|  |
| --- |
| **Git hub Link-**  **https://github.com/jyothiarunkr/Phase-4-Develop-a-Web-Application-using-frontend-stack.git**  **Source Code**  **KITCHEN STORY** |
| **AN E-COMMERCE WEB APPLICATION**  **FOR A SMALL RETAIL FOOD STORE**  **-Dr. Jyothi N.M.** |

# Source code snippet

This section will describe the key implementation processes and some code snippets of this e-commerce web application .kitchen story

The application was structured according to the MVC pattern. This pattern aims to separate the user interface logic from the business logic. Also, this pattern helps to create well defined and organized web applications with efficient code reuse and multiple views.

## Home Page

The home page of the application displays all the available products in the store. In Code Snippet 1, the getProducts helper method is called to retrieve all product data stored in the database. Also, shown in the code snippet is the implementation of the getProducts method, which calls the getEntityManagerFactory static method in order to establish a connection to the database. Then a Java Persistence Query Language (JPQL) is written to query the database to return a list of all available products in the database.

List<Product> products = DBProduct.getProducts(); session.setAttribute("custProducts", products);

// This retrieves product objects from the database.

**public static** List<Product> getProducts() {

EntityManager em = DBUtil.*getEntityManagerFactory*()

.createEntityManager();

String query = "SELECT p FROM Product p ORDER BY p.lastUpdate"; TypedQuery<Product> q = em.createQuery(query, Product.**class**); List<Product> products;

**try** {

products = q.getResultList();

**if** (products == **null** || products.isEmpty()) products = **null**;

} **finally** {

em.close();

}

**return** products;

}

**Code Snippet 1.** Retrieving available products from the database.

## Subscription to Email List

Code Snippet 2 shows the implementation of the subscribeToEmail method. This method retrieves the submitted input data on the subscribe to email form. It then checks that no input parameter is null or empty by calling the checkParam method. It also calls the checkEmail method to ensure that the user enters a valid email address. In order to avoid duplicate email address in the database, the emailExists method is called to check if the entered email address already exists in the database. The addEmailSubscriber method is called to add the entered data into the database if all the input parameters are entered correctly. Lastly, the sendEmail method is called to send an email confirmation to the provided email address and the user is redirected to the confirmation page.

// Insert EmailSubscriber obj into the database.

**private** String subscribeToEmail(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

String firstName = request.getParameter("firstName").trim(); String lastName = request.getParameter("lastName").trim(); String email = request.getParameter("email").trim();

String message = ""; String url = "";

EmailSubscriber emailSubscriber = **new** EmailSubscriber(); emailSubscriber.setFirstName(firstName); emailSubscriber.setLastName(lastName); emailSubscriber.setEmail(email); request.setAttribute("emailSubscriber", emailSubscriber);

**if** (!ValidatorUtil.*checkParam*(firstName)

|| !ValidatorUtil.*checkParam*(lastName)

|| !ValidatorUtil.*checkParam*(email)) { message = "No field can be empty"; request.setAttribute("message", message);

url = "/emailSubscription/index.jsp";

} **else if** (!ValidatorUtil.*checkEmail*(email)) { request.setAttribute("emailError", "email not in the right format"); url = "/emailSubscription/index.jsp";

} **else** {// No two email address can be the same in the database.

**if** (DBEmail.*emailExists*(email)) {

String emailMessage = "This email address already exist. "

+ "<br> Kindly provide another email address."; request.setAttribute("emailMessage", emailMessage);

url = "/emailSubscription/index.jsp";

} **else** {

DBEmail.*addEmailSubscriber*(emailSubscriber);

// Prepare the confirmation email message String recipient = email;

String sender =

**this**.getServletContext().getInitParameter("custServEmailAddress");

String subject = "Email Subscription Confirmation"; String body = "<p>Hello "

+ emailSubscriber.getFirstName()

+ ",</p>\n\n"

+ "<p>Thank you for subscribing to our email list."

