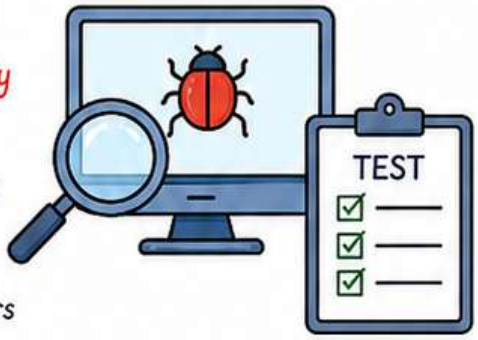
Chapter 23: Testing MERN Applications

23.1 Introduction to Software Testing

23.1.1 Meaning of Software Testing

* Software testing is the process of evaluating and verifying that a software application works as expected.
* It helps to find defects (bugs) and ensures the quality, reliability and performance of the application.
* Testing ensures that the application meets the requirements and provides a good user experience.
* In simple words, testing is about finding problems before the users find them.



Key Points:

- ☑ Testing helps to improve the quality of the application.
- ☑ It reduces the number of bugs and errors.
- ☑ It saves time and cost in the long run.
- ☑ It increases the confidence of users in the application.

23.1.2 Why is Testing Important?

* Testing helps to find bugs and errors before the application is released.
* It ensures that the application works correctly under different conditions.
* It improves the performance, security and usability of the application.
* It builds trust and satisfaction among users.
* It reduces the cost of fixing bugs after the application is deployed.

23.1.3 Testing in MERN Stack

In a MERN application, testing is done at different levels:

Frontend Testing
Testing the UI and user interactions built with React.

Backend Testing
Testing API endpoints and business logic built with Node.js and Express.

Database Testing
Testing database operations and data integrity in MongoDB.

Integration Testing
Testing the interaction between frontend, backend and database together.

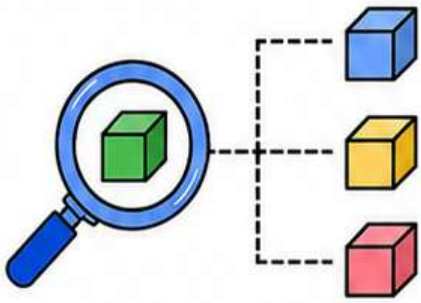
Key Takeaway:

Testing is an essential part of the software development lifecycle. It ensures that the MERN application is bug-free, secure, reliable and delivers a great experience to the users.

23.2 Unit Testing

23.2.1 What is Unit Testing?

* Unit testing is a type of testing where individual units or components of the application are tested in isolation.
* In the MERN stack, units can be functions, methods, components or modules.
* It ensures that each unit works as expected.
* It helps in finding bugs early in the development process.



Testing Individual Units

23.2.2 How Unit Testing Works?

The general flow of unit testing is:

1. Write a unit (function/component) → 2. Write a test case for the unit → 3. Run the test and check result → 4. Fix bugs (if any) → 5. Repeat until all tests pass

Key Points:

- ☑ Unit testing checks a small part of the code.
- ☑ It helps in writing clean, reliable and bug-free code.
- ☑ It makes code easier to refactor and maintain.
- ☑ Unit tests act as documentation for the code.
- ☑ It reduces the cost and time of fixing bugs.

23.2.3 Unit Testing Tools in MERN Stack

* Jest - Unit testing framework for JavaScript applications.
* React Testing Library - For testing React components.
* Supertest - For testing REST APIs (Node.js + Express).

Jest

React Testing Library

supertest

23.2.4 Example - Unit Testing a Function (Jest)

Function (sum.js)

```
function sum(a, b) {
   return a + b;
}
module.exports = sum;
```

Unit Test (sum.test.js)

```
const sum = require('./sum');
test('adds two numbers', () => {
    expect(sum(2, 3)).toBe(5);
});
```

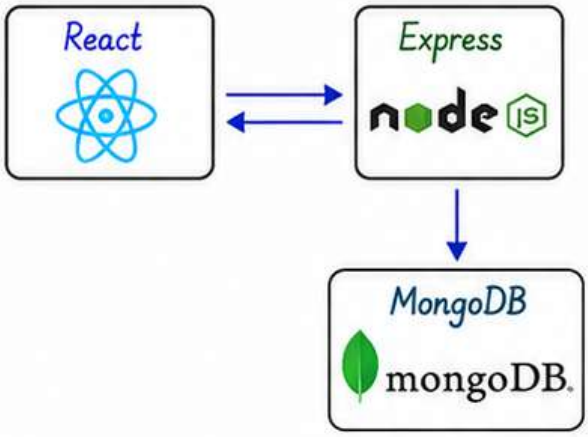
Key Takeaway:

Unit testing helps to test individual units of code in isolation, ensuring that each unit works correctly and the overall application is reliable.

23.3 Integration Testing

23.3.1 What is Integration Testing?

* Integration testing is a type of testing where multiple units or components are combined and tested together.
* In the MERN stack, it ensures that the frontend, backend and database work together as expected.
* It helps to identify issues in the communication between different modules.
* It ensures that data flows correctly between React, Express and MongoDB.



Integration between Components

23.3.2 How Integration Testing Works?

The general flow of integration testing is:

1. Select components or modules to integrate
2. Combine the components and set up the test environment
3. Execute the tests and verify the interaction between modules
4. Identify and fix integration issues
5. Repeat until all integrations work correctly

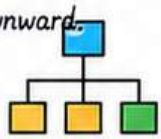
Key Points:

- ☑ It checks the interaction between different units or modules.
- ☑ It ensures data is passed correctly between frontend, backend and database.
- ☑ It helps to detect interface issues and communication problems.
- ☑ It is done after unit testing and before system testing.
- ☑ It improves the reliability and stability of the application.

