Purwadhika
Digital Technology School

AI Fullstack Software Development

Ensuring Quality with Test-Driven Development

Job Connector Program

- Intro to TDD
- Mocking dependencies with Jest

Intro to TDD in REST API

TDD brings numerous benefits to REST API development, including requirement validation, enhanced API design, CI/CD integration, regression testing, improved code quality and maintainability, collaboration, and the ability to test integrations with external systems. It helps ensure that your API functions correctly, remains stable, and evolves smoothly over time.
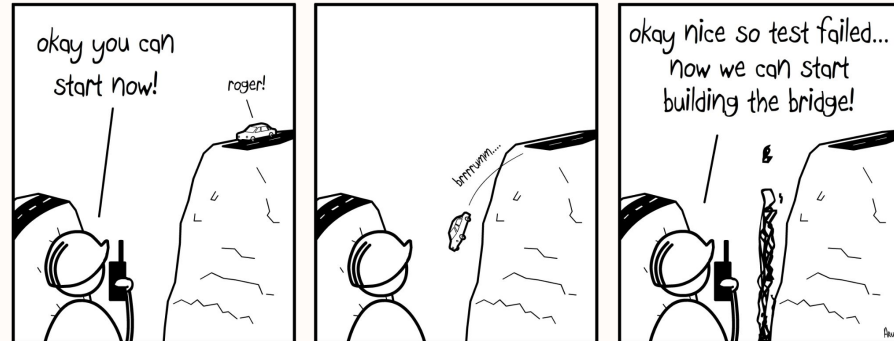
Test Driven Development in REST API

Test-Driven Development (TDD) is particularly valuable in the context of building REST APIs. Here are some reasons why TDD is beneficial for REST API development:

1. **Requirement Validation**: TDD starts with writing tests before writing the actual code. This approach helps you clarify and validate the requirements of your REST API. By explicitly defining the expected behavior through tests, you ensure that your API fulfills the desired functionality.
2. **Enhanced API Design**: Writing tests first forces you to think about the API design from the client's perspective. You define the expected input and output of each API endpoint, which leads to a more thoughtful and well-designed API interface. TDD encourages you to focus on the API contract and how clients will interact with it.

Test Driven Development in REST API

3. **Continuous Integration and Deployment (CI/CD)**: TDD aligns well with CI/CD practices. By having a comprehensive suite of tests, you can automate the testing process, ensuring that new changes or features don't break existing functionality. This allows for a faster and more reliable deployment process, as you can catch issues early in the development cycle.

4. **Regression Testing**: As your REST API evolves, making changes or introducing new features can inadvertently break existing functionality. TDD provides a safety net against regressions. By running your test suite after each change, you can quickly identify if any existing functionality has been affected, ensuring that the API remains stable.

5. **Improved Code Quality and Maintainability:** Following TDD principles encourages modular and loosely coupled code. By writing tests for specific API endpoints, you can ensure that each part of your API functions correctly and independently. This promotes cleaner code architecture and easier maintenance in the long run.

Test Driven Development in REST API

6. **Collaboration and Documentation:** TDD promotes collaboration within development teams. Tests serve as documentation of the API's intended behavior. By having a clear set of tests, different team members can work on different endpoints simultaneously, ensuring that the API as a whole functions correctly.

7. **Mocking and Integration Testing:** TDD facilitates the use of mocks and stubs to isolate dependencies and test specific components or units in isolation. This is particularly useful when testing API integrations with external systems, such as databases or external services.

TDD Example - Setup

In this session, you will learn:

- Testing Express.ts Routes
- Mocking Dependencies with Jest
- Nock to intercept network call

Lets setup your project!

```
mkdir tdd-rest-api
cd tdd-rest-api
npm init --y
npm i express typescript @types/jest @types/express
npm i supertest nodemon ts-node ts-jest
```

TDD Example - Setup

You will need to install the jest and supertest package to use it for testing your Express.ts routes. supertest is a popular library for testing HTTP endpoints in Node.js.

```
mkdir tdd-rest-api
cd tdd-rest-api
npm init --y
npm i express typescript @types/jest @types/express
npm i supertest nodemon ts-node ts-jest
```

TDD Example - Setup

Before we start to code, we must setting ts config and create jest ts config. After that.