+ " We will send you news about our new prod-

ucts and special offers.</p>\n\n"

+ "<p>Sincerely, </p>\n" + "<p>T&T Team</p>\n\n";

**boolean** bodyIsHTML = **true**;

**try** {

// Send email message.

EmailUtil.*sendEmail*(recipient, sender, subject, body,

bodyIsHTML);

} **catch** (MessagingException e) { e.printStackTrace();

}

url = "/emailSubscription/confirmation.jsp";

}

}

**return** url;

}

**Code Snippet 2.** Method for subscribing to the email list.

## Add Product to the Shopping Cart

In Code Snippet 3, the addCartItem method is called when a user clicks the add to cart button on the home page of the application. In this method, a new instance of the Cart

class is created if the retrieved Cart object does not exist in the Session object. Then the addCartItem method of the Cart class is called to add the product to the shopping cart after all the necessary operations and checks. Lastly, the Cart object is saved into the Session object and the user is redirected back to the home page of the application.

// Add product to the shopping cart.

**private** String addCartItem(HttpServletRequest request,

HttpServletResponse response) **throws** IOException { HttpSession session = request.getSession();

Cart cart = (Cart) session.getAttribute("cart");

**if** (cart == **null**) {

cart = **new** Cart();

}

String code = request.getParameter("code");

**if** (!code.isEmpty()) {

Product product = DBProduct.*getProductByCode*(code); cart.addCartItem(product);

}

session.setAttribute("cart", cart);

**return** "/index.jsp";

}

**Code Snippet 3.** Method for adding product to the shopping cart.

## Update Item in the Shopping Cart

The updateCartItem method in Code Snippet 4 is responsible for updating the quantities of items in the shopping cart. The code and quantity parameters are first retrieved from the Request object, and the Cart object retrieved from the Session object. After all the necessary operations and checks, the updateCartItem method of the Cart class is called to perform the necessary updates in the shopping cart, including item quantity.

// Update cart item quantity in the shopping cart.

**private** String updateCartItem(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

String code = request.getParameter("code");

String qtyString = request.getParameter("quantity"); HttpSession session = request.getSession();

Cart cart = (Cart) session.getAttribute("cart");

**int** quantity;

Product product = DBProduct.*getProductByCode*(code);

**try** {

quantity = Integer.*parseInt*(qtyString);

**if** (quantity < 0 || quantity == 0) { quantity = 1;

}

**if** (quantity > product.getQuantity()) { quantity = 1;

request.setAttribute("insuffientQuantity",

"insufficient quantity");

}

} **catch** (NumberFormatException ex) { quantity = 1;

}

**if** (quantity < product.getQuantity()

|| quantity == product.getQuantity()) { cart.updateCartItem(product, quantity);

}

**return** "/cart/cart.jsp";

}

**Code Snippet 4.** Method for updating item quantity in the shopping cart.

## Remove Item from the Shopping Cart

The removeCartItem method in Code Snippet 5 is responsible for removing an item from the shopping cart. After performing all the necessary operations and checks it calls the removeCartItem method of the Cart class to remove the item from the cart.

// Remove cart item from the shopping cart

**private** String removeCartItem(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

String code = request.getParameter("code"); HttpSession session = request.getSession();

Cart cart = (Cart) session.getAttribute("cart"); Product product = DBProduct.*getProductByCode*(code); **if** (product != **null** && cart != **null**) {

cart.removeCartItem(product);

}

**return** "/cart/cart.jsp";

}

**Code Snippet 5.** Method for removing item from the shopping cart.

## Remove all Items from the Shopping Cart

Code Snippet 6 shows the implementation of the clearCart method. It is responsible for removing all items from the shopping cart. It calls the clearCart method of the Cart class to delete all items from the cart after all the necessary operations and checks.

// Remove all items in the cart.

