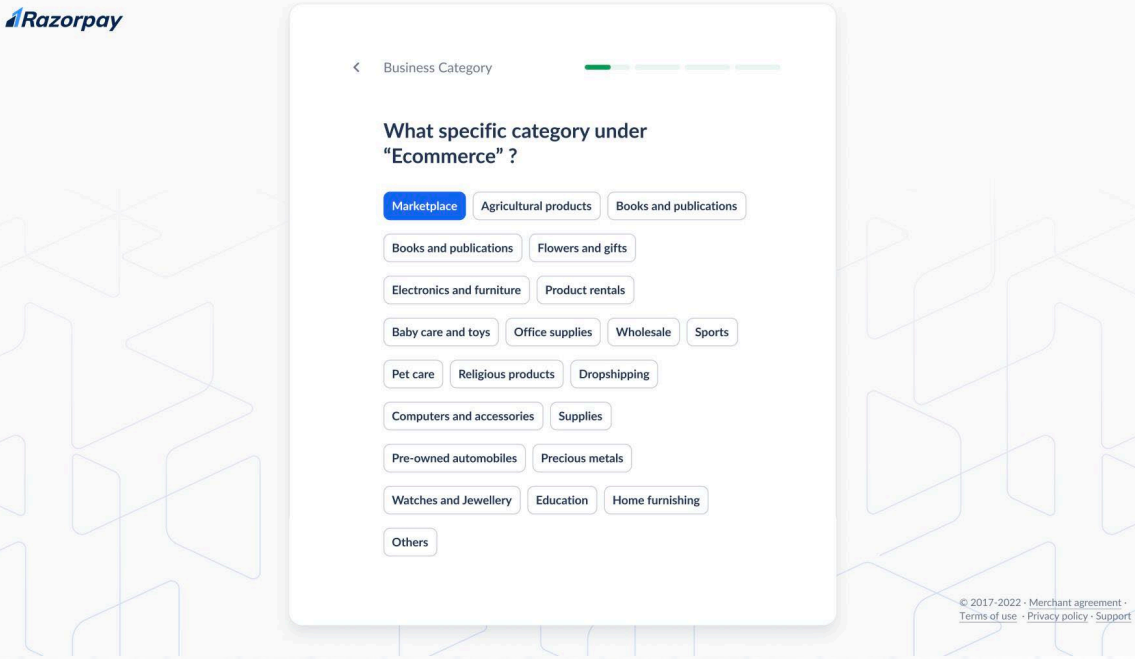
6.Select the relevant Business Category from the list. For example, retail store, online store/marketplace, government and so on.



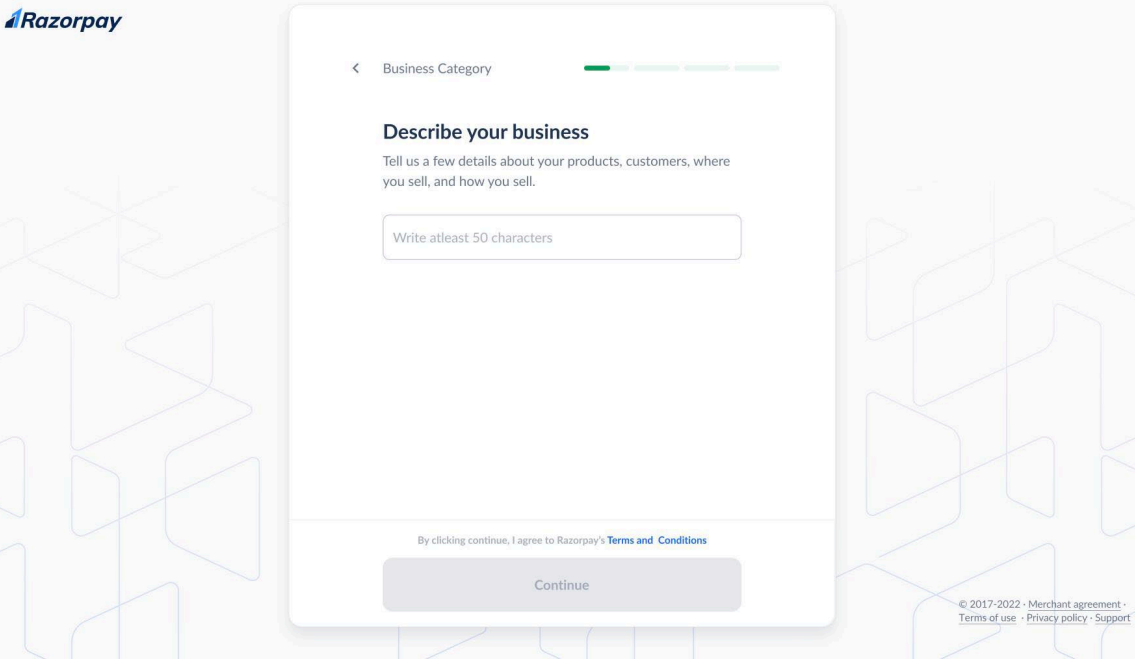
7.Choose a Subcategory from the list. For example, if you selected Online store/Marketplace then specify the category under it from the list.



8.Specify the category under the subcategory selected in the previous step. In case you selected Ecommerce, then specify the category under it from the list.

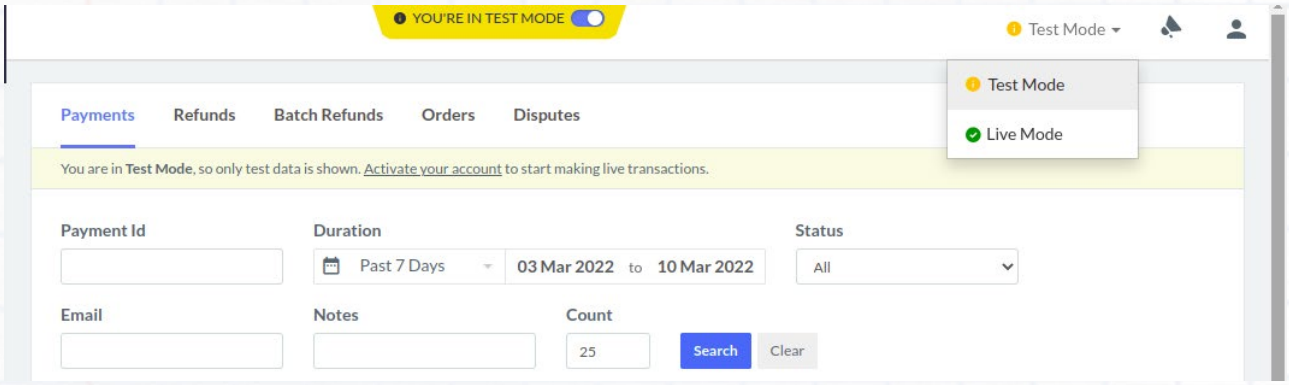


9.Describe your business in at least 50 characters. Click Continue.



3.Test Mode

Once your account is created, you have access to the Test mode on the Dashboard. Test mode is used for testing purposes and does not involve actual money transactions. However, you would need to activate your account in order to accept live payments.



4.API Keys

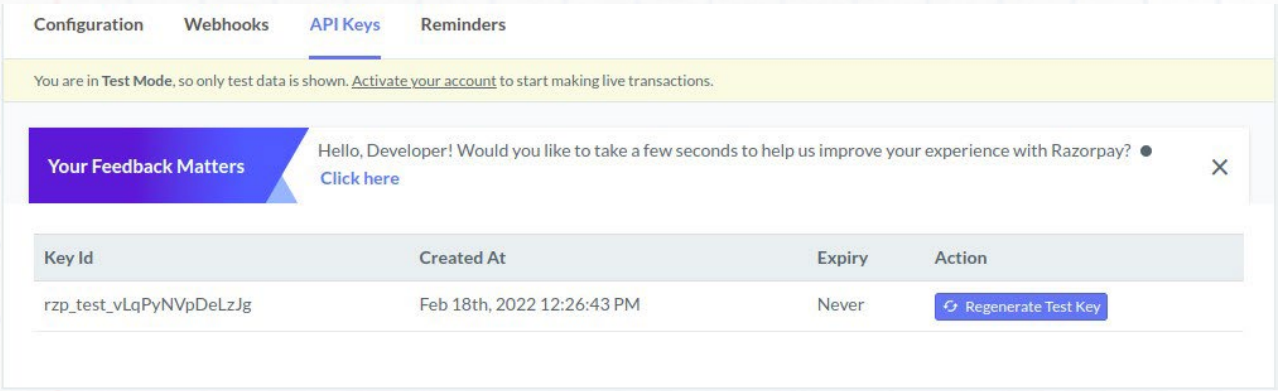
API key is a combination of the key_id and key_secret and is required to make any API request to Razorpay. You also have to implement the API keyin your code as part of your integration process.

5.Generate API Keys

- Log into your Dashboard with appropriate credentials.
- Select the Test mode for which you want to generate the APIkey.
- **Test Mode**: The test mode is a simulation mode that you can use to test your integration flow. Your customers will not be able to make payments inthis mode.

- **Navigate to Settings** → API Keys → Generate Key to generate key for theselected mode.

Once generated, you will be able to see the Key Id, the date the key wascreated and the expiry date for the API Key on screen.



// Write the following code Inside lib/Screen/payment/payment.dart //

```
var options = {
   "key": "rzp_test_4B5CoaTyxFQh3I", // generate key from razorpay website //
   "amount": payment_price * 100, // payment price value get from order summary page //
   "name": "INdigrain",
   "description": "payment for the product",
   "prefill": {"contact": "${Stringvalue.mobilenumber}", "email": "${Stringvalue.email}"},
  };
  try {
   razorpay.open(options);
  } catch (e) {
   print(e.toString());
  }
```

Replace generated key from Razorpay.

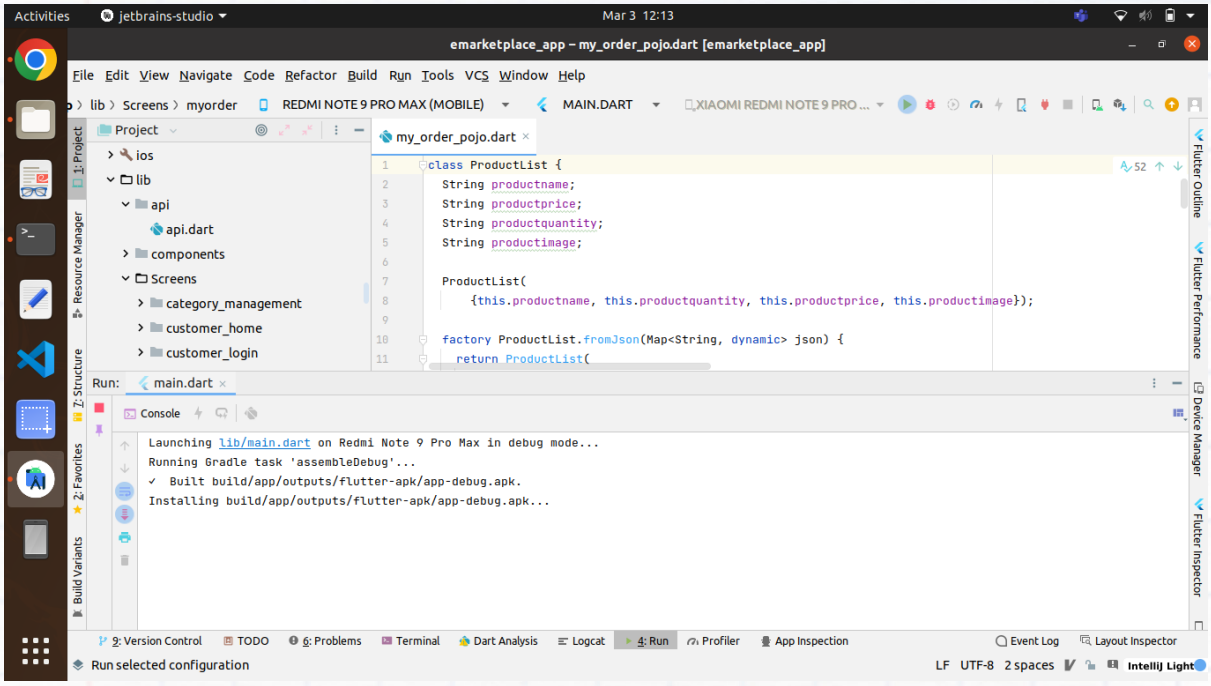
key: < your key >

example: rzp_test_vLqPyNVpDeLzJg

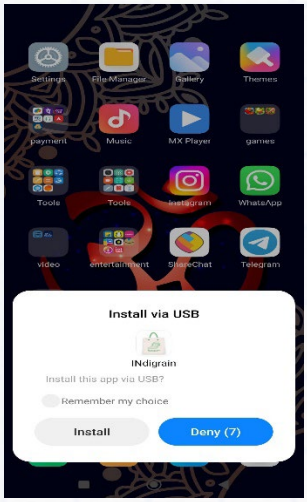
Pass the Checkout options. Ensure that you pass the order_id that you received in the response to the previous step.

Running the application

After Completed all the code and Run the Main.dart File

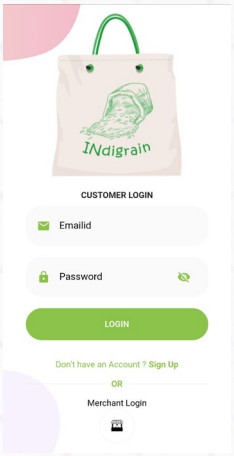


Install App in Mobile

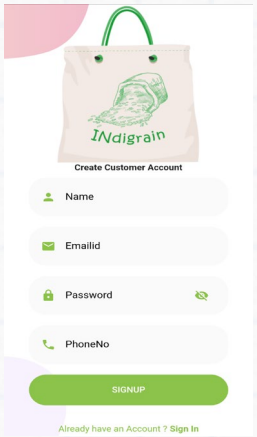


FINAL OUTPUT:

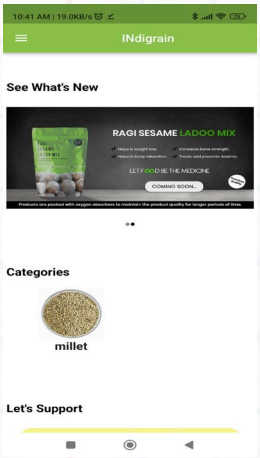
CUSTOMER LOGIN

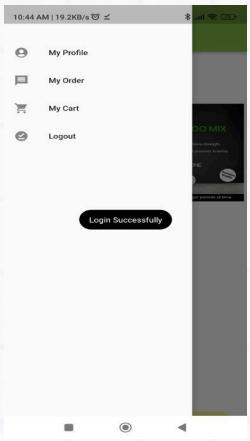


CUSTOMER REGISTRATION

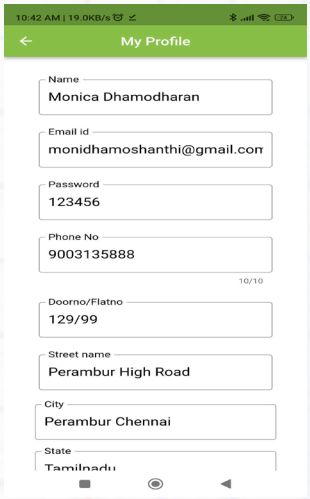


CUSTOMER HOME:

