Contents

[1. CORE FEATURES(CLIENT) 3](#_Toc137414179)

[1.1. USER REGISTRATION/LOGIN 3](#_Toc137414180)

[1.2. OBSERVE TRENDING PRODUCTS & DIVIDED PHONES AND ACCESSORIES 3](#_Toc137414181)

[1.3. PRODUCT DETAILS 3](#_Toc137414182)

[1.4. ADD TO CART 3](#_Toc137414183)

[1.5. RATING & COMMENT 3](#_Toc137414184)

[1.6. ADMIN FEATURES 4](#_Toc137414185)

[2. ACTIVITY STRUCTURE 4](#_Toc137414186)

[2.1. CATEGORY CONTROLLER CLASS 4](#_Toc137414187)

[2.1.1. HOME PAGE FUNCTION. 4](#_Toc137414188)

[2.1.2. ADD CATEGORY FUNCTION 5](#_Toc137414189)

[2.1.3. PRODUCT PAGE 6](#_Toc137414190)

[2.1.4. EDIT CATEGORY FUNCTION 7](#_Toc137414191)

[2.1.5. DELETE CATEGORY FUNCTION 8](#_Toc137414192)

[2.1.6. SAVING UPLOADED IMAGE FILE IN PROJECT 9](#_Toc137414193)

[2.2. PRODUCT CONTROLLER CLASS 9](#_Toc137414194)

[2.2.1. SHOW ADD PRODUCT FORM FUNCTION 10](#_Toc137414195)

[2.2.2. ADD PRODUCT FUNCTION 10](#_Toc137414196)

[2.2.3. SHOW UPDATE PRODUCT FORM FUNCTION 11](#_Toc137414197)

[2.2.4. UPDATE PRODUCT FUNCTION 12](#_Toc137414198)

[2.2.5. DELETE PRODUCT FUNCTION 13](#_Toc137414199)

[2.2.6. SAVING UPLOADED IMAGE FILE IN PROJECT 14](#_Toc137414200)

[2.3. HOME CONTROLLER CLASS 15](#_Toc137414201)

[2.3.1. HOME PAGE FUNCTION 15](#_Toc137414202)

[2.3.2. PRODUCT PAGE FUNCTION 17](#_Toc137414203)

[2.3.3. PHONE PAGE FUNCTION 18](#_Toc137414204)

[2.3.4. ACCESSORIES PAGE FUNCTION 19](#_Toc137414205)

[2.3.5. GET PRODUCT BY CATEGORY ID FUNCTION 20](#_Toc137414206)

[2.4. RATING CONTROLLER CLASS. 21](#_Toc137414207)

[2.4.1. INFO PRODUCT FORM FUNCTION 21](#_Toc137414208)

[2.4.2. RATING PRODUCT 23](#_Toc137414209)

[2.4.3. DATE COMPARE FUNCTION 26](#_Toc137414210)

[2.4.4. REVIEW PRODUCT FUNCTION 27](#_Toc137414211)

[2.4.5. POST REVIEW FUNCTION 28](#_Toc137414212)

[2.4.6. EDIT REVIEW PRODUCT FUNCTION 29](#_Toc137414213)

[2.4.7. UPDATE REVIEW FUNCTION 30](#_Toc137414214)

[2.3. USER CONTROLLER CLASS 31](#_Toc137414215)

[2.3.1. HOME PAGE FUNCTION. 31](#_Toc137414216)

[2.3.2. ADD USER FUNCTION 31](#_Toc137414217)

[2.3.3. CHECK REGISTER FUNCTION 32](#_Toc137414218)

[2.3.4. LOGIN FUNCTION 33](#_Toc137414219)

[2.3.5. CHECK LOGIN FUNCTION 33](#_Toc137414220)

[2.3.6. DETAIL USER FUNCTION 34](#_Toc137414221)

[2.3.7. EDIT PROFILE FUNCTION 35](#_Toc137414222)

[2.3.8. CHECK EDIT PROFILE FUNCTION 35](#_Toc137414223)

[2.3.9. CHANGE PASSWORD FUNCTION 36](#_Toc137414224)

[2.3.10. CHECK CHANGE PASSWORD FUNCTION 37](#_Toc137414225)

[2.3.11. ADD CART FUNCTION 38](#_Toc137414226)

[2.4. USER CONTROLLER CLASS 39](#_Toc137414227)

[2.4.1. VIEW CART FUNCTION 39](#_Toc137414228)

[2.4.2. CLEAR CART FUNCTION 40](#_Toc137414229)

[2.4.3. REMOVE CART FUNCTION 40](#_Toc137414230)

[2.4.4. UPDATE QUANTITY ITEM CART FUNCTION 41](#_Toc137414231)

[2.4.5. CHECK OUT FUNCTION 41](#_Toc137414232)

[2.4.6. CONFIRM CHECK OUT FUNCTION 42](#_Toc137414233)

[2.4.7. VIEW HISTORY FUNCTION 44](#_Toc137414234)

[2.4.8. VIEW DETAIL HISTORY FUNCTION 44](#_Toc137414235)

[2.4.9. VIEW ORDERS FUNCTION (ADMIN SITE) 45](#_Toc137414236)

[2.4.10. VIEW DETAIL ORDER FUNCTION (ADMIN SITE) 46](#_Toc137414237)

[2.4.11. UPDATE STATUS ORDER FUNCTION 46](#_Toc137414238)

# CORE FEATURES(CLIENT)

## USER REGISTRATION/LOGIN

* Only appears once (the first time the application is run)
* Allows the user to register with the Phone selling server
* Enables the user to customize his/her personal detail.

## OBSERVE TRENDING PRODUCTS & DIVIDED PHONES AND ACCESSORIES

* The first page features a part exclusively dedicated to displaying trending products.
* And also divided into two part “PHONES” and “ACCESSORIES”.

## PRODUCT DETAILS

* User can check every single once.
* Allows the user to see if the product being for sale or not.
* Allows the user to see if the product is trending or not.

## ADD TO CART

* User can add the product and the amount they want to their cart
* In the cart they can change the quantity of product and then check out

## RATING & COMMENT

* The user can rate and comment the product only they have bought that product 7-14 days before

## ADMIN FEATURES

* Manage customer and employee accounts.
* Determine the official prices for each item.
* Manage products, add, delete, and update products.
* Manage product categories.
* Manage and process orders.
* The responsibilities of this department include network administration, customer information management, updating product information, updating customer information, ensuring that the database is always updated quickly.