**private** String clearCart(HttpServletRequest request,

HttpServletResponse response) **throws** IOException { HttpSession session = request.getSession();

Cart cart = (Cart) session.getAttribute("cart");

**if** (cart != **null**) {

cart.clearCart();

}

**return** "/cart/cart.jsp";

}

**Code Snippet 6.** Method for removing all items from the shopping cart.

## Payment with PayPal Express Checkout

This e-commerce web store uses PayPal Express Checkout as its payment solution. With this payment solution, the buyer does not enter any information as the buyer’s details can be obtained from PayPal. For the purpose of testing, this application uses a virtual testing environment called PayPal Sandbox, which mimics most of the features of the PayPal production environment. In order to use the Sandbox server, a seller (merchant) and a buyer test accounts were created. Three PayPal Name-Value Pair (NVP) API operations were created to implement the Express Checkout payment solution. The following subsections describe these API operations.

## SetExpressCheckout API Call

Code Snippet 7 shows the method used to implement the SetExpressCheckout API call. This API call is used to initiate the payment transaction. When a buyer clicks the PayPal check out button on the shopping cart page of the application, the callShortcutExpressCheckout method is called. The checkoutDetails parameter of the method contains the request-specific fields retrieved from the clicked button. The returnURL parameter is the page on the e-commerce web store that PayPal redirects to after a successful payment authorization, and the cancelURL is the page on the e- commerce web store that PayPal redirects to if a buyer cancels the payment. From the method implementation, the fields in the checkoutDetails parameter are used to construct an NVP string, and then an HTTP POST request is sent to PayPal API server by calling the httpCall method, which makes the SetExpressCheckout API call. Part of the response to the API call is an acknowledgment status, which indicates a success or failure with or without warning messages. Also, a token is returned as part of the response. The token is a unique string used for identifying each transaction and in making other API calls.

//Method for implementing the SetExpressCheckout API call.

**public** Map<String, String> callShortcutExpressCheckout(

Map<String, String> checkoutDetails, String returnURL, String cancelURL) {

// Construct the parameter string that describes the

// SetExpressCheckout API call

StringBuilder nvpstr = **new** StringBuilder("");

//Append the line item parameters to the nvpstr variable.

**for** (String key : checkoutDetails.keySet()) {

**if** (key.startsWith("L")) {

nvpstr.append("&" + key + "=").append(checkoutDetails.get(key));

}

}

**if** (isSet(checkoutDetails.get("PAYMENTREQUEST\_0\_AMT"))) { nvpstr.append("&PAYMENTREQUEST\_0\_AMT=").append(

checkoutDetails.get("PAYMENTREQUEST\_0\_AMT"));

}

**if** (isSet(checkoutDetails.get("paymentType"))) { nvpstr.append("&PAYMENTREQUEST\_0\_PAYMENTACTION=").append(

checkoutDetails.get("paymentType"));

}

**if** (isSet(returnURL))

nvpstr.append("&RETURNURL=").append(returnURL);

**if** (isSet(cancelURL))

nvpstr.append("&CANCELURL=").append(cancelURL);

**if** (isSet(checkoutDetails.get("currencyCodeType"))) { nvpstr.append("&PAYMENTREQUEST\_0\_CURRENCYCODE=").append(

checkoutDetails.get("currencyCodeType"));

}

**if** (isSet(checkoutDetails.get("PAYMENTREQUEST\_0\_ITEMAMT"))) { nvpstr.append("&PAYMENTREQUEST\_0\_ITEMAMT=").append(

checkoutDetails.get("PAYMENTREQUEST\_0\_ITEMAMT"));

}

**if**(isSet(checkoutDetails.get("LOGOIMG")))

nvpstr.append( "&LOGOIMG="+ checkoutDetails.get("LOGOIMG"));

//Make the API call.