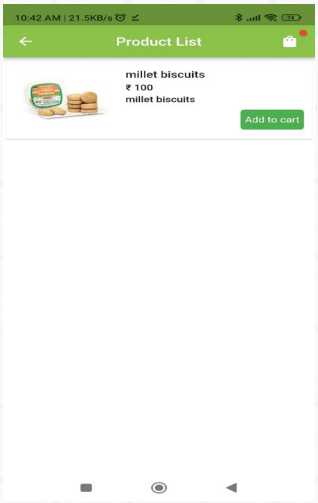




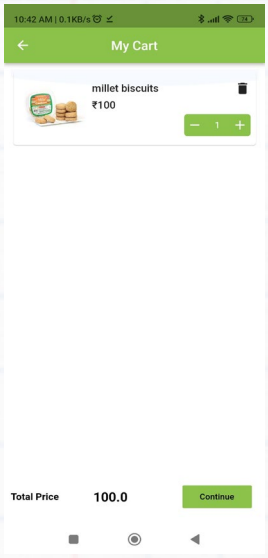
PROFILE



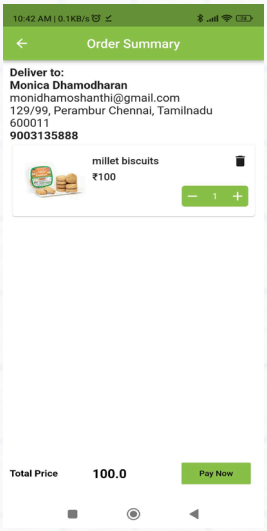
PRODUCT LIST



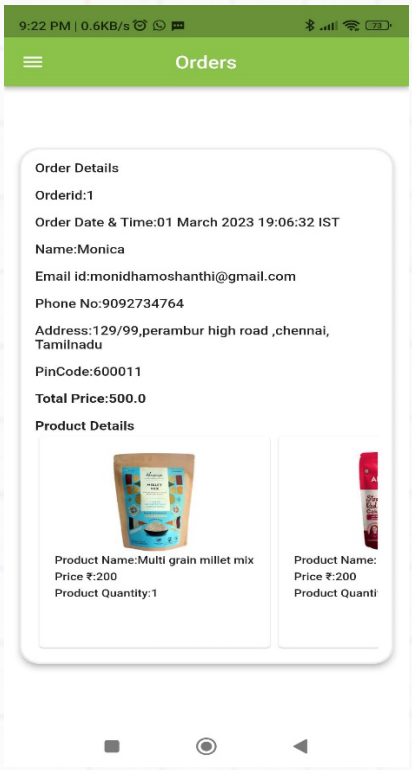
CART SCREEN

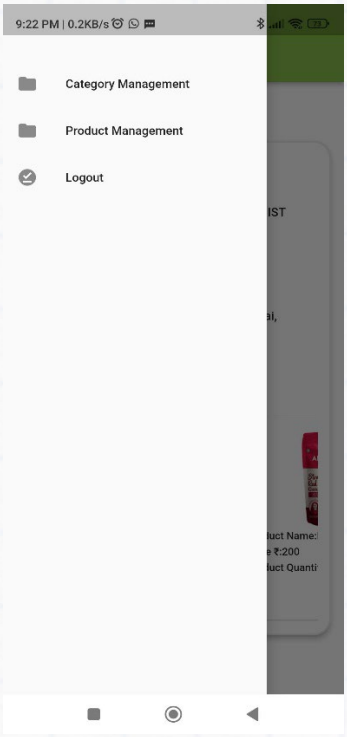


ORDERSUMMARY

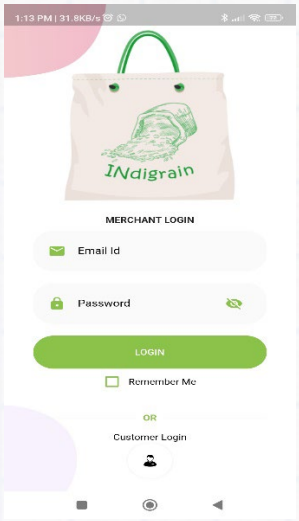


ORDERMANAGEMENT



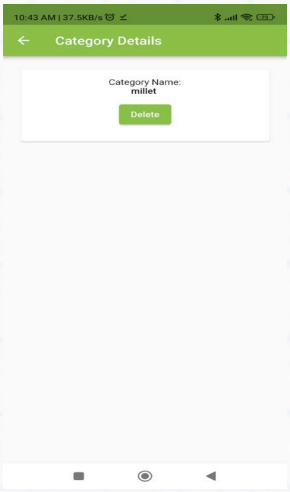


MERCHANT LOGIN



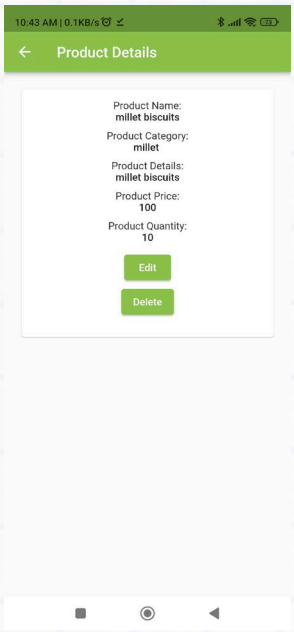
CATEGORY LIST

CATEGORY DETAIL

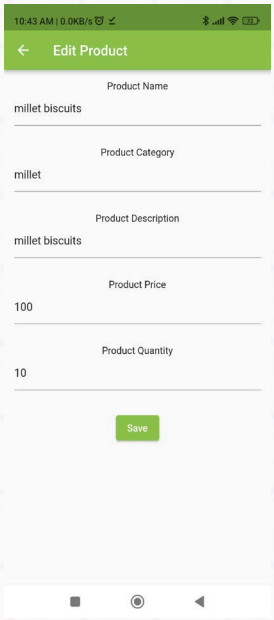


PRODUCTLIST

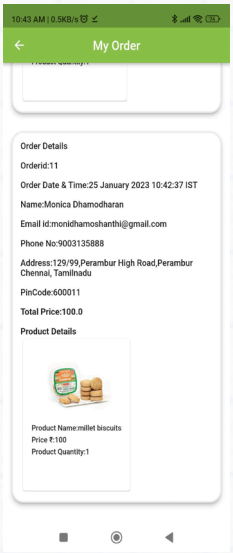
PRODUCTDETAILS



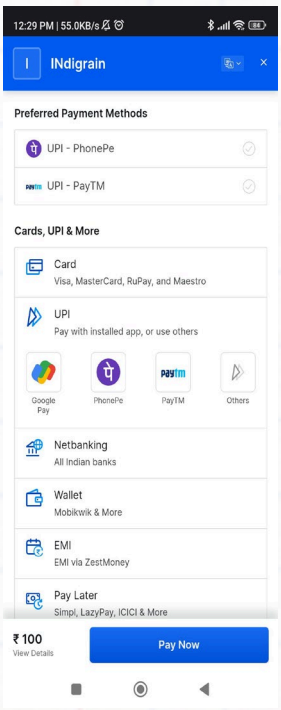
EDIT PRODUCT

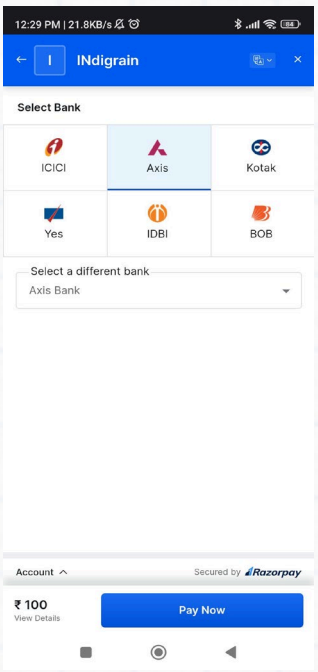


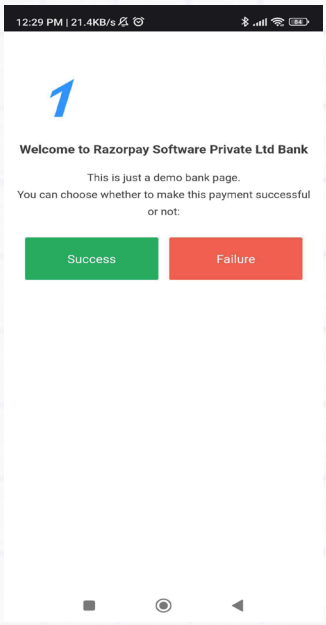
MYORDER



SAMPLE OUTPUT PAYMENT







Part- B Build the Backend and Database for E-marketplace mobile application.

Building the backend of E-marketplace using Spring Boot framework.

Install Java SE 13 (JDK)

Note: Although newer versions of the JDK are available, NetBeans requires a file included in versions 13 and earlier for the installation.

1. Follow this link to download Java SE 13:

https://www.oracle.com/java/technologies/javase/jdk13-archive-downloads.html

2. Select the Windows x64 Installer option for JDK 13.0.2 (scroll down the page to reach this spot). Click the link on the right side of this option to download it.

Note: You may need to create an Oracle User account to download this software. If so, you can use your college email account and address when setting up your account:

3. After downloading, double-click the downloaded file (likely in your Downloads folder) and follow the installation instructions. Leave default settings from the installer as they are.

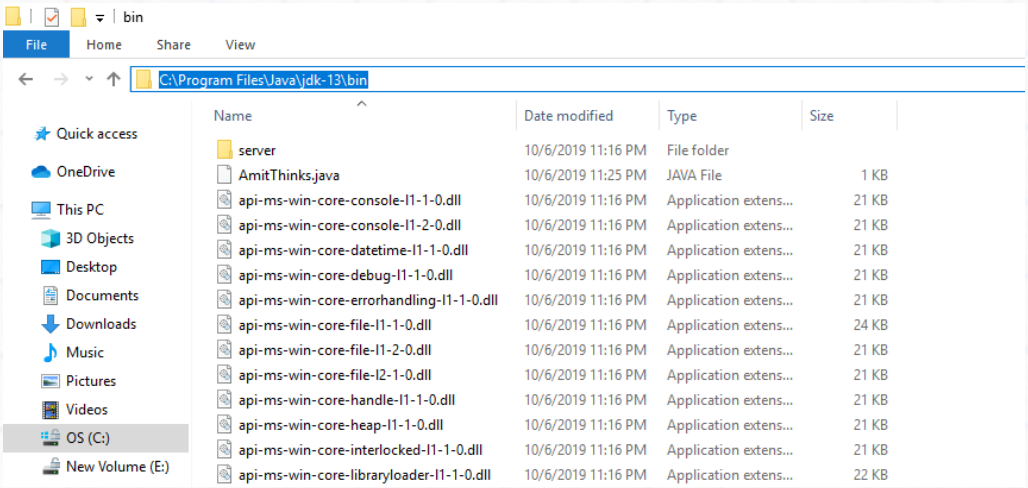
4. Now, let us set the JDK path.

Now, we will see how to set Java JDK Path (Environment Variable).

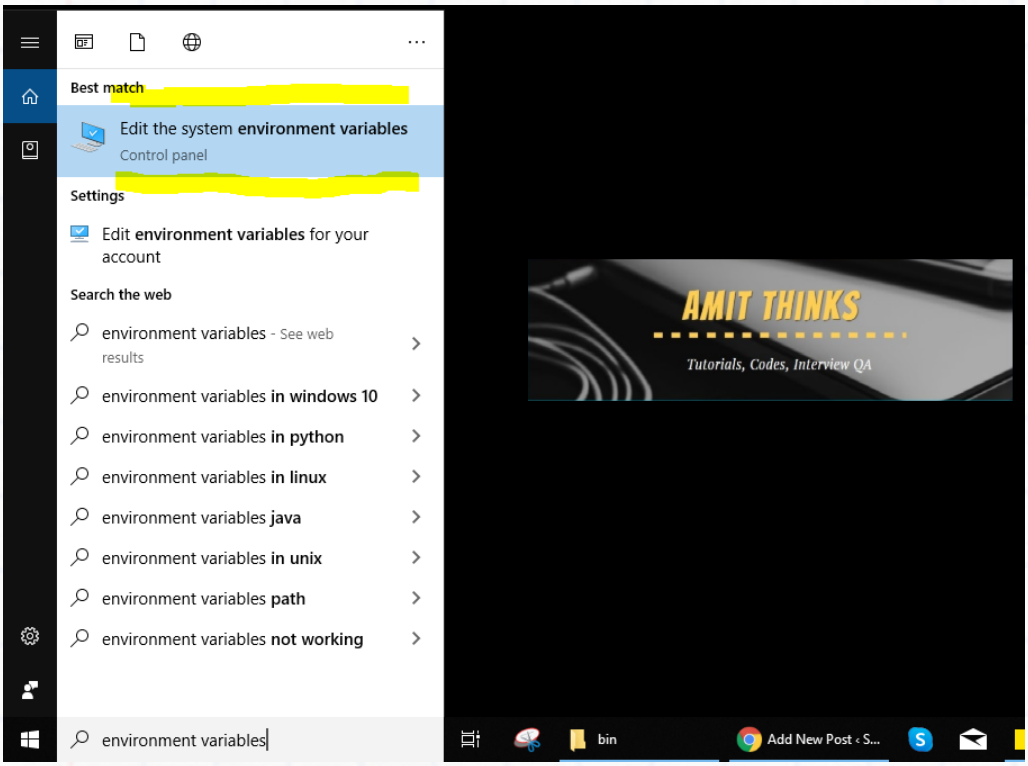
At first, copy the path wherein you installed the Java JDK. Let us copy it first. Remember, we need to copy the bin path i.e. the following on our system:

```
C:\Program Files\Java\jdk-13\bin
```

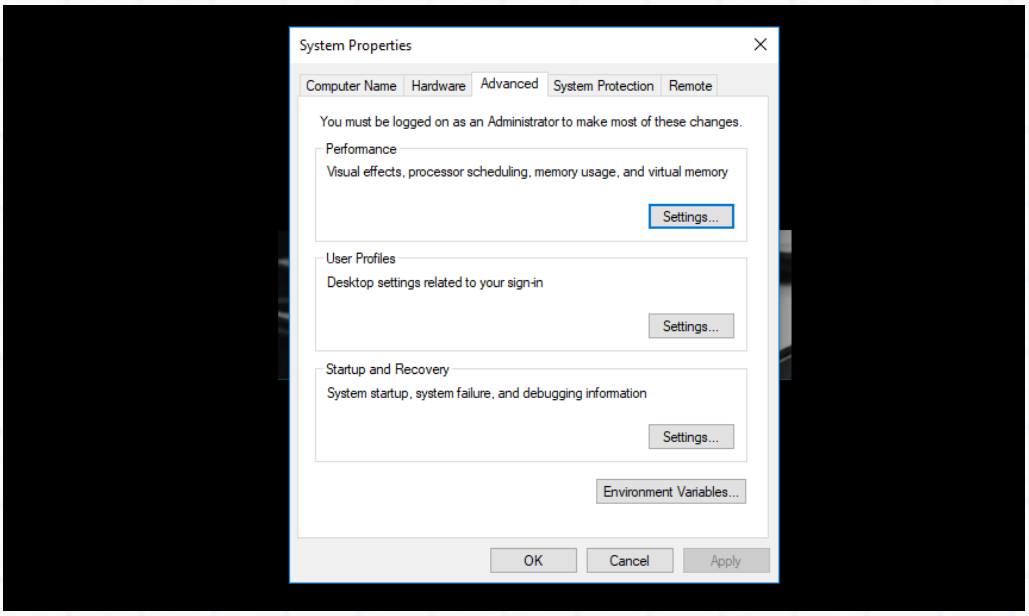
Here's the screenshot of the "bin" path, wherein we installed Java 13:



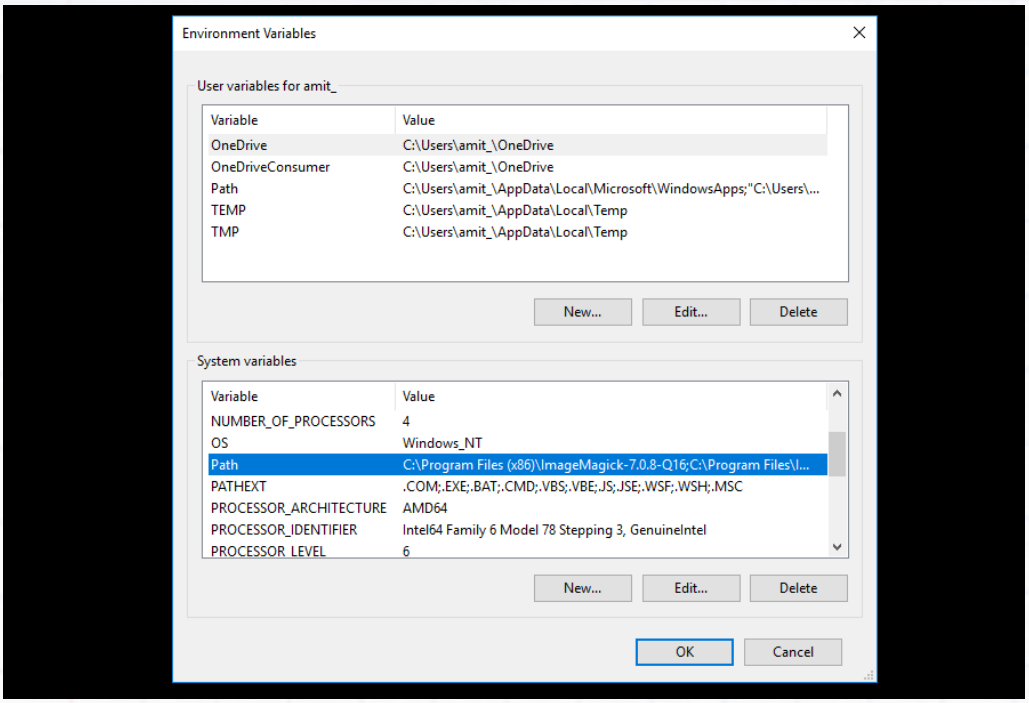
5. To set JDK Path, the easiest way is to type “Environment Variables” on Start. On typing, the following would be visible:



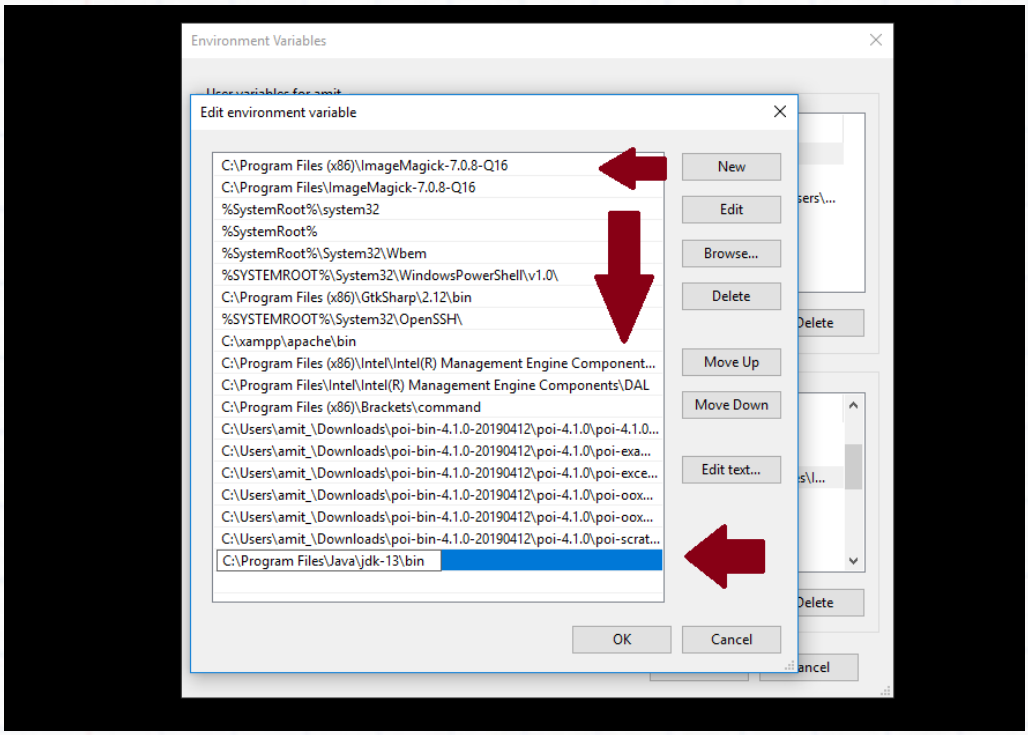
6. Now, click on “Edit Environment Variables” and a new dialog box would be visible:



7. Now, click “Environment Variable” and a new dialog box will open. Go to “User Variables” section.



8. Click “New”. Type PATH in the Variable name and add the Java JDK path “C:\Program Files\Java\jdk-13\bin” as displayed in the below screenshot:



Above, press Ok.

9. Follow a similar process to set System Variables.

10. Now, we will verify the JDK installation.

Now, we can easily verify java installation was successfully or not using the following command on command prompt:

```
java -version
```

Install Apache NetBeans IDE

Note: Don't run the Apache NetBeans installer before Java is installed on your system.

1. Open the web page https://netbeans.apache.org/download/ .

Go to the NetBeans 17 download page by clicking one of the Download buttons.

2. In the next page, make sure to download the Windows 64-bit version of NetBeans.

3. Now go to your Downloads folder (or wherever you had NetBeans downloaded to) and double-click the NetBeans installer file to run it (Apache-NetBeans-17-bin-windows-x64.exe). Click the Next button on the NetBeans 17 installer window.

4. In the License Agreement window, click the checkbox to accept the terms. Then click Next.

5. In the next window, under JDK for the Apache NetBeans IDE, make sure that the

location of the correct JDK has been chosen. You may have multiple versions of JDK on your computer. The version you installed in Step 1 should be specified here (change to the right one if it says different).

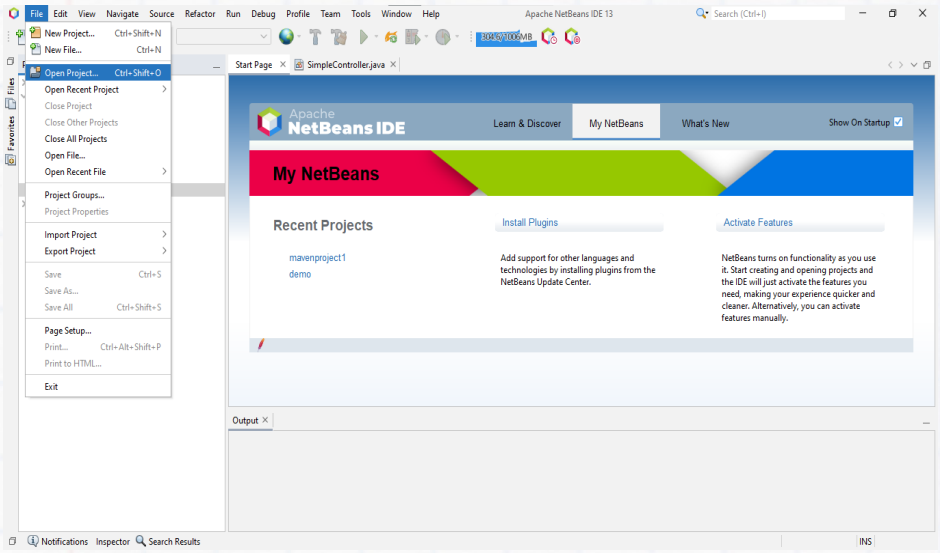
6. Click Install in the next window.

