Fullstack Web Development

Homework Assignment | Weeks 1 & 2

This homework assignment covers the material from **Weeks 1 and 2** of the Fullstack Web Development Bootcamp. Week 1 introduces the React front-end library and the Express back-end framework, giving you the building blocks for full-stack JavaScript applications. Week 2 moves to data persistence with PostgreSQL and the Prisma ORM, then closes with user authentication using bcrypt and JSON Web Tokens. Complete each exercise in your own repository and push your work to GitHub before the submission deadline. Stretch goals are optional but strongly encouraged — they deepen your understanding and make great portfolio pieces.

WEEK 1 — React & Express Foundations

Session 1: React Foundations

Topics covered: JSX, functional components, rendering lists, props, UI composition

1. Hello Component

Create a functional component called **Greeting** that accepts a name prop and renders a personalised message (e.g., *"Hello, Alice!"*). Render it three times in App.jsx with different names.

2. Product Card

Build a **ProductCard** component that receives title, price, and imageUrl as props and displays them in a styled card layout. Demonstrate prop-types or JSDoc for type documentation.

3. Rendering a List

Given the array below, render an unordered list of book titles inside a **BookList** component. Ensure each list item has a unique key prop and explain (in a comment) why React requires it.

- const books = [{ id: 1, title: 'Clean Code' }, { id: 2, title: 'You Don\'t Know JS' }, ...]

4. Composing Components

Build a simple page layout using at least three components: **Header**, **Main**, and **Footer**. Compose them inside App.jsx. Pass a pageTitle prop from App down to Header.

5. Conditional JSX

Create a **Badge** component that renders *"In Stock"* in green or *"Out of Stock"* in red based on an inStock boolean prop. Use a ternary expression inside JSX.

Stretch Goal: Dynamic Product Grid

Fetch a list of products from the public API https://fakestoreapi.com/products using the useEffect hook (preview of next week\'s concepts), render them in a responsive CSS Grid layout using your **ProductCard** component, and show a loading spinner while the data loads.

Session 2: State & Forms

Topics covered: useState, controlled inputs, onChange/onSubmit, conditional rendering

1. Counter Component

Build a **Counter** component with an increment button, a decrement button, and a reset button. Display the current count. Prevent the count from going below 0.

2. Controlled Text Input

Create a **LivePreview** component with a text input and a paragraph below it. As the user types, the paragraph should update in real time using useState and a controlled input pattern.

3. Registration Form

Build a **RegistrationForm** with fields for username, email, and password. On submit, prevent the default browser action and log the form data to the console. Clear all fields after submission.

4. Form Validation

Extend your **RegistrationForm** with client-side validation:

- Username must be at least 3 characters.
- Email must contain an @ symbol.
- Password must be at least 8 characters.
- Display an inline error message beneath each failing field.

5. Toggle Visibility

Add a *"Show Password"* checkbox to the registration form. When checked, the password field should switch its type attribute from password to text, and vice versa.

Stretch Goal: Multi-Step Form

Refactor your registration form into a two-step wizard: Step 1 collects username and email; Step 2 collects password and a confirmPassword field with a match check. Use a single state object to hold all field values and a separate step state (1 or 2) to control which step is shown.

Session 3: Express Foundations

Topics covered: HTTP server setup, GET/POST routes, middleware, JSON responses, nodemon

1. Hello Express

Create a new Node.js project, install Express, and configure nodemon as a dev script. Write a single GET / route that returns a JSON object { message: "Server is running" }.

2. Route Parameters

Add a route GET /users/:id that reads the id from req.params, validates that it is a positive integer, and returns a JSON object { userId: id, name: "Placeholder" }. Return a 400 error if the id is invalid.

3. POST Route with Body Parsing

Add a POST /users route. Use the built-in express.json() middleware to parse the request body. Expect { name, email } in the body and respond with a 201 status and the received data echoed back.

4. Custom Middleware

Write a request-logger middleware function that logs the HTTP method, path, and timestamp to the console for every incoming request. Register it with app.use() before your routes.

5. Query String Filtering

Create a GET /products route. Define a small in-memory array of product objects (at least 5). Support an optional ?category= query parameter that filters the returned products by category. Return all products if the parameter is absent.

Stretch Goal: Global Error Handler

Add a 404 catch-all route and a global error-handling middleware (four-argument function: err, req, res, next). Wrap one of your existing routes in a try/catch and intentionally throw an error to verify the handler responds with a structured JSON error object containing status and message fields.

WEEK 2 — Databases & Authentication

Session 4: PostgreSQL & Prisma

Topics covered: Postgres setup, Prisma schema (models, relations, field types), migrations, Prisma Client queries

1. Schema Design

Create a Prisma schema with two models: **User** and **Post**. A User has many Posts (one-to-many relation). Include appropriate field types (String, Int, DateTime, Boolean) and add @default and @updatedAt decorators where sensible.

