

Day 5 - Testing, Error Handling, and Backend Integration

Refinement

Objective:

On Day 5, the goal is to prepare my marketplace for live deployment by conducting comprehensive testing of all components, fine-tuning performance, and ensuring it can efficiently handle customer traffic. The primary focus will be on validating backend integrations, strengthening error handling mechanisms, and improving the user experience.

1. Functional Testing:

Here's a detailed overview of how each feature working as expected after functional testing.

1. Product Listing:

Test: I loaded the homepage and navigate to product listing pages. Verified all products are displayed correctly, including product names, images, prices, and tags.

Expected Result:

- All products appear without missing information.
- Images load properly across devices without distortion or delays.
- Products are displayed in the correct layout (e.g., grid or list view) as per the design.
- Pagination works correctly, and navigating between pages shows appropriate products.

Test Outcome: ✓ Products are displayed as intended. No issues with missing data, images, or pagination.

2. Filters and Search:

Filter:

Test: I applied filters (e.g. price range and alphabet order).

Expected Result:

- Products displayed match the selected filter criteria.
- I noticed my filter component was not working accurately so I just fixed it.

Test Outcome: ✓ Filter working perfectly, and no irrelevant products are displayed.

Search:

Test: I Searched for products using different keywords (e.g., "Running Shoes," "Nike Air Max"), User can find products by giving product name, category and tag.

Expected Result:

- Results are accurate and include only products matching the search query.

Test Outcome: ✓ Search functionality is accurate, showing only relevant results.

✓ Cart Operations:

Add to Cart:

Test: Added items to the cart and verified the cart is updated with the correct quantity, price, size and product details.

Expected Result:

- Items are added instantly to the cart.
- Cart total updates dynamically based on the added product price.

Test Outcome: ✓ Adding items to the cart works without issues. The total updates as expected.

Remove Items:

Test: Remove items from the cart and check if they disappear and the total updates accordingly.

Expected Result:

- The removed item is no longer displayed in the cart.
- The cart total reflects the change immediately.

Test Outcome: ✓ Removing items from the cart functions properly. The cart updates dynamically.

✓ Dynamic Routing:

Test: Clicked on a product from the product listing page and verify the individual product detail page loads correctly. Checked that all product details (e.g., name, description, price, images) are displayed.

Expected Result:

- Product detail pages load dynamically without any delay or errors.
- The correct product data is fetched and displayed.

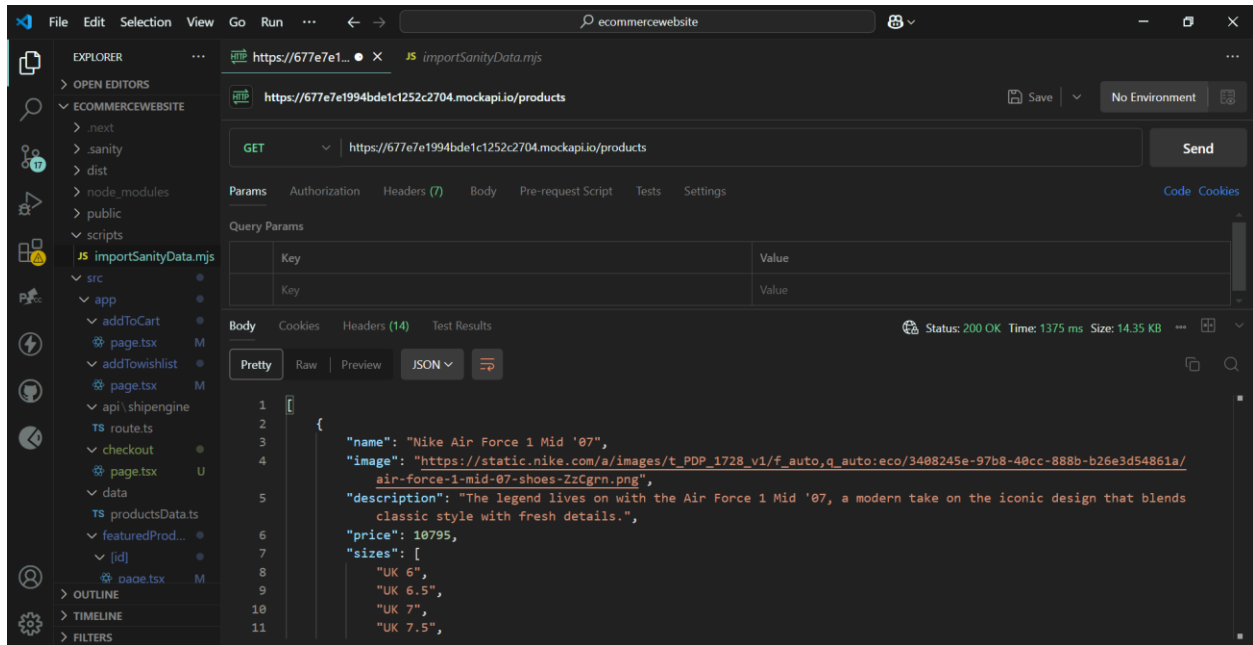
Test Outcome: ✓ The dynamic routing and data fetching for the product detail page worked as expected.