Installation may take a few minutes. After it's done, click the Finish button.

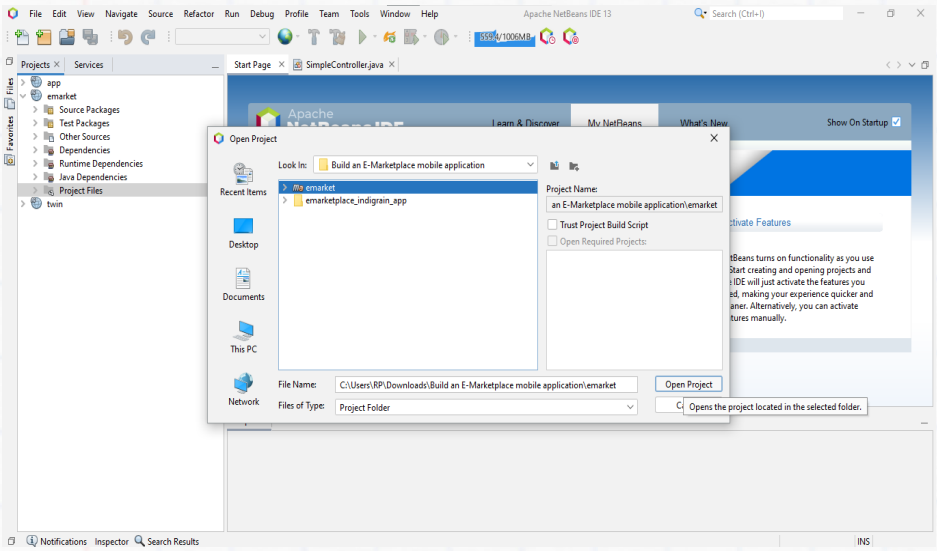
A Java Spring project requires a set of libraries and packages that enable the requested features. For our project, we select Maven as the project management tool. Maven helps to build and manage your Java project. It creates a so-called POM (Project-Object-Model) with all the information and configuration details of the project, which is saved in a pom.xml file.

Importing the Project

1. Open Apache NetBeans, select File › Open Project

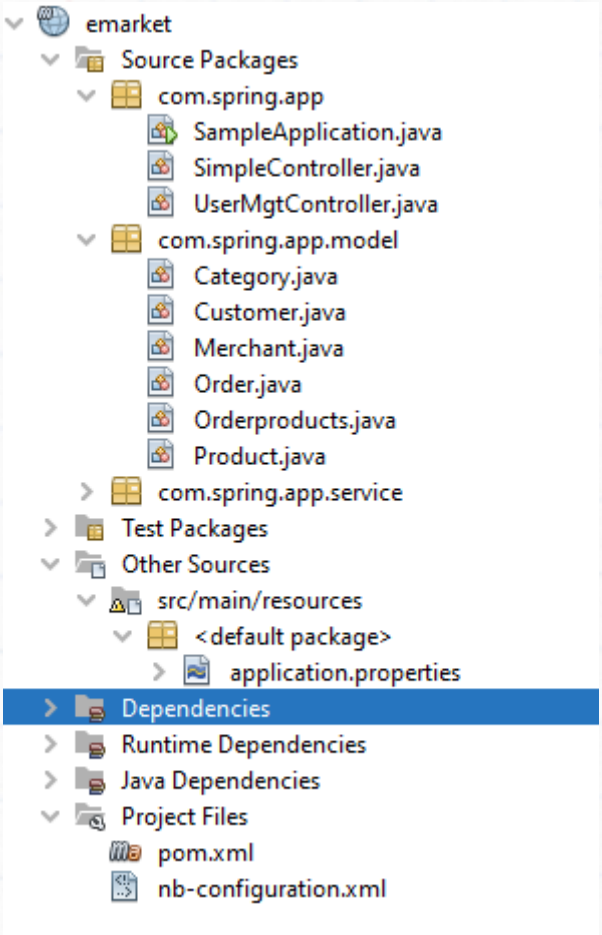


2. Unzip the emarketplace-Copy.zip folder and select the unzip folder containing the Maven project you want to import.



Click Open Project to complete the process.

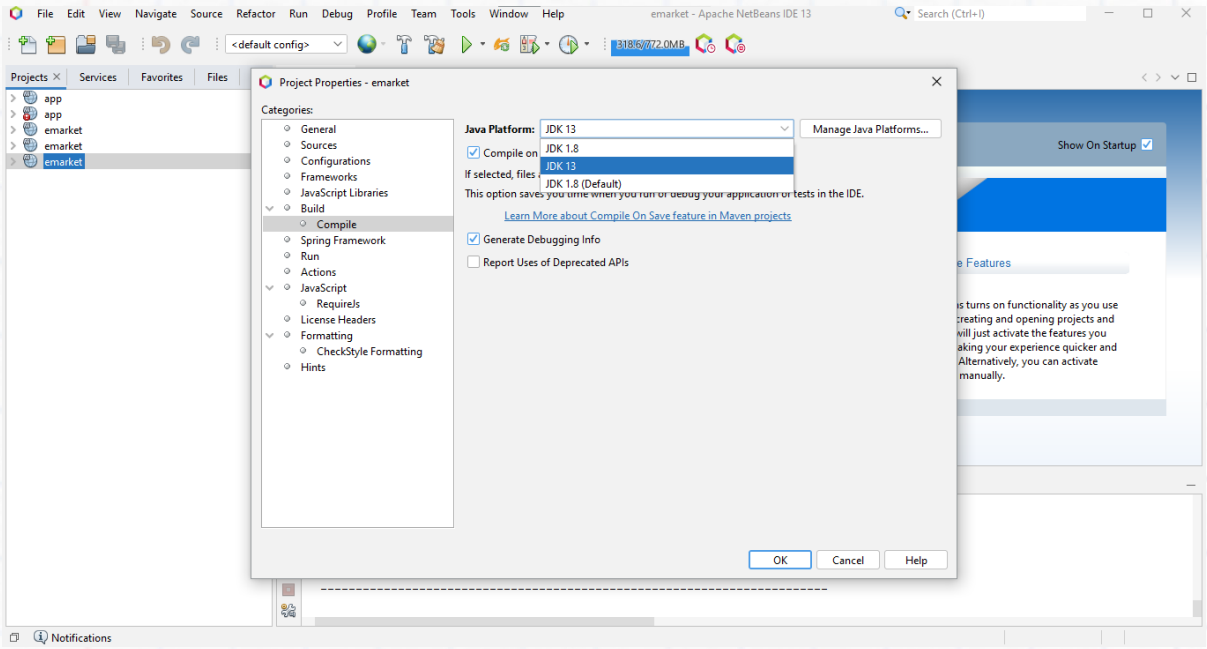
3. The directory structure of the spring boot project will look like this.



To configure your project to use JDK 13:

1. Right-click the project in the Projects window and select Properties from the context menu.

2. In the Project Properties dialog box, choose Build -> Compile and set JDK 13 as the Java Platform



3. Click OK to save the changes.

Create POJOs (plain old Java object) for Merchant, Customer, Category, Product, Order, and Orderproducts.

1. 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, password, addressno, area, city, state, and pincode.

```
    private String id;

    private String email;

    private String password;

    private String phone;
```

```
    private String name;

    private String addressno;

    private String area;

    private String city;

    private String state;

    private String pincode;
```

2. Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). Write the following code.

```
public Customer() {}
```

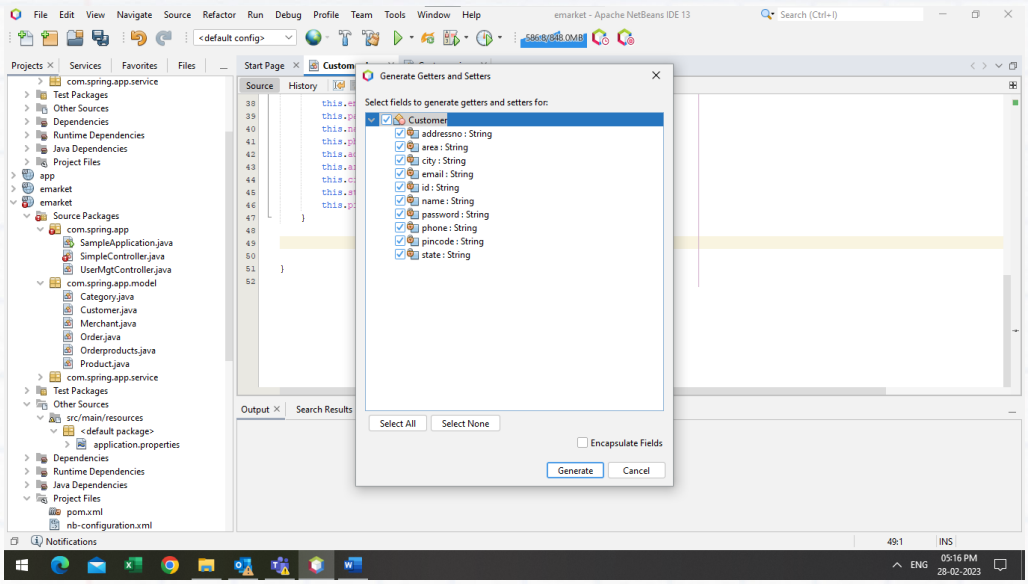
3. Create a constructor with the arguments id, name, email, phone, password, addressno, area, city, state, and pincode. Write the following code.

```
public Customer(String email, String password, String phone, String name, String
id, String addressno, String area, String city, String state, String pincode) {

        this.id = id;

        this.email = email;

        this.password = password;

        this.name = name;

        this.phone = phone;

        this.addressno = addressno;

        this.area = area;

        this.city = city;

        this.state = state;

        this.pincode = pincode;
```

}

4. 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.

5. Merchant.java

In the Projects window, Inside project file > source packages > com.spring.app.model. Open

`Merchant.java` file and write the following code.

Inside Merchant class, Create private fields with their data types for id, name, email, phone, gstno, and password.

```
private String id;

private String email;

private String password;

private String phone;

private String name;

 private String gstno;
```

6. Create an empty constructor (Hibernate, which handles the JPA requires an empty

constructor). Write the following code.

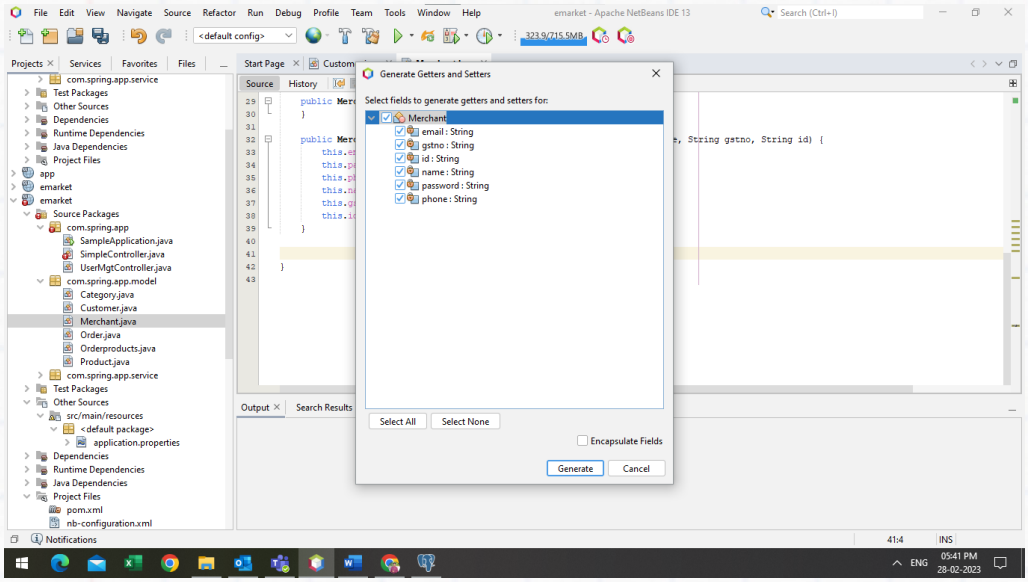
```
public Merchant() {}
```

7. Create a constructor with the arguments id, name, email, phone, gstno, and password. Write the following code.

```
public Merchant(String email, String password, String phone, String name, String gstno, String id) {
        this.email = email;
        this.password = password;
        this.phone = phone;
        this.name = name;
        this.gstno = gstno;
        this.id = id;
    }
```

8. 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 Merchant class.

9. Category.java

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

Inside Category class, Create private fields with their data types for id, name, and image.

```
    private String id;

    private String image;

     private String name;
```

10. Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). Write the following code.

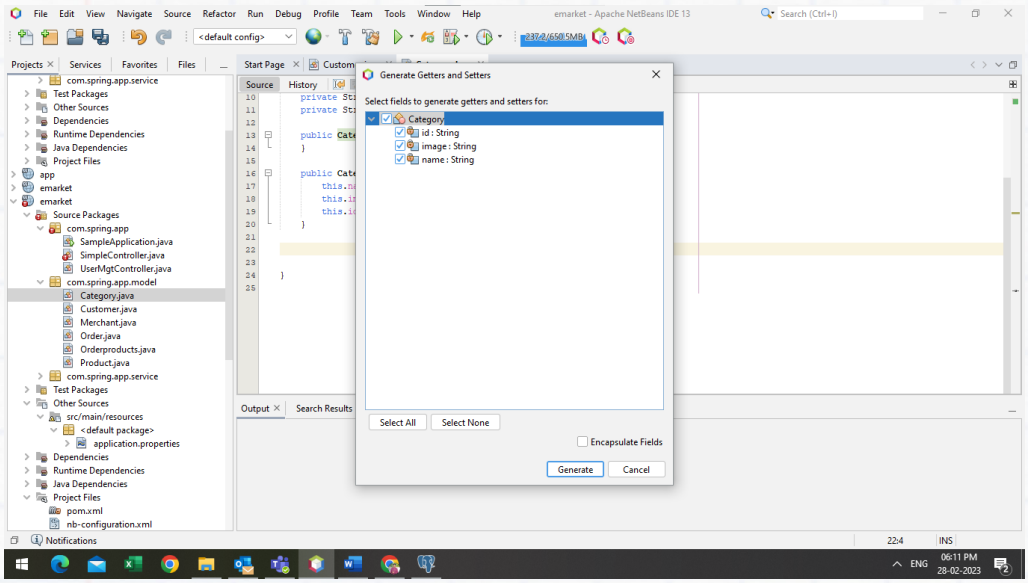
```
  public Category() {}
```

11. Create a constructor with the arguments name, image, and id. Write the following code.

```
public Category(String name, String image, String id) {

        this.name = name;

        this.image = image;

        this.id = id;
    }
```

12. 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 Category class.

13. Product.java

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

Inside Product class, Create private fields with their data types for id, name, description, price, category, quantity, initialquantity and image.

```
private String name;

private String id;

private String description;

private String image;

private String price;

private String category;

private String quantity;

 private String initialquantity ="1";
```

14. Create an empty constructor (Hibernate, which handles the JPA requires an empty

constructor). Write the following code.

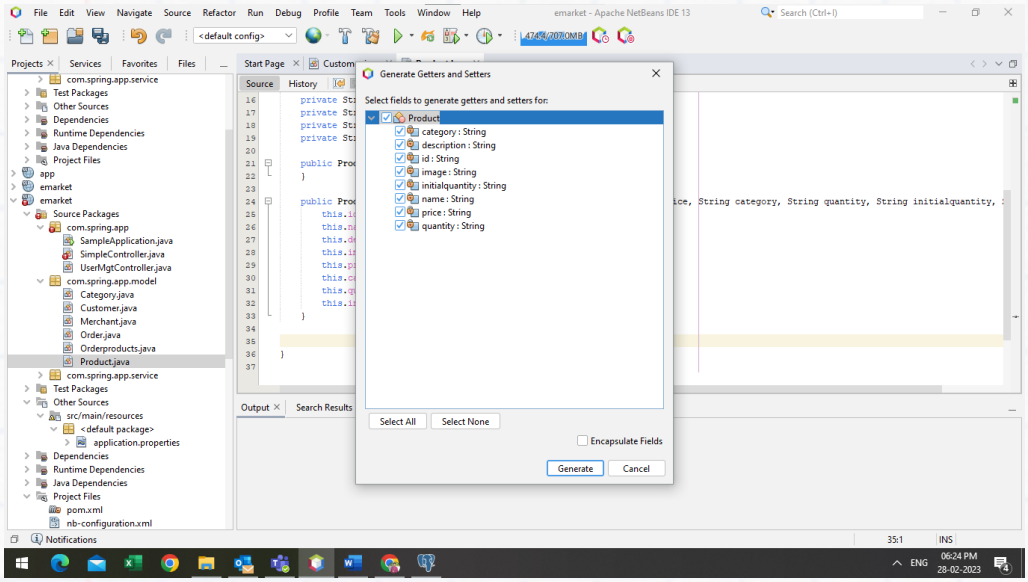
```
public Product() {}
```

15. Create a constructor with the arguments id, name, description, price, category, quantity, initialquantity and image. Write the following code.

```
public Product(String name, String description, String image, String price, String

category, String quantity, String initialquantity, String id) {

        this.id = id;

        this.name = name;

        this.description = description;

        this.image = image;

        this.price = price;

        this.category = category;

        this.quantity = quantity;

        this.initialquantity = initialquantity;

}
```

16. 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 Product class.

17. Order.java

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

Inside Order class, Create private fields with their data types for customerid, customername, customeremail, customerphoneno, customeraddressno, customerarea, customercity, customerstate, customerpincode, totalprice, ordereddatetime, orderid, orderrefid, and productlist.

```
private String totalprice;

private String customerid;

private String customername;

private String customeremail;

private String customerphoneno;

private String customeraddressno;

private String customerarea;

private String customercity;

private String customerstate;
```

```
    private String customerpincode;

    private String ordereddatetime;

    private int orderid;

    private int orderrefid;

     private List<Orderproducts> productlist;
```

18. Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). Write the following code.

```
 public Order() {}
```

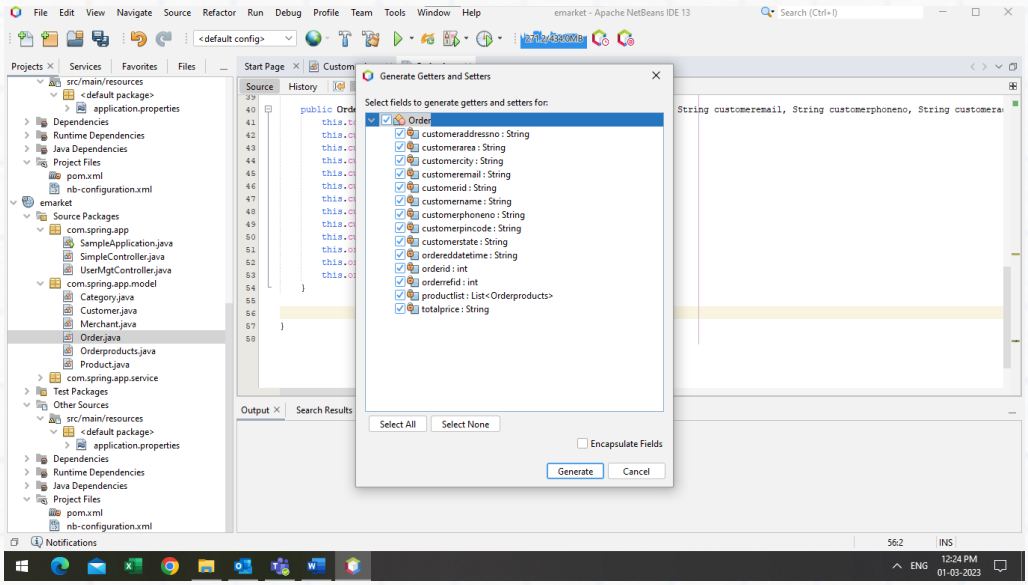
19. Create a constructor with the arguments customerid, customername, customeremail, customerphoneno, customeraddressno, customerarea, customercity, customerstate, customerpincode, totalprice, ordereddatetime, orderid, orderrefid. Write the following code.

```
public Order(String totalprice, String customerid, String customername, String
customeremail, String customerphoneno, String customeraddressno, String
customerarea, String customercity, String customerstate, String customerpincode,
String ordereddatetime, int orderid, int orderrefid) {

        this.totalprice = totalprice;

        this.customerid = customerid;

        this.customername = customername;

        this.customeremail = customeremail;

        this.customerphoneno = customerphoneno;

        this.customeraddressno = customeraddressno;

        this.customerarea = customerarea;

        this.customercity = customercity;
```

```
        this.customerstate = customerstate;

        this.customerpincode = customerpincode;

        this.ordereddatetime = ordereddatetime;

        this.orderid = orderid;

        this.orderrefid = orderrefid;
    }
```

20. 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 Order class.

