Low Level Design

Shopping Cart

Document Control

Change Record:

Version	Date	Author	Comments
0.1	30-11-2024	Febin Anto K K	Initial Architecture Draft
0.2	2-12-2024	Akshaya S S	Added component diagram
0.3	4-12-2024	Sham S	Updated data flow and state management
0.4	6-12-2024	Castro R S Jeev	Refined API integration architecture
0.5	9-12-2024	Jacob Daniel R	Added deployment flow and cloud architecture
0.6	12-12-2024	Febin Anto K K	Final review and documentation updates

Contents

1. Introduction

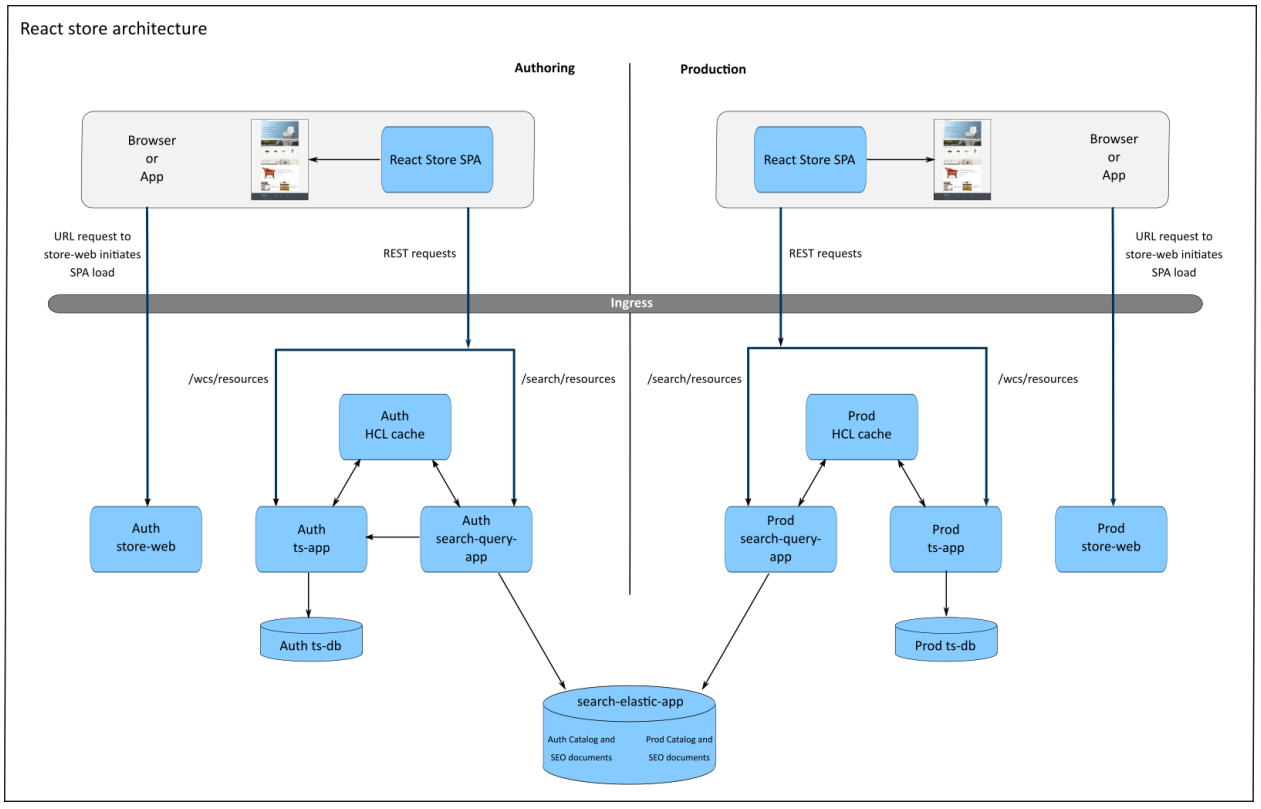
1.1. What is Low-Level design document?

A Low-Level Design (LLD) document is a detailed technical blueprint that describes the internal structure and functionality of a system. It focuses on the implementation details, breaking down the high-level architecture into specific components, modules, and workflows.

1.2. Scope

The Shopping Cart project is a web application developed using React, Redux, Tailwind CSS, and AOS (Animate on Scroll). The goal of the project is to provide a seamless user experience for browsing, selecting, and purchasing products. This document outlines the low-level design scope, covering component structure, state management, styling, and animations.

2. Architecture



3. Architecture Description

3.1 Component-Based Architecture

- **React** is used to build a modular, reusable component structure.
    - Parent components (e.g., App, Header, Footer) encapsulate child components (e.g., ProductCard, CartItem).
    - Components are organized by feature or purpose (e.g., components/Header, components/ProductListing).
- Use React Context API for passing props where Redux state is unnecessary (e.g., theme or locale).