✓ Postman (API Response Testing):

Test: To test the backend APIs to ensure they return the correct responses, status codes, and data structures.

- Send a GET request to /api/products.
- Check the status code "200 for success".
- Verify the structure of the JSON response (e.g., product details like name, price, image).

Test Outcome: ✓ **Success:** The GET request returns a 200 OK status with a valid response body containing the correct product details.



2. Error Handling:

To enhance the user experience , I handled various errors gracefully and provide informative feedback to the user.


3. Network Failures:

Occurs when the client cannot fetch data due to issues like a lost internet connection or server downtime.

Implementation:

Used **try...catch** blocks with **fetch** to handle network request errors.

UI Feedback: Displayed a friendly message like "Unable to fetch data. Please check your internet connection."



```

useEffect(() => {
  const fetchProducts = async () => {
    const query = `*[_type == "products"]{
      _id, name, "imageUrl": image.asset->url, description, price, sizes, rating,
      stock, discount, category, color, details, style, tag, id
    }`;
    try {
      const productsData: Product[] = await client.fetch(query);
      setProducts(productsData.map((product) => ({
        ...product,
        imageUrl: urlFor(product.imageUrl).url(),
      })));
    } catch (error) {
      console.error('Error fetching products:', error);
      setProducts([]);
      alert("Failed to fetch products. Please check your network connection.");
    }
  };

  fetchProducts();
}, [filters, searchQuery]);

```

✓ Invalid or Missing Data:

Happens when the API returns incomplete or malformed data.

Implementation:

- Validated API response data before using it.
- Used conditional rendering to display fallback content if data is invalid or missing.

UI Feedback: Showed a fallback UI, such as "No products available."

```
<div className="w-full lg:w-[80%] grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6 px-4 lg:px-6 md:mt-12 mt-8 lg:mt-0">
  {currentProducts.length > 0 ? (
    currentProducts
      .filter((product) => product._id)
      .map((product) => (
        <Link key={product._id} href={` /FeaturedProducts/${product._id}`} >
          <ProductCard product={product} />
        </Link>
      ))
  ) : (
    <div className="col-span-full text-center text-gray-500">
      No products available. Please try again later.
    </div>
  )}
</div>
```

✓ Unexpected Server Errors:

Errors like 500 Internal Server Error or 404 Not Found.

UI Feedback:

Display user-friendly error messages such as:

- "The requested page was not found (404)."
- "An unexpected error occurred. Please try again later."

Implementation:

- Used fetch error response codes to differentiate between errors.
- Showed relevant messages based on error types.



```

const fetchProducts = async () => {
  setLoading(true);
  const query = `*[_type == "products"]{
    _id, name, "imageUrl": image.asset->url, description, price, sizes, rating,
    stock, discount, category, color, details, style, tag
  }`;
  try {
    const productsData: Product[] = await client.fetch(query);
    setProducts(productsData.map((product) => ({
      ...product,
      imageUrl: urlFor(product.imageUrl).url(),
    })));
    setError(null);
  } catch (error: any) {
    // Check for specific error types
    if (error.status === 500) {
      setError("Internal Server Error (500). Please try again later.");
    } else if (error.status === 404) {
      setError("Products not found (404).");
    } else {
      setError("Failed to fetch products. Please check your network connection.");
    }
    setProducts([]); // Clear products on error
  } finally {
    setLoading(false); // Stop Loading
  }
};

useEffect(() => {
  fetchProducts();
}, [filters, searchQuery]);