# ACTIVITY STRUCTURE

## CATEGORY CONTROLLER CLASS

This is belonged to Admin features for example: homepage function, add Category function, …

### HOME PAGE FUNCTION.

1. When an admin login the system URL will change to ‘…/phone-selling/home’ directly, the first page appear is the home page which is represented by the

View:

* + ‘home.jsp’ file located at the path ‘admin/home.jsp’.

### ADD CATEGORY FUNCTION

After an admin access into ‘…/phone-selling/addCategory’ they will get back the form page use only for adding new category.

1. When an admin submits the required inputs for a Category
   * Name
   * Description
   * Status
   * image.
2. At here we call the addCategory function which is created in

DAO:

* + The addCategory function is created in the CateDAO class.

DAOImplement:

* + The addCategory function is implemented in the CateDAOImpl class.
  + Inside this function, a new Category object will be created and stored in the database.
  + SQL: INSERT INTO categories (name, description, status, image) VALUES (?, ?, ?, ?)

View:

* ‘Category.jsp’ page is responsible for rendering the form page for adding a new category.

After successfully storing the new category into the database, the admin will be redirected to the URL ‘ ...phone-selling/productandcategory ’.

### PRODUCT PAGE

This is a Get method with value : ‘...phone-selling/ productandcategory’.

1. The system will take all the categories and products in the database by using

DAO:

* getAllCategory(): Created in the CateDAO class.
* getAllProducts(): Created in the ProductDAO class.

DAO Implementation:

* getAllCategory() is implemented in the CateDAOImpl class.
  + SQL: select \* from categories
* getAllProducts() is implemented in the ProductDAOImpl class.
  + SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id.

1. The retrieved categories and products are then stored in ArrayLists named 'listCate' and 'listProduct', respectively. The values from these lists are used to render to the

View:

* ‘ Category.jsp ’ file, which is located at ‘ admin/Category.jsp ’.

1. On the page we have two table one for categories one for product. With categories we have 6 columns namely
   * ID, NAME, DESCRIPTION, STATUS, IMAGE and ACTION.
   * Little bit difference with Product table which has 10 columns: NAME, CATEGORY, DESCRIPTION, STATUS, ORIGINAL PRICE, SELLING PRICE, QUANTITY, TRENDING, IMAGE AND ACTION.
   * With everysingle action button has the ID of the category/product in it in order to access to EDIT CATEGORY/PRODUCT FUNCTION or DELETE CATEGORY/PRODUCT FUNCTION

### EDIT CATEGORY FUNCTION

When user choose the category to edit the system responses an edit form

1. To retrieve the information of that category, the system utilizes the getCategoryByID() function from the CateDAO class.

DAO:

* getCategoryByID() function is created in the CateDAO class.

DAO Implementation:

* The getCategoryByID() function is implemented in the CateDAOImpl class.
* This function retrieves all the information of the selected category based on its ID.
* SQL: SELECT \* FROM categories WHERE id = ?

1. The retrieved category information is then displayed on the

View:

* ‘editCategory.jsp’ page, which is located at ‘admin/category/editCategory.jsp’.

1. On the ‘editCategory.jsp’ page, users can modify the values they want to change, such as the name, description, status, and image of the category. The form will have inputs for these fields.

After the user submits the form, the system retrieves the input values:

* Name.
* Description
* Status
* image.

If a new image is uploaded, the system gets the image name and sets it for that Category.

1. To update the category data, the system executes the updateCategory() function, which is created in the CateDAO class and implemented in the CateDAOImpl class.

DAO:

* + updateCategory() function is created in the CateDAO class.

DAO Implementation:

* + The updateCategory() function is implemented in the CateDAOImpl class.
  + This function updates the Category object's data in the database based on the modified values.
  + SQL: UPDATE categories SET name = ?, description = ?,status = ?, image= ? WHERE id = ?

After successfully updating the category data in the database, the user is redirected to the URL ‘...phone-selling/productandcategory’.

### DELETE CATEGORY FUNCTION

This also a GET method feature.

1. When a user selects a category they want to remove by clicking the ‘Delete’ button, the system returns the URL ‘.../phone-selling/delete-category/id’. The ‘id’ parameter in the link contains the ID of the category that needs to be deleted.
2. The system uses this ID to pass it to the deleteCategory() function in the CateDAO class.

DAO:

* deleteCategory() function is created in the CateDAO class.

DAO Implementation:

* The deleteCategory() function is implemented in the CateDAOImpl class.
* This function takes the category ID as a parameter and deletes the corresponding Category object from the database.
* SQL: DELETE FROM categories WHERE id = ?

Once the Category object's data is successfully deleted from the database, the system redirects the user to the URL ‘...phone-selling/productandcategory ’.

### SAVING UPLOADED IMAGE FILE IN PROJECT

1. Purpose:
   * The purpose of the "saveImageInProject" method is to save an uploaded image file into a designated directory within the project.
2. Inputs:
   * MultipartFile file: The uploaded image file to be saved.
3. Outputs:
   * String filePath: The file path where the image file is saved.

Steps:

1. Get the upload directory path within the project.
2. Get the original filename of the uploaded file.
3. Create the complete file path by combining the upload directory path and the original filename.
4. Print the upload directory path for debugging purposes.
5. Write the uploaded file to the specified file path.
6. Return the file path where the image file is saved.

## PRODUCT CONTROLLER CLASS

This is belonged to Admin feature for example: addProdcut, editProduct, ...

### SHOW ADD PRODUCT FORM FUNCTION

When an Admin want to create a new product they need to access to “ …/phone-selling/product ”.

1. Create a Categories list named ‘listCate’.

DAO:

* getAllCategory(): This function is created in CateDAO class.

DAO Implementation:

* The getAllCategory() function is implemented in CateDAOImpl class.
* This function will return back to us a list of categories

1. Put all the data we get into ‘listCate’
2. Add a new modelMap named ‘product’. This attribute will be used to bind the form data and retrieve the user's inputs for creating a new product.
3. Retrieve the list of categories from the CateDAO and add it as an attribute to the model map. This attribute will be used to populate the category dropdown/select options in the add product form.
4. Lastly, the web will represented for the viewer by ‘addProduct.jsp’ which is located in “admin/product/addProduct”.

### ADD PRODUCT FUNCTION

After an admin access into “…/phone-selling/product” they will get back the form page use only for adding new product.

1. When an admin submits the required inputs for a Category

* Category\_id (select by the Category name in dropdown list)
  + Name
  + Description
  + Og\_price
  + Selling\_price
  + Quantity
  + Status
  + Trending
  + image.

