

**Special Topics**

Amr Adi

**Table of Contents**

[Task 1 3](#_Toc209384995)

[References 16](#_Toc209384996)

## Task 1

**Q1-**

The Purpose of the Web Request-Response Cycle (WRRC)

The Web Request-Response Cycle explains how a client (like a web browser or a mobile app) and a server communicate over the internet. Its purpose is to make sure information can travel back and forth in a structured way. Whenever a user clicks a link, submits a form, or opens a page, the browser sends a request to the server, and the server processes it, then sends back a response. This cycle is the foundation of how all web applications work, whether it is loading a static page or fetching dynamic data from a database.

Components of WRRC

Client (Front-End / Browser / App):

* The starting point of the cycle.
* Sends a request to the server, usually through HTTP or HTTPS.
* Examples: typing www.htu.edu.jo into Chrome or clicking “Add to Cart” in a React app.

Request:

* The message sent by the client to the server.
* Includes the URL, method (GET, POST, PUT, DELETE), headers, and sometimes a body with data.
* Example: a POST request with login details.

Server (Back-End):

* Receives the request and decides how to handle it.
* Often written in Node.js/Express, Python/Django, or PHP.
* Responsible for connecting to the database, applying business logic, and preparing the response.

Database (Storage):

* The server may fetch or update data from the database.
* Example: PostgreSQL returning all products in the “Personal Care” category.

Response:

* The message sent back from the server to the client.
* Contains the status code (200 OK, 404 Not Found, 500 Error), headers, and data (HTML, JSON, images, etc.).
* Example: a JSON list of products returned to the React front-end.

Rendering on Client Side:

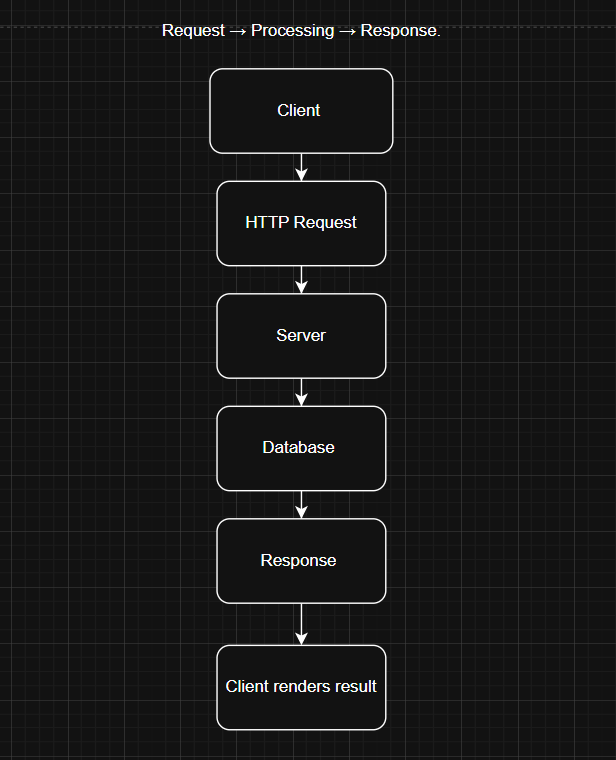
* The client receives the response and displays it to the user.
* In modern apps, frameworks like React take the JSON response and dynamically render it into components.

Client–Server Communication Structure

The communication follows a request → processing → response pattern:

* The client initiates a request (e.g., clicking “Explore”).
* The server receives the request, processes it, and may query the database.
* The server sends a response back (e.g., product details).
* The client interprets and renders the response.

It is a continuous cycle, repeated every time the user interacts with the application. This separation of responsibilities (client focuses on UI, server focuses on logic and data) makes applications scalable and easier to maintain.

****

The WRRC is not just a one-time exchange, but the backbone of how interactive applications function. Without it, the client would not be able to communicate with the server or fetch updated data. For example, in my project (the Pharmacy Store), when a customer clicks on “Explore Personal Care,” the React front-end makes a request to the Express server. The server checks the database for all products in that category, then responds with JSON data. Finally, React dynamically updates the UI without refreshing the page.

This cycle highlights the importance of clear roles: the client handles the presentation, the server handles logic, and the database handles storage. Together, they ensure a smooth and secure user experience. Understanding this cycle also helps in debugging — if something fails, I can check whether the problem is in the request, the server logic, or the database query.