```

3. Performance Testing:

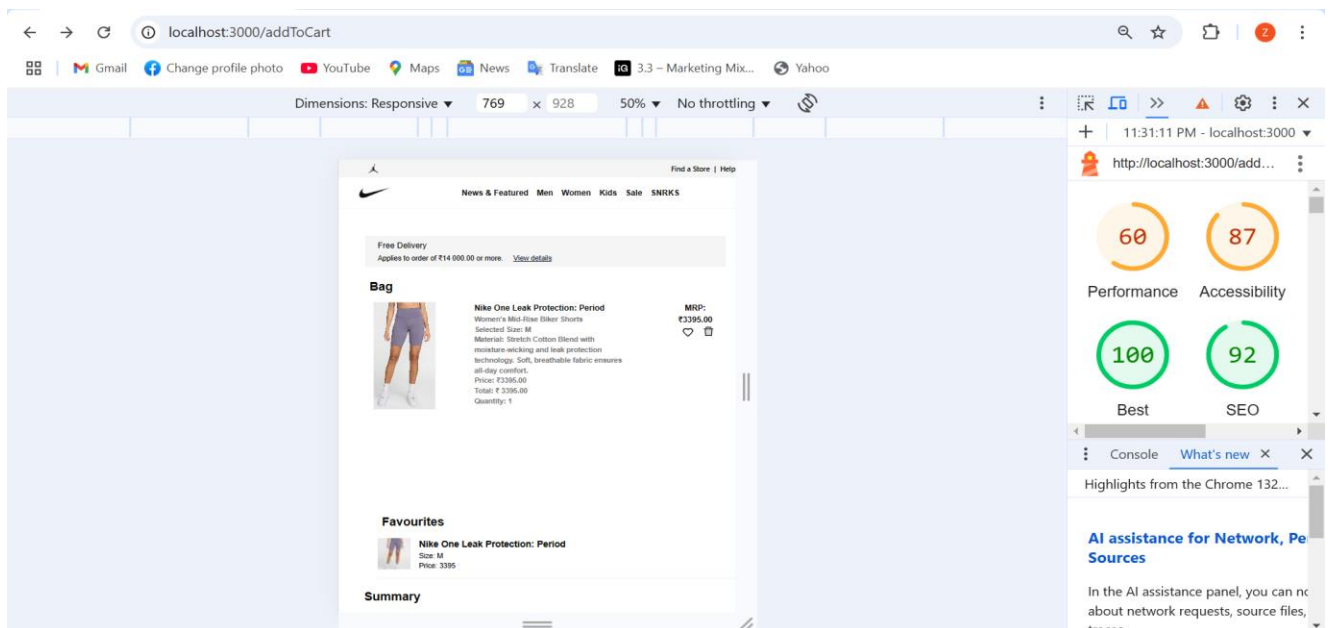
Performance testing is critical for ensuring a website or web application delivers a fast, responsive user experience, especially when it scales with increased traffic or large data. It involves identifying performance bottlenecks, optimizing resource usage, and ensuring that the website can load quickly across different devices and network conditions.

✓ Identifying Bottlenecks Using Tools:

Several tools are available to identify performance bottlenecks that can hinder the speed and efficiency of a website. These tools analyze various aspects of your website's performance and provide actionable insights. I'm using Lighthouse (by Google).

Lighthouse (by Google):

Lighthouse is an open-source, automated tool for improving the quality of web pages. It provides audits for performance, accessibility, SEO, best practices, and more.



✓ Image Optimization:

Images can significantly slow down a website if not optimized. By reducing the file size of images without compromising quality, we can improve load times.

Lazy Loading:

Delaying the loading of images that are outside the viewport (images that are not immediately visible when the page loads).

Responsive Images:

Used the srcset attribute to load different image sizes based on screen size and resolution.

Use WebP format:

WebP offers better compression and smaller file sizes than traditional formats (JPEG, PNG).