**return** httpCall("SetExpressCheckout", nvpstr.toString());

}

**Code Snippet 7.** Method for SetExpressCheckout API call.

## GetExpressCheckoutDetails API Call

In Code Snippet 8, the getShippingDetails implements the GetExpressCheckoutDetails API call. In the method, an NVP string is constructed by using the token returned from the SetExpressCheckout API call. Then, a call is made to the GetExpressCheckoutDetails API by calling the httpCall method. The response of this API call contains the buyer’s details and an acknowledgment status, which indicates a success or a failure with or without warning messages.

//Method for implementing GetExpressCheckoutDetails API call.

**public** Map<String, String> getShippingDetails(String token) {

//Construct the nvp string

String nvpstr = "&TOKEN=" + token;

//Make the API call.

**return** httpCall("GetExpressCheckoutDetails", nvpstr);

}

**Code Snippet 8.** Method for GetExpressCheckoutDetails API call.

## DoExpressCheckoutPayment API Call

As shown in Code Snippet 9, the confirmPayment method implements the DoExpressCheckoutPayment API call, which completes the payment transaction. In this method, the NVP string is constructed with the necessary request-specific fields in the checkoutDetails parameter of the method. Then the API call is made by calling the httpCall method. The response from this API contains a success or a failure with or without warning messages, and other transaction specific information like transaction id.

//Method implementation for DoExpressCheckoutPayment API call.

**public** HashMap<String, String> confirmPayment(

Map<String, String> checkoutDetails, String serverName) {

// Construct the NVP string

String finalPaymentAmount = encode(checkoutDetails

.get("PAYMENTREQUEST\_0\_AMT"));

StringBuilder nvpstr = **new** StringBuilder("");

// mandatory parameters in DoExpressCheckoutPayment call

**if** (isSet(checkoutDetails.get("TOKEN"))) nvpstr.append("&TOKEN=").append(

encode(checkoutDetails.get("TOKEN")));

**if** (isSet(checkoutDetails.get("payer\_id"))) nvpstr.append("&PAYERID=").append(

encode(checkoutDetails.get("payer\_id"))); nvpstr.append("&PAYMENTREQUEST\_0\_SELLERPAYPALACCOUNTID=").append(

**this**.getSellerEmail());

**if** (isSet(checkoutDetails.get("paymentType"))) nvpstr.append("&PAYMENTREQUEST\_0\_PAYMENTACTION=").append(

encode(checkoutDetails.get("paymentType")));

**if** (isSet(serverName))

nvpstr.append("&IPADDRESS=").append(encode(serverName)); nvpstr.append("&PAYMENTREQUEST\_0\_AMT=").append(finalPaymentAmount);

// Check for additional parameters that can be passed in

// DoExpressCheckoutPayment API call

**if** (isSet(checkoutDetails.get("currencyCodeType"))) nvpstr.append("&PAYMENTREQUEST\_0\_CURRENCYCODE=").append(

encode(checkoutDetails.get("currencyCodeType").toString()));

**if** (isSet(checkoutDetails.get("PAYMENTREQUEST\_0\_ITEMAMT"))) nvpstr.append("&PAYMENTREQUEST\_0\_ITEMAMT=").append(

encode(checkoutDetails.get("PAYMENTREQUEST\_0\_ITEMAMT")

.toString()));

**if** (isSet(checkoutDetails.get("PAYMENTREQUEST\_0\_TAXAMT"))) nvpstr.append("&PAYMENTREQUEST\_0\_TAXAMT=").append(

encode(checkoutDetails.get("PAYMENTREQUEST\_0\_TAXAMT")

.toString()));

**if** (isSet(checkoutDetails.get("shippingAmt"))) nvpstr.append("&PAYMENTREQUEST\_0\_SHIPPINGAMT=").append(

encode(checkoutDetails.get("PAYMENTREQUEST\_0\_SHIPPINGAMT")

.toString()));

