LS/EECS

Building E-commerce Applications LAB 2

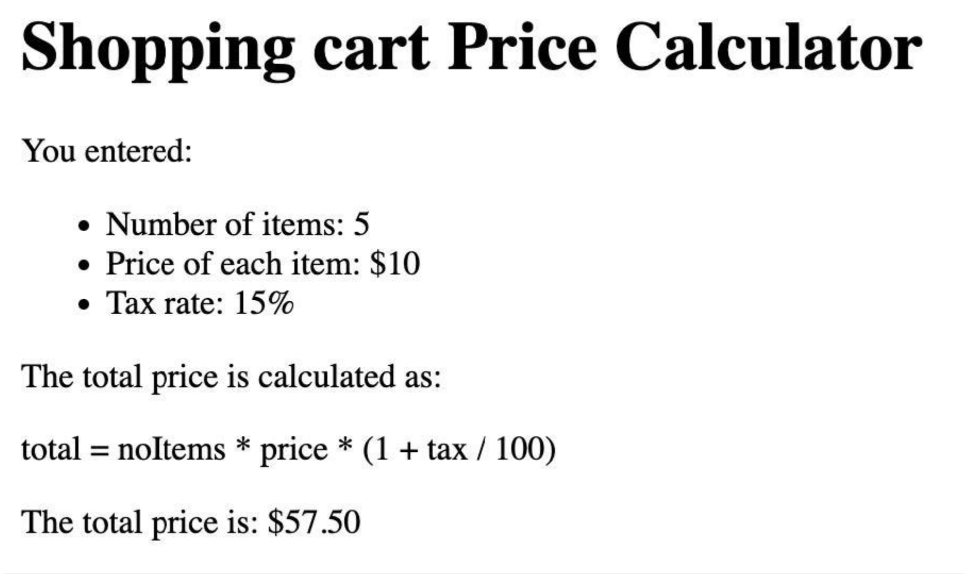
Objectives

By the end of this lab, students should be able to:

- Understand the core components of the Model-View-Controller (MVC) design pattern and their roles in web application development.
- Implement a View layer using HTML and JavaScript for client-side validations.
- Create a Model layer in Java to handle business logic and server-side validations.
- Integrate the View and Model layers through a Controller in a servlet-based web application.
- Test server-side validations using Curl commands.

Before You Start this Lab

Make sure your Shopping Cart Price Calculator project is working correctly. It should allow the client to provide the needed parameters through the query string and it should render calculation results. No need for any validation at this stage. Do not proceed with the next step until you test this functionality:



Shopping cart Price Calculator

You entered:

- Number of items: 5
- Price of each item: $10
- Tax rate: 15%

The total price is calculated as:

total = noItems * price * (1 + tax / 100)

The total price is: $57.50

Export your project (make sure you include the source files in the export) to a “WAR” file LastName_v2.war on the desktop (WAR stands for web archive and it is a standard way of archiving Java EE applications.

Import the war file just created to a new project named LastName_v3.

Introduction

In this lab, we will introduce the Model-View-Controller (MVC) pattern, a design pattern commonly used in web development. The MVC pattern separates an application into three main components:

1. **Model**: The business logic and data manipulation layer. It communicates to the database and updates the View whenever the data changes.
2. **View**: The user interface layer. This is what the end-user interacts with. It displays data from the Model to the user and sends commands to the Controller.
3. **Controller**: The layer that handles user input. It takes the user's requests, processes them (with possible updates to the Model), and returns the output display to the View.

By separating the concerns into different components, the MVC pattern provides a more organized and modular approach to web application development.

