Step 1: Functional Testing

Functional testing ensures that the **core features of the marketplace** work correctly.

✅ What to Test?

- **Product Listing**: Products should display correctly with accurate prices and stock statuses.
- **Filters & Search**: The marketplace should filter products based on category and user queries.
- **Cart Operations**: Users should be able to add, update, and remove items from the cart.
- **Checkout Process**: Ensure order placement works smoothly.
- **User Profiles**: Profiles should store and display user details correctly.

🔧 How to Test?

- **Postman**: Test API responses and ensure data is correctly fetched from the backend.
- **React Testing Library**: Test individual frontend components.
- **Cypress**: Perform **end-to-end testing** by simulating user interactions.

Example: Cypress Test for Product Search

```
describe("Search Functionality", () => {
  it("should filter products based on user input", () => {
    cy.visit("/");
    cy.get("#search-bar").type("Laptop");
    cy.get(".product-card").should("contain", "Laptop");
  });
});
```

Step 2: Error Handling

Error handling prevents application crashes and improves the user experience.

✅ What to Handle?

- **API Failures**: Handle network issues or backend downtime gracefully.
- **Invalid Data**: Prevent crashes if API responses are missing fields.
- **User Input Errors**: Display meaningful validation messages.

🔧 How to Implement?

- **Use `try-catch` for API calls**:

```
const fetchProducts = async () => {
  try {
    const response = await fetch("/api/products");
    if (!response.ok) throw new Error("Failed to fetch products");
    return await response.json();
  } catch (error) {
    console.error(error);
    return [];
  }
};
```

- **Display fallback UI if data is unavailable**:

```
type Product = {
  id: number;
  name: string;
  image: string;
  price: number;
  stock: number;
};

type ProductListProps = {
  products: Product[];
};

const ProductList: React.FC<ProductListProps> = ({ products }) => {
  if (!products.length) return <p>No products available. Please try again later.</p>;

  return (
    <div className="product-listing">
      {products.map((product) => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
};
```

Step 3: Performance Optimization

Optimizing performance improves **page speed, user experience, and reduces server load**.

✅ What to Optimize?

- **Images**: Use compressed formats (WebP) and lazy loading.
- **JavaScript & CSS**: Minify files and remove unused code.
- **Caching**: Use local storage and browser caching to speed up repeated visits.

🔧 How to Implement?

- **Use Lighthouse to Identify Issues**:

```
npx lighthouse https://your-marketplace.com --view
```

- **Implement Lazy Loading for Images**:

```
type LazyImageProps = {
  src: string;
  alt: string; };
const LazyImage: React.FC<LazyImageProps> = ({ src, alt }) => (
  <Image loading="lazy" src={src} alt={alt}  />  );
```

- **Enable Gzip Compression in Next.js**:

```
export async function getServerSideProps() {
  return {
    props: {},
    headers: {
      "Content-Encoding": "gzip",
    }, }; }
```

Step 4: Cross-Browser and Device Testing

Ensures that the marketplace looks and functions correctly on different browsers and devices.

✅ What to Test?

- Chrome, Firefox, Safari, Edge.
- Mobile (iOS & Android), Tablets, and Desktops.

🔧 How to Test?

- Use **BrowserStack or LambdaTest** for browser compatibility testing.
- Use **Chrome DevTools** to simulate different devices.

Example: Media Query for Responsive Design

```
@media (max-width: 768px) {
    .product-grid {
      grid-template-columns: 1fr;
    }
  }
```

Step 5: Security Testing

Protects against **data breaches, injection attacks, and API vulnerabilities**.

✅ What to Secure?

- **Form Validation**: Prevent SQL injection and XSS attacks.
- **HTTPS**: Ensure secure API communication.
- **API Keys**: Store in environment variables.

🔧 How to Implement?

- **Sanitize User Input**:

```
const validateInput = (input: string): string => input.replace(/[^a-zA-Z0-9 ]/g, "");

const secureApiCall = async (endpoint: string): Promise<any> => {
  try {
    const response = await fetch(endpoint, {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${process.env.NEXT_PUBLIC_API_KEY}`,
      },
    });
    if (!response.ok) throw new Error("API request failed");
    return await response.json();
  } catch (error) {
    console.error(error);
    return null;
  }
};
```

- **Use `.env` for Sensitive Data**:

```
NEXT_PUBLIC_API_KEY=your-secret-key
```

- **Run Security Tests using OWASP ZAP**

```
zap-cli scan https://your-marketplace.com
```

Step 6: User Acceptance Testing (UAT)

Ensures **real-world users** can navigate the marketplace easily.

✅ How to Perform?

- **Invite non-technical users to test the platform.**
- **Simulate real customer scenarios**: Searching, filtering, adding products to the cart.
- **Collect feedback** and adjust the UX accordingly.

Example: Feedback Form

```tsx
const FeedbackForm: React.FC = () => {
  const [feedback, setFeedback] = useState<string>("");

  const handleSubmit = (event: React.FormEvent) => {
    event.preventDefault();
    console.log("Feedback submitted:", feedback);
  };

  return (
    <form onSubmit={handleSubmit}>
      <textarea onChange={(e) => setFeedback(e.target.value)} />
      <button type="submit">Submit</button>
    </form>
  );
};
```