✓ Minimize JavaScript and CSS:

Large, unminified JavaScript and CSS files can delay page load times.

Caching Strategies:

Caching helps reduce the time to load a page by storing resources locally in the browser or on a CDN, preventing the need for repeated requests.

Browser Caching:

Set appropriate cache headers for static assets (e.g., images, scripts) so that browsers store them locally and do not request them every time the page loads. Used Cache-Control headers to define how long resources should be cached.

In a Next.js Application: In Next.js, you can use a custom server or configure caching through middleware or static file serving.

Server Static Assets: Store static files in the public/ folder and define headers using next.config.js.

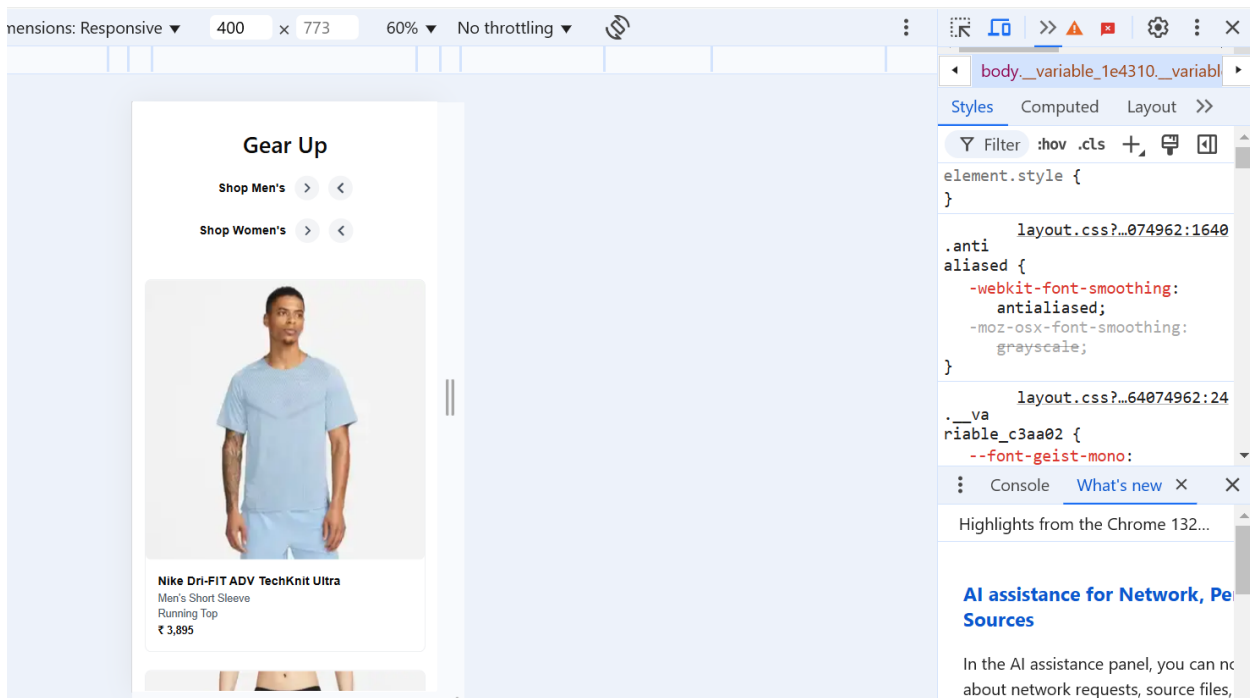
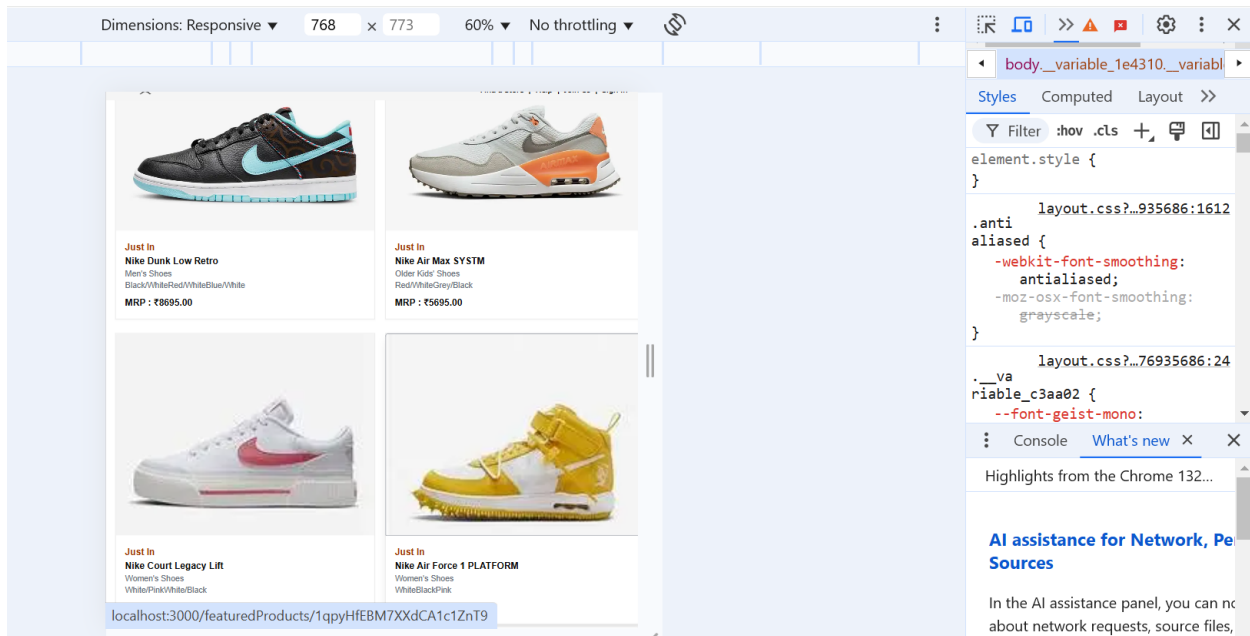
4. Cross-Browser and Device Testing:

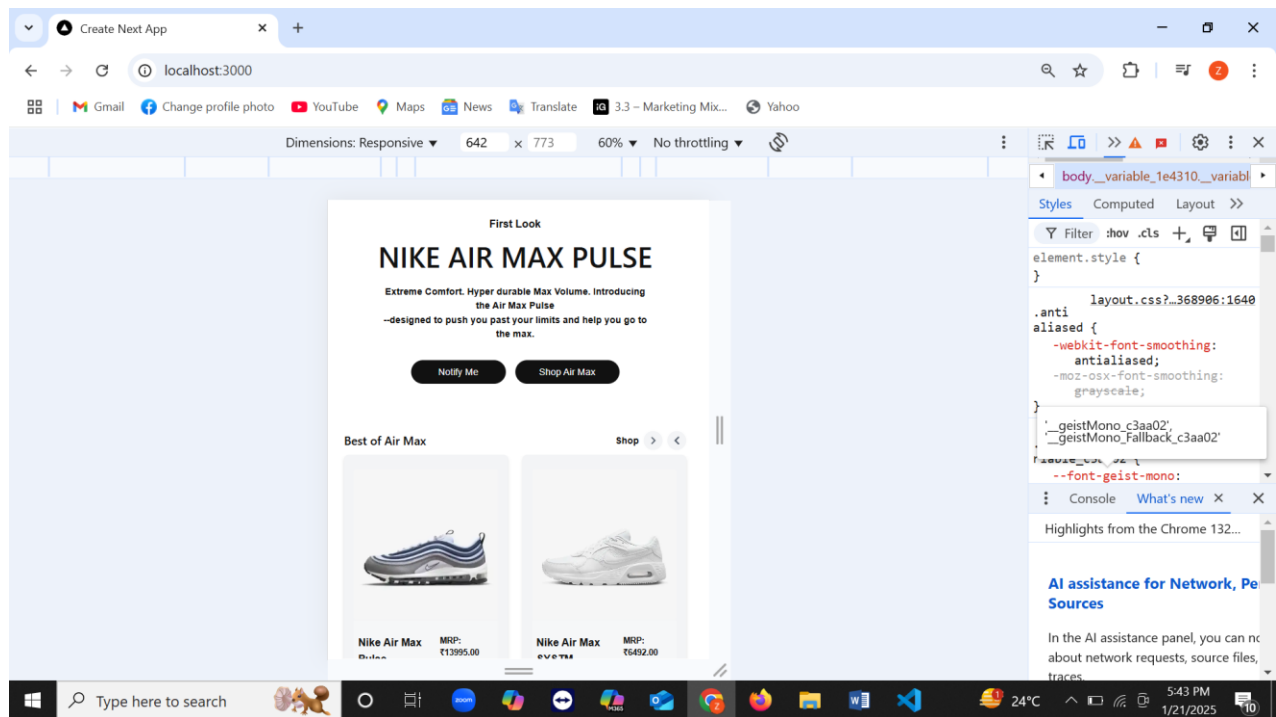
Ensuring that my marketplace works seamlessly across different browsers and devices is essential for providing a consistent user experience. Here's a detailed breakdown of the process:

Tested on Popular Browsers:

Tested my marketplace on the following widely used browsers to check for compatibility issues:

1. Google Chrome
2. Mozilla Firefox





Tested on Different Devices:

Test on various devices to ensure a smooth user experience across platforms:

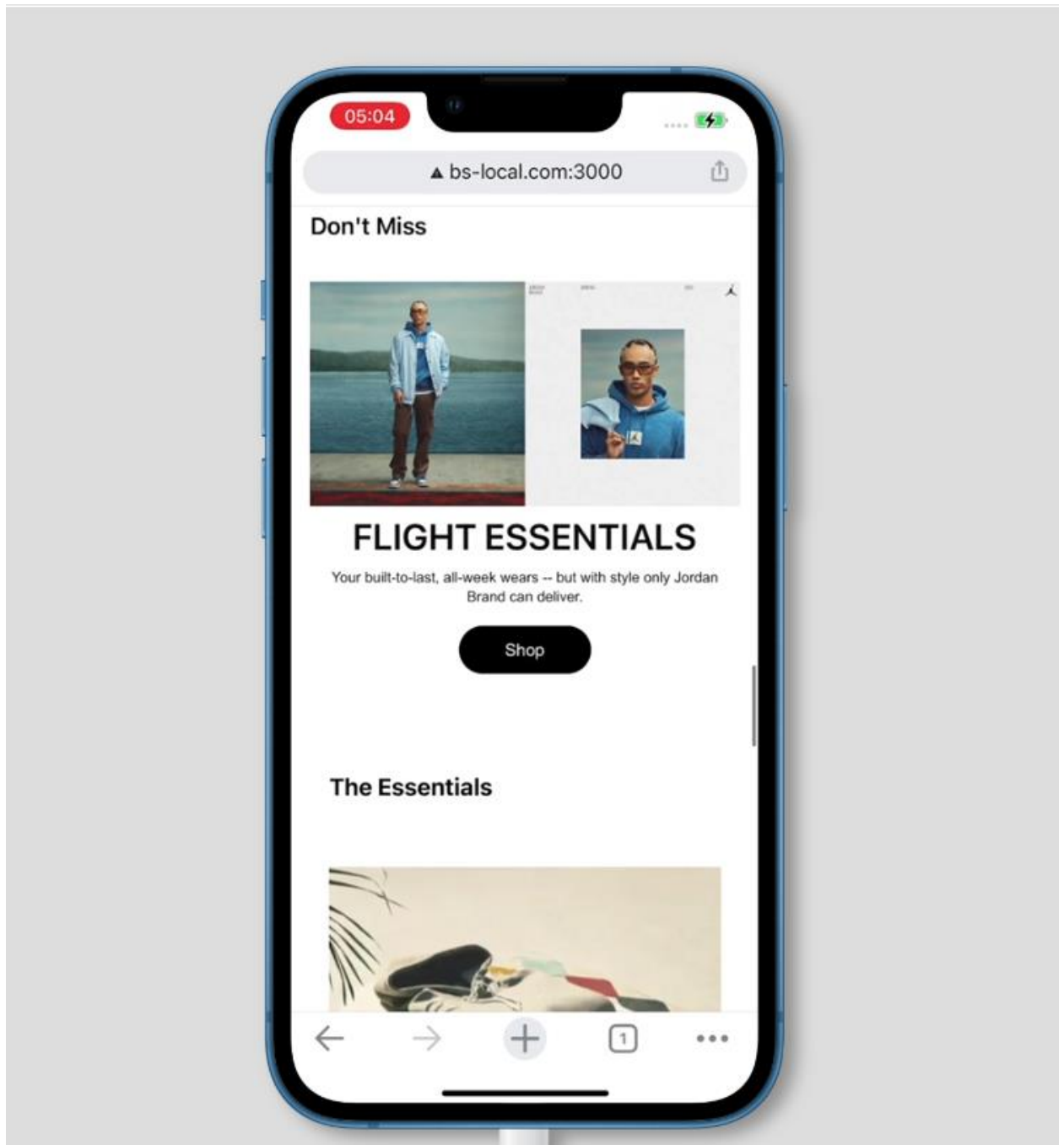
1. **Desktop:** Windows
2. **Tablets:** iPad, Android tablet
3. **Mobile:** iOS (iPhone), Android smartphone

