DAY 6- DEPLOYMENT PREPARATION AND STAGING ENVIRONMENT SETUP

Before deploying your website, it's crucial to ensure everything is prepared and optimized for production. Here’s a checklist to help you:

1. Code and Dependencies

- Remove Unused Code and Dependencies: Clean up any unused code, libraries, or components to reduce bundle size.
- Use .env for Secrets: Store API keys, environment-specific variables, and sensitive data in .env files. Never hardcode sensitive information.
- Check for Warnings/Errors: Ensure there are no errors or warnings in your code during development (npm run dev) or build (npm run build).

2. Environment Configuration

- Set the Correct Environment: Ensure your environment variable NEXT_PUBLIC_NODE_ENV is set to "production".
- API Endpoints: Verify that all API endpoints point to production servers (not staging or local).
- Environment Variables: Ensure all required environment variables are set in the production environment.

3. Static and Dynamic Pages

- Test Static Generation (getStaticProps/getStaticPaths): Ensure all paths and data-fetching logic work correctly during the build process.

- Verify Server-Side Rendering (getServerSideProps): Confirm pages requiring dynamic data load correctly.
- Fallbacks for Dynamic Routes: If using getStaticPaths, ensure fallback is properly configured.

4. Optimization

- Image Optimization: Use Next.js Image Optimization with the <Image /> component for better performance.
- Bundle Analysis: Run next build && next analyze to identify large bundles and optimize them.
- Tree Shaking: Ensure dead code elimination is working, and you are importing only what you need.

5. Security

- HTTPS: Use an SSL certificate to secure your site.
- Helmet or Equivalent: Add headers for security (e.g., Content Security Policy, X-Frame-Options).
- Sanitize User Inputs: Prevent XSS, SQL injection, and other attacks if you're accepting user input.
- Rate Limiting: If you have APIs, implement rate-limiting to prevent abuse.

6. Performance

- Test Load Time: Use tools like Lighthouse, PageSpeed Insights, or WebPageTest to analyze performance.
- Enable Caching: Use Cache-Control headers for static assets and optimize caching for APIs.
- Lazy Loading: Use lazy loading for images and components that aren’t above the fold.

7. Accessibility

- Run Accessibility Tests: Use tools like Axe or Lighthouse to ensure the site is accessible to all users.
- Keyboard Navigation: Verify that all interactive elements can be accessed and used via keyboard.
- Alt Text for Images: Ensure all images have descriptive alt text.

8. Error Handling

- Global Error Pages: Create custom error pages for 404 and 500 errors.
- Error Logging: Integrate error reporting tools like Sentry to monitor runtime errors in production.

9. Final Testing

- Cross-Browser Testing: Test your site in multiple browsers (Chrome, Firefox, Safari, Edge).
- Responsive Design: Verify the site looks good on different screen sizes (mobile, tablet, desktop).
- Test Critical Flows: Ensure all user journeys (e.g., login, checkout, search) work seamlessly.
- I use Lighthouse, Cypress tool for checking performance and components.

10. Deployment

- Build and Export: Run npm run build and verify the output in the .next folder.
- Preview Deployment: Deploy to a staging environment before production.

- Use a CDN: Deploy using a CDN (e.g., Vercel, AWS CloudFront) for better performance and distribution.
- Monitor Deployment: Test the live site thoroughly after deployment.

Step 1: Functional Testing

1. Search Bar

- Expected Functionality:
    - Users can search for products by keywords.
    - Suggestions or auto-completion may appear as the user types.
- Features to Test:
    - Search results update dynamically as the user types.
    - Correct products are displayed based on search terms.
    - Handling of no results (e.g., display a "No results found" message).
    - Case-insensitivity in searches.
    - Search performance for large datasets.
    - Special characters or empty input should not crash the application.

---

2. Add to Cart

- Expected Functionality:
    - Users can add products to the cart with a single click.
    - Quantity selection is available and updates in the cart.
- Features to Test:

- Item successfully added and reflected in the cart.
- Cart total updates correctly with the addition or removal of items.
- Handle duplicate product addition (increase quantity instead of duplicate rows).
- UI feedback after adding an item (e.g., toast notifications or badge updates).
- Persistence of cart data (local storage/session storage).
- Validation for adding invalid items (e.g., out of stock).