Task 1: Add an HTML page (View)

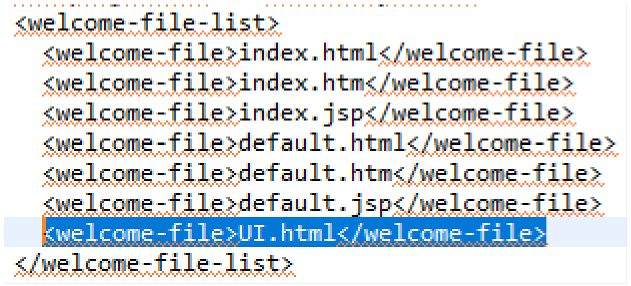
1. Create a new HTML page named `UI.html`.
2. This page will serve as the user interface and should contain form elements that allow the user to input the number of items, price per item, and tax rate.
3. The form should have a submit button that, when clicked, will invoke the APIs you developed in the previous lab.

```
<!DOCTYPE html>
<html>
<head>
    <title>Shopping Cart Price Calculator</title>
</head>
<body>
    <header>
        <h1>Shopping Cart Calculator</h1>
    </header>

    <main>
        <form id="cartForm">
            <fieldset>
                <legend>Enter Shopping Cart Details</legend>
                Number of Items: <input type="text" id="noItems"
name="noItems"><br>
                Price per Item: <input type="text" id="price" name="price" ><br>
                Tax Rate: <input type="text" id="tax" name="tax" ><br>
                <input type="submit" value="Calculate">
            </fieldset>
        </form>
    </main>

    <footer>
        <p>Created for E-Commerce Lab</p>
    </footer>
</body>
</html>
```

4. To invoke the CalculatorServlet you created in the previous lab, **add the Servlet name and the post method i**n the <form> in your UI.html

```
<form action="CalculatorServlet" method="post" id="cartForm" >
```

5. Add UI.html into the <welcome-file-list> in the web.xml



```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
  <welcome-file>UI.html</welcome-file>
</welcome-file-list>
```

Task 2: Add an External CSS File

1. Create a new external CSS file named styles.css.
2. Add styles to make your HTML form visually appealing.
3. Link the `styles.css` file to your HTML file.

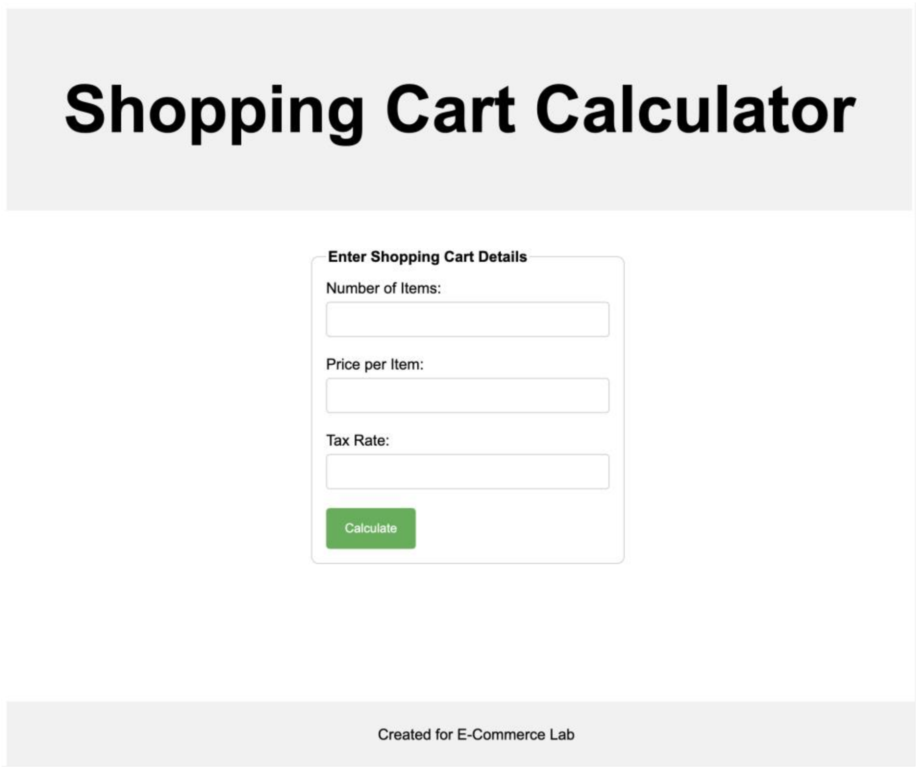
   Here's how you link the CSS file in your HTML in the <head></head>:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