21. Orderproducts.java

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

Inside Orderproducts class, Create private fields with their data types for productname, productprice, productquantity, productquantity, productid, productimage, and productdescription.

```
    private String productname;

    private String productprice;
```

```
    private String productquantity;

    private String productid;

    private String productimage;

     private String productdescription;
```

22. Create an empty constructor (Hibernate, which handles the JPA requires an empty constructor). Write the following code.

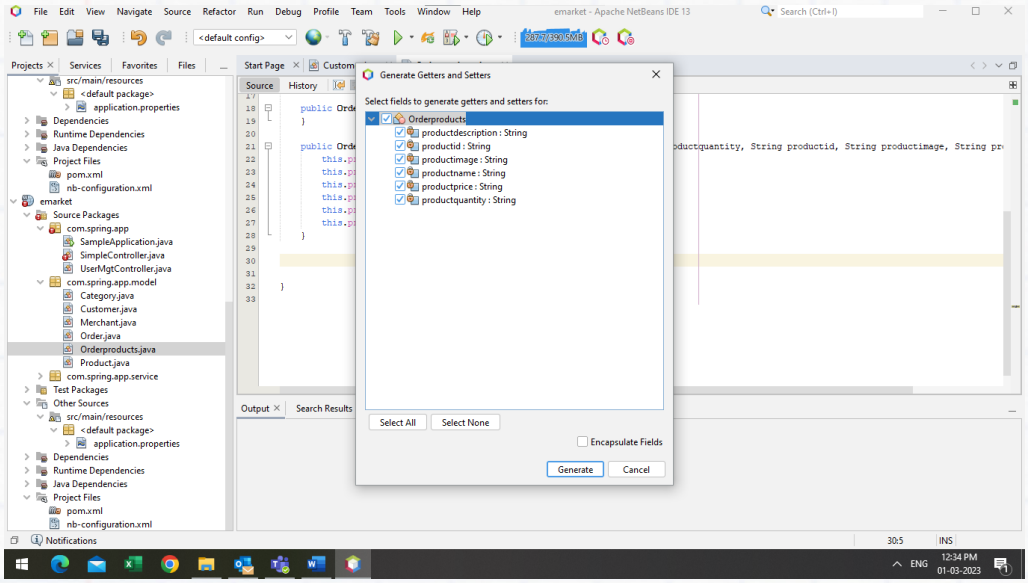
```
 public Orderproducts() {}
```

23. Create a constructor with the arguments productname, productprice, productquantity, productquantity, productid, productimage, and productdescription. Write the following code.

```
public Orderproducts(String productname, String productprice, String

productquantity, String productid, String productimage, String productdescription)

{

        this.productname = productname;

        this.productprice = productprice;

        this.productquantity = productquantity;

        this.productid = productid;

        this.productimage = productimage;

        this.productdescription = productdescription;

 }
```

24. 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 Orderproducts class.

Create Spring Boot API Controller for merchant 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.

Create REST end points that performs the basic database operations such as Create, Read, Update, Delete

Merchant

- Handling merchant login
- Manage Categories
  - List category
  - Insert category
  - Delete category
- Manage Products
  - List product
  - Insert product
  - Update product
  - Delete product
- List Received Order

Customer

- Handling customer login
- Handling customer register
- Manage Profile

- List Order
- List Categories
- List Products
- Order checkout
- Generate Invoice & send via email

1. Handling merchant login

This method is used to login as merchant.

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 = "/merchantlogin", method = RequestMethod.POST)

    public String loginMerchant(@RequestBody Merchant merchant) {

        String s = "select count(*) from merchant where memail=? AND mpassword=?";



        System.out.println("s = " + merchant.getEmail());

        System.out.println("s = " + merchant.getPassword());

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



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

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

        if (count > 0) {

            return "Successfull";

        } else {
```

```
            return "Failure";

        }

    }
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to send email 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 "/merchantlogin" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the loginMerchant method.

Inside loginMerchant method is where you create the query to count data values from the merchant table.

The SQL SELECT statement can be used along with COUNT (*) function to count of all rows present in the merchant 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.

2. List category

This method is used to display categories.

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 = "/category", method = RequestMethod.GET)

    public JSONObject category() {

        String s = "select catcategoryname AS name, catcategoryimage AS image,
catid AS id from category";
```

```
        List<Category> mrlist = jdbc.query(s, new
BeanPropertyRowMapper(Category.class));

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

        JSONObject json = new JSONObject();

        json.put("Category", mrlist);

        if (!mrlist.isEmpty()) {

            json.put("Category", mrlist);

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

            return json;

        }

        return json;

    }
```

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

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

`@RequestMapping(method = RequestMethod.GET)`, and is used to map HTTP GET requests to the mapped controller methods. We used it to return all the categories.

Inside category method is where you create the query to return a list of categories from the category table.

The SQL string contains a query to select all the category details from the category 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.

3. Insert category

This method is used to insert categories.

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 = "/insertcategory", method = RequestMethod.POST)

    public String insertCategory(@RequestBody Category category) {

        String s = "insert into
category(catcategoryname,catcategoryimage)values(?,?)";

        System.out.println("s = " + category.getName());

        System.out.println("s = " + category.getImage());

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

        int a = jdbc.update(s, category.getName(), category.getImage());

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

        if (a == 1) {

            return "Inserted Successfully";

        } else {

            return "Inserted failure";

        }

    }
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to send name and image of a category.

`@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 "/insertcategory" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the insertCategory method.

Inside insertCategory method is where you create the query to insert a category in the category 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 2 values that correspond to 2 '?' respectively.

4. Delete category

This method is used to delete categories.

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 = "/deletecategory", method = RequestMethod.POST)

    public String deleteCategory(@RequestBody Category category) {

        String s = "delete from category where catid= ('" + category.getId() +
"')";

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

        int a = jdbc.update(s);

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

        if (a == 1) {
```

```
            return "Deleted Successfully";

        } else {

            return "Deleted Failure";

        }

    }
```

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

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

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

`@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 "/deletecategory" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the deleteCategory method.

Inside deleteCategory method is where you create the query to delete categories from the category table.

Create a SQL string to delete category by ID from category table. Call the update method of JdbcTemplate and pass the string to be bound to the query.

5. List product

This method is used to list products.

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 = "/listproduct", method = RequestMethod.GET)

    public JSONObject product() {
```

```
        String s = "select proid AS id, proname AS name, proimage AS image,
prodescription AS description, proprice AS price, procategory AS category,
proquantity AS quantity, proinitialquantity AS intialquantity from product";

        List<Product> mrlist = jdbc.query(s, new
BeanPropertyRowMapper(Product.class));

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

        JSONObject json = new JSONObject();

        json.put("Product", mrlist);

        if (!mrlist.isEmpty()) {

            json.put("Product", mrlist);

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

            return json;

        }

        return json;

    }
```

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

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

`@RequestMapping(method = RequestMethod.GET)` is used to map HTTP GET request to the mapped controller methods. We used it to return all the products.

Inside product method is where you create the query to return a list of products from the product table.

The SQL string contains a query to select all the product details from the product 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.

6. Insert product

This method is used to insert products.

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 = "/insertproduct", method = RequestMethod.POST)

    public String insertProduct(@RequestBody Product product) {

        String s = "insert into
product(proname,proimage,proprice,prodescription,procategory,proquantity,proinitia
lquantity)values(?,?,?,?,?,?,?)";



        System.out.println("s = " + product.getName());

        System.out.println("s = " + product.getImage());

        System.out.println("s = " + product.getPrice());

        System.out.println("s = " + product.getDescription());

        System.out.println("s = " + product.getCategory());

        System.out.println("s = " + product.getQuantity());

        System.out.println("s = " + product.getInitialquantity());



        System.out.println("s = " + s);
```

```
        int a = jdbc.update(s, product.getName(), product.getImage(),
product.getPrice(), product.getDescription(), product.getCategory(),
product.getQuantity(), product.getInitialquantity());

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

        if (a == 1) {

            return "Inserted Successfully";

        } else {

            return "Inserted failure";

        }

    }
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to send name, image, price, description, quantity, and initial quantity of a product.

`@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 "/insertproduct" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the insertProduct method.

Inside insertProduct method is where you create the query to insert a product in the product 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 7 values that correspond to 7 '?' respectively.

7. Update product

This method is used to update products.

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 = "/updateproduct", method = RequestMethod.POST)

    public String updateProduct(@RequestBody Product product) {

        String s = "update product set proname= ?, prodescription= ?, procategory=
?, proprice= ?, proquantity= ? where proid='" + product.getId() + "'";

        System.out.println("s = " + product.getName());

        System.out.println("s = " + product.getDescription());

        System.out.println("s = " + product.getCategory());

        System.out.println("s = " + product.getPrice());

        System.out.println("s = " + product.getQuantity());

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

        int a = jdbc.update(s, product.getName(), product.getDescription(),
product.getCategory(), product.getPrice(), product.getQuantity());

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

        if (a == 1) {

            return "Updated Successfully";

        } else {

            return "Updated Failure";

        }
```

```
}
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to send name, image, price, description, quantity and initial quantity of the product.

`@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 "/updateproduct" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the updateProduct method.

Inside updateProduct method is where you create the query to update a product in the product table.

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

The SQL string is used to update the product details by ID and pass the string to the update method of JdbcTemplate followed by object arguments of type string which are the name, description, quantity, price, and category. Note that the ID is only used to find the product to be updated but the ID itself is not updated.

8. Delete product

This method is used to delete products.

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 = "/deleteproduct", method = RequestMethod.POST)

public String deleteProduct(@RequestBody Product product) {

    String s = "delete from product where proid= ('" + product.getId() + "')";

    System.out.println("s = " + s);
```

```
        int a = jdbc.update(s);

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

        if (a == 1) {

            return "Deleted Successfully";

        } else {

            return "Deleted Failure";

        }

    }
```

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

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

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

`@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 "/deleteproduct" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the deleteProduct method.

Inside deleteProduct method is where you create the query to delete product from the product table.

Create a SQL string to delete the products by ID from product table. Call the update method of JdbcTemplate and pass the string to be bound to the query.

9. List Received Order

This method is used to display received orders.

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 = "/merchantorder", method = RequestMethod.POST)
    public JSONObject order() {
        String s = "select distinct osrefid AS orderrefid, max(oscustomername) AS
customername, max(oscustomeremail) AS customeremail, max(oscustomerphone) AS
customerphoneno, max(oscustomeraddressno) AS customeraddressno,
max(oscustomerarea) AS customerarea, max(oscustomercity) AS customercity,
max(oscustomerstate) AS customerstate, max(oscustomerpincode) AS customerpincode,
max(ostotalprice) AS totalprice, max(osordereddatetime) AS ordereddatetime from
ordersummary group by osrefid order by osrefid";
        System.out.println("s = " + s);
        List<Order> orderidList = jdbc.query(s, new
BeanPropertyRowMapper(Order.class));
        System.out.println("orderidList = " + orderidList.isEmpty());
        JSONArray orderArr = new JSONArray();
        JSONObject orderObj = new JSONObject();
        if (!orderidList.isEmpty()) {
            for (Order orObj : orderidList) {
                JSONObject orderDetObj = new JSONObject();
                orderDetObj.put("orderrefid", orObj.getOrderrefid());
                orderDetObj.put("totalprice", orObj.getTotalprice());
                orderDetObj.put("customername", orObj.getCustomername());
                orderDetObj.put("customeremail", orObj.getCustomeremail());
```

```
                orderDetObj.put("customerphoneno", orObj.getCustomerphoneno());
                orderDetObj.put("customeraddressno",
orObj.getCustomeraddressno());
                orderDetObj.put("customerarea", orObj.getCustomerarea());
                orderDetObj.put("customercity", orObj.getCustomercity());
                orderDetObj.put("customerstate", orObj.getCustomerstate());
                orderDetObj.put("customerpincode", orObj.getCustomerpincode());
                orderDetObj.put("ordereddatetime", orObj.getOrdereddatetime());

                String t = "select osproductname AS productname, osproductprice AS
productprice, osproductimage AS productimage, osproductquantity AS
productquantity, osproductprice AS productprice from ordersummary where osrefid="
+ orObj.getOrderrefid();
                System.out.println("t = " + t);
                List<Orderproducts> productlist = jdbc.query(t, new
BeanPropertyRowMapper(Orderproducts.class));
                System.out.println("productlist = " + productlist.isEmpty());
                System.out.println("productlist = " + productlist);
                JSONArray pdlistArr = new JSONArray();
                if (!productlist.isEmpty()) {
                    for (Orderproducts pdlist : productlist) {
                        JSONObject pdlisObj = new JSONObject();
```

```
                    pdlisObj.put("productname", pdlist.getProductname());

                    pdlisObj.put("productquantity",
pdlist.getProductquantity());

                    pdlisObj.put("productprice", pdlist.getProductprice());

                    pdlisObj.put("productimage", pdlist.getProductimage());

                    pdlistArr.add(pdlisObj);

                }

                orderDetObj.put("pdlist", pdlistArr);

                orderArr.add(orderDetObj);

            }

        }

        orderObj.put("orderdetails", orderArr);

    }

    return orderObj;

}
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to send customer details, order details and product details.

Inside order method is where you create the query to return customer details, product details and order details as list from the ordersummary table.

The SQL s string contains a query to select the customer details, product details and order details from the ordersummary 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.

10. Handling customer login

This method is used to login as customer.

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 loginCustomer(@RequestBody Customer customer) {

        String s = "select cemail AS email, cpassword AS password, cname AS name,
cid AS id, cphone AS phone,caddressno AS addressno, carea AS area, ccity AS city,
cstate AS state, cpincode AS pincode from customer where cemail=CAST('" +
customer.getEmail() + "' AS VARCHAR) AND cpassword=CAST('" +
customer.getPassword() + "' AS VARCHAR)";

        System.out.println("s = " + customer.getEmail());

        System.out.println("s = " + customer.getPassword());

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

        List<Customer> mrlist = jdbc.query(s, new
BeanPropertyRowMapper(Customer.class));

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

        JSONObject json = new JSONObject();

        json.put("Customerdetails", mrlist);
```

```
        if (!mrlist.isEmpty()) {

            json.put("Customerdetails", mrlist);

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

            return json;

        }

        return json;

    }
```

`@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 email and password 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 "/customerlogin" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the loginCustomer method.

Inside loginCustomer 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.

11. Handling customer registration

This method is used to register as customer.

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 String customerRegister(@RequestBody Customer customer) {
        String s = "insert into
customer(cname,cemail,cpassword,cphone)values(?,?,?,?)";
        System.out.println("s = " + customer.getEmail());
        System.out.println("s = " + customer.getPassword());
        System.out.println("s = " + customer.getName());
        System.out.println("s = " + customer.getPhone());
        System.out.println("s = " + s);
        int a = jdbc.update(s, customer.getName(), customer.getEmail(),
customer.getPassword(), customer.getPhone());
        System.out.println("a = " + a);
        if (a == 1) {
      return "Registered Successfully";
        } else {
            return "Registeration failure";
        }
    }
```

`@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 name, email ID, password, and phone No 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.

Inside customerRegister method is where you create the query to insert customer details 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.

12. Manage profile

This method is used to update customer profile.

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 = "/updateprofile", method = RequestMethod.POST)

    public String updateProfile(@RequestBody Customer customer) {

        String s = "update customer set cemail= ?, cpassword= ?, cname= ?, cphone=
?, caddressno= ?, carea= ?, ccity= ?, cstate= ?, cpincode= ? where cid=('" +
customer.getId() + "')";

        System.out.println("s = " + customer.getEmail());

        System.out.println("s = " + customer.getPassword());
```

```
        System.out.println("s = " + customer.getName());

        System.out.println("s = " + customer.getPhone());

        System.out.println("s = " + customer.getAddressno());

        System.out.println("s = " + customer.getArea());

        System.out.println("s = " + customer.getCity());

        System.out.println("s = " + customer.getState());

        System.out.println("s = " + customer.getPincode());

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

        int a = jdbc.update(s, customer.getEmail(), customer.getPassword(),
customer.getName(), customer.getPhone(), customer.getAddressno(),
customer.getArea(), customer.getCity(), customer.getState(),
customer.getPincode());

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

        if (a == 1) {

            return "Updated Successfully";

        } else {

            return "Updated Failure";

        }

    }
```

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

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

`@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 "/updateprofile" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the updateProfile method.

Inside updateProfile method is where you create the query to update customer details 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 update the customer details by ID and pass the string to the update method of JdbcTemplate followed by object arguments of type string which are the email, password, name, phone, address no, state, city, area, and pin code. Note that the ID is only used to find the customer to be updated but the ID itself is not updated.

13. List Order

This method is used to display history of orders.

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 = "/customerorder", method = RequestMethod.POST)

    public JSONObject customerOrder(@RequestBody Order order) {

        String s = "select distinct osrefid AS orderrefid, oscid AS customerid,
oscustomername AS customername, oscustomeremail AS customeremail, oscustomerphone
AS customerphoneno, oscustomeraddressno AS customeraddressno, oscustomerarea AS
customerarea, oscustomercity AS customercity, oscustomerstate AS customerstate,
oscustomerpincode AS customerpincode, ostotalprice AS totalprice,
osordereddatetime AS ordereddatetime from ordersummary where oscid=('" +
order.getCustomerid() + "') order by osrefid";

        System.out.println("s = " + s);
```

```
        List<Order> orderidList = jdbc.query(s, new
BeanPropertyRowMapper(Order.class
        ));
        System.out.println("orderidList = " + orderidList.isEmpty());
        JSONArray orderArr = new JSONArray();
        JSONObject orderObj = new JSONObject();
        if (!orderidList.isEmpty()) {
            for (Order orObj : orderidList) {
                JSONObject orderDetObj = new JSONObject();
                orderDetObj.put("orderrefid", orObj.getOrderrefid());
                orderDetObj.put("customerid", orObj.getCustomerid());
                orderDetObj.put("customername", orObj.getCustomername());
                orderDetObj.put("customeremail", orObj.getCustomeremail());
                orderDetObj.put("customerphoneno", orObj.getCustomerphoneno());
                orderDetObj.put("customeraddressno",
orObj.getCustomeraddressno());
                orderDetObj.put("customerarea", orObj.getCustomerarea());
                orderDetObj.put("customercity", orObj.getCustomercity());
                orderDetObj.put("customerstate", orObj.getCustomerstate());
                orderDetObj.put("customerpincode", orObj.getCustomerpincode());
                orderDetObj.put("totalprice", orObj.getTotalprice());
```

```
                orderDetObj.put("ordereddatetime", orObj.getOrdereddatetime());


                String t = "select osproductname AS productname, osproductprice AS
productprice, osproductimage AS productimage, osproductquantity AS
productquantity, osproductprice AS productprice, osproductdescription AS
productdescription from ordersummary where oscid= '" + orObj.getCustomerid() + "'
AND osrefid= '" + orObj.getOrderrefid() + "'";
                System.out.println("t = " + t);
                List<Orderproducts> productlist = jdbc.query(t, new
BeanPropertyRowMapper(Orderproducts.class
                ));
        System.out.println("productlist = " + productlist.isEmpty());
                System.out.println("productlist = " + productlist);
                JSONArray pdlistArr = new JSONArray();
                if (!productlist.isEmpty()) {
                    for (Orderproducts pdlist : productlist) {
                        JSONObject pdlisObj = new JSONObject();
                        pdlisObj.put("productname", pdlist.getProductname());
                        pdlisObj.put("productquantity",
pdlist.getProductquantity());
                        pdlisObj.put("productprice", pdlist.getProductprice());
                        pdlisObj.put("productimage", pdlist.getProductimage());
```

```
                    pdlisObj.put("productdescription",
pdlist.getProductdescription());
                    pdlistArr.add(pdlisObj);
                }
                orderDetObj.put("pdlist", pdlistArr);
                orderArr.add(orderDetObj);
            }
        }
        orderObj.put("orderdetails", orderArr);
    }
    return orderObj;
}
```

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

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

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

`@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 "/customerorder" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the customerOrder method.