A new image is uploaded, the system gets the image name and sets it for that Product.

1. At here we call the addCategory function which is created in

DAO:

* + The addProduct function is created in the ProductDAO class.

DAOImplement:

* + The addProduct function is implemented in the ProductDAOImpl class.
  + By this function, a new Product object will be created and stored in the database.
  + SQL: INSERT INTO products(cate\_id, name, description, og\_price, selling\_price, image, qty, status, trending) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?).

View:

* ‘Category.jsp’ page is responsible for rendering the form page for adding a new category.

After successfully storing the new product into the database, the admin will be redirected to the URL “...phone-selling/productandcategory”.

### SHOW UPDATE PRODUCT FORM FUNCTION

When a user chooses to edit a product, the system responds with an edit form. Here are the steps involved in the process:

This is a Get method with value: ‘edit-product/id’ it access when admin click to ‘edit’ button on the product

1. To retrieve the information of that product, the system utilizes the getProductById() function from the ProductDAO class.

DAO:

* The getProductById() function is created in the ProductDAO class.

DAO Implementation:

* The getProductById() function is implemented in the ProductDAOImpl class.
* This function retrieves all the information of the selected product based on its ID.
* SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id Where p.id = ?

1. The retrieved product information is then displayed on the ‘editProduct.jsp’ page, which is located at “admin/product/editProduct.jsp”.

View:

* The ‘editProduct.jsp’ page displays the existing values of the product, such as name, description, status, original\_price, selling\_price, trending and image, in input fields.

### UPDATE PRODUCT FUNCTION

When admin submit those change the request from admin have the value: ‘edit-product/product/id/edit’.

1. Users can modify these values according to their requirements.

After the user submits the form, the system retrieves the input values:

* Category\_id (select by the Category name in dropdown list)
  + Name
  + Description
  + Og\_price
  + Selling\_price
  + Quantity
  + Status
  + Trending
  + image.

If a new image is uploaded, the system gets the image name and sets it for that product.

To update the product data, the system executes the updateProduct() function, which is created in the ProductDAO class and implemented in the ProductDAOImpl class.

DAO:

* The updateProduct() function is created in the ProductDAO class.

DAO Implementation:

* The updateProduct() function is implemented in the ProductDAOImpl class.
* This function updates the product object's data in the database based on the modified values by Admin.
* SQL: UPDATE products SET cate\_id = ?, name = ?, description = ?, og\_price = ?,selling\_price = ?, image = ?, qty = ?, status = ?, trending = ? WHERE id =?.

View:

* ‘Category.jsp’ page is responsible for rendering the form page for adding a new product.

After successfully updating the product data in the database, the user is redirected to the URL “...phone-selling/productandcategory”.

### DELETE PRODUCT FUNCTION

This also a GET method feature.

1. When a user selects a product they want to remove by clicking the ‘Delete’ button, the system returns the URL ‘.../phone-selling/ delete-product/id’. The ‘id’ parameter in the link contains the ID of the product that needs to be deleted.
2. The system uses this ID to pass it to the deleteProduct () function in the ProductDAO class.

DAO:

* deleteProduct () function is created in the ProductDAO class.

DAO Implementation:

* The deleteProduct () function is implemented in the ProductDAOImpl class.
* This function takes the product ID as a parameter and deletes the corresponding Product object from the database.
* SQL: DELETE FROM products WHERE id = ?.

View:

* ‘Category.jsp’ page is responsible for rendering the data after updating.

After successfully delete the product from the database, the user is redirected to the URL “...phone-selling/productandcategory”.

### SAVING UPLOADED IMAGE FILE IN PROJECT

1. Purpose:
   * The purpose of the "saveImageInProject" method is to save an uploaded image file into a designated directory within the project.
2. Inputs:
   * MultipartFile file: The uploaded image file to be saved.
3. Outputs:
   * String filePath: The file path where the image file is saved.

Steps:

1. Get the upload directory path within the project.
2. Get the original filename of the uploaded file.
3. Create the complete file path by combining the upload directory path and the original filename.
4. Print the upload directory path for debugging purposes.
5. Write the uploaded file to the specified file path.
6. Return the file path where the image file is saved.

## HOME CONTROLLER CLASS

This controller is serving for the client. It take responsible for operating core client’s features.

### HOME PAGE FUNCTION

The **homePage** method is a request handler for the **GET** mapping "/clienthome". It is responsible for rendering the home page based on the user's role.

Parameters

* **modelmap** (ModelMap): An object to hold the model attributes for the view.
* **session** (HttpSession): The HTTP session object for storing and retrieving user session data.

1. Retrieve the logged-in user from the HTTP session using the attribute key "userSession"
2. Check if the logged-in user has the role "admin". If true, return a ModelAndView object with the view name "admin/home" to render the admin home page.
3. If the user does not have the role "admin", continue with the following steps:

* Create a **ModelAndView** object with the view name "client/home".
* Initialize two empty lists: **listProduct** to hold trending products and **listCate** to hold limit categories.
* Retrieve the limit categories by invoking the **getLimitCategory** method of the **CateDAO** class.

DAO

* + Call the **getLimitCategory** function defined in the **CateDAO** class.
  + This function retrieves a list of limit selling categories from the database.

DAO Implementation:

* Implement the **getLimitCategory** method in the **CateDAOImpl** class.
* Execute the SQL query to fetch the limit categories and return the list.
* SQL: SELECT TOP 5 \* FROM categories WHERE status=1 .
* Retrieve the trending products by invoking the **getTrendingProduct** method of the **productDAO** object.

DAO

* + Call the **getTrendingProduct** function defined in the **ProductDAO** class.
  + This function retrieves a list of trending products from the database.

DAOImplementation

* + - Implement the getTrendingProduct method in the ProductDAOImpl class.
    - Execute the SQL query to fetch the trending products and return the list.
  + SQL: SELECT TOP 4 p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id where p.trending = 1 and c.status = 1
* Add the **listProduct** and **listCate** attributes to the modelmap object.

1. Return.

VIEW:

‘home.jsp’ to render the client home page with the appropriate data

### PRODUCT PAGE FUNCTION

The **productPage** method is a request handler for the **GET** mapping "/allProduct". It is responsible for rendering the product page and displaying all products available for the client.

Parameters:

* **modelmap** (ModelMap): An object to hold the model attributes for the view.

1. Create a ModelAndView object with the view located "client/product/index".
2. Initialize an empty list **listProduct** to hold all the products.
3. Retrieve all products for the client by invoking the **getAllProductsClient** method of the **productDAO** object.

