1. Installation

* Install Oracle 11g Edition Express

Follow this [link](https://www.oicbasics.com/2020/01/download-oracle-database-11g-xe-express.html) to install Oracle 11g Express Edition step by step.

* Install MariaDB 11.0

Use this link (<https://mariadb.com/downloads/>) to download the MariaDB 11.0. Don’t forget to add the bin folder to your PATH.

* Install HeidiSQL

Use this link ([Download HeidiSQL](https://www.heidisql.com/download.php)) to download the HeidiSQL.

* Install Python3.10.4

Use this link to download this language (<https://www.python.org/downloads/release/python-3104/>). Add both scripts folder and the python folder into the PATH.

* Install php 7.4.28

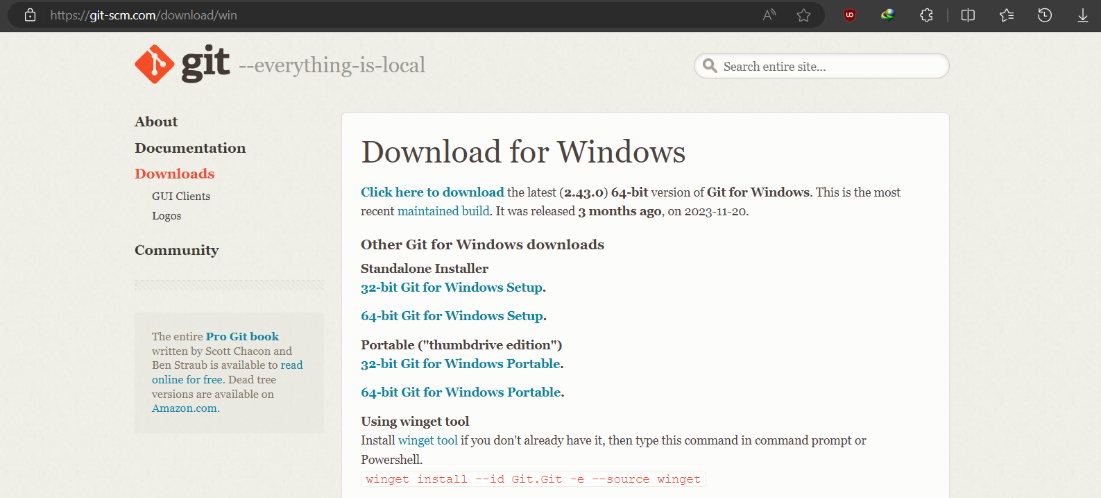
Use this path to download the language ([PHP 7.4.28 (64-bit) Download (filehorse.com)](https://www.filehorse.com/download-php-64/69544/)). Add this folder into PATH.

* Install composer

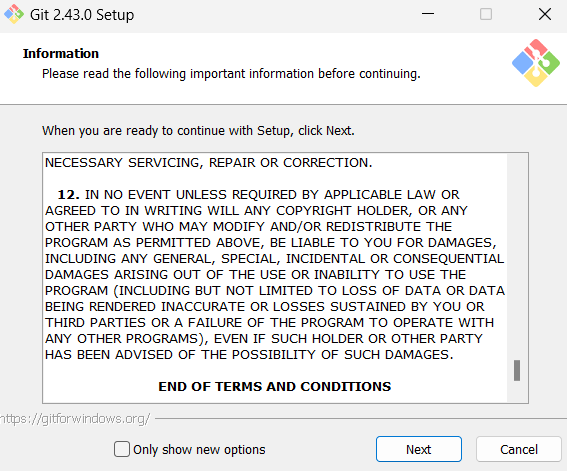
Use this link to download the composer ([Composer (getcomposer.org)](https://getcomposer.org/download/)). Add this folder also into the PATH.

* Installing Git

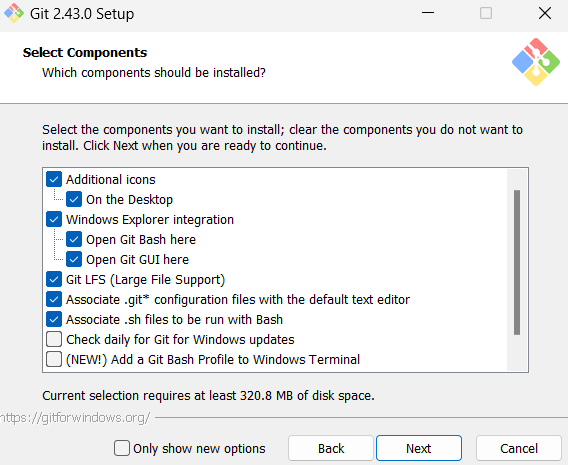
In this project, we will clone the project from the repository using Git (<https://git-scm.com/>) standalone installer.



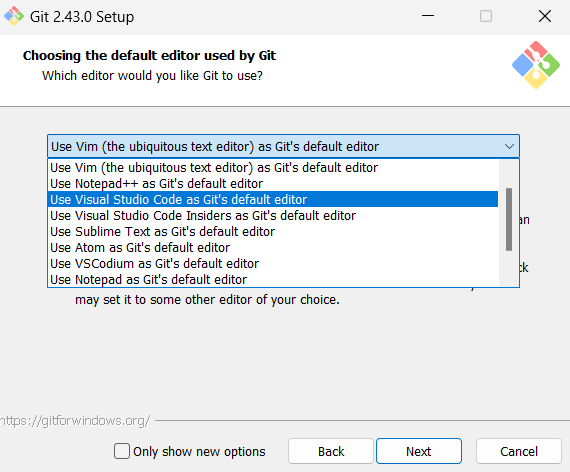
1. Firstly, install the git software. If you want to clone it using another software, for example, GitHub Desktop, then you can skip all the steps and clone it using the software you choose. Open the installer and untick ‘Only show new options’ and click ‘Next’.



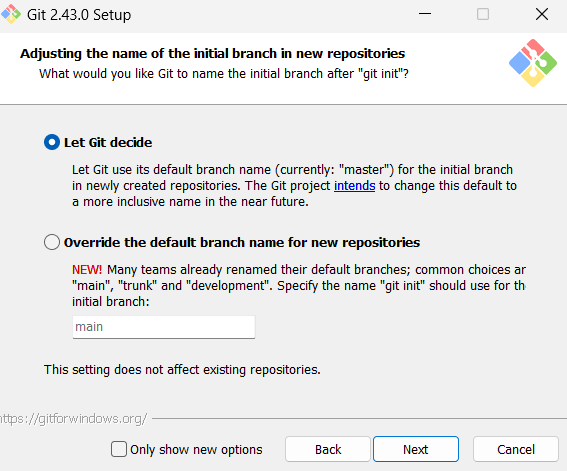
1. Make sure ‘Open Git Bash here’ and ‘Open Git GUI here’ are selected. Click 'Next’.



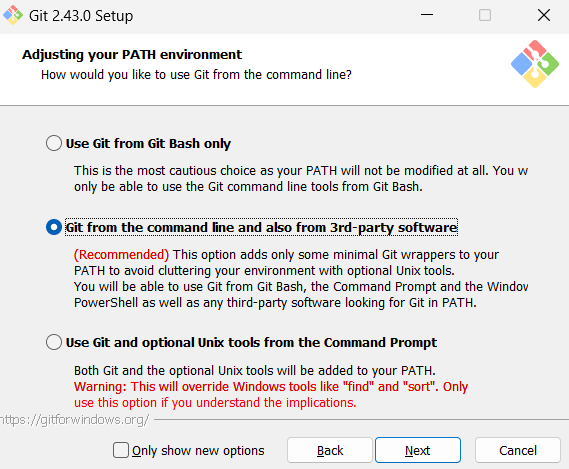
1. Choose your desired default editor used by Git. In this case it is VS Code. Click 'Next’.