After that , add "jest" to package.json to key test in scripts object

After changing the package.json file we can simply run the npx

```
npx tsc --init
npx ts-jest config:init
npx jest
```
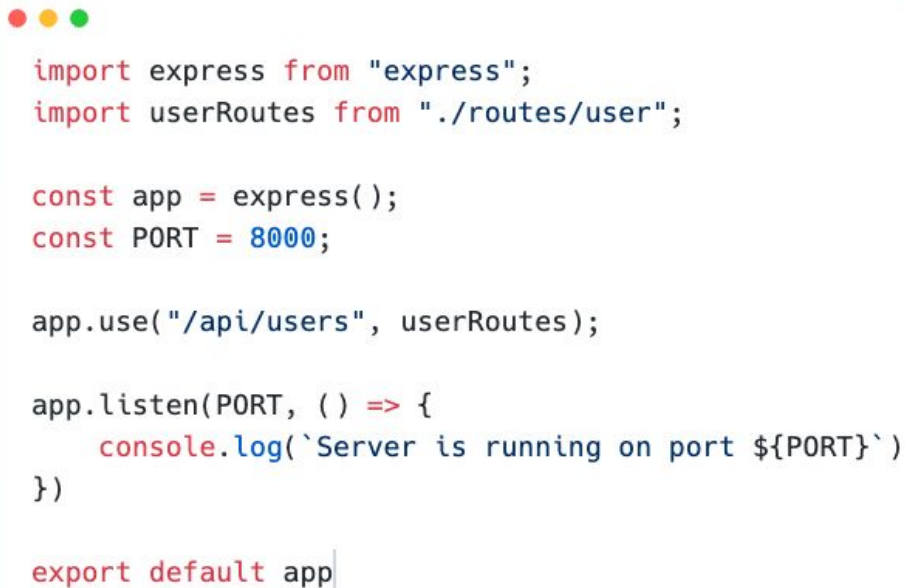
TDD Example - Testing Express.js Routes

In this example, we would like to create a test for routes **/api/users** with **GET** method. From this routes, we know that the expected response is list of users. Create this files in your root of projects:

1. App.ts
2. tests/user.test.ts
3. routes/user.ts

TDD Example - Testing Express.js Routes

Setup app.js in the root of projects. In this example we would run the server on port 4000

```javascript
import express from "express";
import userRoutes from "./routes/user";

const app = express();
const PORT = 8000;

app.use("/api/users", userRoutes);

app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`)
})

export default app
```

TDD Example - Testing Express.js Routes

And then setup the user route to this code. Focus on users where the data is empty array.

```javascript
import express from "express";

const router = express.Router();

router.get("/", (req, res) => {
    res.status(200).send({
        message: "OK",
        users: []
    })
})

export default router
```

TDD Example - Testing Express.js Routes

Create folder and name it as tests. Put your whole test file inside tests folder. Create test file named as user.test.js

In this file we would like to create a test from route /api/users with response are list of users. Try to run this test by writing **`npx jest`** in the command line to execute this test

```javascript
import request from "supertest";
import app from "./../app";

describe("GET /api/users", () => {
    it("should return an array of users", () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body.users).toEqual(["User1", "User2"])
    })
})
```

TDD Example - Testing Express.js Routes

Have you try to run the test? Your test must be run and shown as red right? It happens since you have not write down the codes about the user routes.

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.726 s, estimated 2 s
Ran all test suites.
```

TDD Example - Testing Express.js Routes

Next step, lets update the user routes by following this sample of codes. After that, lets run the `npx jest` command again and check the result!


```
PASS  tests/user.test.ts
  GET /api/users
    √ should return an array of users (48 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.916 s, estimated 2 s
Ran all test suites.
```
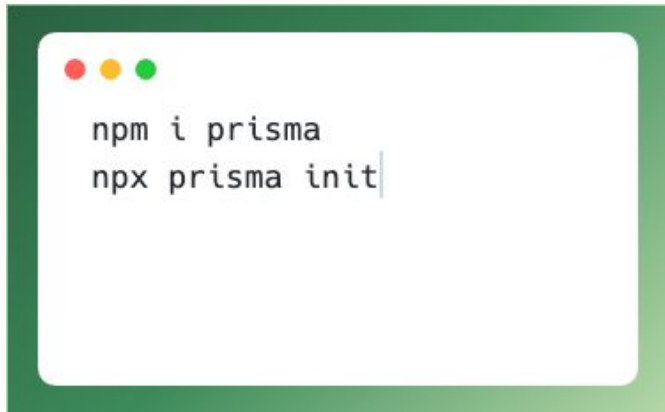
```typescript
import request from "supertest";
import app from "./../app";