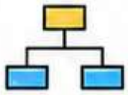
23.3.3 Types of Integration Testing

1. Big Bang Integration Testing
All modules are integrated at once and tested.

2. Top Down Integration Testing
Testing starts from the top module and moves downward.



3. Bottom Up Integration Testing
Testing starts from the lower modules and moves upward.



4. Sandwich Integration Testing
Combination of both top-down and bottom-up approaches.

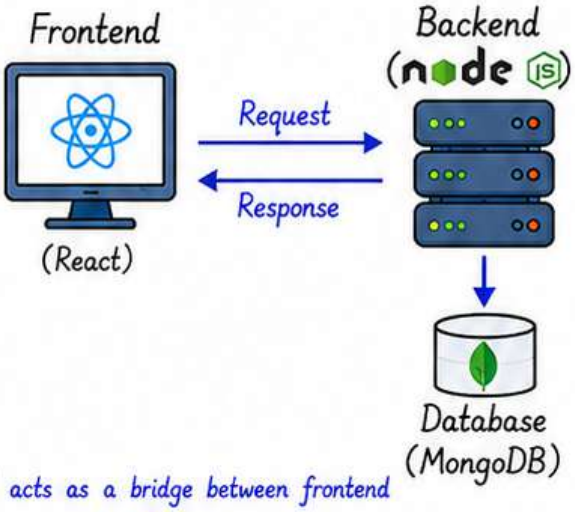
Key Takeaway:

Integration testing ensures that all parts of the MERN application work together seamlessly. It helps to find issues in module interactions and ensures smooth communication between React, Express and MongoDB.

23.4 API Testing

23.4.1 What is API Testing?

- API (Application Programming Interface) testing is the process of testing APIs to ensure they work as expected.
- It checks the functionality, reliability, performance and security of APIs.
- In the MERN stack, APIs are built using Node.js and Express.
- API testing ensures that the backend services return correct responses to the frontend and other systems.



API acts as a bridge between frontend and database.

23.4.2 How API Testing Works?

The general flow of API testing is:

1. Select the API endpoint to test
2. Send the request with required data
3. Receive the response from the API
4. Validate the response (status, data, headers)
5. Log the result and repeat for other endpoints

Key Points:

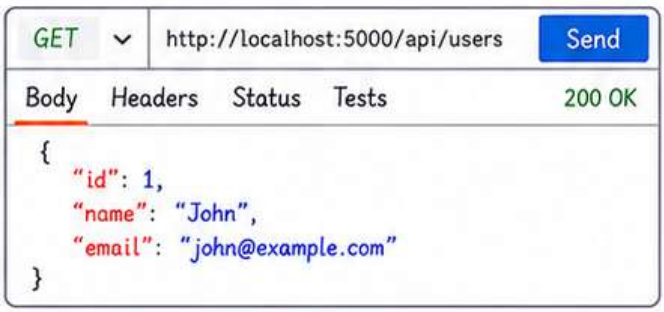
- ☑ API testing checks if the API returns the correct status code and data.
- ☑ It validates request methods like GET, POST, PUT, DELETE.
- ☑ It ensures security, authentication and authorization.
- ☑ It helps in identifying bugs in the backend logic.
- ☑ It can be automated easily.

23.4.3 API Testing Tools

- Postman – Popular tool for testing REST APIs.
- Insomnia – Lightweight tool for API testing.
- Swagger (OpenAPI) – For API documentation and testing.
- Jest + Supertest – For automated API testing in Node.js.
- Cypress – End-to-end testing including API calls.

23.4.4 Example – Testing a GET API (Using Postman)

Step	Action	Example
1	Set the request method	GET
2	Enter the API endpoint URL	http://localhost:5000/api/users
3	Send the request	Click Send
4	Check the response	Status: 200 OK, Data: [...]



Key Takeaway:

API testing ensures that the APIs are reliable, secure, fast and return correct data. It plays a crucial role in building robust and scalable applications.

23.5 Jest

23.5.1 What is Jest?

- Jest is a JavaScript testing framework developed by Facebook.
- It is used to test JavaScript code, especially in React applications.
- Jest is simple, fast and comes with many built-in features.
- It works out of the box, no extra configuration is required for most cases.

Jest

Delightful JavaScript Testing

23.5.2 Features of Jest

- ☑ Zero configuration – works out of the box.
- ☑ Fast and optimized for performance.
- ☑ Built-in test runner.
- ☑ Built-in assertion library (expect).
- ☑ Supports code coverage reporting.
- ☑ Mock functions and timers.
- ☑ Snapshot testing.
- ☑ Easy integration with React.

23.5.3 Basic Syntax in Jest

- test() or it() – Defines a test case.
- expect() – Used to make assertions.
- toBe() – Checks for exact value.
- toEqual() – Checks for deep equality.