DAO

* + Call the **getAllProductsClient** function defined in the **ProductDAO** class.
  + This function retrieves all products from the database that are available for the client, it get the product but the category of it have to be ‘1’.

DAO Implementation:

* Implement the **getAllProductsClient** method in the **ProductDAOImpl** class.
* Execute the SQL query to fetch all the products and return the list
* SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id where c.status = 1.

1. Add the **listProduct** attribute to the modelmap object.
2. Return the ModelAndView.

VIEW:

* Page ‘index.jsp’ is responsible for renderring the product page with the list of products located in"client/product/index".

### PHONE PAGE FUNCTION

The **phonePage** method is a request handler for the **GET** mapping "**/**phone". It is responsible for rendering the phone page and displaying all phone products available for the client.

Parameters:

* **modelmap** (ModelMap): An object to hold the model attributes for the view.

1. Create a ModelAndView object with the view located "client/product/index".
2. Initialize an empty list **listProduct** to hold all the phone products.
3. Retrieve phone products by invoking the **getPhone** method of the **productDAO** class

DAO

* Call the **getPhone** function defined in the **ProductDAO** class.
* This function retrieves all phone products from the database their category status is **1** for the client.

DAOImplementation

* Implement the **getPhone** method in the **ProductDAOImpl** class.
* Execute the SQL query to fetch all the phone products and return the list.
* SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id Where p.cate\_id = 21.

1. Add the **listProduct** attribute to the **modelmap** object.
2. Return the ModelAndView object to render the phone page with the list of phone products.

### ACCESSORIES PAGE FUNCTION

The **accessoriesPage** method is a request handler for the **GET** mapping "/accessories". It is responsible for rendering the accessories page and displaying all accessory products available for the client.

Parameters:

* **modelmap** (ModelMap): An object to hold the model attributes for the view.

1. Create a ModelAndView object with the view located "client/product/index".
2. Initialize an empty list **listProduct** to hold all the accessory products.
3. Retrieve accessory products by invoking the **getOther** method of the **productDAO** class

DAO

* Call the **getOther** function defined in the **ProductDAO** class.
* This function retrieves all accessory products from the database that are available for the client.

DAO Implementation:

* Implement the **getOther** method in the **ProductDAOImpl** class.
* Execute the SQL query to fetch all the accessory products and return the list.
* SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id Where p.cate\_id <> 21 and c.status = 1".

1. Add the **listProduct** attribute to the **modelmap** object.
2. Return the ModelAndView: ‘index.jsp’ file to render the accessories page with the list of accessory products.

### GET PRODUCT BY CATEGORY ID FUNCTION

The **getProductByCategoryID** method is a request handler for the **GET** mapping "/category/id". It retrieves products based on a specific category ID and renders the product page with the filtered products

Parameters

* **id** (Path Variable): An integer representing the ID of the category to retrieve products for.
* **modelmap** (ModelMap): An object to hold the model attributes for the view.

1. Create a ModelAndView object with the view located "client/product/index".
2. Initialize an empty list **listProduct** to hold the products for the specified category.
3. Retrieve products by invoking the **getByCateID** method of the **productDAO** object

DAO

* Call the **getByCateID** function defined in the **ProductDAO** class.
* This function retrieves products from the database that belong to the specified category ID.

DAO Implementation

* Implement the **getByCateID** method in the **ProductDAOImpl** class.
* Execute the SQL query with the provided category ID and return the list of products
* SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id Where c.status=1 and p.cate\_id = ?.

1. Add the **listProduct** attribute to the **modelmap** object.
2. Return the ModelAndView object to render the product page with the filtered products.

## RATING CONTROLLER CLASS.

This class belongs to Client feature, it responsible for handling product ratings and reviews from customers.

### INFO PRODUCT FORM FUNCTION

The **infoProductForm** method is a request handler for the GET mapping "info/id". It retrieves information about a specific product, along with its rating and reviews, and prepares the data to be displayed on the client-side.

Parameters:

* ‘**id**’ (Path Variable): An integer representing the ID of the product to retrieve information for.
* ‘**modelMap**’ (ModelMap): An object to hold the model attributes for the view.
* ‘**httpSession**’ (HttpSession): The HTTP session object for storing and retrieving user session data.

1. Retrieve the logged-in user from the HTTP session using the attribute key "**userSession**".
2. Check if the logged-in user has the role "admin". If true, redirect the user to the "phone-selling/home" page.
3. If not, continue with the following steps:
   1. Retrieve the product information by invoking the getProductById method.

DAO

* Calling ‘**getProductById**’function and passing the ‘id’ parameter. This function is created in ProductDAO class.

DAOImplementation:

* ‘**getProductById**’ is implemented in ProductDAOImpl class after passing the ID parameter this function will return the product information.
  + SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id Where p.id = ?
  1. Retrieve a list of ratings for the product by invoking the **getRateByProductId** method.

DAO

* + This function is called and created in RatingDAO class and use the ‘id’ parameter.

DAOImplementation

* + **getRateByProductId** will handle getting the product’s star rated. Only by using the ‘**id’** parameter. Then return a list of it
  + SQL: select stars\_rated from rate\_prod where prod\_id = ?

Create an Integer list then put the value we get into this. By this list we can do two things for the function:

1. Calculate the number of ratings (‘**rateNum’**) by getting the size of the rating list.
2. Calculate the average rating (‘**average’**) by converting the rating list to an IntStream, mapping each rating to its integer value, calculating the average using the **average()** method, and returning 0 if the stream is empty.
   1. Retrieve a list of reviews for the product by invoking the **getAllReview** method.

DAO

* + **getAllReview** function is created in ReviewDAO class and using the ‘id’ parameter to pass in.

DAOImplementation

* + **getAllReview** will take responsible for retrieving a list of reviews for the product by the ‘id’ is pass into it.
  + SQL: SELECT r.\*, u.name, rt.stars\_rated FROM reviews r JOIN users u ON r.user\_id = u.id join rate\_prod rt on r.user\_id = rt.user\_id and r.prod\_id = rt.prod\_id where r.prod\_id = ?

After that put it in a list and named it ‘**listReview’**

Add the following attributes to the modelMap:

* + "**product**": The retrieved product object.
  + "**rateNum**": The number of ratings for the product.
  + "**rate**": The average rating for the product.
  + "**listReview**": The list of reviews for the product.

1. Return the view name ‘view.jsp’ which located in "client/product/view" to render the product information, rating, and reviews on the client-side.