1. Select ‘Let Git Decide’ and click 'Next’.



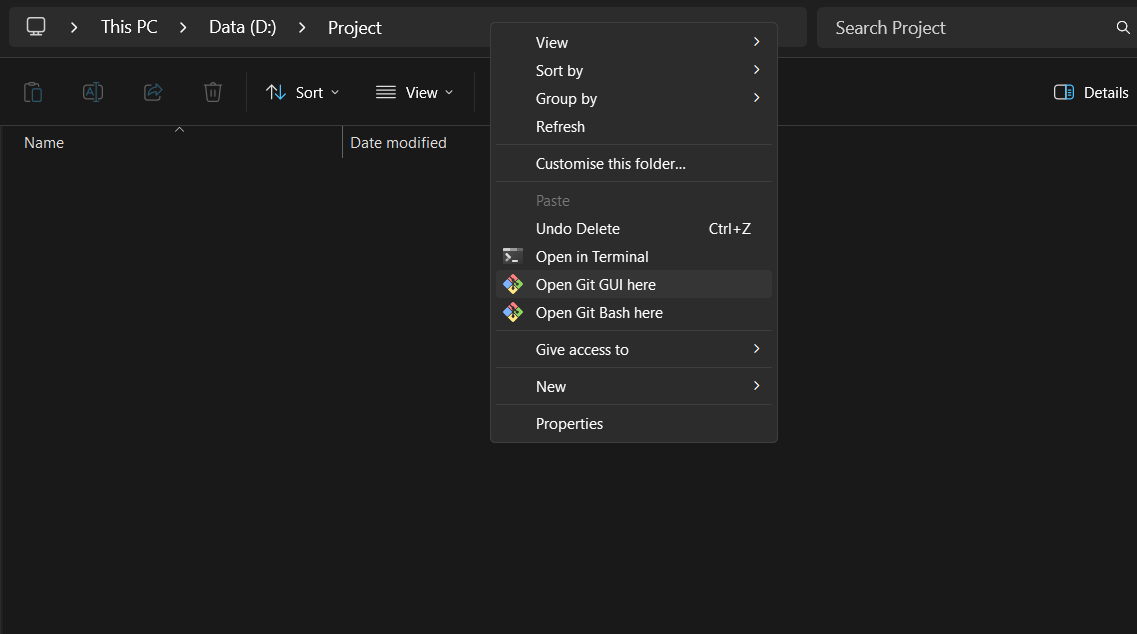
1. Select the recommended option and click 'Next'.



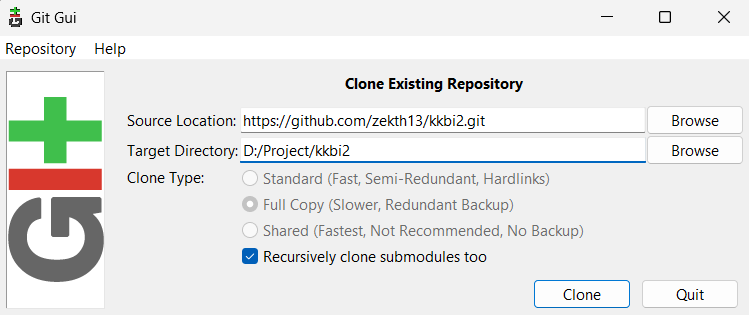
1. Select ‘Use bundeled OpenSSH’ and click ‘Next’.
2. Select ‘Use the OpenSSL library’ and click ‘Next’.
3. Select ‘Checkout Windows-style' and click ‘Next’.
4. Select ‘Use Windows’ default console window’ and click ‘Next’.
5. Select ‘Fast-foward or merge’ and click ‘Next’.
6. Select ‘Git Credential Manager’ and click ‘Next’.
7. Select ‘Enable file system caching’ and click ‘Next’.
8. Unselect both ‘Enable experimental support’ and ‘Enable experimental bult-in' and click ‘Install’.
9. Done.

* Cloning Repository

1. Prepare a folder path for the project folder to be located. In this case, D:\Project.
2. Right click on the empty space in the file explorer and click ‘Open Git GUI here’.

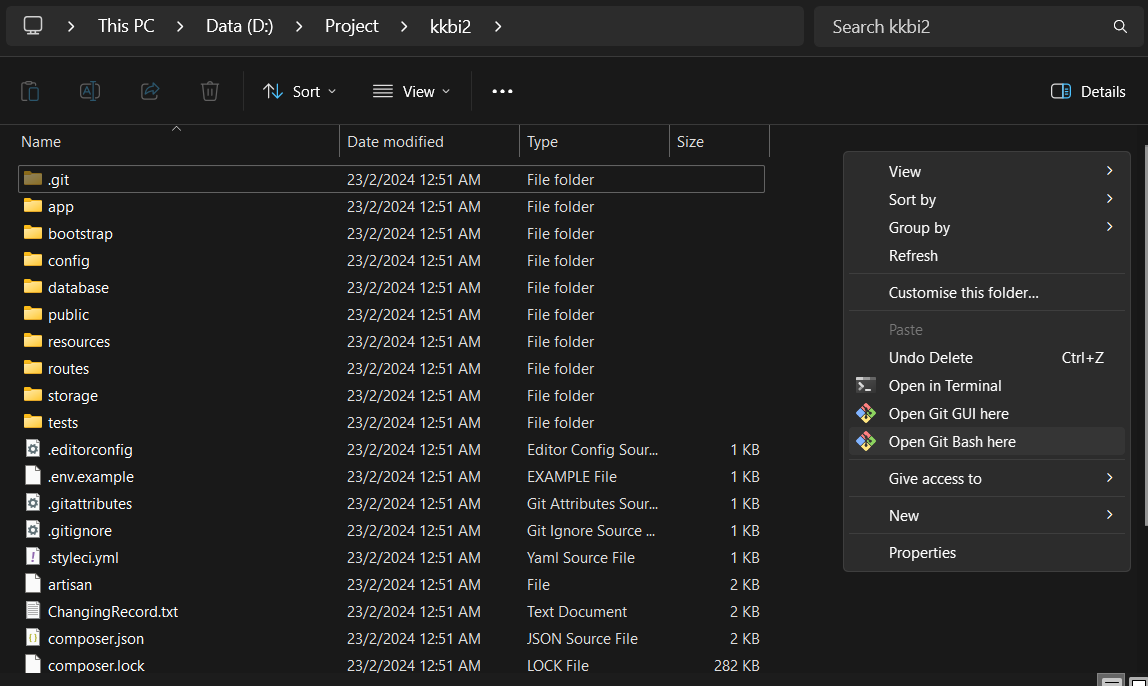


1. Select ‘Clone Existing Repository’.
2. Fill the source location. In this case is <https://github.com/zekth13/kkbi2.git>. Fill the Target Directory. In this case it is ‘D:/Project’ with a new folder, hence fill in ‘D:/Project/kkbi2’. Tick ‘Recursively clone submodule too’ and click ‘Clone’.



1. Done. Now you can close the Git window.

### Installation and Configuration



2. Configuration

* Configure the KKBI folder
  1. Run composer update
  2. Rename .env.example -> .env
  3. Run php artisan key:generate
* Configure .env files
  1. Configure the MARIA\_HOST, MARIA\_PORT, MARIA\_DATABASE, MARIA\_USERNAME, MARIA\_PASSWORD based on the HeidiSQL configuration.