**if** (isSet(checkoutDetails.get("handlingAmt"))) nvpstr.append("&PAYMENTREQUEST\_0\_HANDLINGAMT=").append(

encode(checkoutDetails.get("PAYMENTREQUEST\_0\_HANDLINGAMT")

.toString()));

**if** (isSet(checkoutDetails.get("shippingDiscAmt"))) nvpstr.append("&PAYMENTREQUEST\_0\_SHIPDISCAMT=").append(

encode(checkoutDetails.get("PAYMENTREQUEST\_0\_SHIPDISCAMT")

.toString()));

**if** (isSet(checkoutDetails.get("insuranceAmt"))) nvpstr.append("&PAYMENTREQUEST\_0\_INSURANCEAMT=").append(

encode(checkoutDetails.get("PAYMENTREQUEST\_0\_INSURANCEAMT")

.toString()));

//Make the API call.

**return** httpCall("DoExpressCheckoutPayment", nvpstr.toString());

}

**Code Snippet 9.** Method for DoExpressCheckoutPayment API call.

## Sending Email

Code Snippet 10 shows the method implementation for sending an email confirmation to users after completing an order or subscribing to an email list. The method uses methods and classes from the JavaMail API to automatically send email notifications.

//Method for sending email messages.

**public static void** sendEmail(String recipient, String sender, String subject, String body, **boolean** bodyIsHTML) **throws** MessagingException {

// Get a mail session

Properties properties = **new** Properties(); properties.put("mail.transport.protocol", "smtp"); properties.put("mail.smtps.host", "localhost"); properties.put("mail.smtps.port", 25);

Session session = Session.*getDefaultInstance*(properties); session.setDebug(**true**);

// Create a message

Message message = **new** MimeMessage(session); message.setSubject(subject);

**if** (bodyIsHTML) {

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

} **else** {

}

message.setText(body);

// Here we set the addresses

Address fromAddress = **new** InternetAddress(sender); Address toAddress = **new** InternetAddress(recipient); message.setRecipient(Message.RecipientType.***TO***, toAddress); message.setFrom(fromAddress);

Transport.*send*(message);

}

**Code Snippet 10.** Method for sending an email.

## Admin Login

The admin login process was implemented by the form-based authentication method provided by Apache Tomcat container. Code Snippet 11 shows the authentication configuration. As can be seen from the Code Snippet, all web resources in the admin directory of the application are restricted and a user with manager role is being granted authorization to access the restricted resources. Lastly, customized login and error pages are defined for the authentication.

<security-role>

<description>owner</description>

<role-name>manager</role-name>

</security-role>

<security-constraint>

<!-- Restrict access to the URLs in the admin directory -->

<web-resource-collection>

<web-resource-name>Admin</web-resource-name>

<url-pattern>/admin/\*</url-pattern>

</web-resource-collection>

<!-- Authorize the service and programmer roles -->

<auth-constraint>

<role-name>manager</role-name>

</auth-constraint>

</security-constraint>

<!-- Use form-based authentication to provide access -->

<login-config>

<auth-method>FORM</auth-method>

<form-login-config>

<form-login-page>/login.jsp</form-login-page>

<form-error-page>/login\_error.jsp</form-error-page>

</form-login-config>

</login-config>

**Code Snippet 11.** Form-based authentication configuration for admin login.

## Add Category

The method for adding category data into the database is shown in Code Snippet 12. The method responds to the request sent when an authorized user clicks the add category button on the add category form. The validity of the input data is checked by calling the checkParm method. An appropriate error message is sent whenever something goes wrong otherwise, the category data is inserted into the database by calling the addCategory method.

// Insert Category obj into the database.

**private** String addCategory(HttpServletRequest request,

HttpServletResponse response) **throws** IOException { String name = (request.getParameter("name")).trim();

Date date = **new** Date();

Category addCategory = **new** Category(); addCategory.setName(name); addCategory.setLastUpdate(date); request.setAttribute("category", addCategory); String message = "";

String url = "";

**if** (!ValidatorUtil.*checkParam*(name)) { message = "Field cannot be empty.";

request.setAttribute("message", message); url = "/admin/addCategory.jsp";

} **else** {

// No two categories can have the same name in the database.