### RATING PRODUCT

The ‘**ratingProduct’** method is a request handler for the POST mapping "/add-rating". It handles the process of rating a product by a customer and performs validation checks before storing the rating in the database.

Parameters:

* **rating** (ModelAttribute): An object of type **Rate\_prod** containing the rating details submitted by the customer.
* **redirectAttributes** (RedirectAttributes): An object for adding flash attributes to be accessed in the redirected view.

1. If a client wants to rating a product they have to pass the condition by the **checkOrder** method:

DAO:

* The function is created in OrderItemDAO class, check if an order item exists for the given **user\_id** and **prod\_id.**

DAOImplementation:

* **checkOrder** is implemented in OrderItemDAOImpl class. It will take the parameter: **user\_id** and **prod\_id** passed in when a client posts a Rate\_prod object.
* SQL: SELECT top 1 r.\* FROM order\_details r JOIN orders p ON r.order\_id = p.id where p.user\_id = ? and r.prod\_id= ? ORDER BY ID DESC

1. If **checkOrder** returns a non-null value, continue with the following steps.
   1. Retrieve the ‘**created\_at**’ date of the order item and perform a comparison using the **dateCompare** method.
      * + If the comparison result is "Valid", continue with the next steps.
        + If the comparison result is "Outvalid", add a flash attribute "status" with the value "Fail to rating, out of range" and redirect the user back to the product information page.
        + If the comparison result is any other value, add a flash attribute "status" with the value "Please try it 7 days before rating" and redirect the user back to the product information page.
2. Check if a rating already exists for the given **user\_id** and **prod\_id** by invoking the **findCliendId**

DAO

* Calling **findCliendId** function, created in RatingDAO class, it uses to check if the user has rated the product before.

DAOImplemenation

* **findCliendId** is implemented in RatingDAOImpl class, it uses the **user\_id** and **prod\_id** given before to check.
* SQL: select \* from rate\_prod where user\_id = ? and prod\_id = ?

1. If **findCliendId** returns null, it means the customer has not rated the product before. In this case, proceed with the next steps.
2. Invoke the **addRating** method and pass the input from the form.

* User\_id
* Prod\_id
* Stars\_rated

DAO

* **addRating** function is created in RatingDAO class, this use for creating and store user’s rating to the product into database.

DAOImplementation

* When a client submits their rating the **addRating** function in RatingDAOImpl will handle this
* The function takes the input and work with database
* SQL: INSERT INTO rate\_prod (user\_id, prod\_id, stars\_rated) VALUES (?, ?, ?).

Add a flash attribute "status" with the value "Rating success".

1. But if a rating already exists for the given **user\_id** and **prod\_id**, invoke the **updateRatingByUserId .**

DAO

* **updateRatingByUserId** function is created in RatingDAO class, this use for finding and updating user’s rating to the product into database by passing the rating object.

DAOImplementation

* **updateRatingByUserId** is implemented in RatingDAOImpl class it will find and update that object
* SQL: UPDATE rate\_prod SET stars\_rated = ? WHERE user\_id = ? AND prod\_id = ? .

Add a flash attribute "status" with the value "Rating success".

1. Build the URL for redirecting the user to the product information page by concatenating "phone-selling/info/" with the ‘prod\_id’.
2. Redirect the user to the constructed URL.

### DATE COMPARE FUNCTION

This function it’s use with ratingProduct function to compare with the ‘created\_at’ value.

The dateCompare function compares two dates and determines their difference in days. It follows the following steps:

Parameters:

* **d1**: A string representing the date to compare in the format "yyyy-MM-dd".

Flow:

1. Create a **SimpleDateFormat** object (**sdformat**) with the pattern "yyyy-MM-dd" to parse the dates.
2. Get the current date by creating a **Date** object (**date**).
3. Parse the input date (**d1**) using the **sdformat.parse()** method, and assign it to a **Date** object (**day1**).
4. Format the current date (**date**) using the **sdformat.format()** method and parse it back to a **Date** object (**day2**).
5. Calculate the time difference in milliseconds between **day2** and **day1** using the **getTime()** method, and take the absolute value using **Math.abs().**
6. Convert the time difference to days using the **TimeUnit.DAYS.convert()** method, passing the time difference and the **TimeUnit.MILLISECONDS**.
7. If the calculated **daysDiff** is greater than or equal to 7 and less than or equal to 14, return "Valid".
8. If the **daysDiff** is greater than or equal to 14, return "Outvalid".
9. Otherwise, return "Unvalid".

This function compares the provided date (**d1**) with the current date and determines whether it falls within the valid range of 7 to 14 days. It returns "Valid" if the difference is within the range, "Outvalid" if the difference is greater than 14 days, and "Unvalid" for any other case.

### REVIEW PRODUCT FUNCTION

This is a GET method with value: “...phone-selling/add-review/id/review” to forward to review form by the following steps:

Parameters:

* **id** (Path Variable): An integer representing the **ID** of the product to add a review for.
* **modelMap** (ModelMap): An object to hold the model attributes for the view.

1. Retrieve the product information by invoking the **getProductById** method.

DAO:

* Call the **getProductById** function defined in the **ProductDAO** class, passing the **id** parameter.
* This function retrieves the product information corresponding to the provided **id**. But in the **reviewProduct** function

DAO Implementation:

* Implement the **getProductById** method in the **ProductDAOImpl** class.
* Execute the SQL query with the id parameter and return the **Product** object.
* SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id Where p.id = ?

1. Add the retrieved product object to the modelMap with the attribute name "product".
2. Return the

VIEW:

* Name "review.jsp" located in the "client/product" directory to render the review form for the product.

### POST REVIEW FUNCTION

The **postReview** method is a request handler for the POST mapping "...phone-selling/add-review". It handles the submission of a review form and saves the review to the database with input:

* User\_id
* Prod\_id
* User\_review
* Created\_at

Parameters:

* **review** (ModelAttribute): An object representing the submitted review form data.

1. Retrieve the product **ID** from the submitted review object by invoking the **getProd\_id** method.
2. Add the review to the database by invoking the **addReview** method of the **reviewDAO** object.

DAO

* Call the **addReview** function defined in the **ReviewDAO** class, passing the **Review** object.
* This function adds the review to the database.

DAO Implementation:

* Implement the **addReview** method in the **ReviewDAOImpl** class, it takes the input from the form then creates and stores new Review object in database.
* Execute the SQL query with the values extracted from the review object.
* SQL: INSERT INTO reviews (user\_id, prod\_id, user\_review, created\_at) VALUES (?, ?, ?, ?).