3.2 State Management

- **Redux** is implemented to manage the application's state.
    - The store is structured as follows:
        - **Products**: Contains all available products fetched from the API.
        - **Cart**: Tracks items added to the cart, quantities, and total price.
        - **User**: Manages user login state and preferences (if applicable).
    - Actions are defined for:
        - Adding, removing, and updating items in the cart.
        - Fetching products and handling API errors.
- **Selectors** are used for optimizing state retrieval in components.

3.3 Styling Architecture

- **Tailwind CSS** utility-first framework is used for styling.
    - Components adhere to a consistent design system defined in the Tailwind configuration file.
    - Use custom classes (via @apply) for reusable styles.
    - Responsive design is achieved with Tailwind's grid and breakpoint utilities.

3.4 Animation Architecture

- **AOS (Animate on Scroll)** enhances the user interface by adding scroll-triggered animations.
    - Configurations are set globally (e.g., duration, easing) and overridden locally for specific elements.
    - Animations include fade-in, zoom, and slide effects on components like product cards and headers.

3.5 API Layer Integration

- API interactions are abstracted in a separate api directory with utility functions for:
    - Fetching products.
    - Submitting orders.
    - User authentication (if applicable).
- Axios or Fetch API is used, with error handling and retry mechanisms implemented.

3.6 Testing and Debugging Tools

- **Jest** and **React Testing Library** for unit tests.
- **Redux DevTools** for monitoring and debugging state changes.
- **AOS Debugging**: Ensure animations render correctly in the DOM with appropriate data-aos attributes.

4. Unit Test Cases

1. Add Item to Cart

- **Test Case Name**: should add an item to the cart
- **Description**: Verify that an item is successfully added to the cart.
- **Input**: Product ID, Quantity.
- **Expected Output**: The cart contains the new item with the correct quantity.

2. Update Item Quantity

- **Test Case Name**: should update item quantity in the cart
- **Description**: Ensure that the quantity of an existing item in the cart is updated correctly.
- **Input**: Product ID, Updated Quantity.
- **Expected Output**: The item's quantity in the cart is updated.

3. Remove Item from Cart

- **Test Case Name**: should remove an item from the cart
- **Description**: Verify that an item can be removed from the cart.
- **Input**: Product ID.
- **Expected Output**: The cart no longer contains the removed item.

4. Calculate Total Price

- **Test Case Name**: should calculate total price of items in the cart
- **Description**: Ensure that the total price is calculated correctly based on item prices and quantities.
- **Input**: List of items with prices and quantities.
- **Expected Output**: Correct total price.

5. Handle Empty Cart

- **Test Case Name**: should handle operations on an empty cart
- **Description**: Ensure that the cart behaves as expected when it is empty.
- **Input**: None.
- **Expected Output**: Cart operations return appropriate responses (e.g., empty cart message, no total price).

6. Prevent Adding Duplicate Items

- **Test Case Name**: should not add duplicate items to the cart
- **Description**: Verify that adding an item that already exists updates the quantity instead of creating a duplicate entry.
- **Input**: Product ID already in the cart.
- **Expected Output**: Item quantity is updated; no duplicate entry.

7. Validate Checkout Process

- **Test Case Name**: should validate cart before checkout
- **Description**: Ensure that the checkout process only proceeds when the cart contains valid items.
- **Input**: Empty or invalid cart.
- **Expected Output**: Prevent checkout and display error message.

8. Persist Cart State

- **Test Case Name**: should persist cart state across sessions
- **Description**: Verify that the cart's state is saved and restored correctly.
- **Input**: Add items to cart, reload page.
- **Expected Output**: Cart state remains consistent after reload.

9. Validate User Input

- **Test Case Name**: should validate user input when adding items
- **Description**: Ensure that invalid inputs (e.g., negative quantities) are handled appropriately.
- **Input**: Negative or invalid quantity.
- **Expected Output**: Display an error message; item not added.

10. Integration with Payment Gateway

- **Test Case Name**: should process payment only for valid cart
- **Description**: Ensure that payment processing works correctly for a valid cart and is blocked for invalid carts.
- **Input**: Valid and invalid carts.
- **Expected Output**: Payment proceeds for valid cart; error for invalid cart.

Example Code for Unit Tests (React + Jest)

Here's an example test case for the "Add Item to Cart" functionality:

```
import { render, screen, fireEvent } from '@testing-library/react';
import Cart from './Cart';

test('should add an item to the cart', () => {
  render(<Cart />);
  const addButton = screen.getByText('Add to Cart');
  fireEvent.click(addButton);
  const cartItems = screen.getByTestId('cart-items');
  expect(cartItems).toHaveTextContent('Product Name');
});
```