**if** (DBCategory.*categoryNameExists*(name)) {

message = "This name already exists for another category. "

+ "<br> Kindly provide another name."; request.setAttribute("message", message);

url = "/admin/addCategory.jsp";

} **else** {

}

}

DBCategory.*addCategory*(addCategory); url = getCategories(request, response);

**return** url;

}

**Code Snippet 12.** Method for adding category data into the database.

## Add Product

Code Snippet 13 shows the method for adding product data into the database. The method responds to the request sent when an admin user clicks the add product button on the add product form. The method checks the validity of the entered data by calling the checkParam method. An appropriate error message is sent whenever something goes wrong otherwise, the product data is added into the database by calling the addProduct method.

// Insert Product obj into the database.

**private** String addProduct(HttpServletRequest request,

HttpServletResponse response) **throws** IOException { String code = request.getParameter("code").trim();

String description = request.getParameter("description").trim(); **double** price = Double.*parseDouble*(request.getParameter("price")); **byte**[] image = uploadFile(request, response);

**int** quantity = Integer.*parseInt*(request.getParameter("quantity")); String categoryName = request.getParameter("categoryName"); Category category = DBCategory.*getCategoryByName*(categoryName); Product product = **new** Product();

product.setCode(code); product.setDescription(description); product.setPrice(price); product.setImage(image); product.setQuantity(quantity); product.setCategory(category); product.setLastUpdate(**new** Date()); request.setAttribute("product", product); String message = "";

String url = "";

**if** (!ValidatorUtil.*checkParam*(code)

|| !ValidatorUtil.*checkParam*(description)) { message = "No field can be empty"; request.setAttribute("message", message);

url = "/admin/addProduct.jsp";

} **else** {

// No two product can have the same code.

**if** (DBProduct.*productCodeExists*(code)) {

message = "This code already exist for another product. "

+ "<br> Kindly provide another code."; request.setAttribute("message", message);

url = "/admin/addProduct.jsp";

}// A product must belong to only one existing category in the

// database.

**else if** (category == **null**) {

message = "The selected category no longer exist. "

+ "<br> Kindly select another category."; request.setAttribute("message", message);

url = "/admin/addProduct.jsp";

} **else** {

}

}

**return** url;

}

DBProduct.*addProduct*(product);

url = getProducts(request, response);

**Code Snippet 13.** Method for adding product data into the database.

## Download Reports in Excel Format

One of the methods for downloading reports in Microsoft excel format is shown in Code Snippet 14. The method uses classes and methods from the Apache POI API to create the email subscribers report in excel format.

// This creates an excel workbook of all email subscribers.

**public static** Workbook getEmailSubscribersReport() {

List<EmailSubscriber> emailSubscribers = DBEmail.*getEmailSubscribers*();

// create a workbook, sheet and its title XSSFWorkbook workbook = **new** XSSFWorkbook();

XSSFSheet sheet = workbook.createSheet("Email Subscribers Report"); XSSFRow row = sheet.createRow(0); row.createCell(0).setCellValue("The Email Subscribers Report");

// create row headers

row = sheet.createRow(2); row.createCell(0).setCellValue("LastName"); row.createCell(1).setCellValue("FirstName"); row.createCell(2).setCellValue("Email");

// create data rows

**int** i = 3;

**for** (EmailSubscriber e : emailSubscribers) { row = sheet.createRow(i);

row.createCell(0).setCellValue(e.getLastName()); row.createCell(1).setCellValue(e.getFirstName()); row.createCell(2).setCellValue(e.getEmail()); i++;

}

**return** workbook;

}

**Code Snippet 14.** Method for downloading reports in excel format.