And a sample `styles.css`:

```
body {
    font-family: Arial, sans-serif;
}


form {
    margin: 20px auto;
    padding: 20px;
    width:320px;
}


input[type="text"] {
    width: 280px;
    padding: 10px;
    margin: 5px 0 20px 0;
    border: 1px solid #ccc;
    border-radius: 4px;
}


input[type="submit"] {
    background-color: #4CAF50;
    color: white;
    padding: 14px 20px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
}


input[type="submit"]:hover {
    background-color: #45a049;
}

header {
    background-color: #f1f1f1;
    padding: 20px;
    text-align: center;
    font-size: 35px;
}

footer {
    background-color: #f1f1f1;
    padding: 10px;
    text-align: center;
    position: fixed;
    width: 100%;
    bottom: 0;
}

fieldset {
    border: 1px solid #ccc;
    border-radius: 8px;
    padding: 15px;
    margin-bottom: 20px;
}

legend {
    font-weight: bold;
}
.calculator-results {
      margin: 20px auto;

      padding: 20px;

      width:320px;
}
```

6. When you now run the dynamic web application on http://localhost:8080/<LastName>_Lab2/, you should now see your webpage:



Task 3: Create a Java Class (Model)

1. Create a **new Java class** named `Cart.java`.
2. Move all the shopping cart calculation logic from `CalculatorServlet` to this new Model class.
3. The `Cart.java` class should have methods to calculate the total price, including tax, based on the number of items, price per item, and tax rate.

```
public class Cart {
    public static double calculateTotal(int noItems, double price, double
tax) {
        // Perform the calculation
        double total = noItems * price * (1 + tax / 100);
        return total;
    }
}
```

Task 4: Add Server-side Validations

1. Add server-side validations in the Cart.java class.
2. Validate the following:
   - The number of items should be a positive integer.
   - The price per item should be a positive number.
3. The tax rate should be a positive number and less than or equal to 100.

```
public class Cart {
    public static String validateInput(int noItems, double price, double
tax) {
        if (noItems <= 0) return "Number of items must be a positive
integer.";
        if (price <= 0) return "Price must be a positive number.";
        if (tax < 0 || tax > 100) return "Tax rate must be between 0 and
100.";
        return null;
    }

    public static double calculateTotal(int noItems, double price, double
tax) {
        String validationError = validateInput(noItems, price, tax);
        if (validationError != null) {
            throw new IllegalArgumentException(validationError);
        }
        double total = noItems * price * (1 + tax / 100);
        return total;
    }
}
```

Task 5: Add Client-side Validations(Optional)

1. Create a new JavaScript file named `validations.js`.
2. Modify the HTML code to include a link to the `validations.js` file:

```
<!DOCTYPE html>
<html>
<head>
    <title>UI for Shopping Cart</title>
    <script src="validations.js"></script> <!-- Link to external JavaScript
file -->
</head>
<body>
    <form action="CalculatorServlet" method="post" id="cartForm" >
        <!-- ... -->
    </form>
</body>
</html>
```

3. Implement the same set of validations as you did on the server-side.

```
document.addEventListener('DOMContentLoaded', function() {
    document.getElementById('cartForm').addEventListener('submit',
function(event) {
        event.preventDefault();
        var noItems = parseInt(document.getElementById('noItems').value);
        var price = parseFloat(document.getElementById('price').value);
        var tax = parseFloat(document.getElementById('tax').value);

        if (noItems <= 0) {
            alert("Number of items must be a positive integer.");
            return;
        }
```

```
                if (price <= 0) {
                    alert("Price must be a positive number.");
                    return;
                }
                if (tax < 0 || tax > 100) {
                    alert("Tax rate must be between 0 and 100.");
                    return;
                }

                // If validation passes, call the server API
        event.currentTarget.submit();
            });
        });
```

