**Understanding Unit Testing and Integration Testing**

**Unit Testing**: This involves testing individual components (functions or methods) of the code in isolation. The goal is to ensure that each unit performs as expected independently.

**Tools for Unit Testing**:

* **JavaScript**: Jest, Mocha, Jasmine
* **Python**: unittest, pytest
* **Java**: JUnit

**Example**: If you have a function add (a, b), a unit test will ensure that add (2, 3) returns 5.

**Integration Testing**: This tests how different modules or components work together as a whole. Integration tests are broader than unit tests and cover scenarios where the system interacts with external services (e.g., databases or APIs).

**Tools for Integration Testing**: Postman, Cypress, Selenium, or JUnit (for Java).

**Example**: Testing whether a function saves data to a database and retrieves it correctly.

**Why Testing is Important**:

* Ensures the functionality of individual components.
* Increases reliability by catching bugs early.
* Enhances performance by spotting inefficiencies or bottlenecks.
* Improves the maintainability of the codebase.

**Installing and Setting up Postman for API Testing**

Postman is an excellent tool for testing API endpoints.

1. **Download and Install Postman**: You can download Postman from the official website.S
2. **Create a Collection**: A collection is a group of API requests that belong to a specific project.
   * Open Postman.
   * Click the Collections tab, then Create a Collection.
   * Name the collection (“Expense Tracking API”).
3. **Add API Requests to the Collection**: You can now add different requests like GET, POST, PUT, and DELETE.
   * I Created a new request inside the collection for each API endpoint of my expense tracking app. **CRUD Operations**: These are essential operations of a REST API.

**Create (POST)**: For adding a new expense (e.g., /api/expenses).

**Read (GET)**: For retrieving all expenses or a single expense (e.g., /api/expenses/:id).

**Update (PUT)**: For modifying an expense (e.g., /api/expenses/:id).

**Delete (DELETE)**: For removing an expense (e.g., /api/expenses/:id).

**Authentication**:

Test login functionality using a POST request with credentials (e.g., /api/auth/login).

Include access tokens in headers for protected routes, such as Bearer <token>.

**Error Handling**:

Ensure appropriate status codes are returned, e.g., 400 Bad Request for invalid input, 401 Unauthorized for failed authentication.

Use Postman’s Tests feature to write simple scripts to check the response (e.g., status code and error message).

**Debugging Techniques Using Browser Developer Tools (Frontend)**

**Using Chrome DevTools**:

**Open DevTools**: Press F12 or right-click on your page and select Inspect.

**Console Tab**: For viewing errors, logs, and debugging JavaScript.

**Sources Tab**: For setting breakpoints and stepping through code.

**Network Tab**: For inspecting API requests, responses, and performance.

**Debugging Strategies**:

**Inspecting Variables**: Use console.log() to track variable values in the Console.

**Breakpoints**: Set breakpoints in the Sources tab to pause code execution and inspect values.

**Network Issues**: Use the Network tab to check failed requests, latency, and payload.

**Putting It All Together**

Create unit tests for your backend functions (e.g., for expense calculation logic) using a tool like Jest or Mocha.

Perform integration tests using Postman to ensure your APIs are working correctly.

Debug the frontend using Chrome DevTools and inspect network requests.

* + Debug the backend using the VS Code debugger and logging strategie
  + Fill in the necessary details (URL, headers, body data, etc.).

**Testing CRUD Operations, Authentication, and Error Handling Using Postman**

**CRUD Operations**: These are essential operations of a REST API.

* **Create (POST)**: For adding a new expense.