Example: sum.js

```
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

Example: sum.test.js

```
const sum = require('./sum');
test('adds two numbers', () => {
  expect(sum(2, 3)).toBe(5);
});
```

23.5.4 Common Matchers in Jest

Matcher	Description	Example
toBe(value)	Checks for exact equality.	expect(a).toBe(5);
toEqual(value)	Checks for deep equality.	expect(obj).toEqual({a: 1});
toBeTruthy()	Checks if value is truthy.	expect(value).toBeTruthy();
toBeFalsy()	Checks if value is falsy.	expect(value).toBeFalsy();
toContain(item)	Checks if item exists in array/string.	expect(arr).toContain('test');

23.5.5 Example – Testing a Function

multiply.js

```
function multiply(a, b) {
  return a * b;
}
module.exports = multiply;
```

multiply.test.js

```
const multiply = require('./multiply');
test('multiplies two numbers', () => {
  expect(multiply(3, 4)).toBe(12);
});
```

Test Result

```
PASS  multiply.test.js
  ✓ multiplies two numbers (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.345 s
```

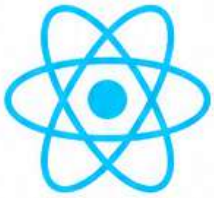
Key Takeaway:

Jest makes testing JavaScript code easy and enjoyable.
It helps in writing reliable, maintainable and bug-free code.

23.6 React Testing Library

23.6.1 What is React Testing Library?

- React Testing Library is a lightweight testing library for React components.
- It focuses on testing components the way users interact with them.
- It encourages good testing practices by avoiding implementation details.
- It works well with Jest as the test runner.



React Testing Library

Simple and user-focused testing for React components

23.6.2 Features of React Testing Library

- ☑ Simple and easy to learn.
- ☑ Uses DOM testing utilities.
- ☑ Encourages testing like a user.
- ☑ Lightweight and fast.
- ☑ Works with Jest.
- ☑ Supports async testing.
- ☑ Great for component testing.
- ☑ Less boilerplate code.

23.6.3 Basic Syntax

- render() - Renders the component for testing.
- screen - Access elements from the rendered output.
- fireEvent - Simulate user events.
- userEvent - Simulate real user interactions.
- waitFor() - Wait for async updates.

Example Component (Button.jsx)

```jsx
import React, { useState } from 'react';
export default function Button() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

23.6.4 Example - Testing a Component

Button.test.js

```jsx
import { render, screen,
       fireEvent } from
'@testing-library/react';
import userEvent from
'@testing-library/user-event';
import Button from './Button';
```

Test Case

```jsx
test('increments count when
 button is clicked', async () => {
  render(<Button />);
  expect(screen.getByText(/Count: 0/))
       .toBeInTheDocument();
  await userEvent.click(
    screen.getByRole('button', {
      name: /increment/i
    })
  );
  expect(screen.getByText(/Count: 1/))
    .toBeInTheDocument();
});
```

Test Result (Jest)

```
PASS  Button.test.js
  ✓ increments count when
    button is clicked (45 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.123 s
Ran all test suites.
```

23.6.5 Common Queries

Query	Description	Example
getByText(text)	Find element by its text content.	screen.getByText('Hello')
getByRole(role)	Find element by role (button, textbox, etc.).	screen.getByRole('button')
getByLabelText(label)	Find form elements by label.	screen.getByLabelText('Name')
queryByText(text)	Like getByText but returns null if not found.	screen.queryByText('Error')
findByText(text)	Finds element asynchronously.	await screen.findByText('Data')

Queries help us find elements on the screen the same way users do.

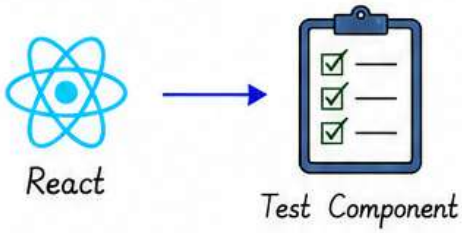
Key Takeaway :

React Testing Library helps write tests that are simple, reliable and focused on user behavior. It promotes maintainable tests and improves confidence in React applications.

23.7 Testing React Components

23.7.1 What is Component Testing?

- Component testing is the process of testing individual React components in isolation.
- It ensures that each component works as expected.
- It helps in finding bugs early and improves code quality.
- React Testing Library and Jest are commonly used for component testing.



Test individual components in isolation to ensure they work correctly.

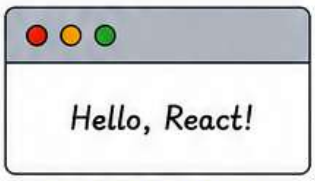
23.7.2 Benefits of Testing React Components

- ☑ Ensures components render correctly.
- ☑ Validates user interactions.
- ☑ Helps detect bugs early.
- ☑ Improves code reliability.
- ☑ Makes refactoring safe and easier.
- ☑ Encourages modular and reusable code.
- ☑ Increases confidence in application changes.

23.7.3 Example Component (Message.jsx)

```jsx
import React from 'react';
export default function Message({ text }) {
  return (
    <div>
      <h2>{text}</h2>
    </div>
  );
}
```

- This is a simple functional component that displays a message passed as a prop called text.



Rendered Output

23.7.4 Writing Test Case for Message Component

Message.test.js

```jsx
import { render, screen } from '@testing-library/react';
import Message from './Message';
test('renders message correctly', () => {
  render(<Message text="Hello, React!" />);
  const heading = screen.getByText('Hello, React!');
  expect(heading).toBeInTheDocument();
});
```

Explanation

- render() renders the Message component.
- screen.getByText() finds the text on the screen.
- toBeInTheDocument() verifies that the text exists in the document.

Test Result (Jest)

```
PASS Message.test.js
  ✓ renders message correctly (18 ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.256 s
Ran all test suites.
```

23.7.5 Testing User Interaction (Button Component)

Button.jsx

```jsx
import React, { useState } from 'react';
export default function Button() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>
        Clicked {count} times
      </button>
    </div>
  );
}
```

Button.test.js

```jsx
import { render, screen, fireEvent }
   from '@testing-library/react';
import Button from './Button';
test('increments count on button click', () => {
  render(<Button />);
  const button = screen.getByRole('button');
  expect(button).toHaveTextContent('Clicked 0 times');
  fireEvent.click(button);
  expect(button).toHaveTextContent('Clicked 1 times');
});
```

Explanation

- getByRole('button') finds the button element.
- Initially the count is 0.
- fireEvent.click(button) simulates a user click.
- After click, count increments to 1.