**Q2-**

When I compare building interfaces with a front-end framework like React.js versus the traditional approach of plain HTML, CSS, and JavaScript, I see clear differences in how I design, code, and maintain a web app.

By traditional, I mean writing static HTML for structure, CSS for styling, and imperative JavaScript that directly manipulates the DOM (or light helpers like querySelector, addEventListener, and maybe a bit of jQuery). There’s no component system, no built-in routing, and I manually decide when and how to update the page.

Advantages of React.js over Traditional HTML, CSS, and JavaScript

1 Components make the UI modular and reusable

* React: I break the UI into components (e.g., a product card, a navbar, a form). Each component owns its markup, behavior, and sometimes styles. I can reuse a component in many places and change it once to update it everywhere.
* Traditional: I tend to copy/paste markup and wire up separate event handlers. Repetition creeps in and small changes can require touching multiple files.

Result: Faster iteration, less duplication, and a more consistent interface.

2 State-driven rendering (declarative, not manual DOM work)

* React: I describe what the UI should look like given the current state. When state changes (like a counter, a filter, or a list from the server), React re-renders the right parts automatically.
* Traditional: I write the how imperative code to find elements, update text, add/remove classes, attach/detach listeners, and keep everything in sync. This gets fragile as features grow.

Result: Fewer UI bugs from missed DOM updates and a codebase that’s easier to reason about.

3 Client-side routing for a smooth app experience

* React Router: I get navigation between views (home, details, cart, admin, etc.) without full page reloads. State (like a cart count) stays in memory while moving around the app.
* Traditional: I either reload whole pages or build my own history handling, which is more work and easy to get wrong.

Result: Faster-feeling navigation and simpler state management across screens.

4 Predictable data fetching with lifecycle hooks

* React: Hooks like useEffect give me a defined place to fetch data and clean up subscriptions or timers. I can avoid stale data and race conditions more easily.
* Traditional: I sprinkle fetch calls in different scripts and manually coordinate when to update or cancel, which becomes messy.

Result: Cleaner data flow and fewer timing-related bugs.

5 Strong ecosystem & tooling

* React + Vite: Fast dev server, hot reload, JSX, and great build performance.
* Libraries: Mature options for forms, tables, routing, animations, testing, etc. As the app grows, I can bring in high-quality packages rather than reinventing the wheel.

Result: Higher productivity and better quality with less custom code.

6 Performance patterns that scale

* React: Virtual DOM diffing, memoization (memo, useMemo, useCallback), and code splitting are available when I need them.
* Traditional: Lots of ad-hoc DOM updates can become slow and harder to optimize as the UI becomes complex.

Result: A smoother path from small prototype to larger, more dynamic app.

7 Maintainability over time

* React: A clear file/folder structure, explicit data flow, and component boundaries make refactoring safer. If I add TypeScript later, it’s even easier to prevent regressions.
* Traditional: Logic tends to spread out (HTML attributes, scattered handlers, inline scripts), which makes refactors riskier.

Result: Code stays understandable for me (and for teammates) months later.

8 Testability

* React: Component tests can render a piece of UI in isolation and simulate user actions. This is far more reliable than poking at a live DOM built from many unrelated scripts.
* Traditional: Tests often become brittle because the UI is glued together imperatively.

Result: Confidence to change code without breaking user flows.

9 A path to other platforms

* React skills transfer to React Native (mobile) and Next.js (SSR/SEO). The mental model carries over.
* Traditional: Not directly portable.

Result: What I learn with React stays useful across projects and platforms.

|  |  |  |
| --- | --- | --- |
| Area | Traditional (HTML/CSS/JS) | React.js |
| Structure | Templates + manual DOM | Reusable components |
| Updates | Imperative DOM changes | Declarative: UI = f(state) |
| Routing | Full reloads / custom logic | React Router (SPA) |
| Data fetching | Scattered callbacks | useEffect lifecycle |
| Scalability | Gets tangled | Stays modular and testable |
| Tooling | Minimal | Vite + huge ecosystem |
| Performance | Harder to optimize | Virtual DOM + memoization |
| Testing | Brittle DOM tests | Component-focused tests |

For small, static pages, traditional HTML/CSS/JS is perfectly fine. But the moment I need multiple views, shared state, dynamic lists, or updates across the interface, a framework like React pays off quickly. It gives me a clean component model, predictable rendering, great tooling, and a huge ecosystem—making the app easier to build, test, and maintain over time.