### 3. Folder usage

#### 3.1 KKBI2

There are a lot of files but emphasize important things

1. app -> Http -> Controllers # All the function in every page in here

Controllers

-> Controller

-> forecastcontroller.php #All the forecast function

-> HomeController #All functions in Home page

-> inventorycontroller #All functions in inventory page

-> reportcontroller #All functions in report page

-> Salescontroller #All functions in sales page

-> storeforecast #All functions in store forecast page

-> suppliercontroller #All functions in supplier page

1. resources -> views

-> auth folders #User interface code for the login page

-> inventory folders #User interface code for inventory page

-> layouts folders #Layout for every page

-> reports folders #User interface code for report page

-> sales folders #User interface code for the sales summary page, outlets sales page, products sales page

-> suppliers folder #User interface code for suppliers sales page

-> forecast.blade.php #User interface code for forecast page

-> [home.blade.php](http://home.blade.php) #User interface code for dashboard page

-> loading.blade.php #User interface code for loading page in forecasting page

-> storeforecast.blade.php #User interface code for storeforecasting page

-> welcome.blade.php #User interface code for welcoming page

1. routes

->api.php #Contain all the API routes

->web.php #Contain routes to web pages

1. .env files #Contain all the DB connections

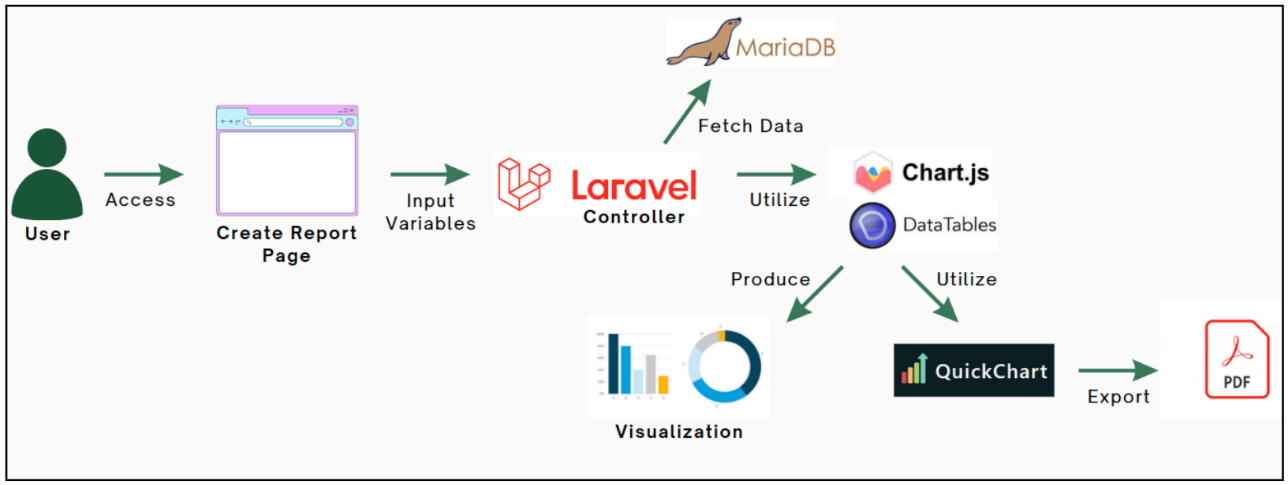
#### 3.2 Flask API

1. annual\_prediction.py #Produce the annual sales forecasting using XGBoost model and Polynomial Regression model
2. connector.py #Connecting to MariaDB, get data from it
3. main.py #contain routes for web page, initialize application
4. monthly\_prediction.py #Produce the monthly sales forecasting using 2 models.
5. poly\_model.pkl #A pickle files contain polynomial regression model for forecasting
6. poly\_storemodel.pkl #A pickle files contain polynomial regression model for sales forecasting for each stores
7. str\_no\_prediction.py #Produce the monthly sales forecasting for every stores
8. XGBoost\_model.pkl #A pickle files contain XGBoost model for forecasting
9. XGBoost\_storemodel.pkl #A pickle files contain XGBoost model for forecasting each stores.

## 4. Handling the code for data visualization and reporting modules

### 4.1 Overall Process

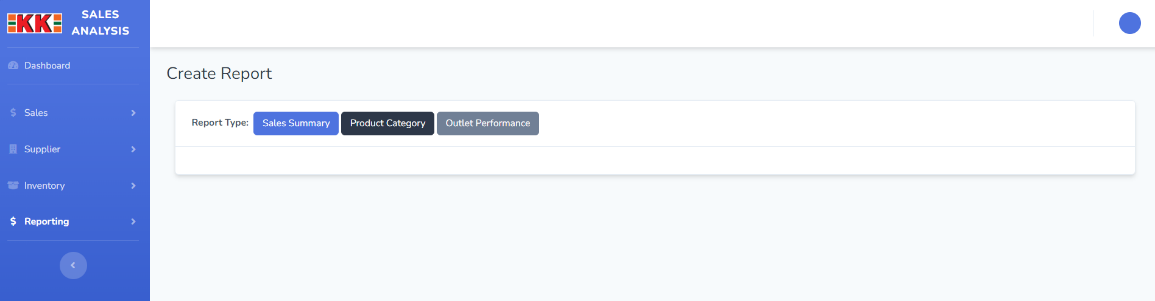
To understand how the dashboard is coded, let's see the overall process flow.



First, the user logs into the dashboard and accesses a page such as Create Report Page. On this page, the user must input the required variables according to the data that needs to be visualized. For example, for the Product Category Report, the user needs to select multiple product categories to be compared, as well as the time interval (monthly, quarterly, or annually) and year. These input variables are sent to the Laravel View, where the front-end of the dashboard is located. Then, these variables and the necessary information about the data to be visualized are sent to the Laravel Controller, where the back-end processes take place. From the input variables, an SQL query is run via Eloquent, an object relational mapper (ORM), to fetch the data from MariaDB.

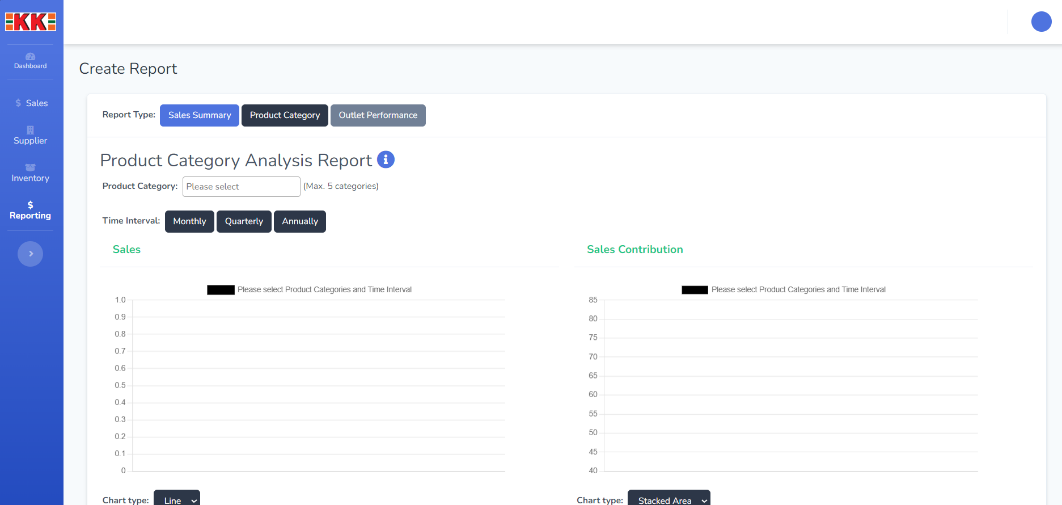
The data then undergoes the necessary calculations, sorting, arrangement and hierarchy to be sent and used later in the front-end JavaScript view. In the view, Chart.js and DataTables are used to visualize the data. In the controller, QuickChart API is used to convert the Chart.js charts into images to be included in the PDF report. The empty PDF templates for all report types are ready in the view, so that in the controller, the final processed data is used to fill in the PDF template with the correct data that the user queries. The PDF is then exported and can be downloaded by the user.

### 4.2 Front-end

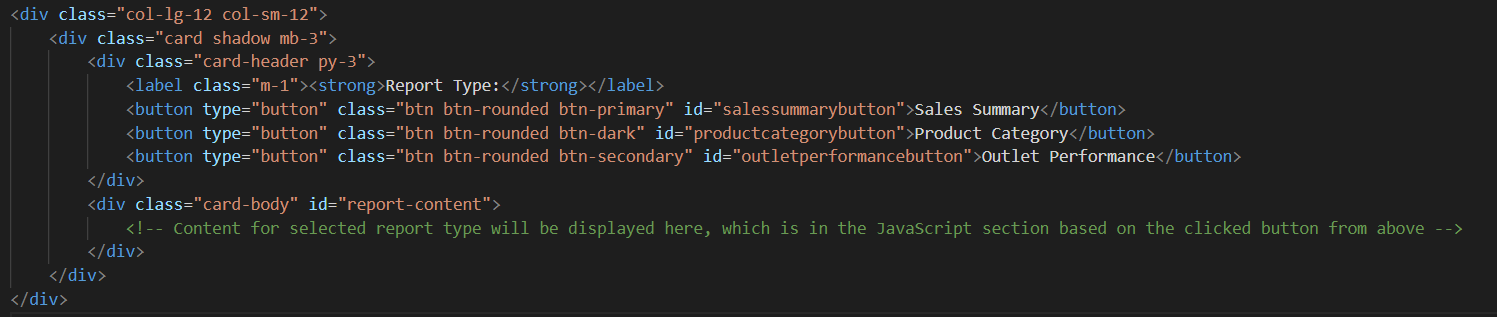


The front-end of the page can be seen in app/resources/views/reports/index.blade.php, where all the report types of category page are written, which are Sales Summary, Product Category and Outlet Performance. For this Example, we’ll use **Product Category** as an example.