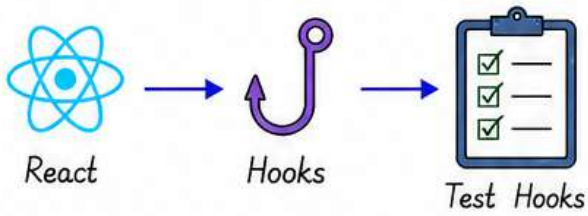
Key Takeaway :

Testing React components ensures they render correctly and behave as expected. It helps in building robust, maintainable and bug-free applications.

23.8 Testing Hooks

23.8.1 What is Hook Testing?

- Hooks allow us to use state and other React features in functional components.
- Testing hooks ensures they work correctly in isolation.
- We use React Testing Library's renderHook() function to test custom hooks.
- It helps in verifying the logic and side effects of hooks.



Test hooks in isolation to ensure their logic works as expected.

23.8.2 Features of Hook Testing

- ☑ Test hooks in isolation.
- ☑ Easy to set up and write tests.
- ☑ Supports async and side effects.
- ☑ Verifies return values and state changes.
- ☑ Works well with Jest.
- ☑ Reduces bugs and improves reliability.
- ☑ Ensures hooks behave as expected.

23.8.3 Basic Syntax

- renderHook() – Renders the hook for testing.
- result.current – Accesses the current value returned by the hook.
- act() – Wraps updates to the hook that trigger state changes.

Example Hook (useCounter.js)

```
import { useState } from 'react';
export function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  return { count, increment, decrement };
}
```

23.8.4 Writing Test Case for useCounter Hook

useCounter.test.js

```
import { renderHook, act } from
  '@testing-library/react';
import { useCounter } from './useCounter';
test('initial value is correct', () => {
  const { result } = renderHook(() =>
    useCounter(5));
  expect(result.current.count).toBe(5);
});
```

Explanation

- renderHook() renders the hook.
- result.current.count gets the current count value.
- expect() verifies the expected value.

Test Result (Jest)

```
PASS useCounter.test.js
  ✓ initial value is correct (12 ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.842 s
Ran all test suites.
```

23.8.5 Testing State Updates

useCounter.test.js

```
test('increment and decrement work', () => {
  const { result } = renderHook(() =>
    useCounter(0));
  act(() => result.current.increment());
  expect(result.current.count).toBe(1);
  act(() => result.current.decrement());
  expect(result.current.count).toBe(0);
});
```

Explanation

- act() is used to wrap state updates.
- This ensures React processes the updates.
- We test both increment and decrement logic.

Test Result (Jest)

```
PASS useCounter.test.js
  ✓ increment and decrement work (15 ms)
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.933 s
Ran all test suites.
```

23.8.6 Common Hooks to Test

Hook	Purpose	What to Test	Example
useState	Manages state in components	Initial state, updates	Counter, Toggle
useEffect	Handles side effects	Effect execution, cleanup	Data fetching, subscriptions
useContext	Accesses context values	Context value changes	Theme, Auth context
useReducer	Manages complex state logic	Actions, state transitions	Todo list, cart reducer
Custom Hooks	Reused logic in hooks	Return values, state updates	useFetch, useCounter

Testing these hooks ensures your logic is reliable and bug-free.

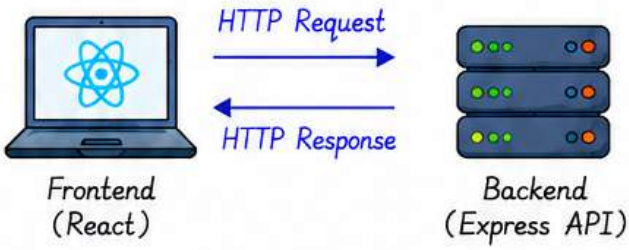
Key Takeaway:

Testing hooks in isolation helps ensure they work as expected.
It improves code quality, prevents bugs and builds confidence in your React applications.

23.9 Testing Express APIs

23.9.1 What is API Testing?

- API testing ensures that backend endpoints work correctly.
- We test routes, responses, status codes and data.
- In Node.js (Express), we use tools like Jest + Supertest.



23.9.2 Why Test Express APIs?

- ☑ Ensures APIs return correct data.
- ☑ Validates proper status codes.
- ☑ Checks error handling.
- ☑ Improves security and reliability.
- ☑ Prevents breaking changes.
- ☑ Ensures database operations work.
- ☑ Builds confidence in backend services.

23.9.3 Example Express Route

```
const express = require('express');
const app = express();
app.use(express.json());

app.get('/api/hello', (req, res) => {
  res.json({ message: 'Hello from API!' });
});

module.exports = app;
```

23.9.4 Tools Used

- ☑ Jest – Test runner
- ☑ Supertest – HTTP assertions
- ☑ Chai – Assertion library
- ☑ Axios – HTTP client (optional)

Supertest

23.9.5 Writing Test Case for GET API

hello.test.js

```
const request = require('supertest');
const app = requrss('./app');

test('GET /api/hello returns message', async () => {
  const res = await request(app).get('/api/hello');
  expect(res.statusCode).toBe(200);
  expect(res.body).toHaveProperty('message');
  expect(res.body.message).toBe('Hello from API!');
});
```

Explanation

- request(app).get('/api/hello') – sends GET request.
- res.statusCode – should be 200 (OK).
- res.body – should contain message.