4. If the validation fails, display an appropriate error message on the UI.

You can also add validation while using the EventListener on the form fields instead of sending out an alert. Let's try it for the Tax field.
On the styles.css, add:

```
.error {
width: 100%;
padding: 10px;
}

.error.active {
margin: 20px auto;
font-size: 80%;
color: white;
background-color: #900;
border-radius: 0 0 5px 5px;
box-sizing: border-box;
}
```

Replace the HTML for the Tax Rate field and add a label and a span that references the css class:

```
<label for="tax">
Tax Rate: <input type="text" name="tax" id="tax">
<span class="error" aria-live="polite"></span>
</label>
```

On the validations.js, add another listener for your input fields **nested** inside the "*document.addEventListener('DOMContentLoaded', function() {* " line :

```
const taxError = document.querySelector("#tax + span.error");
document.getElementById('tax').addEventListener("input", (event) => {

// Each time the user types something, we check if the
// form fields are valid.
var tax = parseFloat(document.getElementById('tax').value);
if (tax < 0 || tax > 100) {
// In case there is an error message visible, if the field
taxError.textContent = "Tax rate must be between 0 and 100.";
taxError.className = "error active";
} else {
taxError.textContent = ""; // Reset the content of the message
taxError.className = "error"; // Reset the visual state of the message
}
});
```

You can also use HTML5 to add validation. To prevent empty fields, add “required” to all your <input> tags, for example:

```
Number of Items: <input type="text" name="noItems" id="noItems" required >
```

For further reading on client-side validation, look at: https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation

Task 6: Create the results view

1. Create a new JSP file named `ShoppingCartCalculator.jsp`.
2. In this JSP file, use the attributes set in the Servlet to display the number of items, price per item, tax rate, and the calculated total price.

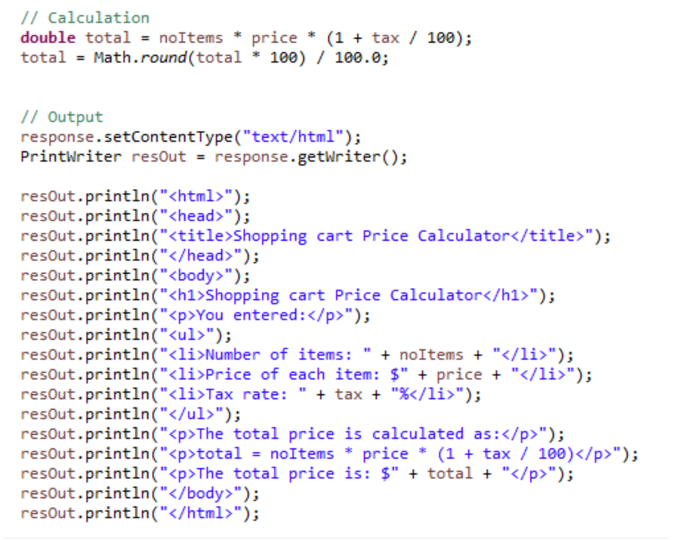
   Here's a sample code snippet for the JSP file:

```
        <%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
        <!DOCTYPE html>
        <html>
        <head>
            <title>Shopping cart Price Calculator</title>
            <link rel="stylesheet" type="text/css" href="styles.css">
        </head>
        <body>
<header>
                     <h1>Shopping Cart Calculator</h1>
              </header>
<main>
<div class="calculator-results">
                     <fieldset>
              <legend>Shopping Cart</legend>
              <p>You entered:</p>
              <ul>
              <li>Number of items: <%= request.getAttribute("noItems") %></li>
              <li>Price of each item: $<%= request.getAttribute("price") %></li>
              <li>Tax rate: <%= request.getAttribute("tax") %> %</li>
               </ul>
              <p>The total price is calculated as:</p>
              <p>total = noItems * price * (1 + tax / 100)</p>
              <p>The total price is: $<%= request.getAttribute("total") %></p>
              </fieldset>
              </div>
       </main>
       <footer>
              <p>Created for E-Commerce Lab</p>
       </footer>
       </body>
       </html>
```