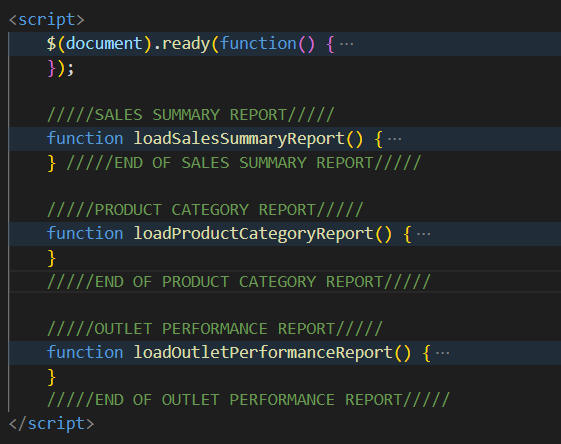
When user clicked one of the report types, the page will load it.



In the **index.blade.php**, this is the code for the report types buttons,

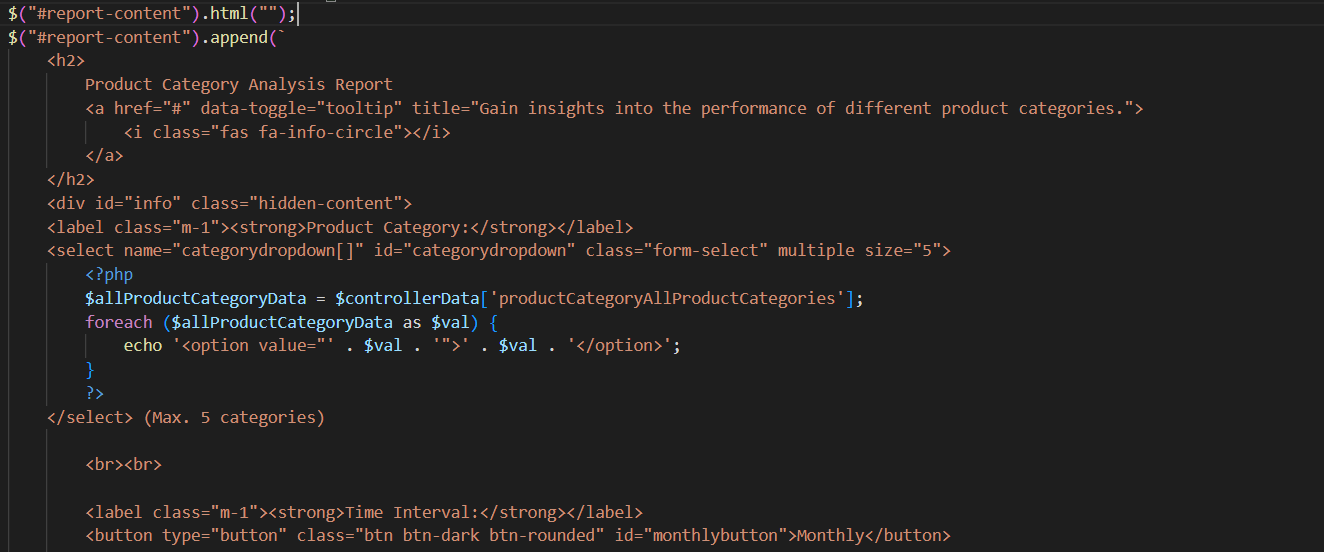


In which it will call one of the below functions (the functions are collapsed for better view) to load the page,

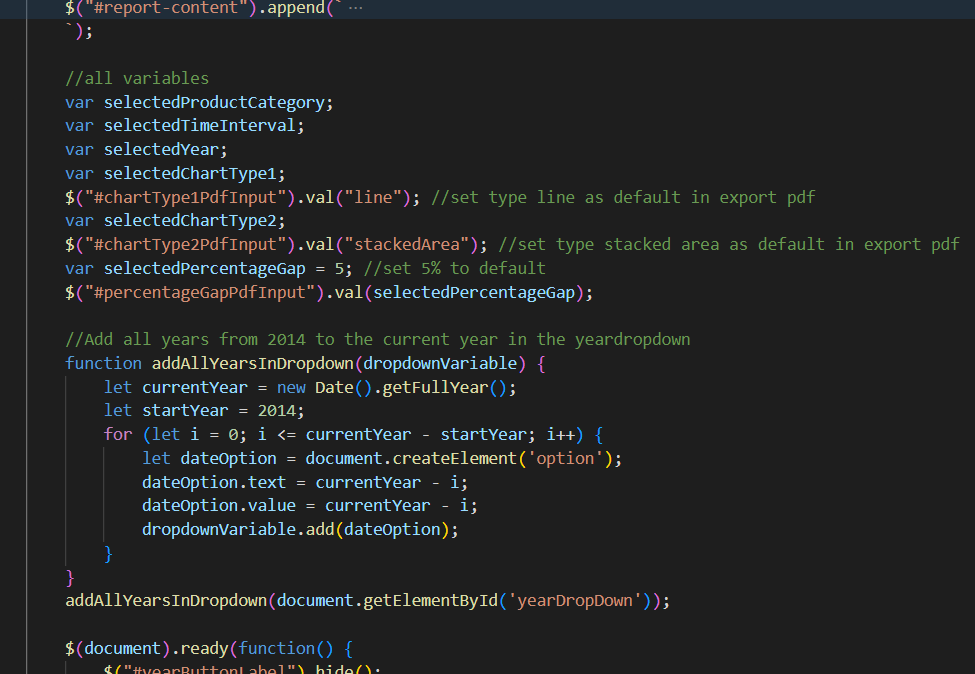


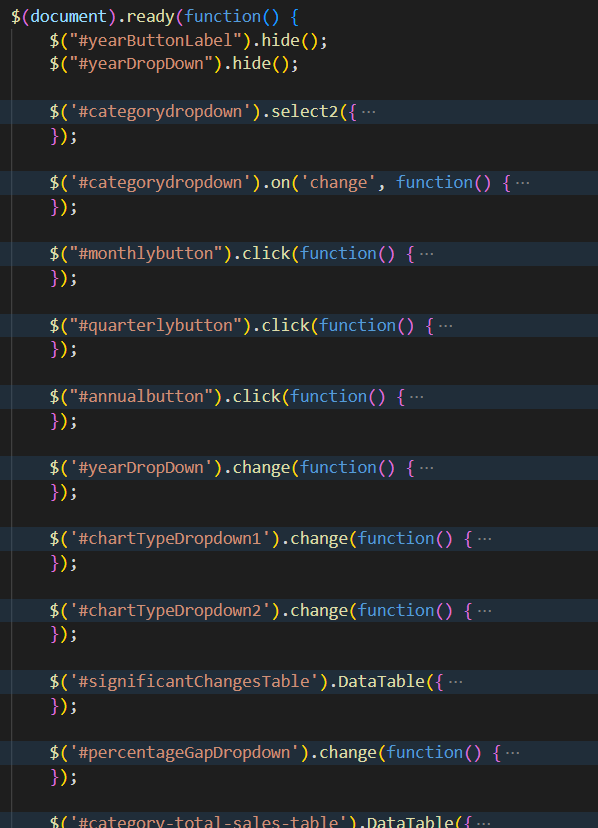
For this example, it will load the function **loadProductCategoryReport().**

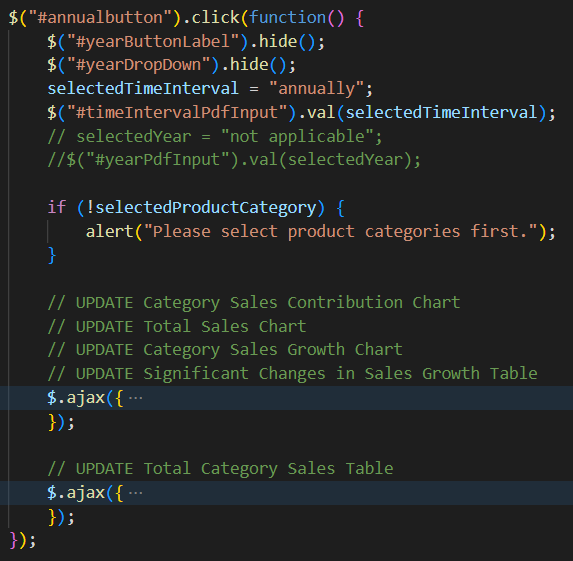
In the function, in JavaScript, we can see that it appends the html for the report content,



and then JavaScript for the elements inside the page, which are dropdowns, buttons, table and charts.