Test Result (Jest)

```
PASS hello.test.js
  ✓ GET /api/hello returns message (25 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.721 s
Ran all test suites.
```

23.9.6 Testing POST API

Example Route

```
app.post('/api/users', (req, res) => {
  const { name } = req.body;
  if (!name) {
    return res.status(400).json({
      error: 'Name is required'
    });
  }
  res.status(201).json({ id: 1, name });
});
```

Test Case (POST)

```
test('POST /api/users creates user', async () => {
  const res = await request(app)
    .post('/api/users')
    .send({ name: 'React User' });
  expect(res.statusCode).toBe(201);
  expect(res.body).toHaveProperty('id');
  expect(res.body.name).toBe('React User');
});
```

Error Handling Test

```
test('POST /api/users without name', async () => {
  const res = await request(app)
    .post('/api/users')
    .send({});
  expect(res.statusCode).toBe(400);
  expect(res.body.error).toBe('Name is required');
});
```

23.9.7 Common Assertions

Assertion	Description	Example
toBe(200)	Check status code	expect(res.statusCode).toBe(200)
toHaveProperty(key)	Check key exists in response body	expect(res.body).toHaveProperty('id')
toEqual(value)	Check exact response	expect(res.body.name).toBe('John')
toContain(value)	Check value exists in response	expect(res.body.message).toContain('Hello')

23.9.8 Best Practices

- Test all HTTP methods (GET, POST, PUT, DELETE).
- Validate status codes and response data.
- Test both success and failure scenarios.
- Keep tests independent and isolated.

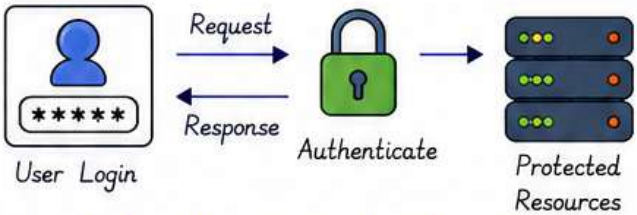
Key Takeaway:

Testing Express APIs ensures your backend is reliable, secure and bug-free. Using Jest + Supertest helps validate endpoints easily and efficiently.

23.10 Authentication Testing

23.10.1 What is Authentication Testing?

- Authentication testing ensures that only authorized users can access protected resources.
- It verifies login, logout, token validation and access control.
- It helps prevent unauthorized access and protects sensitive data.



Authentication testing ensures only valid users can access protected resources.

23.10.2 What We Test in Authentication?

- ☑ User registration and login.
- ☑ Valid credentials – success response.
- ☑ Invalid credentials – failure response.
- ☑ Token / session generation.
- ☑ Access with valid token.
- ☑ Access with invalid / expired token.
- ☑ Logout and token invalidation.
- ☑ Role based access control (RBAC).
- ☑ Password reset / change.
- ☑ Rate limiting / brute force protection.

23.10.3 Example Flow – Login

1. Send credentials (email & password)
2. Server validates credentials
3. If valid, returns JWT token
4. Use token in Authorication header for protected routes
5. Server verifies token and returns response

JWT (JSON Web Token) is commonly used in MERN applications for authentication.

23.10.4 Writing Test Case for Login API

login.test.js (Jest + Supertest)

```
const request = require('supertest');
const app = require('../app');

describe('POST /api/auth/login', () => {
  it('should login with valid credentials', async () => {
    const res = await request(app)
      .post('/api/auth/login')
      .send({ email: 'test@gmail.com', password: '123456'})
      .expect(200);
    expect(res.body).toHaveProperty('token');
    expect(res.body.user).toHaveProperty('id');
    expect(res.body.user.email).toBe('test@gmail.com');
  });
});
```

Explanation

- We send POST request with valid credentials.
- Expect 200 OK status.
- Response should contain a token.
- User details should be present in response.

Test Result (Jest)

```
PASS  login.test.js
  ✓ should login with valid credentials (112 ms)

Test Suites:  1 passed, 1 total
Tests:        1 passed, 1 total
Snapshots:    0 total
Time:         1.342 s
Ran all test suites.
```

23.10.5 Negative Test Cases

Test Case	Description	Expected Result
Invalid Email	Login with wrong email	401 Unauthorized
Invalid Password	Login with wrong password	401 Unauthorized
Missing Fields	Login without email or password	400 Bad Request
Empty Payload	Login with empty body	400 Bad Request
SQL Injection	Attempt SQL injection in email/password	401 Unauthorized
Brute Force	Multiple failed login attempts	429 Too Many Requests

We also test:

- Token expiration
- Token tampering
- Accessing protected routes without token
- Accessing with wrong role

Negative testing helps identify security vulnerabilities.

23.10.6 Example – Testing Protected Route

Protected Route Test (protected.test.js)

```
describe('GET /api/profile (protected)', () => {
  it('should access with valid token', async () => {
    // Login and get token
    const login = await request(app)
      .post('/api/auth/login')
      .send({ email: 'test@gmail.com', password: '123456'});

    const token = login.body.token;

    const res = await request(app)
      .get('/api/profile')
      .set('Authorization', `Bearer ${token}`)
      .expect(200);
    expect(res.body).toHaveProperty('email');
  });
});
```

Explanation

- First, we login and get JWT token.
- Then, we access protected route with Bearer token.
- Expect 200 OK and correct user data.

Test Result (Jest)

```
PASS  protected.test.js
  ✓ should access with valid token (98 ms)

Test Suites:  1 passed, 1 total
Tests:        1 passed, 1 total
Snapshots:    0 total
Time:         1.105 s
Ran all test suites.
```

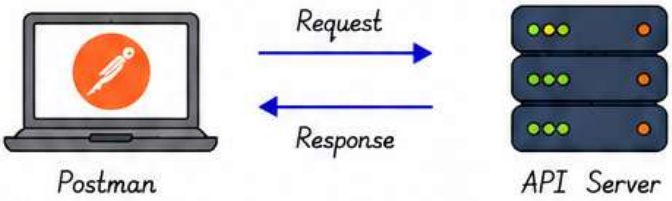
Key Takeaway:

Authentication testing ensures that only authorized users can access your application. Always test positive, negative and edge cases to make your API secure and reliable.