**Q3-** Amr’s Pharmacy is a simple storefront where a single user can browse categories, add/remove items to the cart, change quantities, and complete a basic checkout (cash on delivery). There’s no authentication flow. The admin manages the product catalog and categories directly in the database (adds/edits/deletes medicines, updates stock, price, and images).

User requirements:

Shopper

* view categories and open a category page to see its products.
* add items to the cart, remove items, and change quantities, so I can control my order.
* Showed the cart total to update instantly without reloading the page.
* checkout by entering name, phone, and address and see a confirmation with an order ID.

Admin

* add/edit/delete products (name, price, stock, description, image, category) in the database.
* add/rename/hide categories directly in the database.

System Requirements:

* The system shall provide an API to retrieve products and categories from the database.
* The system shall provide an API endpoint to create new orders with user details.
* The system shall validate input (name, phone, address) before storing an order.
* The system shall serve product images from a static folder.
* The system shall allow admins to update product/category data directly in the database.

**Q4-** I am using adalo:

https://app.adalo.com/apps/69a0cbf1-c1d0-46bd-8247-1a318616d8c6/screens

home page

**A screenshot of a computer

AI-generated content may be incorrect.**

**Products page**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Cart page**

**A screenshot of a computer

AI-generated content may be incorrect.**

**About us page**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Checkout page**

**A screenshot of a computer

AI-generated content may be incorrect.**

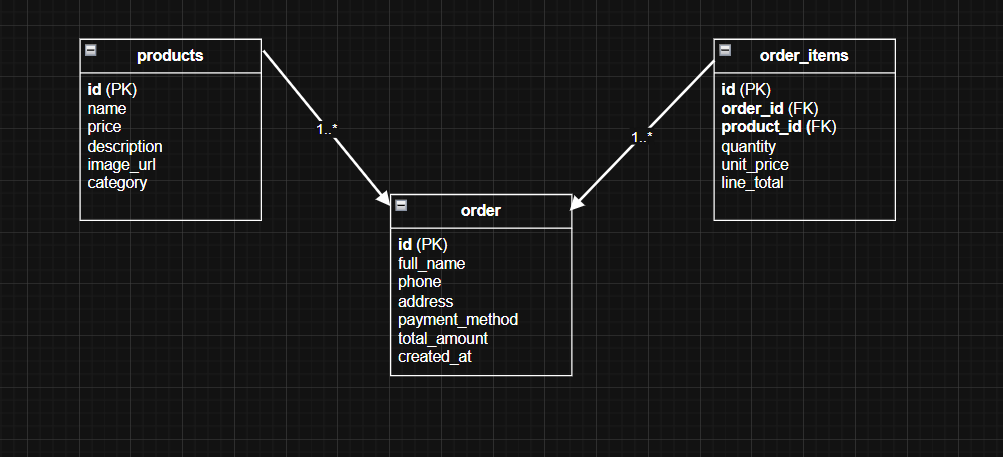
**Orders page:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Q5-**

**The ERD digrame:**

****

ntities and keys

* + products( id (PK), name, price, description, image\_url, category )
  + orders( id (PK), full\_name, phone, address, payment\_method, total\_amount, created\_at )
  + order\_items( id (PK), order\_id (FK), product\_id (FK), quantity, unit\_price, line\_total )

Relationships

* + orders: order\_items (one order has many order items)
  + products: order\_items (one product can appear in many order items)

## References

**backend.turing.edu. (n.d.). *How the Web Works, HTTP Request/Response Cycle - Back-End Engineering Curriculum - Turing School of Software and Design*. [online] Available at: https://backend.turing.edu/module2/lessons/how\_the\_web\_works\_http.**

**‌**

**AppsRhino (2024). *Advantages of Using React for Front-End Development*. [online] Medium. Available at: https://appsrhino.medium.com/advantages-of-using-react-for-front-end-development-da419ce77fc5.**

**‌**

**www.w3schools.com. (n.d.). *React Components*. [online] Available at: https://www.w3schools.com/react/react\_components.asp.**

**‌**