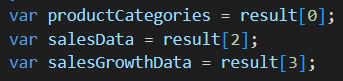




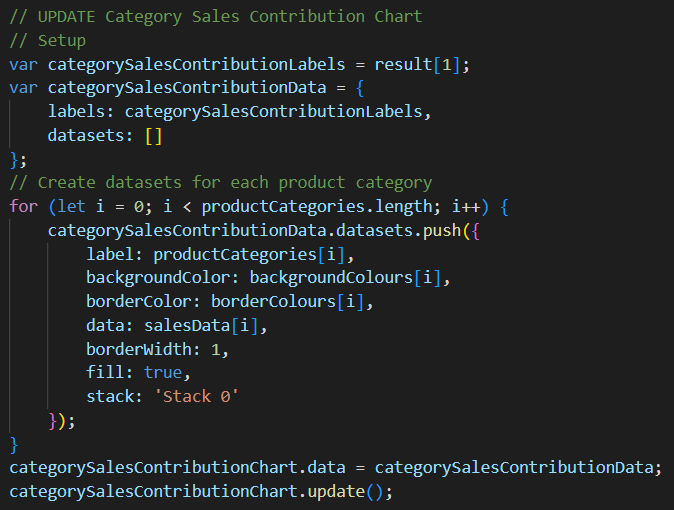
This is where the dashboard is dynamically updated based on the user input. For example, when the **user clicks the annual button**, the $("#annualbutton").click(function() send ajax request to get the related data and do the following tasks:

1. Hide the Year button
2. Hide the year dropdown
3. UPDATE Category Sales Contribution Chart
4. UPDATE Total Sales Chart
5. UPDATE Category Sales Growth Chart
6. UPDATE Significant Changes in Sales Growth Table
7. UPDATE Total Category Sales Table

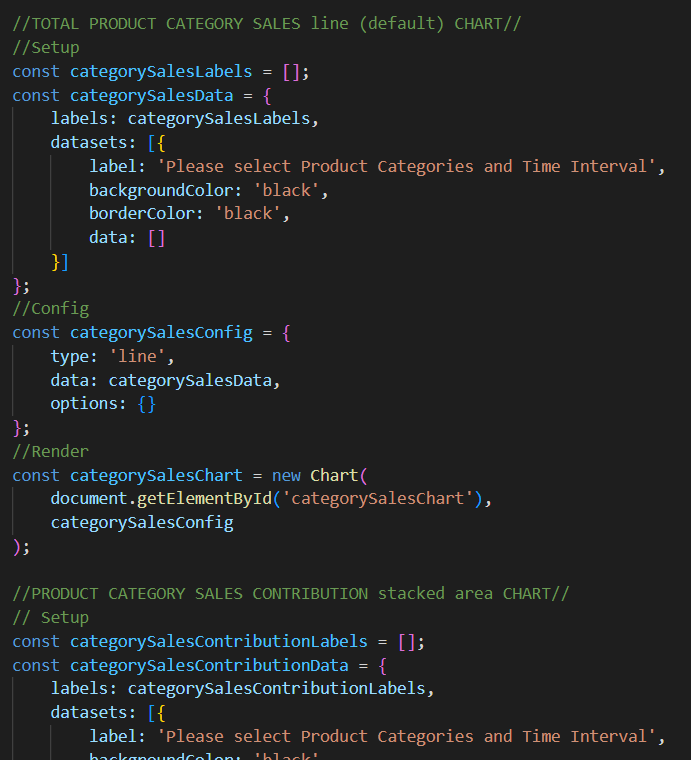
This is the way data is gotten to update them is from the data pass by the Laravel controller and we will see it later in the back-end section.



The variables store the data from the result array which is passed from the Laravel controller in the back-end.



There is also JavaScript to initiate and display the empty charts.



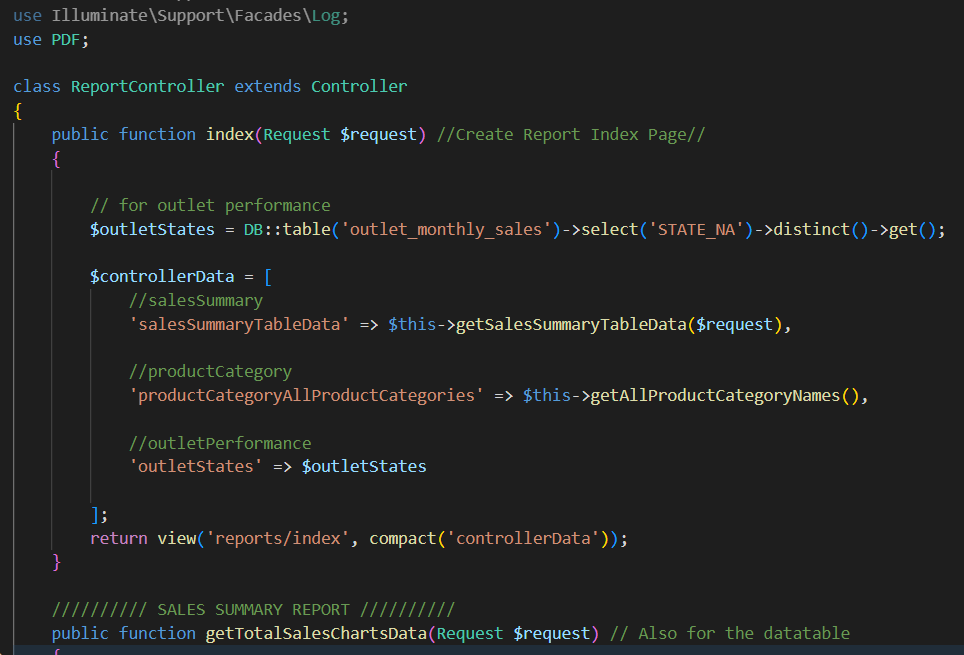
All other functions are called when for their own actions, for example:

* $('#categorydropdown').on('change', function() is called when users make changes to the category dropdown.
* $("#monthlybutton").click(function() is called when users click the monthly button.
* $('#chartTypeDropdown1').change(function() is called when users select another chart type from the change chart type dropdown.
* and so on.