---

3. Product Card

- Expected Functionality:
  - Each card displays product details (name, price, description, image).
  - Click actions navigate to the product detail page or perform another action.
- Features to Test:
  - All data displays correctly on the product card.
  - Card responsiveness across various screen sizes.
  - Image loading and fallback if an image is missing.
  - Click actions on "Add to Cart" or "View Details" buttons.
  - Hover effects and accessibility features (e.g., focus state).

---

4. Filter

- Expected Functionality:
  - Users can filter products based on categories, price range, ratings, etc.
  - Multiple filters can be applied simultaneously.
- Features to Test:

- Filter criteria are correctly applied to the product list.
- Filters are reset when the user clears selections.
- Combined filtering (e.g., category + price range).
- Performance with large datasets and multiple filters.
- Clear messaging when no products match the applied filters.
- Responsive design for filter options.

---

5. Dynamic Cards

- Expected Functionality:
  - Cards dynamically display data based on user interactions or API responses.
- Features to Test:
  - Cards update correctly with API data or dynamic changes.
  - No duplication or missing cards.
  - Handle errors gracefully when dynamic data fails to load.
  - Performance for loading large numbers of cards.
  - UI behavior when the number of cards exceeds the visible area (e.g., infinite scroll or pagination).

Step 2:Error Handling

1. API Connectivity

- Ensure the API endpoints are reachable and return a **200 OK** status.
- Verify secure connections using HTTPS.

- Test the API with **valid** and **invalid** tokens (if authentication is required).

---

2. Sanity Integration

- Confirm that Sanity CMS is correctly configured and data is being fetched using GROQ queries or other mechanisms.
- Validate that Sanity's dataset matches the structure your frontend expects.
- Ensure real-time updates from Sanity (if using its real-time listener).

---

3. Data Accuracy

- Validate the API data matches the structure and fields required on your website (e.g., titles, descriptions, images, etc.).
- Test data integrity by cross-checking with Sanity Studio.
- Handle cases where data is incomplete or fields are missing gracefully.

---

4. Error Handling

- Test for API failures (e.g., 404 Not Found, 500 Internal Server Error) and ensure appropriate error messages are displayed to the user.
- Handle **timeout scenarios** gracefully without breaking the UI.
- Display fallback content or messages when the API fails to fetch data.

---

5. Performance

- Measure API response times and ensure they are within acceptable limits.
- Optimize queries in Sanity (e.g., limit the fields fetched to only those required).
- Use caching mechanisms (e.g., stale-while-revalidate or getStaticProps/getServerSideProps in Next.js).

---

6. Data Rendering on the Website

- Ensure the data fetched from APIs dynamically populates the components correctly (e.g., product cards, filters, etc.).
- Handle edge cases, such as empty data arrays or unexpected data formats.
- Test dynamic routing or URL-based data rendering (e.g., /product/:id).

---

7. Pagination and Lazy Loading

- Test API responses for paginated data (if implemented) and validate next/previous page calls.
- Verify that lazy loading or infinite scroll fetches data correctly without duplication.

---

8. API Security

- Validate that sensitive data (e.g., API keys) is not exposed in the frontend code.
- Use environment variables (process.env) to store keys securely in the build.

---

9. Cross-Browser Compatibility

- Ensure API-driven data renders consistently across different browsers and devices.
- Test for older browser versions to ensure compatibility.

---

10. Real-Time Updates (if applicable)

- If using real-time updates (e.g., WebSockets, Sanity's real-time listener), validate that changes in Sanity reflect instantly on the website.
- Ensure live updates don't cause performance bottlenecks.

---

11. Accessibility

- Ensure API data renders in a way that supports accessibility (e.g., readable by screen readers, properly labeled ARIA roles).

---

12. Localization (if applicable)

- If the API serves data for multiple languages, validate that the correct language data is displayed based on user settings.

---

13. Testing Automation

- Automate the API testing using tools like **Postman**, **Cypress**, or **Jest** to validate responses for expected scenarios.
- Write end-to-end tests in Cypress to simulate user interactions that trigger API calls and validate their behavior.

Step3: Api Testing

Here are some key points you might consider when working with and testing your API:

1. Understanding the API

- **Base URL**: https://fakestoreapi.com/products
- **Purpose**: Provides product data for a mock e-commerce platform.