23.11 Postman

23.11.1 What is Postman?

- Postman is a popular API development and testing tool.
- It helps to build, test, document and share APIs easily.
- It supports REST, SOAP and GraphQL APIs.
- Postman is available as Web, Desktop and Chrome Extension.

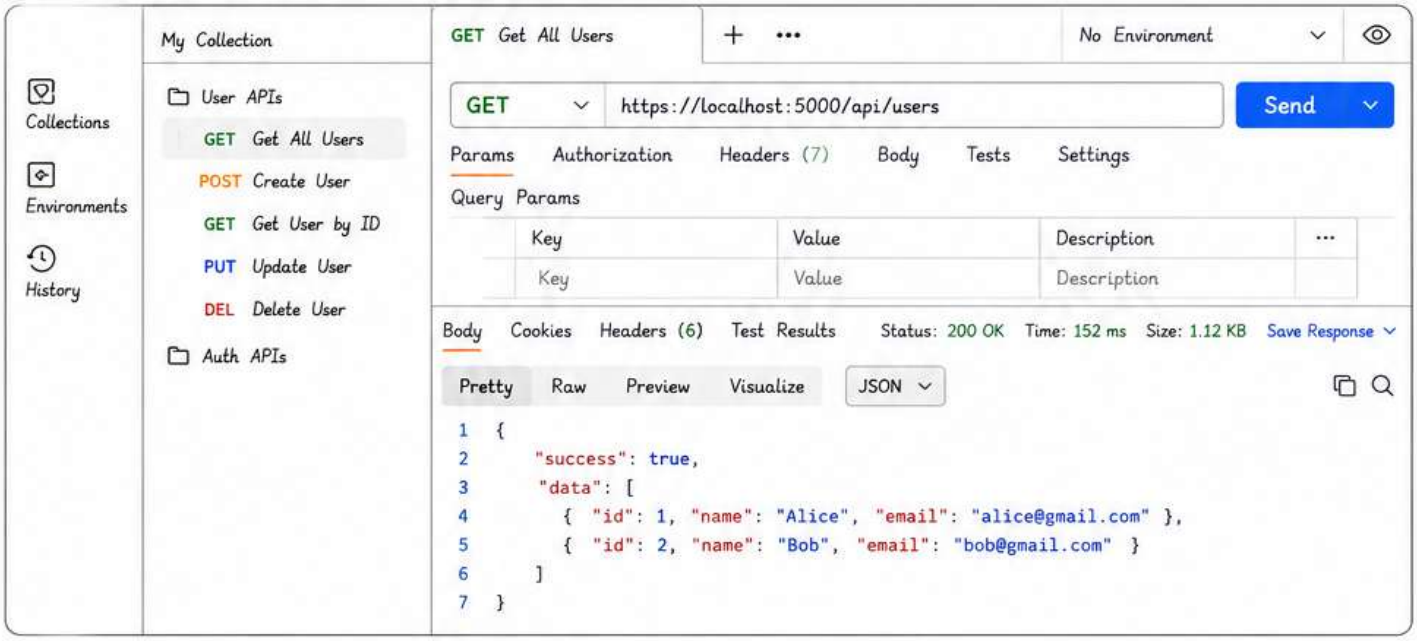


Postman helps us send requests to APIs and view responses in a user-friendly way.

23.11.2 Key Features of Postman

- ☑ Send REST, SOAP and GraphQL requests.
- ☑ Set headers, parameters, body and authorization.
- ☑ View responses in multiple formats (JSON, XML, etc.).
- ☑ Save requests in collections.
- ☑ Organize APIs in folders.
- ☑ Environment variables for dynamic values.
- ☑ Automate API tests using Tests (JavaScript).
- ☑ Mock servers for testing without real backend.
- ☑ Generate API documentation.
- ☑ Collaborate and share APIs.

23.11.3 Postman Interface Overview



23.11.4 Example - GET Request

1. Select GET method
2. Enter URL
3. Click Send
4. View Response

Method : GET
URL : https://localhost:5000/api/users
Response :

```
{
    "success": true,
    "data": [
      { "id": 1, "name": "Alice" },
      { "id": 2, "name": "Bob" }
    ]
}
```

23.11.5 Example - POST Request

Method : POST
URL : https://localhost:5000/api/users
Headers : Content-Type: application/json
Body (raw - JSON):

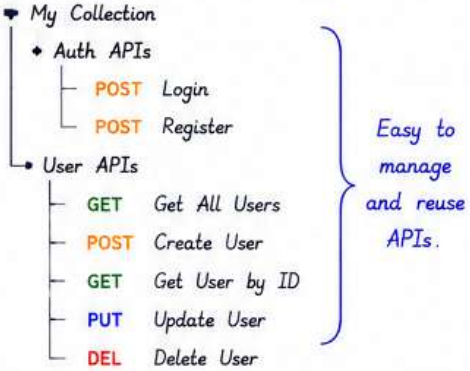
```
{
    "name": "Charlie",
    "email": "charlie@example.com"
}
```

Response :

```
{
  "success": true,
  "message": "User created successfully",
  "data": { "id": 3, "name": "Charlie"
}
```

23.11.6 Postman Collections

- Collections help to group related API requests.
- Example Collection Structure:



Key Takeaway :

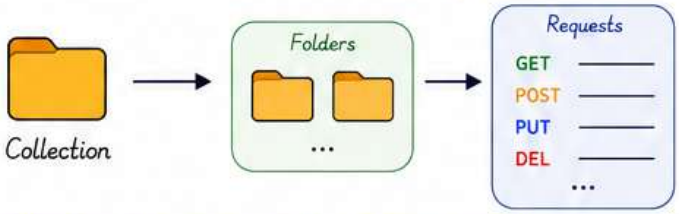
Postman simplifies API development and testing.
It helps in sending requests, validating responses, automating tests and documenting APIs efficiently.