### 4.3 Back-End

#### 4.3.1 Visualization

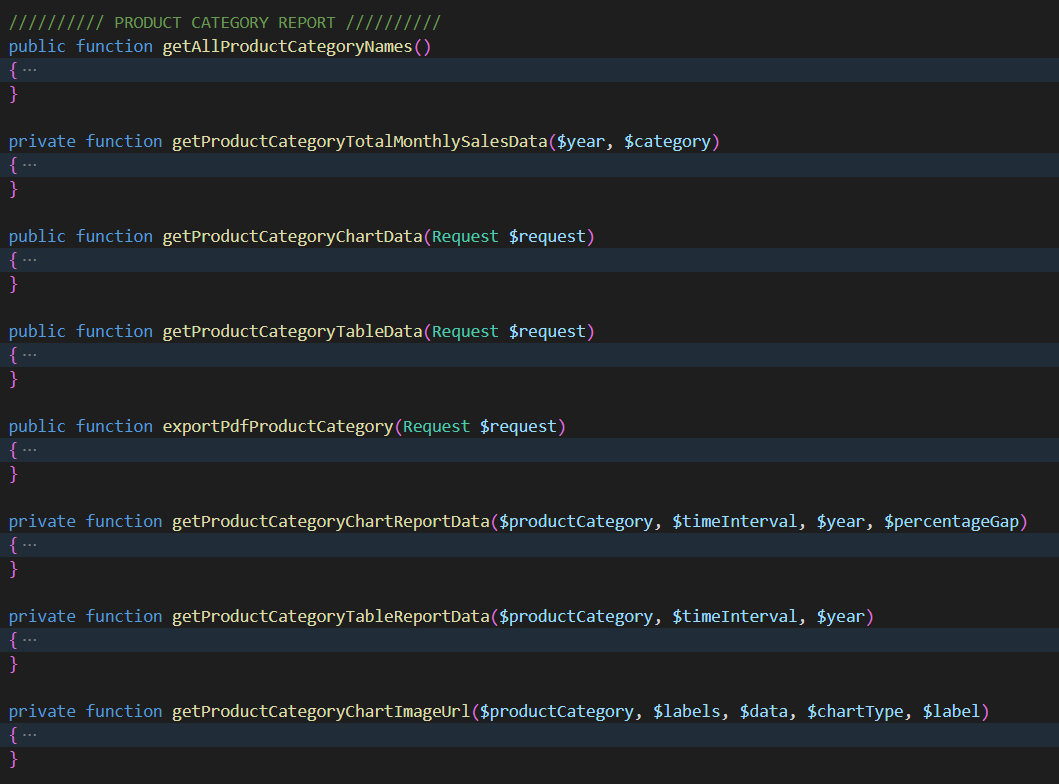
All of the data is fetched from the database which is written in the laravel’s controller. In this case, the controller file name is ReportController.php (app/Http/Controllers/ReportController.php). A controller method in a Laravel application is responsible for handling incoming HTTP requests and returning appropriate responses. At the top of the ReportController code is the function to handle the initial load of the index page for the Create Report Index Page.



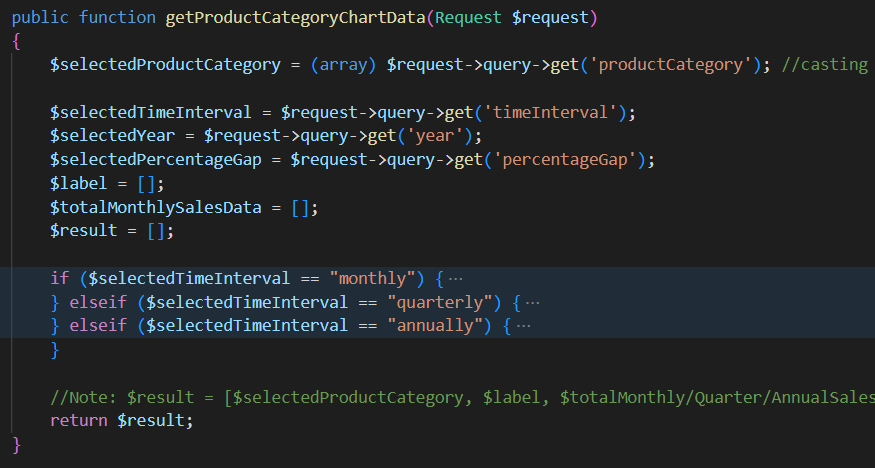
public function index(Request $request) does the following:

1. **Parameter**: It takes a $request object as a parameter. In Laravel, the $request object represents the current HTTP request, and it contains information about the request such as input data, headers, and other details.
2. **Database Query**: It queries the database using Laravel's query builder (DB::table) to retrieve distinct values of the STATE\_NA column from the outlet\_monthly\_sales table. These values are related to outlet states.
3. **Data Preparation**: It prepares an array called $controllerData, which includes data for different sections of a report page.
4. **salesSummaryTableData**: Calls the getSalesSummaryTableData method to get data related to sales summary.
5. **productCategoryAllProductCategories**: Calls the getAllProductCategoryNames method to get data related to all product categories.
6. **outletStates**: Stores the distinct STATE\_NA values obtained from the database query.
7. **View Rendering**: It returns a view called 'reports/index' along with the prepared data in the $controllerData array using Laravel's compact function.

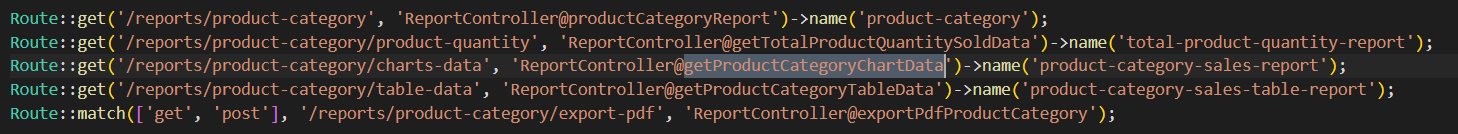
After the index method are the methods for each of the report types. In this case, we will see the product category report as an example. The public function are called during the ajax request, while the private function are called in the public function.



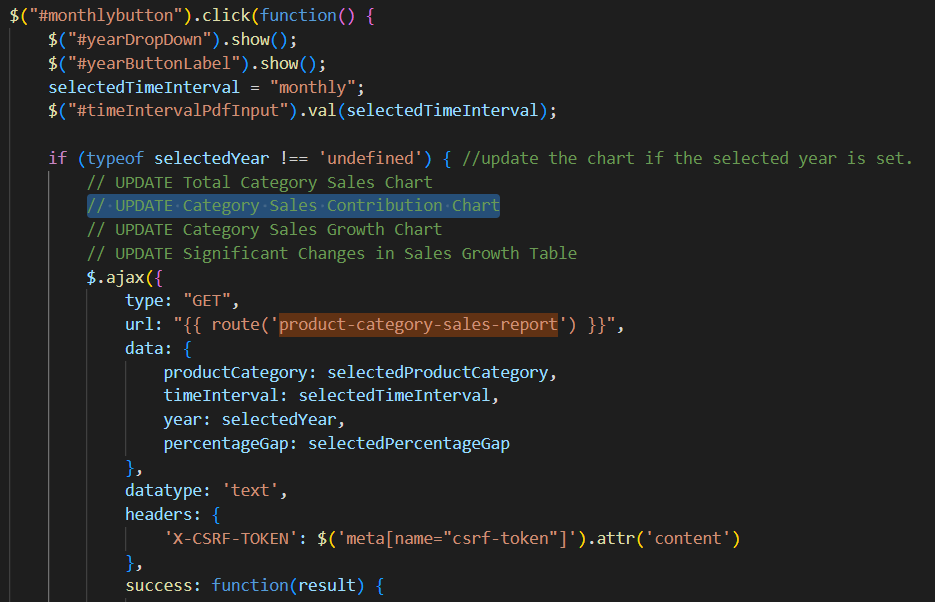
Lets take a function, getProductCategoryChartData(Request $request) as an example.