describe("GET /api/users", () => {
    it("should return an array of users", () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            message: "OK",
            users: ["User1", "User2"]
        })
    })
})
```
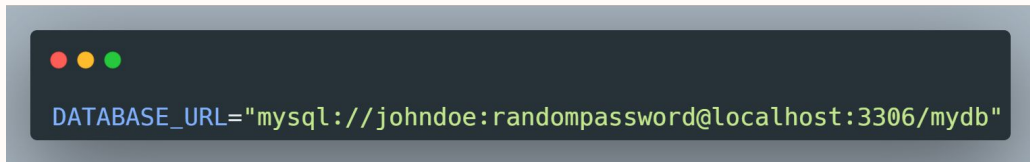
Mocking Dependencies with Jest

In this part you will learn how to mocking dependencies with jest library. In this example, we would mock a sequelize method with jest. But first, lets setup the project with prisma and mysql!

After that setting, the database url in .env file
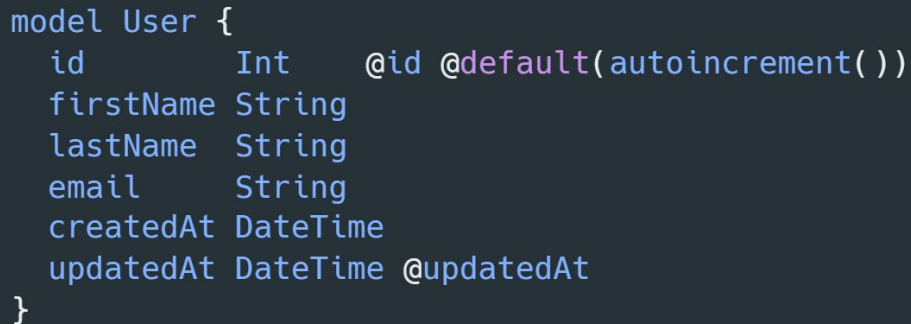
```
npm i prisma
npx prisma init
```

```
DATABASE_URL="mysql://johndoe:randompassword@localhost:3306/mydb"
```

Mocking Dependencies with Jest

Next step, we would like to create table name it as User. Follow this code to generate the User table with several field configurations. Lets add model User in schema.prisma

```
model User {
  id        Int     @id @default(autoincrement())
  firstName String
  lastName  String
  email     String
  createdAt DateTime
  updatedAt DateTime @updatedAt
}
```

Mocking Dependencies with Jest

And Then , Run this migration command on terminal. Add `npx` in the front if the command not work.

```
npx prisma migrate dev
```
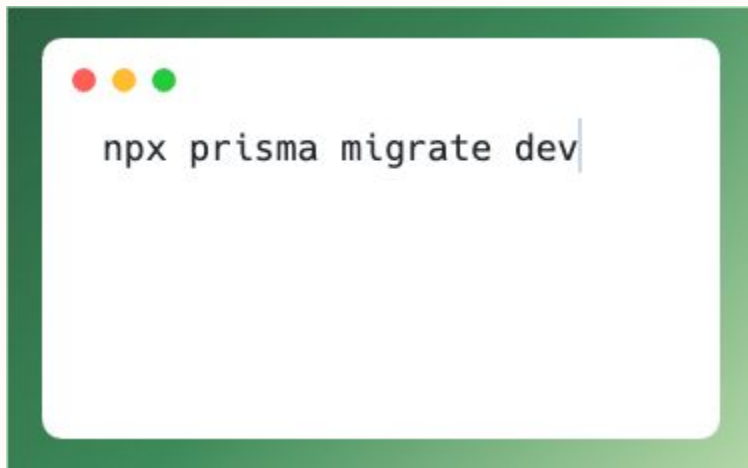
Mocking Dependencies with Jest

Now we are setup the seeding.

Add some setting in package.json first.

And then, Make file seed.ts inside prisma folder and Write this code for seeding.

```json
"prisma": {
    "seed": "ts-node prisma/seed.ts"
}
```

```typescript
import { PrismaClient } from "@prisma/client"

const prisma = new PrismaClient()

async function main() {
    await prisma.user.create({
        data: {
            firstName: "john",
            lastName: "doe",
            email: "john.doe@gmail.com",
            createdAt: new Date(),
        }
    })
    await prisma.user.create({
        data: {
            firstName: "jane",
            lastName: "dine",
            email: "jane.dine@gmail.com",
            createdAt: new Date(),
        }
    })
}

main()
    .catch(e => {
        console.log(e)
        process.exit(1)
    })
    .finally(() => {
        prisma.$disconnect()
    })
```

Mocking Dependencies with Jest

After that run the seed command in terminal.

Mocking Dependencies with Jest

Final step, lets modify our routes/user.ts file. Last time we send the response using array of string, but now we replace that line with query result from database.

Until this step we just completed to setup the api/users route with get method that connected to database.