23.12 API Test Collections

23.12.1 What are API Test Collections?

- A Collection in Postman is a group of API requests related to a specific feature or module.
- It helps to organize, manage and run related tests together.
- Collections make API testing structured, reusable and easy to maintain.

Collection Hierarchy



Collections help group requests, set common settings and run them together.

23.12.2 Benefits of Using Collections

- ☑ Organize APIs into modules and folders.
- ☑ Run multiple requests in a defined order.
- ☑ Share collections with team members.
- ☑ Reuse collections across projects.
- ☑ Store environment variables for dynamic values.
- ☑ Add tests to each request for automation.
- ☑ Generate collection documentation.
- ☑ Monitor collection runs and view results.

23.12.3 Example Collection Structure

- Example: E-Commerce API Collection
- It contains folders for different modules like Auth, Users, Products, Orders.
- Each folder contains related API requests.
- This structure keeps APIs well organized and easy to test.

Collection / Folder	Requests (Examples)	Description
E-Commerce API Collection		Root collection for the API
Auth	POST Login POST Register POST Logout	Authentication APIs
Users	GET Get All Users GET Get User by ID PUT Update User DEL Delete User	User management APIs
Products	GET Get All Products GET Get Product by ID POST Create Product PUT Update Product DEL Delete Product	Product management APIs
Orders	GET Get All Orders POST Create Order GET Get Order by ID	Order management APIs

23.12.4 Creating a Collection in Postman

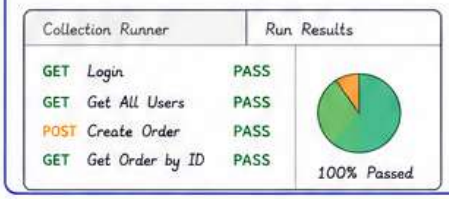
1. Click on 'New' → Select 'Collection'.
2. Give a name to the collection.
3. Create folders inside the collection.
4. Add requests in respective folders.
5. Set environment variables if needed.
6. Save the collection.

23.12.5 Running a Collection

- Click on the collection.
- Choose 'Run'.
- Select environment (if any).
- Set iteration count and delay.
- Click 'Run Collection'.
- View results in the runner window.

23.12.6 Collection Runner (Features)

- Run all requests in order.
- Set environment variables.
- Stop on failure option.
- Parallel or iterative execution.
- Detailed test report and logs.



23.12.7 Sharing a Collection

- Collections can be shared with team members.
- Export collection as JSON.
- Import the collection in another workspace.
- Use Postman Workspaces for collaboration and version control.

23.12.8 Best Practices

- ☑ Use clear collection and request names.
- ☑ Organize requests in meaningful folders.
- ☑ Use environment variables for dynamic data.
- ☑ Write tests for all important endpoints.
- ☑ Keep requests independent and idempotent.
- ☑ Use pre-request scripts for setup.
- ☑ Validate status codes and response data.
- ☑ Regularly run collections to catch issues early.

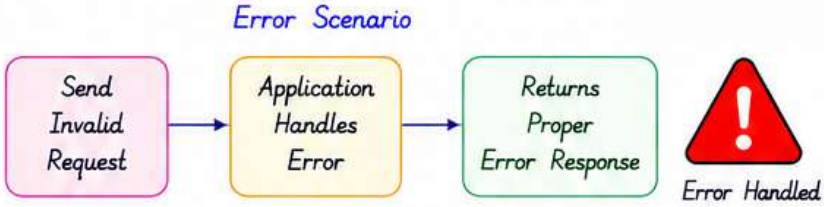
Key Takeaway:

API Test Collections in Postman help organize, automate and manage API tests efficiently. They improve productivity, ensure consistency and make API testing scalable and reliable.

23.13 Error Testing

23.13.1 What is Error Testing?

- Error testing checks how the application handles invalid or unexpected inputs.
- It ensures the application does not crash and shows proper error messages.
- Helps improve reliability, security and user experience.



Error testing ensures the application responds correctly even when something goes wrong.

23.13.2 Why Error Testing is Important?

- ☑ Prevents application crashes.
- ☑ Helps find bugs and edge cases.
- ☑ Improves system stability.
- ☑ Provides meaningful error messages.
- ☑ Enhances security.
- ☑ Improves user experience.
- ☑ Ensures proper error status codes.
- ☑ Builds reliable and production-ready applications.

23.13.3 Common Error Scenarios

Error Scenario	Description	Description	Expected Status Code
Invalid Input	Wrong data type / format	400 Bad Request	400 Bad Request
Missing Fields	Required fields not provided	400 Bad Request	400 Bad Request
Unauthorized Access	No or invalid token	401 Unauthorized	401 Unauthorized
Access Denied	Insufficient permissions	403 Forbidden	403 Forbidden
Resource Not Found	Invalid ID / endpoint	404 Not Found	404 Not Found
Server Error	Unexpected server error		500 Internal Server Error

23.13.4 Example - Testing Error Responses

error.test.js (Jest + Supertest)

```
const request = require('supertest');
const app = require('../app');

describe('Error Handling Tests', () => {
 it('should return 400 for missing fields', async () => {
   const res = await request(app)
     .post('/api/users')
     .send({ name: '' });
   expect(res.status).toBe(400);
   expect(res.body.message).toMatch(/required/i);
 });

 it('should return 404 for invalid route', async () => {
   const res = await request(app).get('/api/invalid');
   expect(res.status).toBe(404);
   expect(res.body.message).toMatch(/not found/i);
 });
});
```

23.13.5 Testing with Postman

- Send request with invalid or missing data.
- Check the status code and error message.
- Verify response structure.
- Test edge cases and negative scenarios.