Inside order method is where you create the query to return customer details, product details and order details as list based on customer ID from the ordersummary table.

The SQL s string contains a query to select the customer details, product details and order details by customer ID from the ordersummary 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.

14. List Products

This method is used to display products.

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 = "/product", method = RequestMethod.POST)

    public JSONObject product(@RequestBody Product product) {

        String s = "select procategory AS category, proname AS name,
prodescription AS description, proprice AS price, proimage AS image,
proinitialquantity AS initialquantity, proquantity AS quantity, proid AS id from
product where procategory=CAST('" + product.getCategory() + "' AS VARCHAR)";

        System.out.println("select procategory AS category, proname AS name,
prodescription AS description, proprice AS price, proimage AS image,
proinitialquantity AS initialquantity, proquantity AS quantity, proid AS id from
product where procategory=CAST('" + product.getCategory() + "' AS VARCHAR)");

        List<Product> mrlist = jdbc.query(s, new
BeanPropertyRowMapper(Product.class

        ));

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

        JSONObject json = new JSONObject();

        json.put("Productdetails", mrlist);

        if (!mrlist.isEmpty()) {
```

```
            json.put("Productdetails", mrlist);

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

            return json;

        }

        return json;

    }
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to return all the products.

`@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 "/product" URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the product method.

Inside product method is where you create the query to return products based on category from the product table.

The SQL string contains a query to select the product details by category from the product 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.

15. Order checkout

This method is used to display order summary.

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 = "/ordersummary", method = RequestMethod.POST)
    public String insertOrder(@RequestBody List<Order> odlist) {
        String maxid = "select coalesce(max(osrefid),0) AS refid from
ordersummary";
        int maxrefid = jdbc.queryForObject(maxid, Integer.class);

        if (!odlist.isEmpty()) {
            for (Orderproducts odlist1 : odlist.get(0).getProductlist()) {
                int quantity = Integer.parseInt(odlist1.getProductquantity());
                String s = "select proid AS id, proname AS name, proimage AS
image, prodescription AS description, proprice AS price, procategory AS category,
proquantity AS quantity, proinitialquantity AS intialquantity from product where
proid = " + Integer.parseInt(odlist1.getProductid()) + "";
                List<Product> mrlist = jdbc.query(s, new
BeanPropertyRowMapper(Product.class));
                System.out.println("totalquantity = " + mrlist);

                for (int i = 0; i < mrlist.size(); i++) {
                    int totalquantity =
Integer.parseInt(mrlist.get(i).getQuantity());
                    if (quantity > totalquantity) {
                        return "Some Products are Out of Stock";
```

```
                }

            }

        }

    }


        String s = "insert into
ordersummary(osproductname,osproductprice,osproductquantity,osproductimage,ostotal
price,osordereddatetime,ospid,oscid,oscustomername,oscustomeremail,oscustomerphone
,oscustomeraddressno,oscustomerarea,oscustomercity,oscustomerstate,oscustomerpinco
de,osrefid)"
                + "values(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";
        List<Object[]> dataObjList = new ArrayList<>();
        if (!odlist.isEmpty()) {
            Date d = new Date();
            SimpleDateFormat formatter = new SimpleDateFormat("dd MMMM yyyy
HH:mm:ss z");
            String strDate = formatter.format(d);
            String id = odlist.get(0).getCustomerid();
            String totalprice = odlist.get(0).getTotalprice();
            String name = odlist.get(0).getCustomername();
            String email = odlist.get(0).getCustomeremail();
            String phone = odlist.get(0).getCustomerphoneno();
```

```
String address = odlist.get(0).getCustomeraddressno();

String area = odlist.get(0).getCustomerarea();

String city = odlist.get(0).getCustomercity();

String state = odlist.get(0).getCustomerstate();

String pincode = odlist.get(0).getCustomerpincode();



for (Orderproducts odlist1 : odlist.get(0).getProductlist()) {

    Object[] dataObjArr = new Object[17];

    dataObjArr[0] = odlist1.getProductname();

    dataObjArr[1] = odlist1.getProductprice();

    dataObjArr[2] = odlist1.getProductquantity();

    dataObjArr[3] = odlist1.getProductimage();

    dataObjArr[4] = totalprice;

    dataObjArr[5] = strDate;

    dataObjArr[6] = Integer.parseInt(odlist1.getProductid());

    dataObjArr[7] = Integer.parseInt(id);

    dataObjArr[8] = name;

    dataObjArr[9] = email;

    dataObjArr[10] = phone;

    dataObjArr[11] = address;

    dataObjArr[12] = area;
```

```
                dataObjArr[13] = city;

                dataObjArr[14] = state;

                dataObjArr[15] = pincode;

                dataObjArr[16] = maxrefid + 1;

                System.out.println("dataObjArr = " + Arrays.toString(dataObjArr));



                dataObjList.add(dataObjArr);

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

      }

        }


        int[] a = jdbc.batchUpdate(s, dataObjList);

        System.out.println("a = " + Arrays.toString(a));

        System.out.println("a.length = " + a.length);

        if (a.length >= 1) {

            String t = "";

            for (Orderproducts odlist1 : odlist.get(0).getProductlist()) {

                t += "update product set proquantity = (CAST(proquantity AS
INTEGER)-" + odlist1.getProductquantity() + ") where proid = (" +
odlist1.getProductid() + ");";

            }
```

```
            System.out.println("t = " + t);

            jdbc.update(t);

            String htmlCnt = sendMail(dataObjList);

            return "Inserted Successfully";


        } else {

            return "Inserted failure";

        }

    }
```

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

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

`@RequestMapping(method = RequestMethod.POST)` is used to map HTTP POST request to the mapped controller methods. We used it to send order details, product details and customer details.

`@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 “/ordersummary” URL with a Post JSON body, the HttpMessageConverters converts the JSON request body into a Post object and passes it to the insertOrder method.

Inside order method is where you create the query to insert customer details, product details in the ordersummary table and update product quantity in the product table.

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

The SQL string “s” 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.

The SQL string “t” is used to update the product quantity by ID and pass the string to the update method of JdbcTemplate. Note that the ID is only used to find the customer to be updated but the ID

itself is not updated.

16. Generate Invoice & send via email

This method is used to send invoice via Gmail.

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 = "/sendmail", method = RequestMethod.POST)

    public String sendMail(List<Object[]> dataObjList) {



        // Recipient's email ID needs to be mentioned.

        String to = "";

        String OrderId = "";

        String OrderedDateTime = "";

        String Customername = "";

        String Customernumber = "";

        String Customeraddress = "";

        String Totalprice = "";



        // Sender's email ID needs to be mentioned

        String from = "indigrainmarketplace@gmail.com";



        final String username = "indigrainmarketplace@gmail.com";
```

```
        final String password = "qgcilfhiyxhviqvt";

        String HtmLFinal = "";

        String HtmlCntTableRow = "";



        for (int i = 0; i < dataObjList.size(); i++) {

            int sNo = i + 1;



            Object[] s = dataObjList.get(i);

            System.out.println("Object = " + s[0]);

            System.out.println("Object = " + s[1]);

            System.out.println("Object = " + s[2]);

            Totalprice = s[4].toString();

            OrderedDateTime = s[5].toString();

            to = s[9].toString();

            Customername = s[8].toString();

            Customernumber = s[10].toString();

            Customeraddress = s[11] + "," + s[12] + "," + s[13] + "," + s[14] +
","," + s[15].toString();

            OrderId = s[16].toString();
```

```
            HtmlCntTableRow += "<tr><td style='border: 1px solid #25a7e7; border-
collapse: collapse; text-align: center;'>" + sNo + "</td><td style='border: 1px
solid #25a7e7; border-collapse: collapse;'>" + s[0] + "</td><td style='border: 1px
solid #25a7e7; border-collapse: collapse; text-align: right;'>" + s[2] + "</td><td
style='border: 1px solid #25a7e7; border-collapse: collapse; text-align: right;'>"
+ s[1] + "&#8377;</td></tr><tr>";

        }

        String htmlCnt1 = "<html>"

                + "<head>"

                + "</head>"

                + "<body>"

                + "<h1 style='color: #25a7e7; text-align:
center;'>Invoice</h1><div><table style='width:
100%'><tr><td><table><tr><td><b>Bill To:</b></td></tr><tr><td>Order ID: " +
OrderId + "</td></tr><tr><td>Ordered Date & Time: " + OrderedDateTime +
"</td></tr><tr><td>Customer Name: " + Customername + "</td></tr><tr><td>Contact
Number: " + Customernumber + "</td></tr><tr><td>Address: " + Customeraddress +
"</td></tr><tr><td>Email ID: " + to + "</td></tr><tr><th></tr></table></td><td
align='right'><img width='60'
src=\"https://lh3.googleusercontent.com/PmbcUFjN8f9K8_5fR-6M-m8K-
uHjNqeEI69X_arC2kVkqQcJeDzd5bhhVHJ3gv5mX3Qyc0OhZYECu_Vpx-CuHYr3Hdebey5t01QJstD-
04q8ge7uynwkFcDQWhbhhOKxZ4dnluJwsGaGUsVXfJFzBXGjvBgV_6yscWIdphFjdCCqFCxa3QWtP3wZll
6pk548FN55wPFjEirszVgsfMDxlI4Qin7VeYotLaikRfDMjiApqm3ifCFGymeWMAKvZaKrC2Km8SMGMhpS
xRS9yi_zvguEfxTavCKL10EbFi2HbbGdBmkaoc9wjuAU7Zam0Uu-FsN6prMCNVVdORz0PaX-
pH1YJEHco3ssQ_LsHRG5HC5K09OayzWD5KAiINY-dlt1JB6ny50VQha778ZxT7Uz-
sLZcXG4W11u8iquzLb2JSmvp44RyiPGfuT2fDLGybYG2AP67cH9Azhq3P_6biCckbzJLt9Mo-
```

```
61yygpi1i4eg0NiIqoTlEo4N8ytEI2-
tFbEaRBfjQKYnW0eAmGh2oY7j3wLYwYENIYkUiUKmnp5AJzkd5nuEIn1ddRyowULr3ducnonywGSDkuHqQ
ofgUetupW1La7B9sWxUSgdEbSCP8Vxf9jKo_Nx13p4aiydZawcqW7tidFkpopf3AoFDUkUfsWKaX4Wmh7g
HUfrd6V8uDKw58u1DlaDsbCfvmvFkq3qkJndy_OSW_dHiSF5fP3IPB_RBY9bJQWwsuJwvoaf-
oCUIf2h5URUSSLkh7e=w285-h358-no?authuser=0\" /><br
/><b>Indigrain</b></td></tr></table></div><br /><table style='width: 100%; border:
1px solid #25a7e7; border-collapse: collapse;'><tr><td style='border: 1px solid
#25a7e7; border-collapse: collapse; background-color: #25a7e7; color: white; text-
align: center;'>Item #</td><td style='border: 1px solid #25a7e7; border-collapse:
collapse; background-color: #25a7e7; color: white; text-align: center;'>Product
Name</td><td style='border: 1px solid #25a7e7; border-collapse: collapse;
background-color: #25a7e7; color: white; text-align: center;'>Quantity</td><td
style='border: 1px solid #25a7e7; border-collapse: collapse; background-color:
#25a7e7; color: white; text-align: center;'>Total Price</td></tr>";


        String htmlCnt2 = "<tr><td colspan=\"3\" style='border: 1px solid #25a7e7;
border-collapse: collapse; text-align: right;'><b>Bill Amount</b></td><td
style='text-align: right;'><b>" + Totalprice + "&#8377;</b></td></tr></table>"
                + "</body>"
                + "</html>";
        HtmLFinal = htmlCnt1 + HtmlCntTableRow + htmlCnt2;
        Properties prop = new Properties();
        prop.put("mail.smtp.host", "smtp.gmail.com");
        prop.put("mail.smtp.port", "465");
        prop.put("mail.smtp.auth", "true");
```

```
        prop.put("mail.smtp.socketFactory.port", "465");
        prop.put("mail.smtp.socketFactory.class",
"javax.net.ssl.SSLSocketFactory");


        Session session = Session.getInstance(prop,
                new javax.mail.Authenticator() {
            @Override
            protected PasswordAuthentication getPasswordAuthentication() {
                return new PasswordAuthentication(username, password);
            }
        });


        try {
            // Create a default MimeMessage object.
            Message message = new MimeMessage(session);


            // Set From: header field of the header.
            message.setFrom(new InternetAddress(from));


            // Set To: header field of the header.
            message.setRecipients(Message.RecipientType.TO,
```

```
                InternetAddress.parse(to));


            // Set Subject: header field

            message.setSubject("Invoice");


            // Send the actual HTML message, as big as you like

            message.setContent(HtmLFinal, "text/html");


            // Send message

            Transport.send(message);

          System.out.println("Sent message successfully....");


        } catch (MessagingException e) {

            throw new RuntimeException(e);

        }

        return "Invoice Generated";

    }
```

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

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

`@RequestMapping(method = RequestMethod.POST` is used to map HTTP POST request to the

mapped controller methods. We used it to send invoice via mail to customer.

Configure pom.xml.

Open `pom.xml` file

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

Configure Spring Data source.

`application. Properties` is used to add the Spring Boot application's configurations such as: database configuration (PostgreSQL), server configuration.

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

Since we're using PostgreSQL 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.jpa.hibernate.ddl-auto=none

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres

spring.datasource.username=postgres

spring.datasource.password=chonar@13

spring.mvc.hiddenmethod.filter.enabled=true

spring.datasource.hikari.maximum-pool-size=2
```

Run the Spring Boot Project file.

Right-click on the project file and click on "Clean and Build".

Installing Resin

1. Go to link. Click on Download for Resin 4.0
2. Unzip resin-4.0.x.zip
3. Define the environment variable RESIN_HOME to the location of Resin, for

example `C:\Users\RP\Downloads\resin`

4. Follow the similar process like setting Environment Variables in Java to set RESIN_HOME

5. Execute `resin.exe` or run-in command prompt

```
resin/bin ./start.bat;
tail -f ../log/jvm-app-0.log;
```

Note: The resin server listens at port 8080 in the default configuration.
To fix 8080 ports already in use
Step 1: Open command prompt as administrator and find the process id that is using the port 8080.

```
netstat -ano | findstr 8080
```

Step 2: Kill the process using process id in above result.