```typescript
import express, { Router, Request, Response } from "express"
import { PrismaClient } from "@prisma/client"

const prisma = new PrismaClient()

const router: Router = express.Router()

router.get("/", async function (req: Request, res: Response) {

    try {
        const users = await prisma.user.findMany()

        return res.status(200).send({
            message: "OK",
            users: users
        })
    } catch (err: any) {
        return res.status(500).send({
            message: "OK",
            users: JSON.stringify(err)
        })
    }


})

export default router
```

Mocking Dependencies with Jest

Lets focus on our tests file on tests/user.test.js

Remember, last time we change the response from /api/users into data from database. If you try to run the test, it would running into failed. This mean we should adjust the test since there are changes on the routes.

```javascript
import request from "supertest"
import app from "./../app"

describe("GET /api/users", () => {
    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
          "message": "OK",
          "users": ["User 1", "User 2", "User 3"]
        }) // update here
    })
})
```

Mocking Dependencies with Jest

Take a look at the terminal. There are expected and received with some information where expected show 5 line but in received show 1 line. From this information, we know that the test did not run as expected. It happens caused by data / response did not connected to prisma.

```
● GET /api/users › should return an array of users

  expect(received).toEqual(expected) // deep equality

  - Expected  - 5
  + Received  + 1

    Object {
      "message": "OK",
  -   "users": Array [
  -     "User 1",
  -     "User 2",
  -     "User 3",
  -   ],
  +   "users": Array [],
    }

    48 |          const response = await request(app).get("/api/users");
    49 |          expect(response.status).toBe(200)
  > 50 |          expect(response.body).toEqual({
       |                                        ^
    51 |             "message": "OK",
    52 |             "users": ["User 1", "User 2", "User 3"]
    53 |          })

    at tests/user.test.ts:50:31
    at fulfilled (tests/user.test.ts:5:58)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        2.261 s
Ran all test suites.
```

Mocking Dependencies with Jest

Lets take a look, in this code you can see several code has been added into the tests/user.test.js

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {
    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Add PrismaClient into tests/user.test.js to do the query inside the test

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {
    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Create array of objects. This purpose form variable sampleUsers is, to input value from this variable into table users.

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {

    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Take a look at the code **beforeAll,**

This means that beforeAll runs a function before any of the tests in this file run. If the function returns a promise, Jest waits for that promise to resolve before running tests.

On this case, we would like to connect to prisma first.

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {
    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Take a look at the code **beforeEach,**

This means that **beforeEach r**uns a function before each of the tests in this file runs. If the function returns a promise or is a generator, Jest waits for that promise to resolve before running the test. Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before aborting

On this case, we would like to insert sample data into database using createMany with sampleUsers as the sample data.

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {

    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Take a look at the code **afterEach,**

This means that **afterEach** runs a function after each one of the tests in this file completes. If the function returns a promise or is a generator, Jest waits for that promise to resolve before continuing.

On this case, we would like to delete the data from the database. Why do we need to clean up the data? We will find out this later.

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {

    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Take a look at the code **afterAll,**

This means that **afterAll** runs a function after all the tests in this file have completed. If the function returns a promise or is a generator, Jest waits for that promise to resolve before continuing. Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before aborting.

On this case, we would like to close the connection into database since we just finish the test flow

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {

    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Try to run the test again, and you will found it on terminal similar to this result. What happen in this test case?

```
FAIL  tests/user.test.ts
GET /api/users
  × should return an array of users (156 ms)

● GET /api/users › should return an array of users

  expect(received).toEqual(expected) // deep equality

  - Expected  -  3
  + Received  + 16

    Object {
      "message": "OK",
      "users": Array [
  -     "User 1",
  -     "User 2",
  -     "User 3",
  +     Object {
  +       "createdAt": "2024-01-05T02:31:17.000Z",
  +       "email": "john.doe@gmail.com",
  +       "firstName": "john",
  +       "id": 1,
  +       "lastName": "doe",
  +       "updatedAt": "2024-01-05T02:31:17.000Z",
  +     },
  +     Object {
  +       "createdAt": "2024-01-05T02:31:17.000Z",
  +       "email": "jane.dine@gmail.com",
  +       "firstName": "jane",
  +       "id": 2,
  +       "lastName": "dine",
  +       "updatedAt": "2024-01-05T02:31:17.000Z",
  +     },
      ],
    }
```

Mocking Dependencies with Jest

Check out the next line of the result. It show that expected result is not the same as the received value. This could happen since we have not adjust the expected result on line 33.