23.13.6 Example Error Responses

400 Bad Request

```
{
  "success": false,
  "message": "Name is required"
}
```

404 Not Found

```
{
  "success": false,
  "message": "User not found"
}
```

401 Unauthorized

```
{
  "success": false,
  "message": "Invalid token"
}
```

23.13.7 Best Practices

- ☑ Test all negative and edge cases.
- ☑ Verify correct HTTP status codes.
- ☑ Check error message clarity.
- ☑ Validate response structure.
- ☑ Test authentication and authorization errors.
- ☑ Handle unexpected server errors.
- ☑ Automate error tests with Jest / Supertest.
- ☑ Log and monitor errors for debugging.

Key Takeaway :

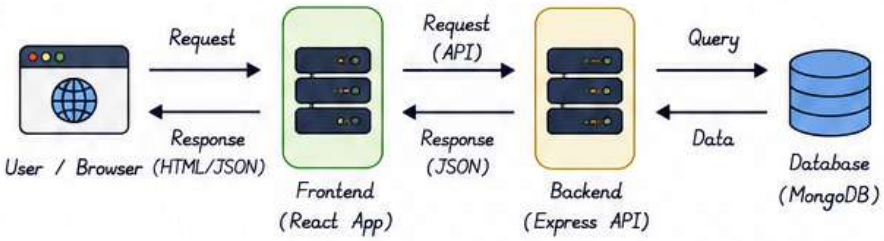
Error testing ensures the application handles invalid inputs and unexpected situations gracefully. It helps build a secure, reliable and user-friendly application.

23.14 End-to-End Testing Concepts

23.14.1 What is End-to-End Testing?

- End-to-End (E2E) testing tests the complete flow of an application from start to finish.
- It simulates real user scenarios across the UI, server, database and external services.
- It helps ensure all parts of the system work together as expected.
- E2E testing focuses on user workflows, not individual components.

End-to-End Test Flow



E2E testing validates the entire application flow as a real user would experience it.

23.14.2 Why is End-to-End Testing Important?

- ☑ Validates real user scenarios.
- ☑ Ensures integration between all components.
- ☑ Detects issues missed by unit or integration tests.
- ☑ Improves confidence in releases.
- ☑ Helps prevent production bugs.
- ☑ Verifies UI, APIs, database and business logic together.
- ☑ Ensures performance and stability in real workflows.

23.14.3 Typical End-to-End Test Flow

1. User Action — User performs an action in the UI.
2. UI Interaction — UI sends a request to the server.
3. API Processing — Server processes the request and accesses DB.
4. Data Retrieval — DB returns data to the server.
5. Response Sent — Server sends response back to the UI.
6. UI Update — UI updates and shows results to the user.

23.14.4 Common E2E Testing Tools

Cypress
- Fast & reliable
- Built for web apps
- Time travel, mocks
- Easy debugging

Playwright
- Microsoft tool
- Cross-browser
- Auto waits
- Parallel execution

Selenium
- Widely used
- Supports many languages
- Good for web automation

Puppeteer
- Built by Chrome team
- Headless Chrome
- Good for testing and scraping

Note:
Cypress is great for local development.
Playwright is good for cross-browser and CI/CD.

23.14.5 Example E2E Test Scenario (Login)

1. Open the application in the browser.
2. Navigate to the login page.
3. Enter valid email and password.
4. Click on Login button.
5. Verify user is redirected to dashboard.
6. Verify welcome message or user details.
7. Logout and verify redirection to login page.

23.14.6 E2E Test Example using Cypress

```
describe('Login Flow E2E Test', () => {
  it('should login with valid credentials', () => {
    cy.visit('http://localhost:3000/login');
    cy.get('input[name="email"]').type('test@gmail.com');
    cy.get('input[name="password"]').type('123456');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
    cy.contains('Welcome, Test User').should('be.visible');
    cy.get('#logout').click();
    cy.url().should('include', '/login');
  });
});
```

23.14.7 E2E Test Coverage

- ☑ User authentication flows
- ☑ CRUD operations
- ☑ Form validations
- ☑ Search and filtering
- ☑ Checkout / payment flow
- ☑ Role-based access
- ☑ Error scenarios
- ☑ Responsive UI checks

23.14.8 Best Practices

- ☑ Write tests for critical user flows only.
- ☑ Keep tests independent and isolated.
- ☑ Use stable selectors (data-testid).
- ☑ Avoid hard coding data; use test data.
- ☑ Use beforeEach and afterEach for setup/cleanup.
- ☑ Run tests in CI/CD pipeline.
- ☑ Use screenshots/videos on failure.
- ☑ Keep tests fast and reliable.
- ☑ Maintain tests regularly as app changes.

23.14.9 E2E vs Other Testing Levels

Testing Level	Scope	Focus	Speed	Example Tools
Unit Testing	Individual functions/units	Logic correctness	Fast	Jest, Mocha
Integration Testing	APIs, DB, modules	Component interaction	Medium	Supertest, Jest
End-to-End Testing	Complete application flow	User workflow	Slow	Cypress, Playwright

23.14.10 Key Takeaway

End-to-End testing validates the entire application from the user's perspective. It ensures that all parts work together, providing a reliable and seamless user experience.

E2E tests build confidence in your application by testing real user journeys.
Use them wisely, keep them stable and focused on critical flows.