Unit Testing with Jest: Notes

1. What is Unit Testing?

- Unit testing is testing individual components or functions in isolation.
- Ensures each part of the application works correctly.

2. Why Use Jest?

- Simple and easy to set up.
- Fast execution with parallel testing.
- Built-in support for mocks, spies, and timers.
- Supports snapshot testing.
- Provides code coverage reports.

3. Installing Jest

$ npm install --save-dev jest

4. Writing a Basic Test

Create `math.js`:

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

Write a test in `math.test.js`:

```
const add = require("./math");
test("adds 2 + 3 to equal 5", () => { expect(add(2, 3)).toBe(5); });
```

Run tests: $ npm test

5. Common Jest Methods:

- test(): Defines a test case.
- expect(value): Asserts the value.
- toBe(value): Checks strict equality.
- toEqual(obj): Checks object equality.
- toContain(value): Checks if array contains a value.

- toThrow(): Checks if function throws an error.

6. Testing Asynchronous Code:

```
const fetchData = () => new Promise((resolve) => { setTimeout(() => resolve("Jest Rocks!"), 1000);
});
test("fetches Jest message", async () => { const data = await fetchData(); expect(data).toBe("Jest Rocks!"); });
```

7. Mocking Functions:

```
const fetchData = jest.fn(() => "Mocked Data");
test("mock function test", () => { expect(fetchData()).toBe("Mocked Data"); });
```

8. Testing Hashed Passwords:

```
const bcrypt = require("bcrypt");
function hashPassword(password) { return bcrypt.hashSync(password, 10); }
function comparePassword(password, hash) { return bcrypt.compareSync(password, hash); }
module.exports = { hashPassword, comparePassword };
```

Write a test in `auth.test.js`:

```
const { hashPassword, comparePassword } = require("./auth");
test("hashes and compares password correctly", () => {
  const password = "securepassword";
  const hashed = hashPassword(password);
  expect(comparePassword(password, hashed)).toBe(true);
});
```

9. Snapshot Testing:

```
import renderer from "react-test-renderer";
import MyComponent from "./MyComponent";
test("renders correctly", () => {
  const tree = renderer.create(<MyComponent />).toJSON();
  expect(tree).toMatchSnapshot();
});
```

10. Generating Code Coverage Reports:

$ npm test -- --coverage

11. Best Practices:

- Keep tests isolated.
- Use mocks for external dependencies.
- Test edge cases and invalid inputs.
- Ensure 100% coverage of critical functions.
- Automate tests using CI/CD pipelines.

12. Conclusion:

Jest is a fast, simple, and powerful testing framework for JavaScript applications. Writing unit tests improves code reliability and reduces bugs. Start testing today!