2. Common API Operations

- **GET**: Fetch all products or a single product by ID.
- **POST**: Add a new product (if supported by the API).
- **PUT/PATCH**: Update existing product data.
- **DELETE**: Remove a product.

3. Testing Points

- **Response Status Codes**: Ensure the API returns appropriate HTTP status codes:
    - 200 OK for successful operations.
    - 404 Not Found for invalid endpoints or missing resources.

    - 201 Created for successful POST operations.
- **Response Data**: Verify the structure and content of the response:
    - Are all expected fields (id, title, price, description, category, etc.) present?
    - Validate data types (e.g., price is a number, title is a string).
- **Error Handling**: Test for appropriate error messages and status codes when:
    - Providing invalid IDs.
    - Missing required fields in POST/PUT requests.
- **Performance**: Measure response times using Thunder Client/Postman.

4. Sanity Checks

- Test the API with a minimal set of inputs to confirm it's functional.
- Verify default behaviors for optional parameters.

5. Examples

- **GET All Products**: GET https://fakestoreapi.com/products
- **GET Single Product**: GET https://fakestoreapi.com/products/1
- **POST Example**:

json

CopyEdit

```
{
  "title": "New Product",
  "price": 29.99,
  "description": "A brand-new item",
  "category": "electronics",
  "image": "https://example.com/image.jpg"
}
```

- **PUT Example**:

json

CopyEdit

{

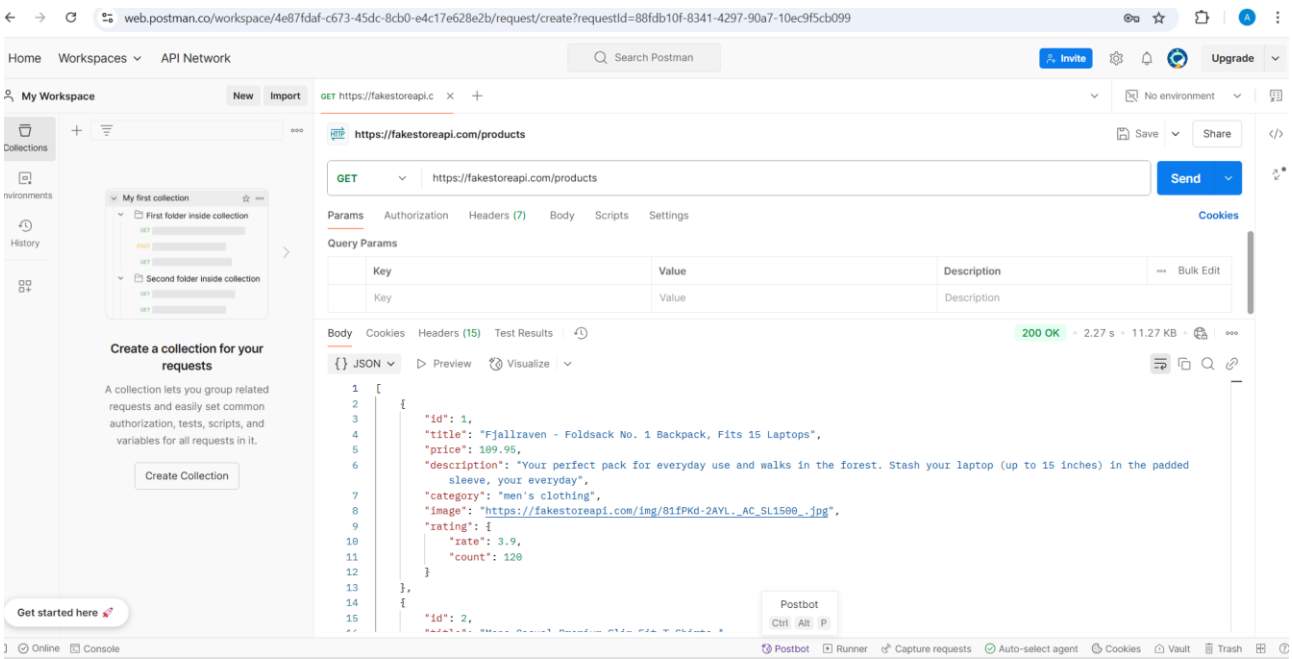
"title": "Updated Product",

"price": 25.99

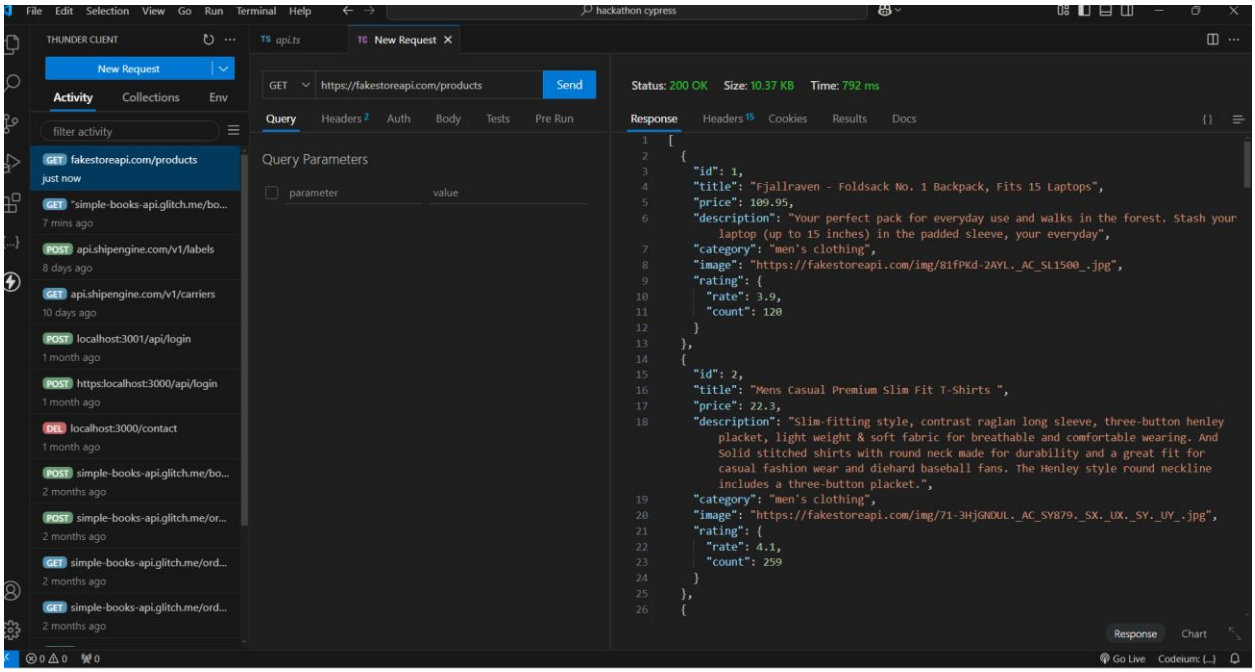
}

6. Integration with Your Website (Bandage)

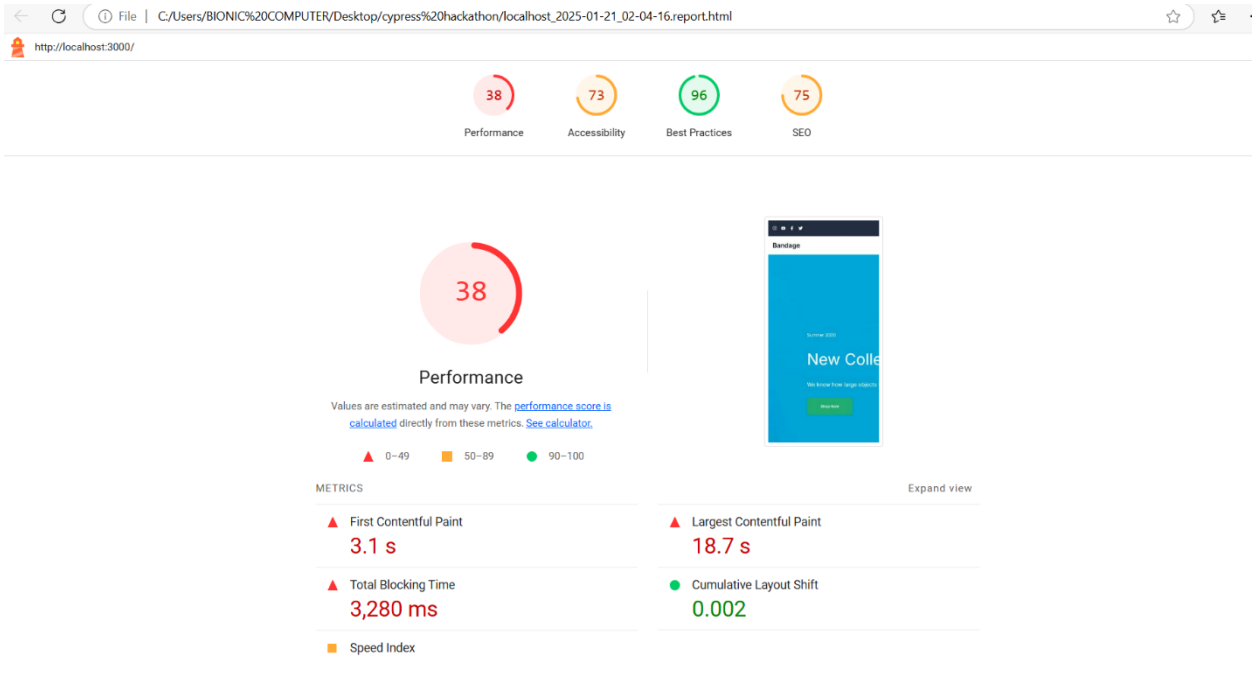
- **Fetch and Display**: Use the GET method to fetch product data and render it dynamically.
- **Form Submissions**: Utilize POST for adding new items to the catalog.
- **CMS (Sanity)**: Integrate with Sanity to manage and store product content effectively.



Postman



Thunder ClientUsing Light House



File | C:/Users/BIONIC%20COMPUTER/Desktop/cypress%20hackathon/localhost_2025-01-21_02-04-16.report.html

http://localhost:3000/

38 73 96 75

View Treemap

Show audits relevant to: All FCP LCP TBT CLS

DIAGNOSTICS

- Minimize main-thread work — 6.5 s
- Reduce JavaScript execution time — 4.5 s
- Largest Contentful Paint element — 18,680 ms
- Eliminate render-blocking resources — Potential savings of 2,010 ms
- Preconnect to required origins — Potential savings of 350 ms
- Reduce unused CSS — Potential savings of 16 KiB
- Reduce the impact of third-party code — Third-party code blocked the main thread for 770 ms
- Page prevented back/forward cache restoration — 4 failure reasons
- Minify CSS — Potential savings of 3 KiB

http://localhost:3000/

38 73 96 75

96

Best Practices

GENERAL

- Browser errors were logged to the console
- Detected JavaScript libraries
- Missing source maps for large first-party JavaScript