1. Create the URL string for redirecting to the product information page by concatenating "...phone-selling/info/" with the **prod\_id**.
2. Return a redirect response to the constructed URL to display the updated product information page, including the newly added review.

### EDIT REVIEW PRODUCT FUNCTION

The **editReviewProduct** method is a request handler for the **GET** mapping "/edit-review/id/userreview". It retrieves the details of a specific review and prepares the data to be displayed in an edit review form.

Parameters:

* **id** (Path Variable): An integer representing the ID of the review to be edited.
* **modelMap** (ModelMap): An object to hold the model attributes for the view.

1. Retrieve the review object from the database by invoking the **findReview** method of the **reviewDAO** object.

DAO

* Call the **findReview** function defined in the **ReviewDAO** class, passing the **id** parameter.
* This function retrieves the review with the specified **ID** from the database.

DAO Implementation

* Implement the **findReview** method in the **ReviewDAOImpl** class.
* Execute the SQL query with the id parameter and return the retrieved review object.
* SQL: SELECT r.\*, p.name FROM reviews r JOIN products p ON r.prod\_id = p.id where r.id =? .

1. Add the retrieved review object to the **modelMap** as an attribute with the key "**review**".
2. Return

VIEW

* Name ‘edit.jsp’ located in "client/product/edit " to render the edit review form.

### UPDATE REVIEW FUNCTION

The **updateReview** method is a request handler for the POST mapping "/update-review". It is responsible for updating an existing review with new review provided by the user.

Parameters

* **review** (ModelAttribute): An object representing the updated review data.

1. Update the review in the database by invoking the updateReview method of the reviewDAO object.

DAO

* Call the **updateReview** function defined in the **ReviewDAO** class, passing the **review** object.
* This function updates the corresponding review in the database with the new review data.

DAO Implementation

* Implement the **updateReview** method in the **ReviewDAOImpl** class, base the parameter that passed in the Review object will be changed and stored.
* Execute the SQL update query with the values from the review object.

1. Retrieve the **prod\_id** (product ID) from the updated review object.
2. Create a URL string by concatenating "/info/" and the prod\_id.
3. Redirect the user to the specified URL using the "redirect" prefix, which will trigger a GET request to the redirected URL.
4. The user will be redirected to the "info" page of the corresponding product, where the updated review will be displayed.

## USER CONTROLLER CLASS

This is belonged to Admin features for example: homepage function, add Category function, …

### HOME PAGE FUNCTION.

1. When an admin login the system url will change to ‘…/phone-selling/home’ directly, the first page appear is the home page which is represented by the

View:

* + ‘home.jsp’ file located at the path ‘admin/home.jsp’.

### ADD USER FUNCTION

After an client access into ‘…/phone-selling/addUser they will get the form page use only register account.

1. When an client submits the required inputs for a user information
   * Name
   * Password
   * Confirm password
   * Email.
   * Phone
   * Address
   * Gender
   * Titles of address

### CHECK REGISTER FUNCTION

After an client access into ‘…/phone-selling/checkRegister they will post the form page use only register account, after submitting the button “Create”

1. When an client submits the required inputs for a User
   * Name
   * Password
   * Confirm password
   * email.
   * Phone
   * Address
   * Gender
   * Titles of address
2. At here we call the **checkRegister(), saveUser()** function which is created in

DAO:

* + The **checkRegister(), saveUser()** function is created in the CateDAO class.

DAOImplement:

* + In the function **saveUser()**, we use SQL: INSERT INTO users (name, email, password, phone, address, role, state, gender, titles\_of\_address) VALUES (?, ?, ?, ?, ?, ?, ?,?,?)
  + In the function **checkRegister(),** we do the validation for checking required inputs in the register form

View:

* ‘add-user.jsp’ page is responsible for rendering the form page for adding a new user.

After successfully storing the new user into the database, the admin will be redirected to the URL ‘ ...phone-selling/addUser ’.

### LOGIN FUNCTION

After an client access into ‘…/phone-selling/login they will get the form page use only login an account.

1. When an user submits the required inputs for a user information
   * Email
   * Password

### CHECK LOGIN FUNCTION

After an client access into ‘…/phone-selling/checkLogin they will post the form page use only login account, after submitting the button “login”

1. When an client submits the required inputs for a User account
   1. Email
   2. Password
2. At here we call the **checkLogin(), getUserById(), getUserByEmail()** function which is created in

DAO:

* + The **checkLogin(), getUserById(), getUserByEmail(), getAllUser()** function is created in the CateDAO class.

DAOImplement:

* + In the function **getUserById()**, we use SQL: SELECT \* FROM users WHERE id = ?
  + In the function **getUserByEmail()**, we use SQL: SELECT \* FROM users WHERE email = ?
  + In the function **checkLogin(),** we do the validation for checking required inputs in the login form
  + In the function **getAllUser(),** we use SQL: SELECT \* FROM users

View:

* ‘login.jsp’ page is responsible for rendering the form page for logining an user account.

After successfully logining, the user will be redirected to the URL ‘ ...phone-selling/home ’ if user is an admin, ‘ ...phone-selling/clienthome ’ if user is a client.

* “home.jsp” in the folder “views/admin” page is responsible for rendering home page of the admin, which is User list page – listing user of the system.
* “home.jsp” in the folder “views/client” page is responsible for rendering home page of the client, which is listing all product for client to buy.

### DETAIL USER FUNCTION

After an client access into ‘…/phone-selling/detailUser/{id} they will get the page viewing the information of specific user, after clicking the button “detail” in table user list at URL: “...phone-selling/home”

At here we call the **getUserById()** function which is created in

DAO:

* + The **getUserById()** function is created in the CateDAO class.

DAOImplement:

* + In the function **getUserById(),** we use SQL: SELECT \* FROM users WHERE id = ?

View:

* ‘detail-user.jsp’ in the folder “views/admin/users” is responsible for rendering the page for viewing detail information of an user account.

### EDIT PROFILE FUNCTION

After an client access into ‘…/phone-selling/editProfile/{id}’ if user id admin or ‘…/phone-selling/editProfileUser/{id}’ is client they will get the page viewing the information of specific user, after clicking the button “Profile”

At here we call the **getUserById()** function which is created in

DAO:

* + The **getUserById()** function is created in the CateDAO class.

DAOImplement:

* + In the function **getUserById()**, we use SQL: SELECT \* FROM users WHERE id = ?