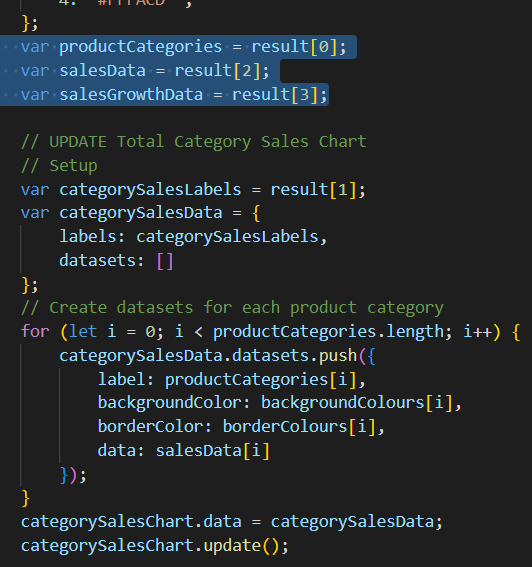
That specific function is named in a route in the web.php file (app/routes/web.php), and it is named as product-category-sales-report



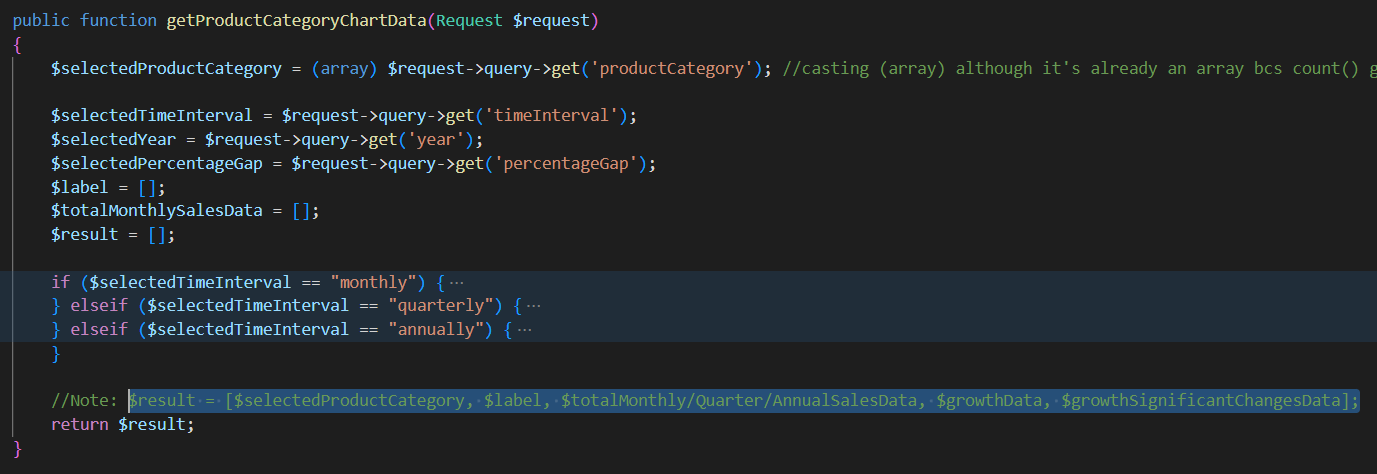
In the view front-end, that name is used in the ajax requests to update related tasks. For example, to update the Category Sales Contribution Chart when the users click the Monthly button,



‘data:‘ contains the variables to be passed to the controller function, and the returned result is used like this:

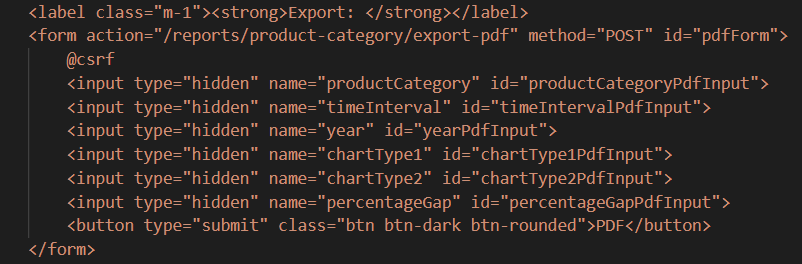


The result array elements arrangement is based on what sorted in the controller:

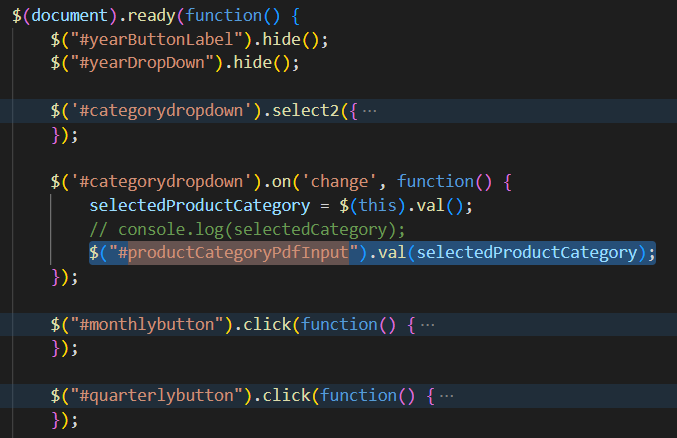


#### 4.3.2 PDF Generation

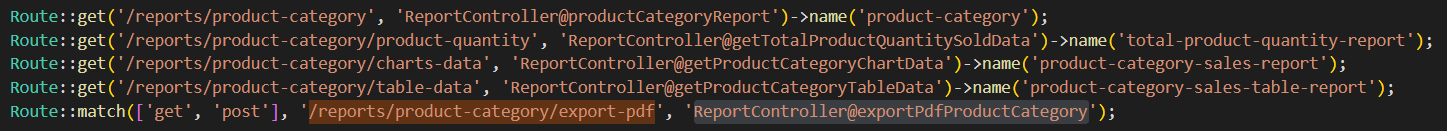
Next is how the data is used to generate the PDF report. Every report type has its a **html view template**. In this case is resources\views\reports\product\_category\_pdf\_template.blade.php. In the front-end view, this is how the users trigger the data to be sent to the controller to generate the PDF,



Each of the input variable, for example, productCategoryPdfInput is set every time the UPDATE occurs in the javascript. For the product category, the value is set when users select products in the product category dropdown.



The first picture above shows that the form method is POST to /reports/product-category/export-pdf. The controller function for that is already set in the route, which is exportPdfProductCategory in ReportController.



The exportPdfProductCategory function handles user input, retrieves relevant data from the database, generates URLs for charts, compiles all the necessary data, and generates a PDF report containing charts and tabular data related to the selected product category. The user can then download the generated PDF file.

Details:

1. **Request Handling**:

* It takes a $request object as a parameter, which contains data from the HTTP request.
* It extracts specific input values from the request, such as selected product categories, time interval, year, chart types, and percentage gap.

1. **User Authentication**:

* It retrieves the authenticated user's information using auth()->user()->pass\_no.

1. **Data Retrieval**:

* It calls the getProductCategoryChartReportData method to obtain chart-related data for the selected product categories, time interval, year, and percentage gap.
* It calls the getProductCategoryTableReportData method to obtain tabular data for the selected product categories, time interval, and year.

1. **Chart Image URLs**:

* It calls three different methods (getProductCategoryChartImageUrl) to obtain URLs for three different charts: total sales chart, sales contribution chart, and sales growth chart. These URLs will be used to display charts in the PDF.
* QuickChart (quickchart.io) is used to generate the charts as images.

1. **Data Preparation**:

* It compiles all the necessary data into an array called $data, including selected time interval, year, user information, table time interval header, chart data, table data, chart image URLs, and percentage gap.

1. **PDF Generation**:

* It sets the time zone to 'Asia/Kuala\_Lumpur'.
* It generates a unique filename for the PDF using the current date and time.
* It uses the PDF class (laravel-dompdf) to load a Blade view (reports.product\_category\_pdf\_template) **with the compiled data** and generate a PDF document.
* It sets the paper size to 'A4' in portrait orientation.
* Finally, it returns the generated PDF for download with the specified filename.

To generate the chart image url, the getProductCategoryChartImageUrl function does:

1. **Parameters**:

* $productCategory: An array containing product category names.
* $labels: An array containing labels for the chart.
* $data: An array containing data points for the chart.
* $chartType: A string specifying the type of chart (e.g., "line," "bar," "stackedArea," "stackedBar").
* $label: A string specifying the title or label for the chart.

1. **Chart Configuration**:

* The function initializes arrays for background and border colors based on the number of product categories.
* It sets the width and length of the chart image.

1. **Chart Data Configuration**:

* Depending on the specified $chartType, it configures the chart data differently:
* For "line" and "bar" charts, it sets up datasets for each product category.
* For "stackedArea" and "stackedBar" charts, it configures the chart to support stacking.

1. **JSON Encoding**:

* It converts the chart data configuration array into JSON format using json\_encode.
* The JSON string is then URL-encoded.

1. **Chart URL Construction**: The function constructs the final URL for the chart image using the QuickChart API base URL, appending the URL-encoded JSON configuration, chart width, and chart length as parameters.
2. **Return**: The generated URL for the QuickChart image is returned.