Used Responsive Design Testing Tool:

I speed up the process by using online tool to simulate browsers and devices.

Browser Stack:

- A cross-browser and device testing platform.
- Allows you to test websites on real browsers and devices (e.g., iPhones, Androids, desktops).



5. Security Testing:

- ✓ Validated input fields to prevent injection attacks:

Injection attacks occur when an attacker sends malicious code through input fields or other user-controlled inputs, exploiting vulnerabilities in how applications handle user data. Common types include SQL Injection, Cross-Site Scripting (XSS), and Command Injection.

- Ensured that all input fields (e.g., forms, search bars, or URLs) validate data based on strict rules.
- Used regular expressions or input validation libraries to restrict input types (e.g., only accept numbers for numeric fields, email format for email fields).
- Sanitized inputs by removing or escaping special characters (<, >, ', ", --, ;) that could be used in malicious scripts.

✓ **Use HTTPS for Secure Communication:**

HTTPS encrypts communication between the user's browser and your server using SSL/TLS, preventing eavesdropping, tampering, and data theft.

It also ensures data integrity and authenticity by verifying the server's identity.

✓ **Avoid Exposing Sensitive API Keys in Your Frontend Code:**

Exposing API keys in frontend code allows attackers to misuse your APIs or gain unauthorized access to sensitive data.

✓ **Use Environment Variables:**

Store API keys in environment variables on the server side and never hard-code them in frontend files.

✓ **Proxy API Calls:**

Instead of making API calls directly from the frontend, create a backend endpoint to handle the call. This ensures that the API key stays secure on the server.

✓ **OWASP ZAP:**

OWASP ZAP (Zed Attack Proxy) is an open-source tool used for automated security and vulnerability scanning of web applications.

Testing Report (CSV Format):

A detailed testing report in CSV (Comma-Separated Values) format is an effective way to organize and share test results. Here's an explanation of the key columns typically included in such a report and their purpose

	A	B	C	D	E	F	G	H	I
1	Test Case ID	Test Case Description	Test Steps	Expected Result	Actual Result	Status	Severity Level	Assigned To	Remarks
2	TC001	Validate product listing page	Open product page > Verify products	Products displayed correctly	Products displayed correctly	Passed	High	-	No issues found
3	TC002	Test API error handling	Disconnect API > Refresh page	Show fallback UI with error message	Error message shown	Passed	Medium	-	Handled gracefully
4	TC003	Check cart functionality	Add product to cart > Verify cart contents	Cart updates with added product	Cart updates as expected	Passed	High	-	Works as expected
5	TC004	Ensure responsiveness on mobile	Resize browser window > Check layout	Layout adjusts properly to screen size	Responsive layout working as intended	Passed	Medium	-	Test successful
6									
7									

Conclusion:

Day 5 marked the final steps in refining and preparing the marketplace for deployment. By thoroughly testing functionalities, implementing robust error handling, and optimizing performance, the platform is now polished to deliver an exceptional user experience. Security measures and cross-browser/device compatibility were prioritized to ensure a reliable and secure platform for customers. Detailed testing documentation and CSV-based reports were submitted, demonstrating a professional and structured approach to testing and optimization. The marketplace is now ready to handle real-world traffic with speed, responsiveness, and customer-focused design.