TRUST AND SAFETY

- Ensure CSP is effective against XSS attacks
- Use a strong HSTS policy
- Ensure proper origin isolation with COOP

Test Case ID	Test Case Description	Test Steps	Expected Result	Actual Result	Status	Security Level	Assigned to	Remarks
TC001	Validate product listing	Open product page>verify products	Products displayed correctly	Products displayed correctly	Passed	Low	-	No issue Found
TC002	Test API error Handling	Use postman and thunder client	no error	array of object shown	Passed	Medium	-	Successful
TC003	Check Cart functionality	Add products to cart>check functionality	Cart updated with added product	Card update	Passed	High	-	Successful
TC004	Ensure responsiveness on mobile	Check layout	layout adjust properly to screen	Responsive layout	Passed	Medium	-	Successful

---

est ase	Test Case Description	Test Steps	Actual Result	Status	Security level	Remarks
C001	Validate product listing	Open product page>verify products	Products displayed correctly	Passed	Low	No issue found
C002	Test API error Handling	Use postman and thunder client	array of object shown	Passed	Medium	Successf
C003	Check Cart functionality	Add products to cart>check functionality	Card update	Passed	High	Successf
C004	Ensure responsiveness on mobile	Check layout	Responsive layout	Passed	Medium	Successf

Checklist for Day 6:

Deployment Preparation:

- ✓

Staging Environment Testing:

- ✓

Documentation:

- ✓

Form Submission:

- ✓

Final Review:

- ✓