### 4.4 Conclusion

In conclusion, this section provides a comprehensive overview of the process involved in handling code for data visualization and reporting in a Laravel-based dashboard. The document delves into the overall process flow, front-end structure and back-end operations. It outlines how users interact with the dashboard, inputting variables to visualize data and how these inputs are processed through Laravel controllers to fetch and manipulate data from a MariaDB database. The front-end, coded in Blade templates and JavaScript, dynamically updates the dashboard based on user actions. The back-end, represented by the Laravel controller, retrieves data, prepares it for visualization and facilitates PDF report generation. The document concludes by detailing the steps involved in generating PDF reports, including data retrieval, chart image URL creation using QuickChart API and PDF document generation using laravel-dompdf. This guide equips users with a comprehensive understanding of the intricacies of the dashboard's codebase, facilitating effective utilization and customization.

## 5. Handling the code for code prediction module

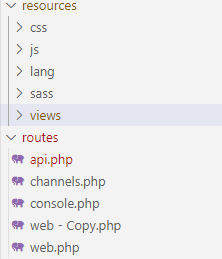
\*\*How it works\*\*

Step 1: User Interaction

Users access the web interface provided by the Laravel application to utilize the forecasting feature.

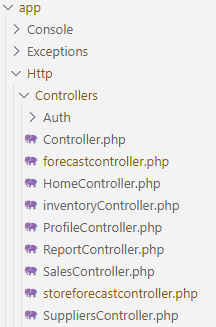
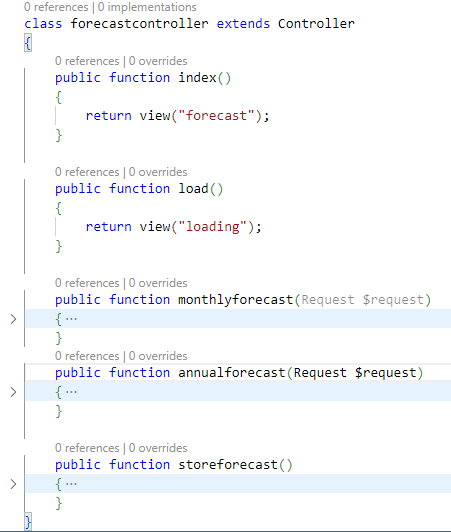
Step 2: Frontend Request

Upon accessing the forecasting page (forecast.blade.php), the frontend initiates AJAX requests to the Laravel backend.

AJAX requests are sent to the monthlyforecast and annualforecast routes defined in forecastcontroller.php.

Step 3: Laravel Backend Processing

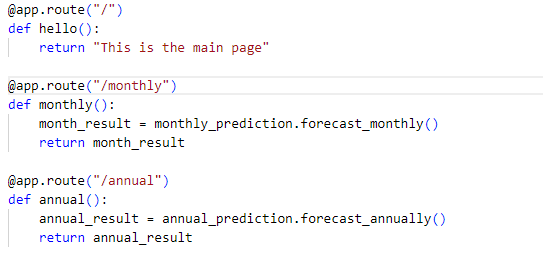
In the Laravel backend, the monthlyforecast and annualforecast methods in forecastcontroller.php handle the AJAX requests.



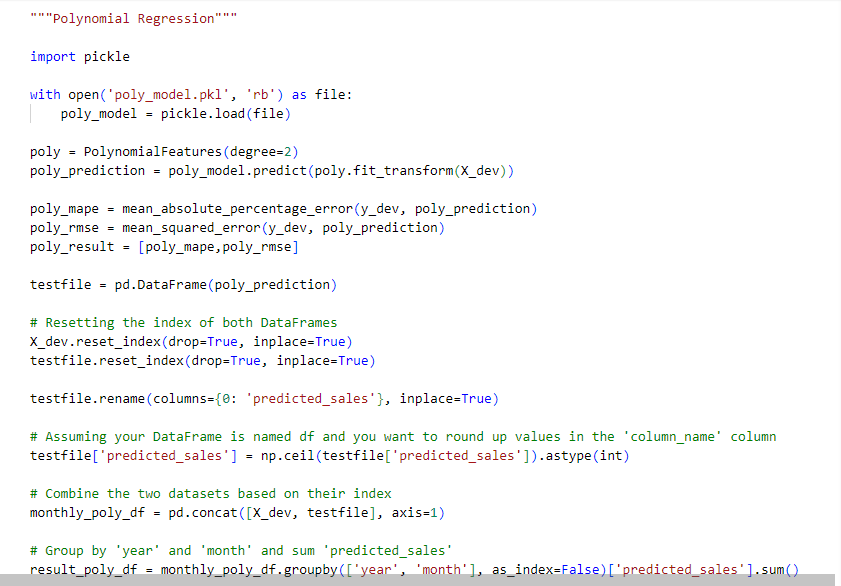
Each method sends an HTTP GET request to the corresponding Flask endpoint (/monthly and /annual) to fetch forecast data. The endpoint is located in the Flask API folder.

Step 4: Flask Endpoint Handling

The Flask application receives GET requests at the /monthly and /annual routes in main.py files. Upon receiving requests, Flask calls functions from forecasting modules (monthly\_prediction and annual\_prediction) to generate forecast data.

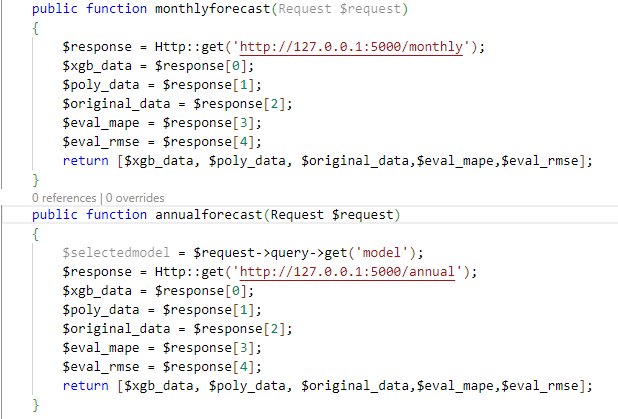


Step 5: Forecasting Logic

The monthly\_prediction and annual\_prediction modules contain forecasting algorithms and logic. These modules process historical data and generate forecasts for the desired time periods (monthly or annual). This is the example function to generate forecast, respectively.

Step 6: Data Response

Forecast data generated by the Flask application is returned as JSON responses to the Laravel backend. This is the code to handle the JSON responses in forecastcontroller.php .

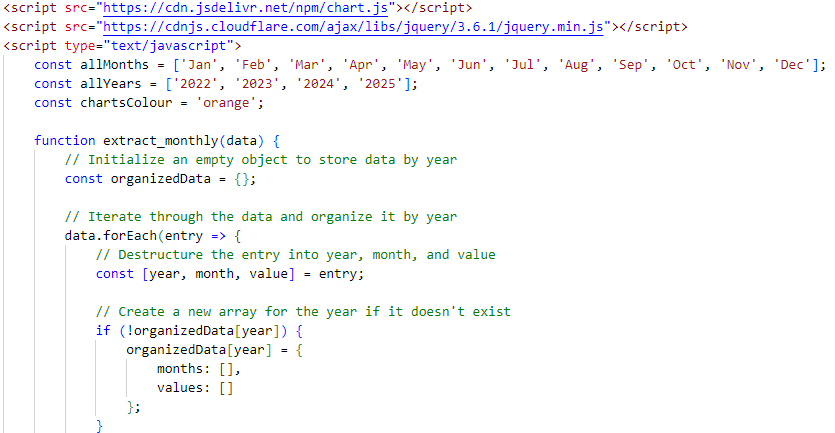


Step 7: Laravel Backend Response

In the Laravel backend, fetched forecast data is processed and returned as responses to the frontend AJAX requests.

Step 8: Frontend Data Rendering

The frontend receives forecast data from the Laravel backend.

Using JavaScript and the Chart.js library, the frontend renders forecasted sales data as charts on the forecasting page (forecast.blade.php).

Step 9: End of Process

Users interact with the forecasting feature as needed, and the application completes the requested actions.