```
taskkill /F /pid 1088
```

Deploying war file in the resin.

1. Go to spring boot project folder -> inside target folder you will find emarker.war file
2. Copy the .war file (E.g.: emarket.war) -> inside resin folder -> webapps folder
3. Start the resin server

   Execute `resin.exe`

   or run-in command prompt

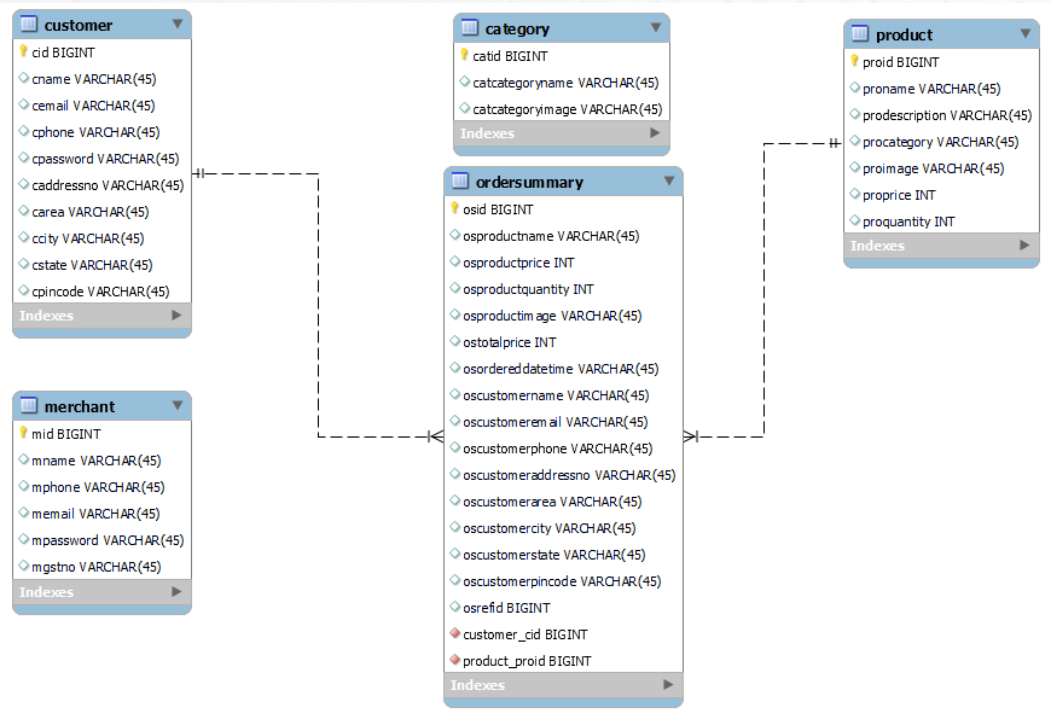
```
resin/bin ./start.bat;
tail -f ../log/jvm-app-0.log;
```

4. Your .war file will be extracted automatically to a folder that has the same name (without extension) (E.g.: webapp)

Creating a database for E-marketplace in PostgreSQL.

1. Create a e market database and Create merchant, customer, category, product, and order summary table, populate the table with data, retrieve and store data for future use, or delete if needed

2. Database Design

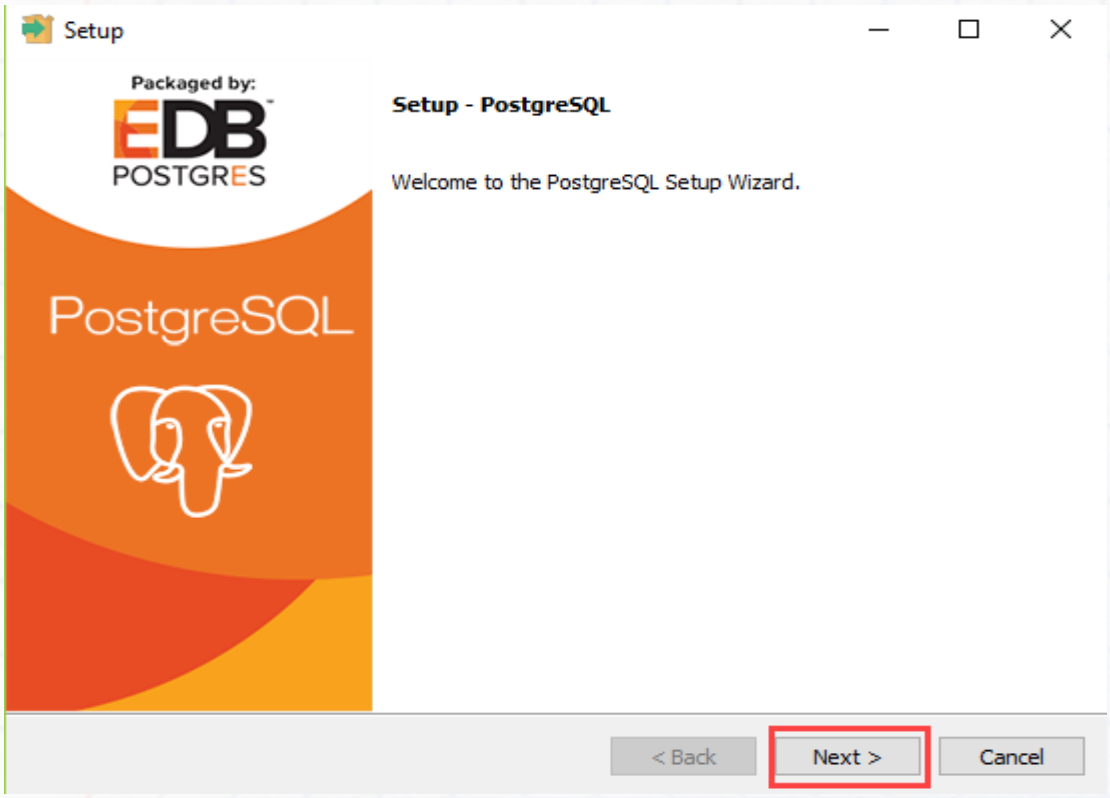


3. Downloading PostgreSQL Installer for Windows

Go to link. Download PostgreSQL

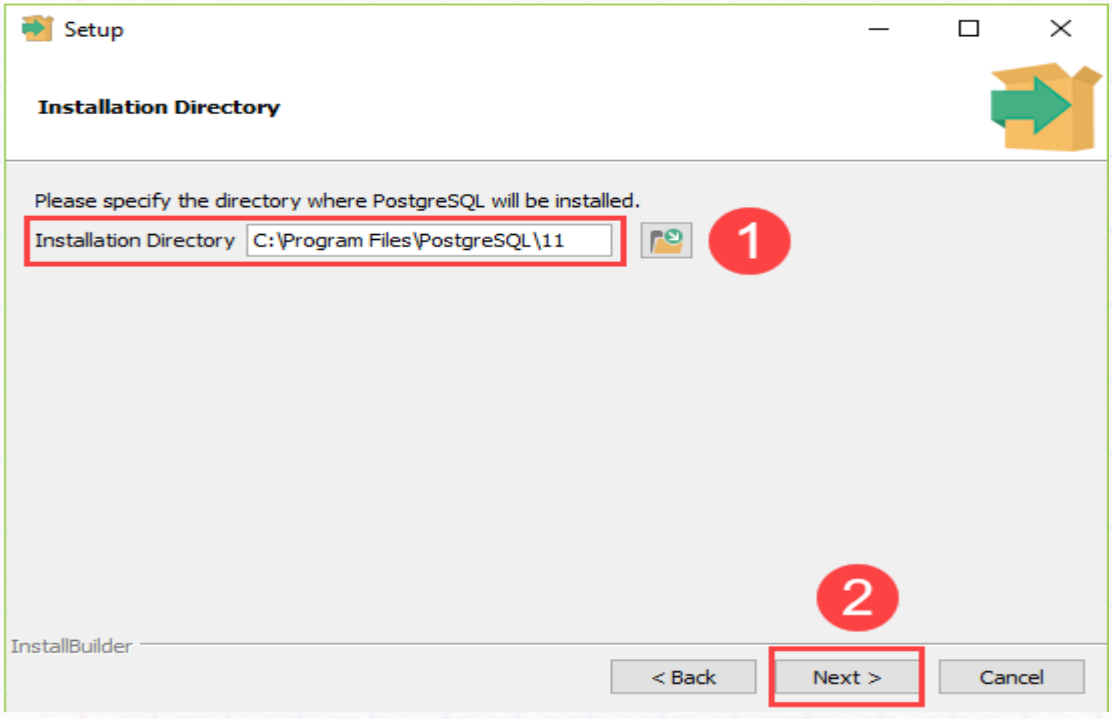
4. Installing the PostgreSQL installer

After downloading the installer double click on it and follow the below steps:

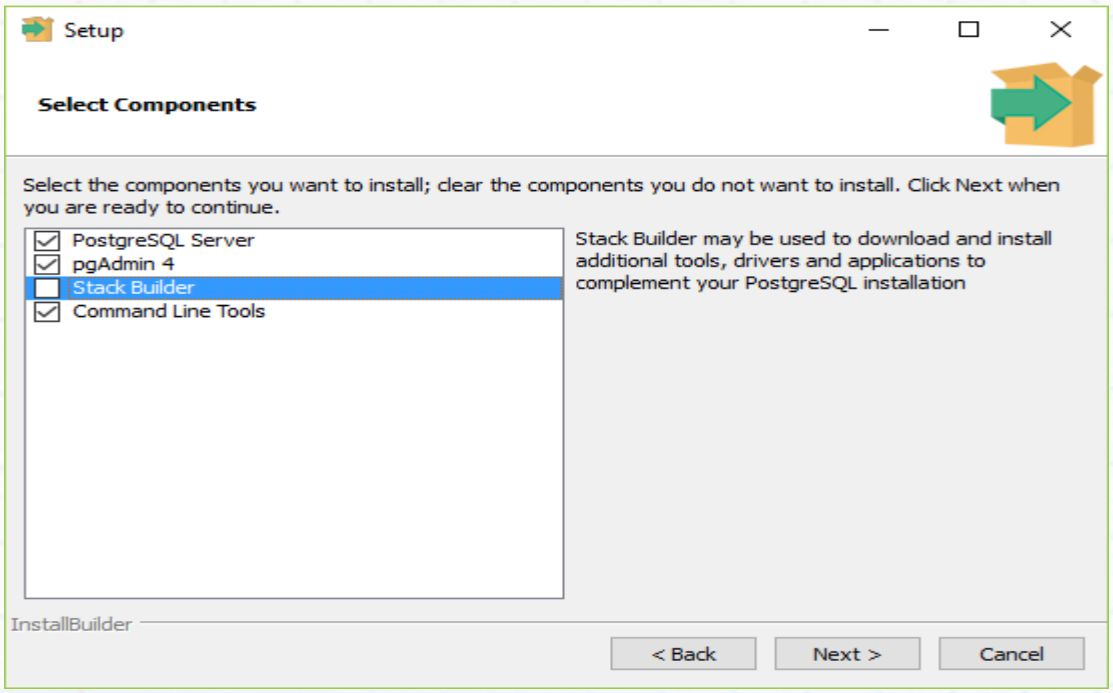


Step 1: Click the Next button.

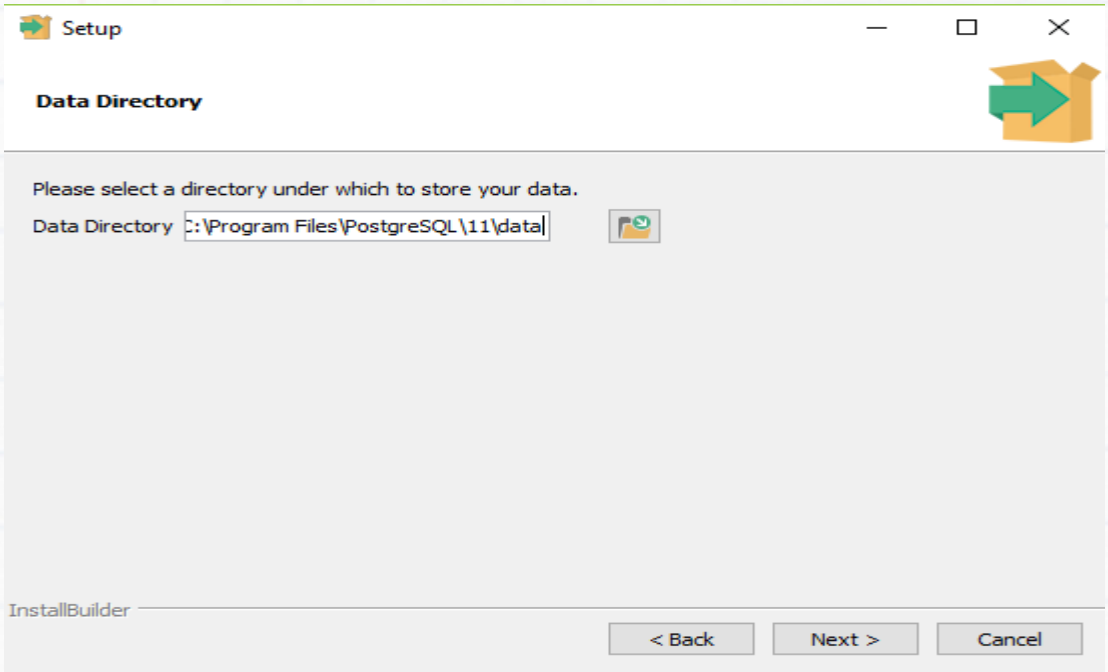
Step 2: Choose the installation folder, where you want PostgreSQL to be installed, and click on Next.



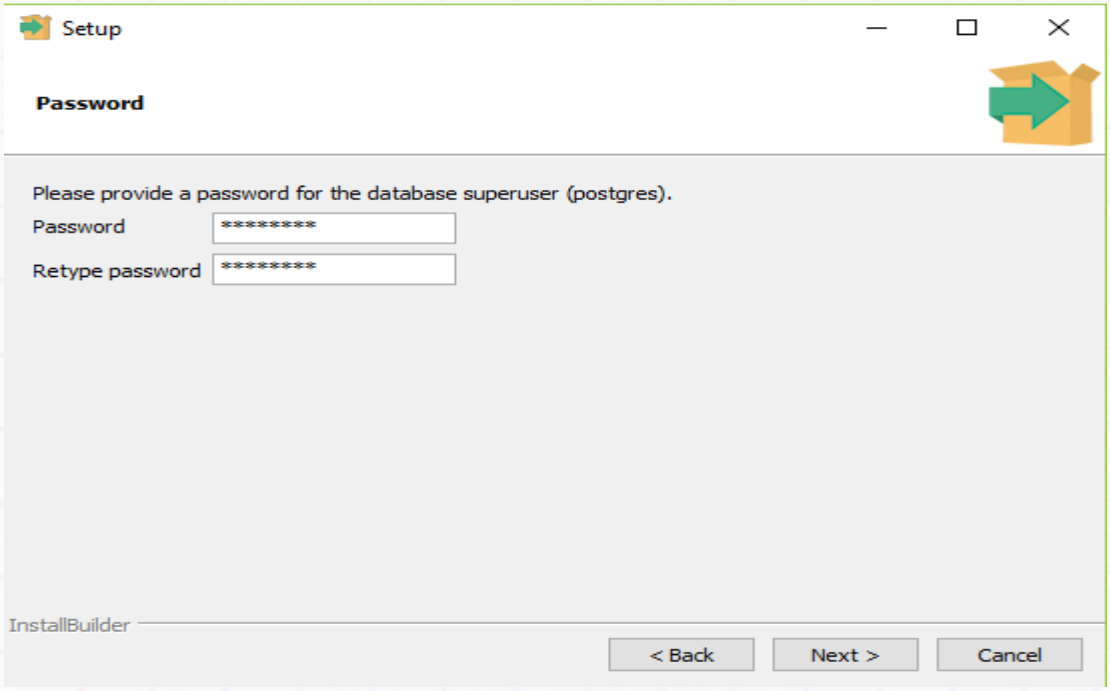
Step 3: Select the components as per your requirement to install and click the Next button.



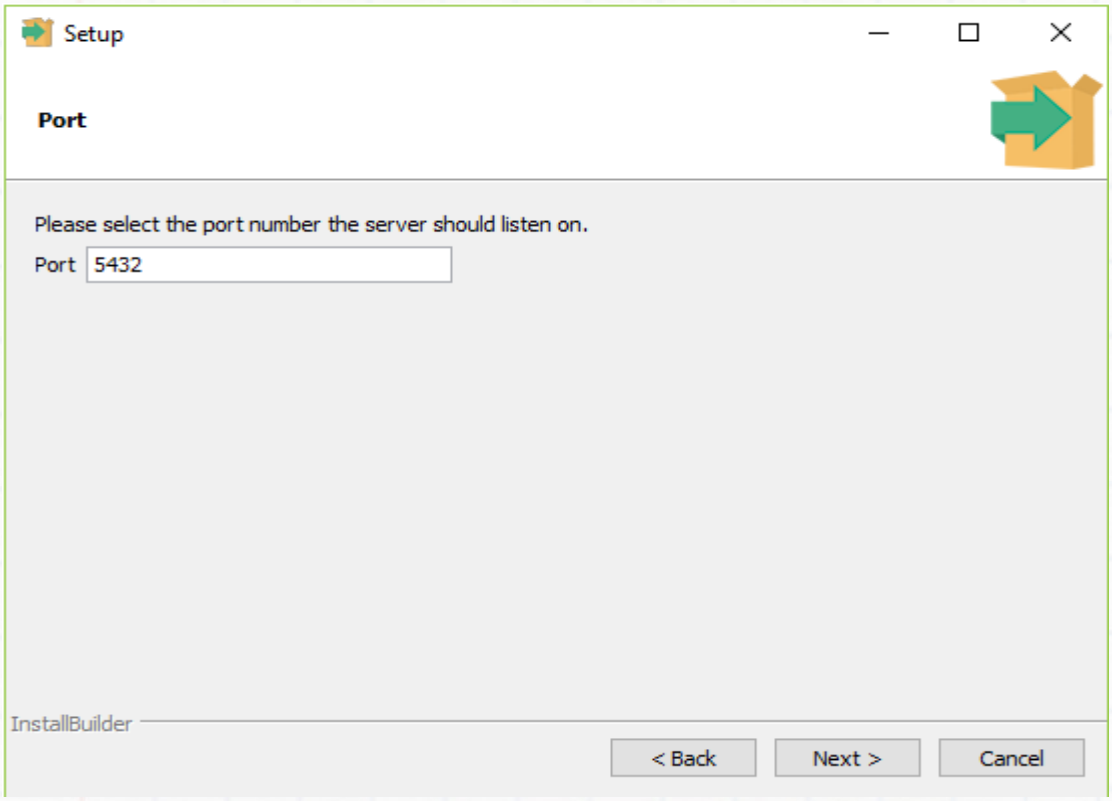
Step 4: Select the database directory where you want to store the data and click on Next.



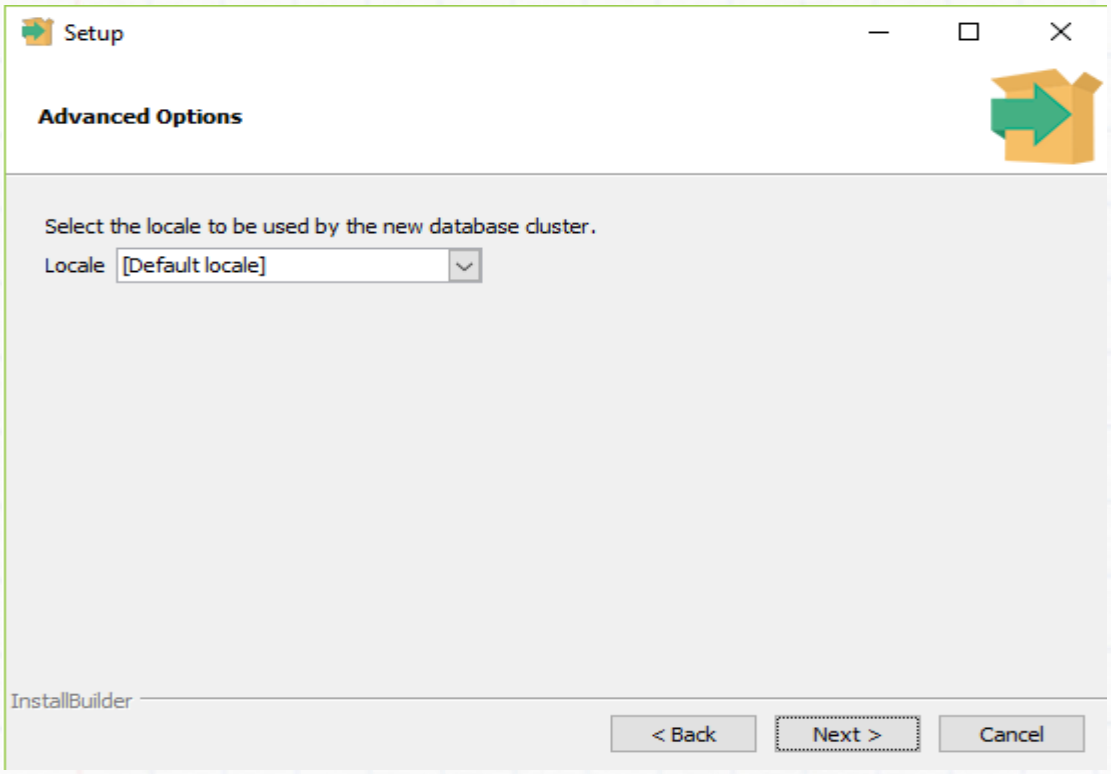
Step 5: Set the password for the database superuser (Postgres)



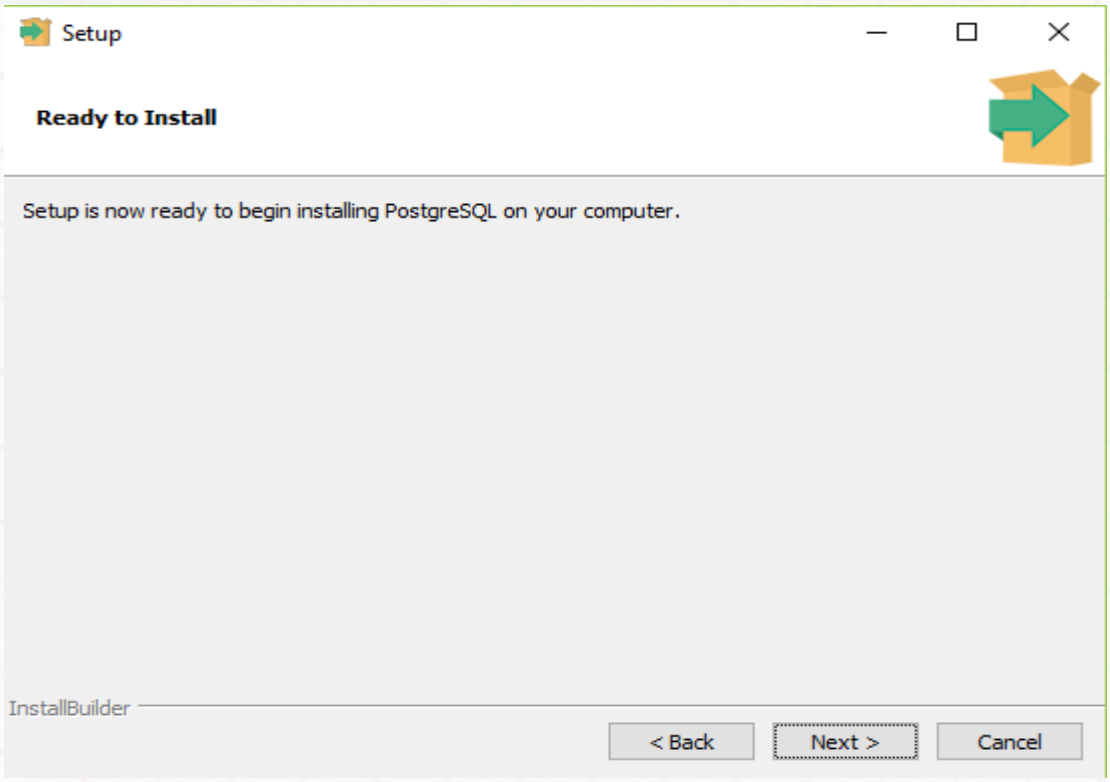
Step 6: Set the port for PostgreSQL. Make sure that no other applications are using this port. If unsure leave it to its default (5432) and click on Next.



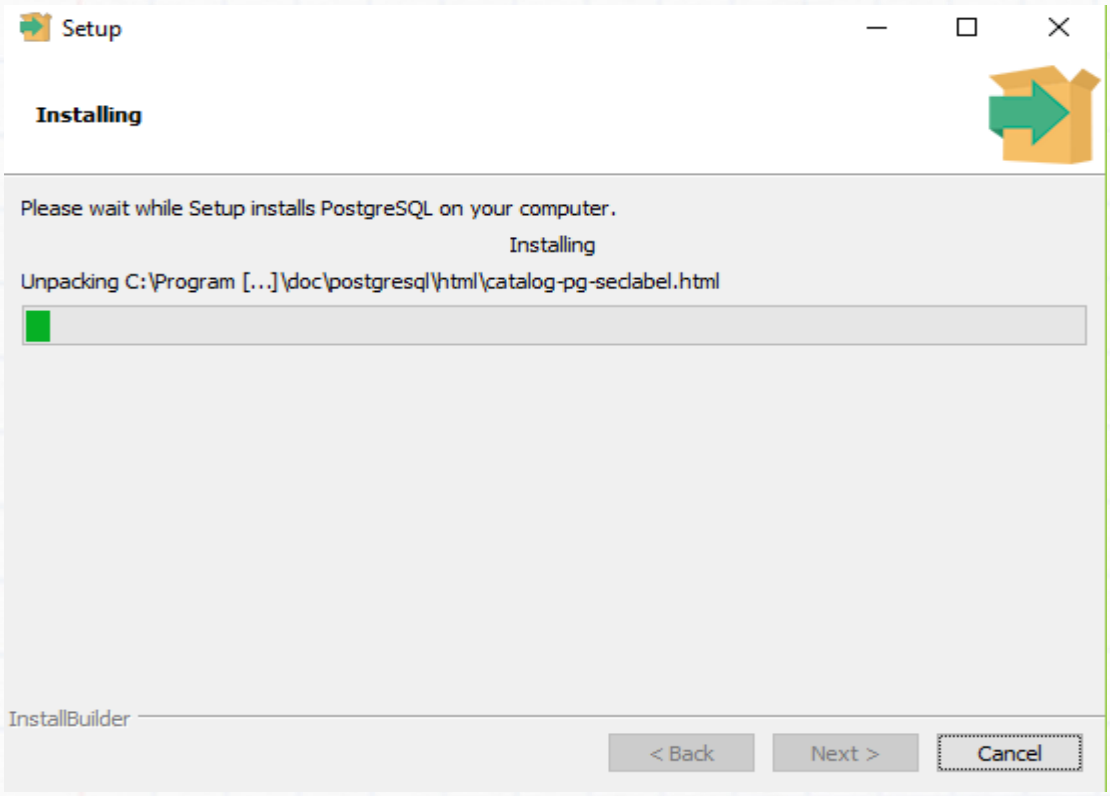
Step 7: Choose the default locale used by the database and click the Next button.



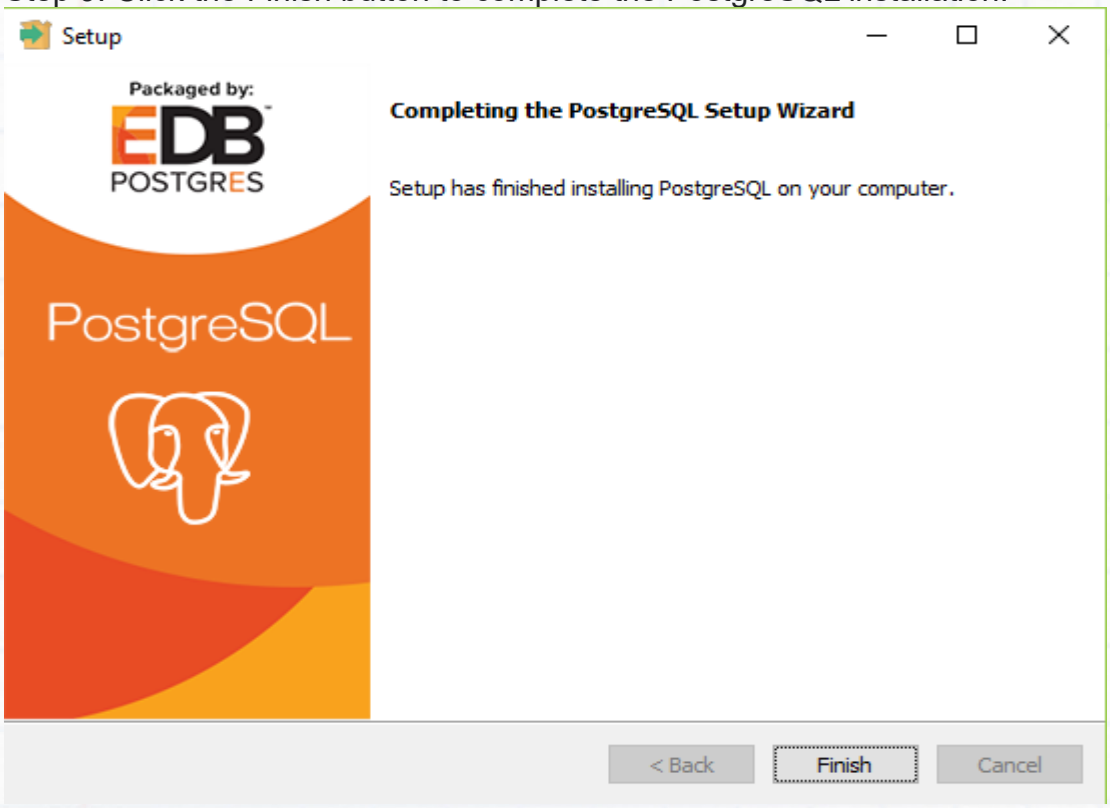
Step 8: Click the Next button to start the installation.



Wait for the installation to complete, it might take a few minutes.