Task 7: Modify the Servlet

1. Open your existing Servlet file, presumably named `ShoppingCartServlet.java`.
2. Modify the `doGet` method to get the parameter values for the number of items, price per item, and tax rate from the request.
3. Use these parameters to call the `calculateTotal` method from your Model class (`Cart.java`) to perform the calculation.
4. Set the calculated total and input parameters as attributes in the request object.
5. Forward the request to a JSP file named `ShoppingCartCalculator.jsp` for rendering the output.
6. If you finished the previous lab, remove the resOut.println ouput block and the Calculation.

Remove this block of code:



```
// Calculation
double total = noItems * price * (1 + tax / 100);
total = Math.round(total * 100) / 100.0;


// Output
response.setContentType("text/html");
PrintWriter resOut = response.getWriter();

resOut.println("<html>");
resOut.println("<head>");
resOut.println("<title>Shopping cart Price Calculator</title>");
resOut.println("</head>");
resOut.println("<body>");
resOut.println("<h1>Shopping cart Price Calculator</h1>");
resOut.println("<p>You entered:</p>");
resOut.println("<ul>");
resOut.println("<li>Number of items: " + noItems + "</li>");
resOut.println("<li>Price of each item: $" + price + "</li>");
resOut.println("<li>Tax rate: " + tax + "%</li>");
resOut.println("</ul>");
resOut.println("<p>The total price is calculated as:</p>");
resOut.println("<p>total = noItems * price * (1 + tax / 100)</p>");
resOut.println("<p>The total price is: $" + total + "</p>");
resOut.println("</body>");
resOut.println("</html>");
```

7. Replace the calculation with:

```
double total = Cart.calculateTotal(noItems, price, tax);
```

8. Set the attributes from the UI.html request to pass to the JSP, and forward the request to the JSP file at the end of doGet:

```
request.setAttribute("noItems", noItems);
request.setAttribute("price", price);
request.setAttribute("tax", tax);
request.setAttribute("total", total);
```

```
RequestDispatcher dispatcher =
request.getRequestDispatcher("/ShoppingCartCalculator.jsp");
dispatcher.forward(request, response);
```

Here's a sample code snippet for the Servlet **(Note: This is just a sample without the default and session values created in the previous lab)**:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;
public class CalculatorServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        // Get parameters from the request
        int noItems = Integer.parseInt(request.getParameter("noItems"));
        double price = Double.parseDouble(request.getParameter("price"));
        double tax = Double.parseDouble(request.getParameter("tax"));

        // Use the Model class to perform the calculation
        double total = Cart.calculateTotal(noItems, price, tax);

        // Set attributes to pass to the JSP
        request.setAttribute("noItems", noItems);
        request.setAttribute("price", price);
        request.setAttribute("tax", tax);
        request.setAttribute("total", total);

        // Forward the request to the JSP file
        RequestDispatcher dispatcher =
request.getRequestDispatcher("/ShoppingCartCalculator.jsp");
        dispatcher.forward(request, response);
    }
}
```

Testing

After completing the above tasks, you can test the server-side validations using Curl commands. Here are some example Curl commands to test the different scenarios:

1. Valid Input
   ```
   curl
   "http://localhost:8080/YourApp/CalculatorServlet?noItems=5&price=10&tax=15"
   ```
2. Invalid Number of Items
   ```
   curl "http://localhost:8080/YourApp/CalculatorServlet?noItems=-
   5&price=10&tax=15"
   ```
3. Invalid Price
   ```
   curl "http://localhost:8080/YourApp/CalculatorServlet?noItems=5&price=-
   10&tax=15"
   ```
4. Invalid Tax Rate
   ```
   curl
   ```

```
"http://localhost:8080/YourApp/CalculatorServlet?noItems=5&price=10&tax=150
"
```