View:

* ‘edit-profile.jsp’ in the folder is responsible for rendering the form for viewing and updating detail information of an user account.

### CHECK EDIT PROFILE FUNCTION

After an client access into ‘…/phone-selling/editProfile’ if user is admin or ‘…/phone-selling/editProfileUser’ if user is client they will post the form page use only login account, after submitting the button “save”

When an client submits the required inputs for a User

* 1. Name
  2. Phone
  3. Address

At here we call the **editProfile(), getUserByEmail(), updateUser()** function which is created in

DAO:

* + The function **editProfile(), getUserByEmail(), updateUser()** function is created in the CateDAO class.

DAOImplement:

* + In the function **getUserByEmail(),** we use SQL: SELECT \* FROM users WHERE email = ?
  + In the function **editProfile(),** we do the validation for checking required inputs in the form
  + In the function **updateUser(),** we use SQL: UPDATE users SET name = ?, email = ?, phone =?, address = ? WHERE id = ?

View:

* ‘edit-profile.jsp’ in the folder “views/admin/users” is responsible for rendering the form for viewing and updating detail information of an admin account.
* ‘edit-profile.jsp’ in the folder “views/client/users” is responsible for rendering the form for viewing and updating detail information of an client.

### CHANGE PASSWORD FUNCTION

After an client access into ‘…/phone-selling/changePassword’ they will get the form page use only change password an account.

1. When an user submits the required inputs for a user information
   * Email
   * Old password
   * New password
   * Confirm new password

### CHECK CHANGE PASSWORD FUNCTION

After an client access into ‘…/phone-selling/checkChangePassword ‘ if user is admin or ‘…/phone-selling/checkChangePasswordUser ‘ if user is client they will post the form page use only login account, after submitting the button “save”

When an client submits the required inputs for a User account

* 1. Email
  2. Old password
  3. New password
  4. Confirm new password

At here we call the **changePassword(), getUserByEmail(), saveChangePassword()** function which is created in

DAO:

* + The **changePassword(), getUserByEmail(), saveChangePassword()** function is created in the CateDAO class.

DAOImplement:

* + In the function **getUserByEmail(),** we use SQL: SELECT \* FROM users WHERE email = ?
  + In the function **changePassword(),** we do the validation for checking required inputs in the change password form
  + In the function **saveChangePassword(),** we use SQL: UPDATE users SET password = ? WHERE id = ?

View:

* ‘change-password.jsp’ in the folder “views/admin/users” is responsible for rendering the form page for changing password an admin account.
* ‘change-password.jsp’ in the folder “views/client/users” is responsible for rendering the form page for changing password a client account.

After successfully changing password, the user will be redirected to the URL ‘ ...phone-selling/ changePassword’ if user is an admin, ‘ ...phone-selling/changePasswordUser’ if user is a client.

### ADD CART FUNCTION

After an client access into ‘…/phone-selling/addtoCart ‘ they will post a product to the cart and store to database

At here we call the **add()** function which is created in

DAO:

* + The **add()** function is created in the CateDAO class.

DAOImplement:

* + In the function **add()** call function **checkItemIsAdded()** to check is item is added to the cart. If the item is added to the cart, call function **save()** to store item to database. If the item is not added to the cart, call function **updateQty()** to update quantity of the item in database.
  + The function **save(),** SQL code: INSERT INTO cart\_items (user\_id, prod\_id, name, qty, price) VALUES (?, ?, ?, ?, ?)
  + The function **updateQty(),** SQL code: UPDATE cart\_items SET qty = ? WHERE prod\_id = ? AND user\_id =?
  + The function **checkItemIsAdded(),** SQL code: SELECT \* FROM cart\_items WHERE user\_id = ? AND prod\_id = ?

View:

* ‘cart-items.jsp’ in the folder “views/client” is responsible for rendering the form page for listing items added
* After Add to cart, it’ll redirect to page at URL: ‘…/phone-selling/info/{prod\_id}’ . This page ‘view.jsp’ in folder ‘views/client/product/’ is responsible for rendering the information of an item.

## HOME CONTROLLER CLASS

### VIEW CART FUNCTION

After an client access into ‘…/phone-selling/views ‘ they will get form page to list items added to cart

At here we call the **getAllItemsByUserID(), getAmount(**) function which is created in

DAO:

* + The **getAllItemsByUserID(), getAmount()** function is created in the CateDAO class.

DAOImplement:

* + The function **getAllItemsByUserID()**, SQL code: SELECT \* FROM cart\_items WHERE user\_id = ?
  + The function **getAmount()**, SQL code: SELECT \* FROM cart\_items WHERE user\_id = ? . Amount is sum of (quantity \* price)

View:

* ‘cart-items.jsp’ in the folder “views/client” is responsible for rendering the form page for listing items added and total amount.

### CLEAR CART FUNCTION

After an client access into ‘…/phone-selling/clearCart ‘ they will delete all products from the cart and from database

At here we call the **clear()** function which is created in

DAO:

* + The **clear()** function is created in the CateDAO class.

DAOImplement:

* + The function **clear(),** SQL code: DELETE FROM cart\_items WHERE user\_id = ?

View:

* After clear all products from the cart, it’ll redirect to page at URL: ‘…/phone-selling/info/{prod\_id}’ . This page ‘view.jsp’ in folder ‘views/client/product/’ is responsible for rendering the information of an item.

### REMOVE CART FUNCTION

After an client access into ‘…/phone-selling/delete{id} ‘ they will delete a product from the cart and from database

At here we call the **remove()** function which is created in

DAO:

* + The **remove()** function is created in the CateDAO class.

DAOImplement:

* + The function **remove(),** SQL code: DELETE FROM cart\_items WHERE prod\_id = ? AND user\_id = ?

View:

* After remove a product from the cart, it’ll redirect to page at URL: ‘…/phone-selling/info/{prod\_id}’ . This page ‘view.jsp’ in folder ‘views/client/product/’ is responsible for rendering the information of an item.

### UPDATE QUANTITY ITEM CART FUNCTION

After an client access into ‘…/phone-selling/update ‘ they will update quantity of a product in the cart and database

At here we call the **updateQty()** function which is created in

DAO:

* + The **updateQty()** function is created in the CateDAO class.

DAOImplement:

* + The function **updateQty()**, SQL code: UPDATE cart\_items SET qty = ? WHERE prod\_id = ? AND user\_id =?

View:

* After remove a product from the cart, it’ll redirect to page at URL: ‘…/phone-selling/info/{prod\_id}’ . This page ‘view.jsp’ in folder ‘views/client/product/’ is responsible for rendering the information of an item.