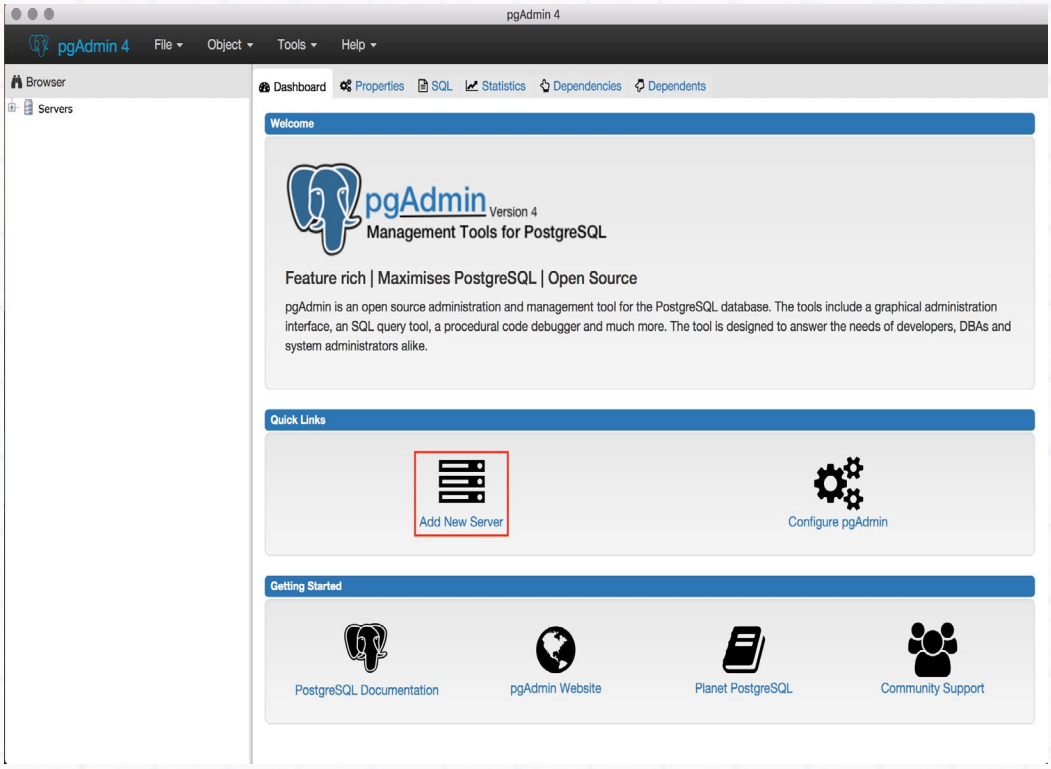
Step 9: Click the Finish button to complete the PostgreSQL installation.



4. When you install PostgreSQL, pgAdmin is installed. Start pgAdmin from your start menu.

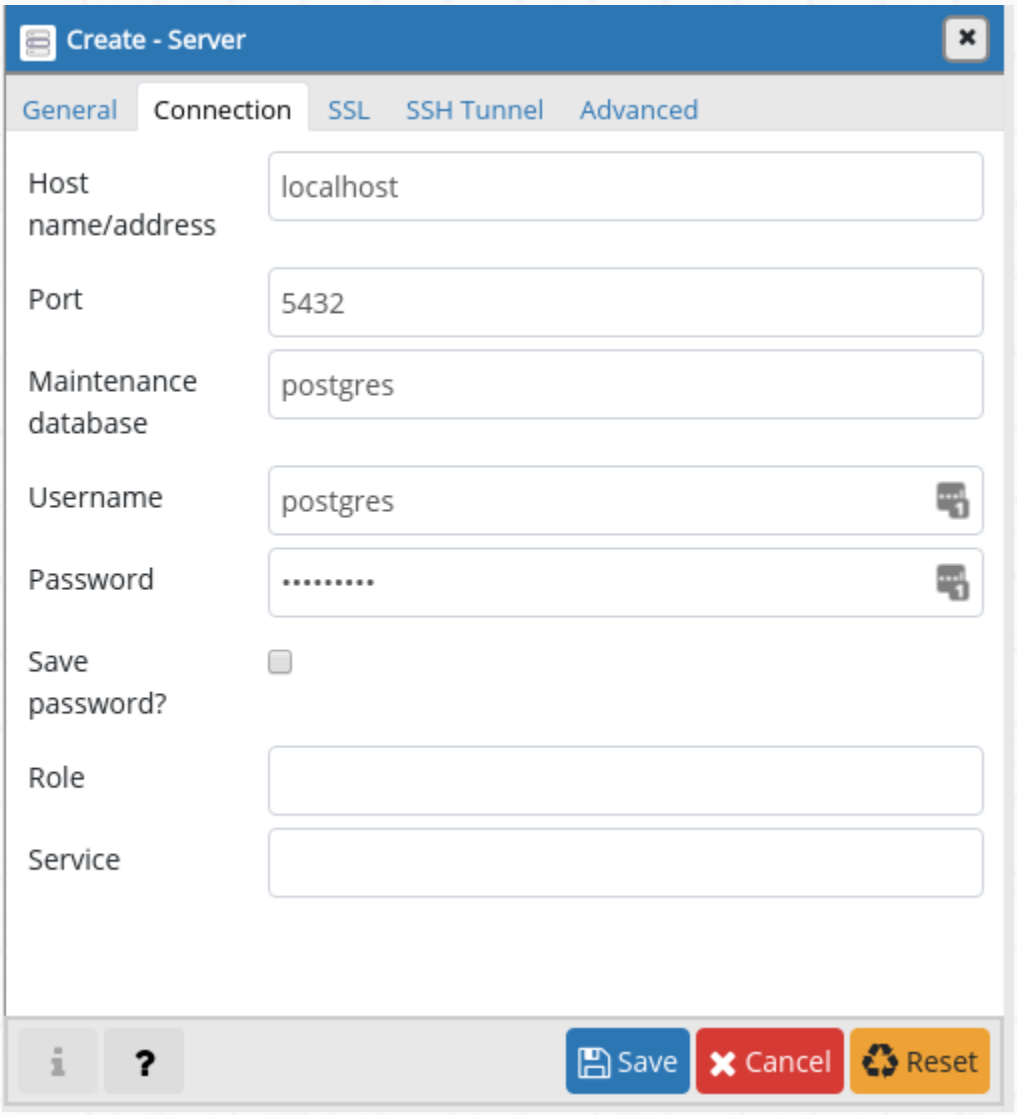
5. Create Server

Go to the “Dashboard” tab. In the “Quick Link” section, click “Add New Server” to add a new connection.



6. In the General tab, enter the name for this server.

7. Select the “Connection” tab in the “Create-Server” window.



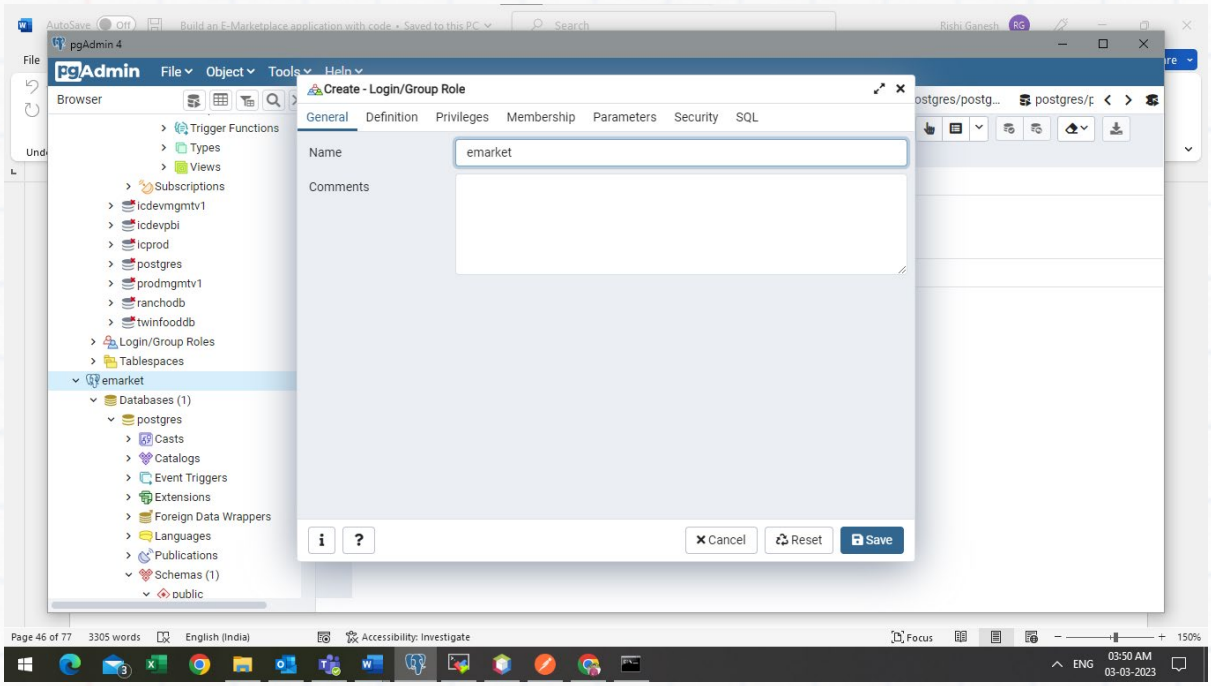
In the default PostgreSQL setup, the administrator user is postgres with an empty password. In the connection tab be sure to have the host set to localhost.
Click Save afterwards.

8. Create New User

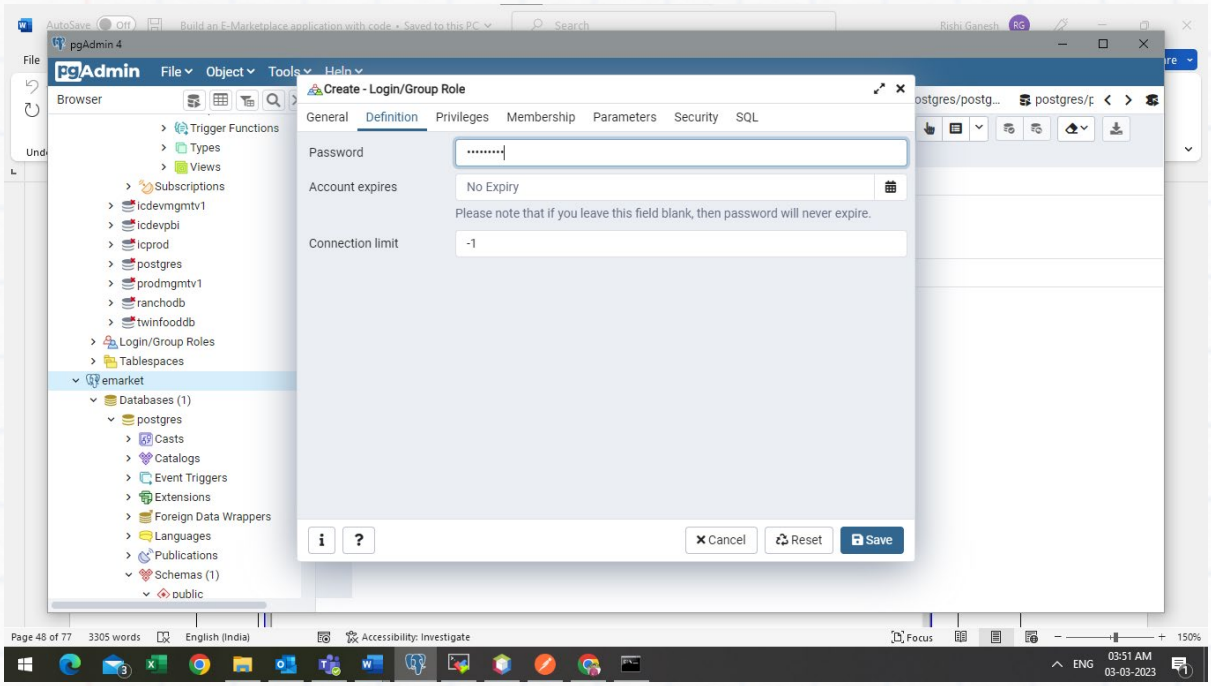
First, connect to the database by double-clicking on the instance name you created above.

Right click on Login/Group Roles, select Create and click on Login/Group Roles… for creating new user.

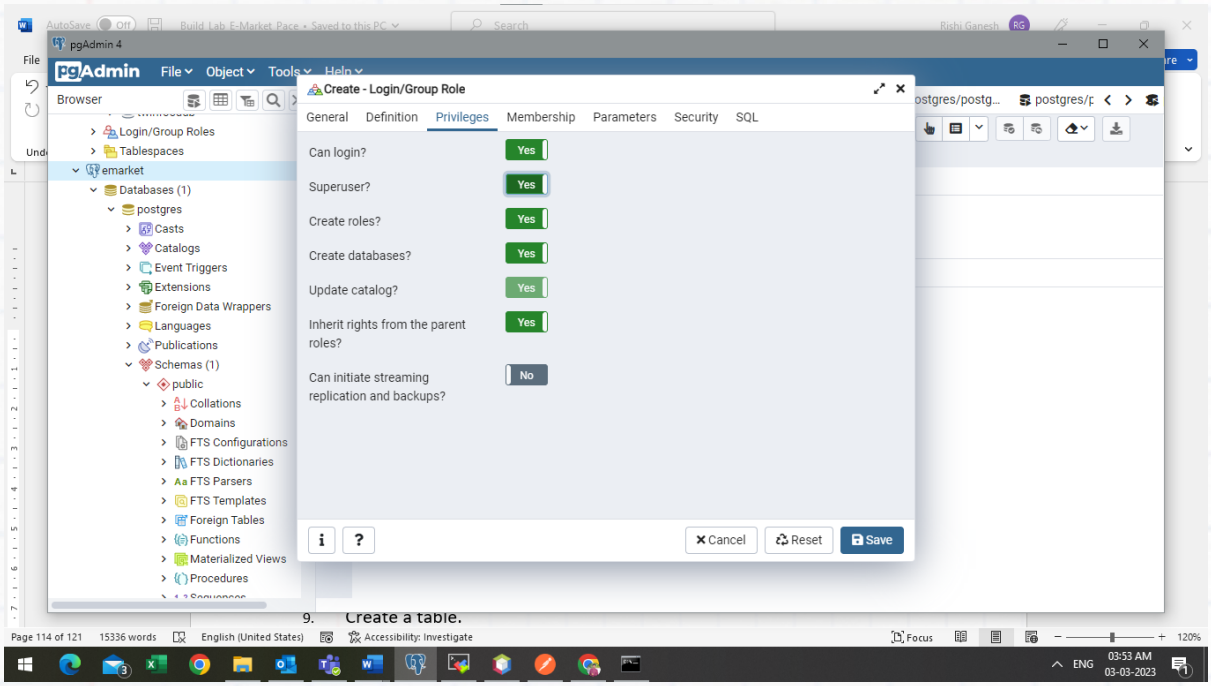
The following Create dialog box will appear. Type ‘emarket in Name field as user’s name and click on Definition tab.



Type login password for the user admin and click on Privileges tab. If you want to create user for limited time, then set the Account expires data and time value.

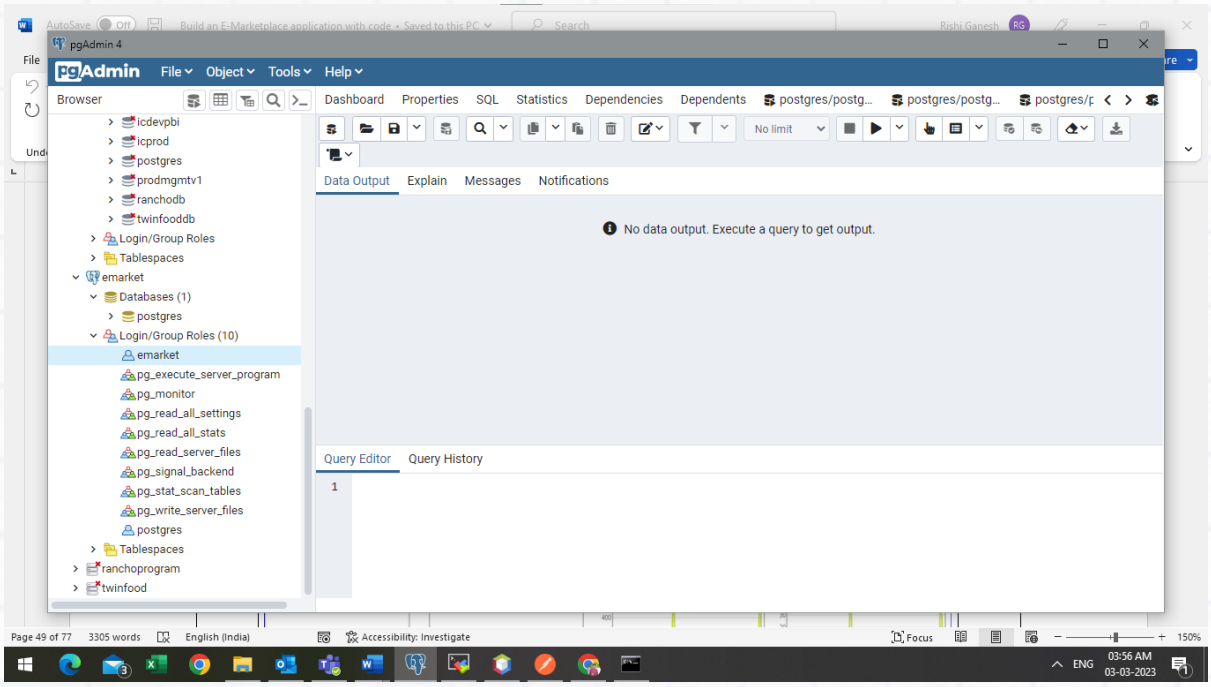


To set all permissions to emarket user make all options to ‘Yes’. Click ‘Save’ button to create the user.



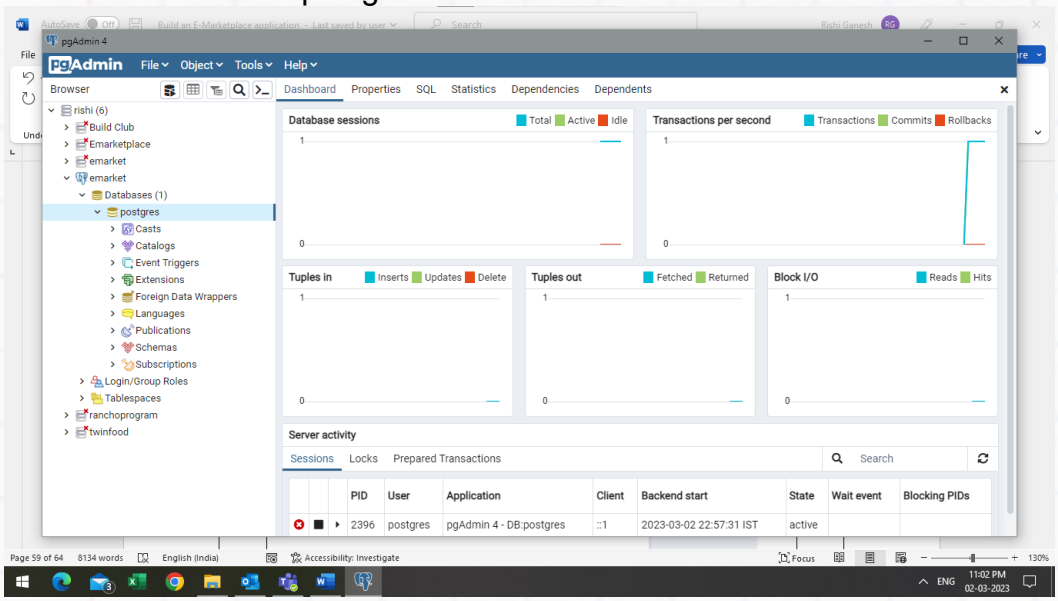
9. Create a table.

‘emarket user entry will be shown in Login/Group Roles section.



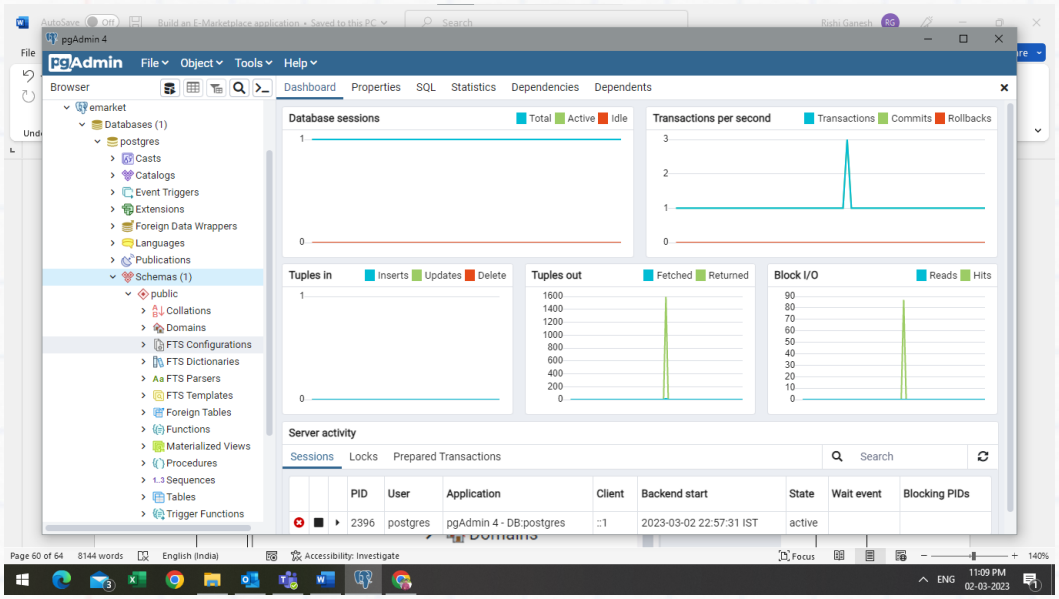
9. Create a table.

Left click on the Database section and select the required database, in this case the name of database is postgres



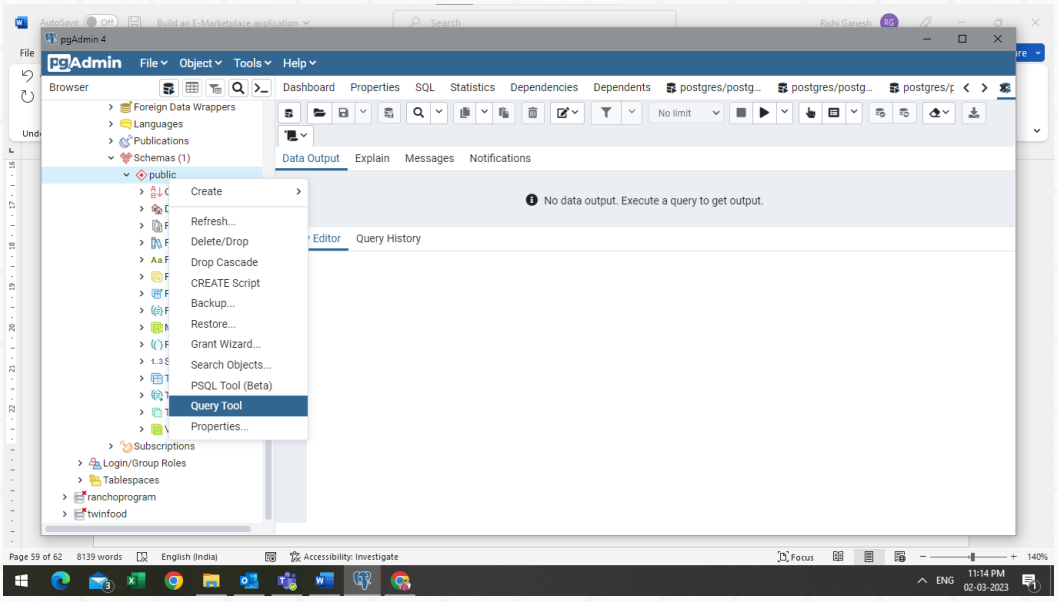
10. PostgreSQL- Database Selection

Now left click on the database and then select the Schemas section using the left mouse button. In this case we left click on postgres



11. PostgreSQL- Selecting Schemas

Now right click on the public section to select the Create option from the drop-down menu.



12. PostgreSQL- Create Table

Create a table named customer:

Type the following query in the Query editor panel.