2. Run Migrations

Run prisma migrate dev --name init to apply your schema to a local Postgres database. Inspect the generated SQL migration file and write a brief comment explaining what each CREATE TABLE statement does.

3. Prisma Studio Exploration

Launch npx prisma studio and manually create two User records and three Post records linked to those users. Take a screenshot to include with your submission.

4. Basic Queries

In a standalone seed.ts (or seed.js) script, use Prisma Client to:

- Find all users and log their emails.
- Find a single user by id and include their related posts.
- Find all posts where published is true.

5. Filtering and Ordering

Write a query that fetches all Posts ordered by createdAt descending, limited to 5 results. Then write a second query that filters Posts whose title contains a specific search string (use Prisma\'s contains filter).

Stretch Goal: Add a Category Model

Extend your schema with a **Category** model and create a many-to-many relation between **Post** and **Category** using an explicit join table. Run a new migration and write queries to assign categories to posts and retrieve all posts belonging to a given category.

Session 5: Prisma CRUD

Topics covered: Create, read, update, delete via Express routes, input validation, error handling middleware

1. **Create User Endpoint**
   Add a POST /users route in your Express app that accepts { name, email } in the request body. Use Prisma to insert a new User record. Return the created user with a 201 status. Return a 409 conflict error if the email already exists.
2. **List and Get User Endpoints**
   Implement GET /users (returns all users) and GET /users/:id (returns a single user with their posts). Return a 404 JSON error if the user is not found.
3. **Update User Endpoint**
   Implement PATCH /users/:id that accepts partial updates (name and/or email). Only update the fields provided in the request body. Use Prisma\'s update method.
4. **Delete User Endpoint**
   Implement DELETE /users/:id. If the user has related Posts, decide whether to cascade-delete them (configure in Prisma schema with onDelete: Cascade) or return a 400 error explaining that posts must be removed first.
5. **Input Validation Middleware**
   Write a reusable validation middleware that checks incoming request bodies against a simple set of rules (e.g., required fields, minimum string length). Apply it to your POST /users and PATCH /users/:id routes. Return a structured 400 response listing all validation errors.

Stretch Goal: Pagination

Add ?page= and ?limit= query parameters to GET /users. Implement offset-based pagination using Prisma\'s skip and take options. Include pagination metadata in the response: { data, total, page, totalPages }.

Session 6: Authentication

Topics covered: bcrypt password hashing, JWT generation/verification, authMiddleware, client-side JWT storage

1. **Register Endpoint**
   Create a POST /auth/register route. Hash the user\'s password with bcrypt (use a salt rounds value of 10) before storing it in the database. Never store or return the plain-text password. Return a 201 status with the new user object (excluding the password hash).

2. Login Endpoint

Create a POST /auth/login route. Look up the user by email, use bcrypt.compare to verify the password, and on success generate a JWT signed with a secret stored in an environment variable. Return the token in the response body. Return a generic 401 Unauthorized error for both invalid email and invalid password (do not reveal which one failed).

3. Auth Middleware

Write an authMiddleware function that reads the JWT from the Authorization: Bearer <token> header, verifies it using jsonwebtoken, and attaches the decoded payload to req.user. If the token is missing or invalid, respond with 401.

4. Protected Route

Apply authMiddleware to GET /me. This route should use req.user to look up and return the currently authenticated user\'s profile (without the password hash). Test it with a valid and an expired token.

5. Client-Side Storage

In your React front-end, implement a login form. On successful login, store the JWT in localStorage. Include the token in the Authorization header for all subsequent API requests. Add a logout button that clears the token from storage and redirects to the login page.

Stretch Goal: Token Refresh

Implement a refresh-token flow: on login, issue both a short-lived access token (15 minutes) and a long-lived refresh token (7 days). Store the refresh token in an HTTP-only cookie. Add a POST /auth/refresh endpoint that validates the refresh token and issues a new access token without requiring the user to log in again.

Submission Instructions

Push all of your work to a **single GitHub repository** named fullstack-bootcamp-hw. Organise your code using the following folder structure:

week1/session1/ week1/session2/ week1/session3/

week2/session4/ week2/session5/ week2/session6/

Before submitting, ensure the following:

- Each session folder has its own package.json (or is clearly part of a shared project with instructions on how to run it).
- A root-level README.md briefly describes what each session covers and lists any environment variables required.
- No .env files are committed to Git. Use .env.example files with placeholder values instead.
- All code runs without errors (npm install then npm run dev or npm start).
- Stretch goal work is clearly marked in your README or in a separate stretch/ subfolder.

Deadline: Submit the GitHub repository link via the course portal before the start of Week 3, Session 1. Late submissions will be accepted up to 48 hours after the deadline with a note in the pull request.

Good luck — and remember, the goal is to learn by doing. If you get stuck, consult the session notes, official documentation, or post a question in the course Slack channel before the next session.