### CHECK OUT FUNCTION

After an client access into ‘…/phone-selling/checkout ‘ they get a form page of cart include basic information of client and total amount of the cart.

At here we call the **updateQty()** function which is created in

DAO:

* + The **getAmount(), getUserById()** function is created in the CateDAO class.

DAOImplement:

* + The function **getAmount()**, SQL code: SELECT \* FROM cart\_items WHERE user\_id = ?. Amount is sum of (quantity \* price)
  + In the function **getUserById()**, we use SQL: SELECT \* FROM users WHERE id = ?

View:

* ‘check-out.jsp’ in the folder “views/client” is responsible for rendering the form page for basic information of client and total amount of items added

### CONFIRM CHECK OUT FUNCTION

After an client access into ‘…/phone-selling/ confirmCheckout‘ they POST information of cart include basic information of client and total amount of the cart.

When an client submits the required inputs for a basic information and total amount of the cart:

* + Email
  + Name
  + Phone
  + Address
  + Total amount

At here we call the **saveOrder(), saveOrderItem, getAllItemByUserID(), getProductById(), updateProductAfterSelling(), clear()** function which is created in

DAO:

* + The **saveOrder(), saveOrderItem(), getAllItemByUserID(), getProductById(), updateProductAfterSelling(), clear()** function is created in the CateDAO class.

DAOImplement:

* + The function **saveOrder()**, SQL code: INSERT INTO orders (id, user\_id, name, email, phone, address, payment\_mode, status, total\_price) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
  + The function **saveOrderItem()**, SQL code: INSERT INTO order\_details (order\_id, prod\_id, qty, price, created\_at, updated\_at) VALUES (?, ?, ?, ?, ?, ?)
  + The function **getAllItemsByUserID()**, SQL code: SELECT \* FROM cart\_items WHERE user\_id = ?
  + In the function **getProductById()**, we use SQL: SELECT p.\*, c.name AS category\_name FROM products p INNER JOIN categories c ON p.cate\_id = c.id" + " Where p.id = ?
  + In the function **updateProductAfterSelling()**, we use SQL: UPDATE products SET qty = ? WHERE id = ?
  + The function **clear()**, SQL code: DELETE FROM cart\_items WHERE user\_id = ?. We’ll clear cart after an order is created.

View:

* ‘check-out.jsp’ in the folder “views/client” is responsible for rendering the form page for basic information of client and total amount of items added. After an order is created, the page will redirect to URL: ‘…/phone-selling/checkout’

### VIEW HISTORY FUNCTION

After an client access into ‘…/phone-selling/ historyUser‘ they get information of orders.

At here we call the **getAllOrderByUserID()** function which is created in

DAO:

* + The **getAllOrderByUserID()** function is created in the CateDAO class.

DAOImplement:

* + The function **getAllOrderByUserID()**, SQL code: SELECT \* FROM orders WHERE user\_id = ?

View:

* ‘history.jsp’ in the folder “views/client/history” is responsible for rendering the page for basic information of client’s orders.

### VIEW DETAIL HISTORY FUNCTION

After an client access into ‘…/phone-selling/detailHistoryUser/{orderID}‘ they get details information of an order.

At here we call the **getAllOrderByUserID(), getAllOrderItemByOrderID()** function which is created in

DAO:

* + The **getAllOrderByUserID(), getAllOrderItemByOrderID()** function is created in the CateDAO class.

DAOImplement:

* + The function **getAllOrderByUserID()**, SQL code: SELECT \* FROM orders WHERE user\_id = ?
  + The function **getAllOrderItemByOrderID()**, SQL code: SELECT oi.prod\_id, oi.order\_id, oi.id, oi.qty, oi.price, oi.created\_at, products.name AS nameProduct FROM products INNER JOIN order\_details oi ON products.id = oi.prod\_id WHERE oi.order\_id = ?

View:

* ‘history-detail.jsp’ in the folder “views/client/history” is responsible for rendering the page for detail information of a client’s order.

### VIEW ORDERS FUNCTION (ADMIN SITE)

After an client access into ‘…/phone-selling/getAllOrderAdmin ‘ they get information of all client’s orders.

At here we call the **getAllOrder(), getIncome()** function which is created in

DAO:

* + The **getAllOrder(), getIncome()** function is created in the CateDAO class.

DAOImplement:

* + The function **getAllOrder()**, SQL code: SELECT \* FROM orders
  + The function **getIncome()**, SQL code: SELECT FORMAT(CONVERT(DATE, o.created\_at, 103), 'dd/MM/yyyy') AS created\_at, SUM(o.price \* o.qty) as price FROM order\_details o INNER JOIN orders e ON e.id = o.order\_id WHERE e.status = 1 GROUP BY FORMAT(CONVERT(DATE, o.created\_at, 103), 'dd/MM/yyyy');

View:

* ‘history.jsp’ in the folder “views/admin/history” is responsible for rendering the page for basic information of all client’s orders.

### VIEW DETAIL ORDER FUNCTION (ADMIN SITE)

After an client access into ‘…/phone-selling/detailOrderUserByAdmin/{orderID}‘ they get details information of a client’s order.

At here we call the **getAllOrderByUserID(), getAllOrderItemByOrderID()** function which is created in

DAO:

* + The **getAllOrderByUserID(), getAllOrderItemByOrderID()** function is created in the CateDAO class.

DAOImplement:

* + The function **getAllOrderByUserID()**, SQL code: SELECT \* FROM orders WHERE user\_id = ?
  + The function **getAllOrderItemByOrderID()**, SQL code: SELECT oi.prod\_id, oi.order\_id, oi.id, oi.qty, oi.price, oi.created\_at, products.name AS nameProduct FROM products INNER JOIN order\_details oi ON products.id = oi.prod\_id WHERE oi.order\_id = ?

View:

* ‘history-detail.jsp’ in the folder “views/admin/history” is responsible for rendering the page for detail information of a client’s order.

### UPDATE STATUS ORDER FUNCTION

After an client access into ‘…/phone-selling/UpdateStatusOrder/{orderID}‘ they get details information of a client’s order.

At here we call the **saveStatus()** function which is created in

DAO:

* + The **saveStatus()** function is created in the CateDAO class.

DAOImplement:

* + The function **saveStatus()**, SQL code: UPDATE orders SET status = ? WHERE id = ?

View:

* After updating status of a client’s order from “unpaid” to “paid”, it will redirect to the URL: ‘…/phone-selling/getAllOrderAdmin’