```
CREATE TABLE IF NOT EXISTS public.customer

(
```

```
    cid serial,
    cname text COLLATE pg_catalog."default" NOT NULL,
    cemail text COLLATE pg_catalog."default" NOT NULL,
    cphone text COLLATE pg_catalog."default",
    cpassword text COLLATE pg_catalog."default" NOT NULL,
    caddressno text COLLATE pg_catalog."default",
    carea text COLLATE pg_catalog."default",
    ccity text COLLATE pg_catalog."default",
    cstate text COLLATE pg_catalog."default",
    cpincode text COLLATE pg_catalog."default",
    CONSTRAINT cid PRIMARY KEY (cid),
    CONSTRAINT uniqemail UNIQUE (cemail),
    CONSTRAINT uniqmobile UNIQUE (cphone)
)
```

In the admin table “cid”, “cname”, “cemail”, “cphone”, “cpassword”, “caddressno”, “carea”, “ccity”, “cstate”, and “cpincode” represents the name of the columns. INT and TEXT are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “cid” 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.

After entering query, select the Execute/Refresh icon from the toolbar.

13. Create a table named merchant:

Type the following query in the Query editor panel.

```
CREATE TABLE IF NOT EXISTS public.merchant
(
    mid serial,
    mname text COLLATE pg_catalog."default",
    mphone text COLLATE pg_catalog."default",
    memail text COLLATE pg_catalog."default" NOT NULL,
    maddress text COLLATE pg_catalog."default",
    mgstno text COLLATE pg_catalog."default",
    mpassword text COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT mid PRIMARY KEY (mid)
)
```

In the merchant table “mid”, “mname”, “mpassword”, “memail”, “maddress”, “mgstno”, and “mpassword” represents the name of the columns. INT and TEXT are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “mid” 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.

After entering query, select the Execute/Refresh icon from the toolbar.

14. Create a table named category:

Type the following query in the Query editor panel.

```
CREATE TABLE IF NOT EXISTS public.category
(
    catid serial,
    catcategoryname text COLLATE pg_catalog."default" NOT NULL,
```

```
    catcategoryimage text COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT catid PRIMARY KEY (catid)
```

In the category table “catid”, “catcategoryname” and “catcategoryimage” represents the name of the columns. INT and TEXT are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “mid” 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.

After entering query, select the Execute/Refresh icon from the toolbar.

15. Create a table named product:

Type the following query in the Query editor panel.

```
CREATE TABLE IF NOT EXISTS public.product
(
    proid serial,
    proname text COLLATE pg_catalog."default" NOT NULL,
    prodescription text COLLATE pg_catalog."default" NOT NULL,
    procategory text COLLATE pg_catalog."default" NOT NULL,
    proimage text COLLATE pg_catalog."default" NOT NULL,
    proprice text COLLATE pg_catalog."default" NOT NULL,
    proquantity text COLLATE pg_catalog."default" NOT NULL,
    proinitialquantity text COLLATE pg_catalog."default",
    CONSTRAINT proid PRIMARY KEY (proid)
)
```

In the product table “proid”, “proname”, “prodescription”, “procategory”, “proimage”, “proprice”, “proquantity”, and “proinitialquantity” represents the name of the columns. INT and TEXT are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “proid” 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.

After entering query, select the Execute/Refresh icon from the toolbar.

16. Create a table named ordersummary:

Type the following query in the Query editor panel.

```
CREATE TABLE IF NOT EXISTS public.ordersummary
(
    oscid bigint,
    osproductquantity text COLLATE pg_catalog."default",
    osproductname text COLLATE pg_catalog."default",
    osproductprice text COLLATE pg_catalog."default",
    ostotalprice text COLLATE pg_catalog."default",
    osordereddatetime text COLLATE pg_catalog."default",
    ospid bigint,
    osid serial,
    oscustomername text COLLATE pg_catalog."default",
    oscustomeremail text COLLATE pg_catalog."default",
    oscustomerphone text COLLATE pg_catalog."default",
    oscustomeraddressno text COLLATE pg_catalog."default",
    oscustomerarea text COLLATE pg_catalog."default",
```

```
    oscustomercity text COLLATE pg_catalog."default",
    oscustomerstate text COLLATE pg_catalog."default",
    oscustomerpincode text COLLATE pg_catalog."default",
    osproductimage text COLLATE pg_catalog."default",
    osproductdescription text COLLATE pg_catalog."default",
    osrefid bigint,
    CONSTRAINT osid PRIMARY KEY (osid)
)
```

In the admin table “osid”, “ospid”, “osproductname”, “osproductdescription”, “osproductprice”, “osproductimage”, “osproductquantity”, “ostotalprice”, “osordereddatetime”, “oscid”, “oscustomername”, “oscustomeremail”, “oscustomerphone”, “oscustomeraddressno”, “oscustomerarea”, “oscustomercity, “oscustomerstate”, “oscustomerpincode”, and “osrefid” represents the name of the columns. INT, BIGINT, and TEXT are data types and NOT NULL defines the column constraint, NOT NULL means no acceptance of NULL values in that column. Here, “osid”, “oscid” 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.

After entering query, select the Execute/Refresh icon from the toolbar.

17. Insert a record into the merchant table.

Type the following query in the Query editor panel.

```
INSERT INTO merchant (mname, mphone, memail, maddress, mgstno, mpassword)
VALUES ('indigrain', '1234567890', 'indigrain@gmail.com', 'chennai', '9876543210',
'indi@123');
```

The ‘merchant is an already created table. Now we are adding a new row of records under the respective columns with the corresponding values: 'indigrain', '1234567890', 'indigrain@gmail.com', 'chennai', '9876543210', 'indi@123'.

After entering query, select the Execute/Refresh icon from the toolbar.

Testing the backend with the mobile application.

Note: Make sure your computer and phone are on the same Wi-Fi network.

1. Change the api call URL from http://121.242.232.216:7070/emarket/ to http://<Wifi ipaddress>:8080/emarket/

In flutter project file ->

- Inside lib/Screen/api/api.dart
- Inside lib/Screen/category_management/api_service.dart
- Inside lib/Screen/customerhome/ApiServiceProjectDetail.dart
- Inside lib/Screen/product_management/api_service.dart
- Inside lib/Screen/ordersummary/ordersummary.dart

Example: (http:// 192.168.68.27:8080/emarket/)

2. Start the resin server.

   Execute `resin.exe`

   or run-in command prompt

```
resin/bin ./start.bat;
tail -f ../log/jvm-app-0.log;
```

3. Perform functional tests and validate if all the functionalities work according to requirements.

    1. Merchant Login
        1. Add, View, Remove Categories
        2. Add, View, Edit, Remove Products
        3. Check Incoming/New Orders
    2. Customer Registration

3. Customer Login
    1. View Categories
    2. View Products
    3. Add to Cart
        a. Add/ Remove Items
        b. Increase/ Decrease Quantity
        c. Quantity Check
    4. Update Profile, Address
    5. Make payment using different payment modes.
    6. Check history of orders