```
31 |      const response = await request(app).get("/api/users");
32 |      expect(response.status).toBe(200);
> 33 |      expect(response.body).toEqual(["User 1", "User 2", "User 3"]);
     |                     ^
34 |   });
35 | });
36 |

at Object.toEqual (tests/user.test.js:33:27)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.344 s, estimated 2 s
Ran all test suites.
```

Mocking Dependencies with Jest

Since you execute the test before, check out your Users table. Suddenly your data is gone. Is this caused by the error test case? Absolutely not.
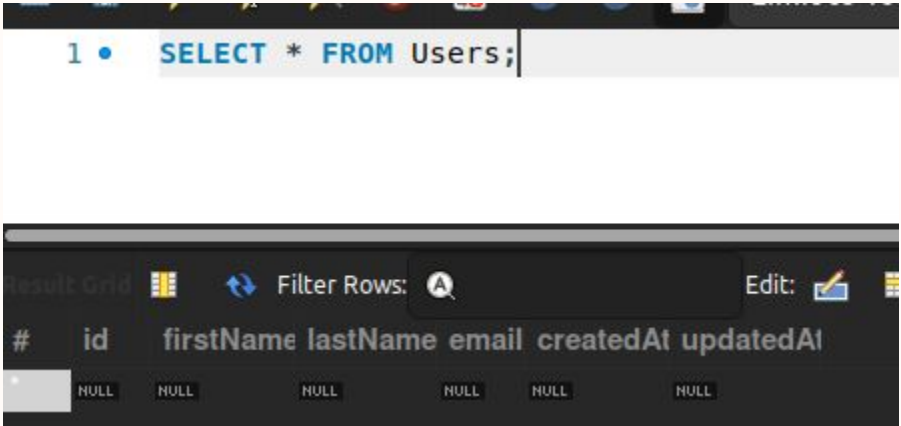
Mocking Dependencies with Jest

Take a look at the code **afterEach,**

On this scope block of code, it is removing all the data from Users table through the user.deleteMany() method.

Still curious about this? Try to remove this line of code and execute the test again. Check out your database. The Users table have two new rows now.

Dont forget to turn it back the code into active again!

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {
    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)
        expect(response.body).toEqual({
            "message": "OK",
            "users": ["User 1", "User 2", "User 3"]
        })
    })
})
```

Mocking Dependencies with Jest

Lets update the expect result from the test. Since we do not need to write down all the field from the data, lets replace **toEqual** with **toMatchObject** method. This matcher to compare the response.body against an array of expected objects. The matcher checks that each object in the response.body has the specified keys and values, but it doesn't require an exact match for additional properties.

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {

    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)

        expect(response.body).toMatchObject({
            "message": "OK",
            "users": sampleUsers.map((item) => {
                return {
                    id: item.id,
                    firstName: item.firstName,
                    lastName: item.lastName,
                    email: item.email,
                }
            })
        })
    })
})
```

Mocking Dependencies with Jest

In this test case we expect that response.body (data that store in database) are same as the variable sampleUsers.

Lets execute the test again!

```javascript
import request from "supertest"
import app from "./../app"
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient()

describe("GET /api/users", () => {

    const sampleUsers = [
        {
            "id": 1,
            "firstName": "john",
            "lastName": "doe",
            "email": "john.doe@gmail.com",
            "createdAt": new Date()
        },
        {
            "id": 2,
            "firstName": "jane",
            "lastName": "dine",
            "email": "jane.dine@gmail.com",
            "createdAt": new Date()
        }
    ]

    beforeAll(async () => {
        await prisma.$connect()
    })

    beforeEach(async () => {
        const users = await prisma.user.findMany()
        if (users.length == 0) {
            await prisma.user.createMany({
                data: sampleUsers
            })
        }
    })

    afterEach(async () => {
        await prisma.user.deleteMany({ where: {} })
    })

    afterAll(async () => {
        await prisma.$disconnect()
    })

    it("should return an array of users", async () => {
        const response = await request(app).get("/api/users");
        expect(response.status).toBe(200)

        expect(response.body).toMatchObject({
            "message": "OK",
            "users": sampleUsers.map((item) => {
                return {
                    id: item.id,
                    firstName: item.firstName,
                    lastName: item.lastName,
                    email: item.email,
                }
            })
        })
    })
})
```

Mocking Dependencies with Jest

The test would go as a green or completed since the expected that we set is match as the expected result. But we have another problem in here. The database is directly impacted by the test. This is not a best practice to write the code if the test that we just create affected into database.

```
PASS  tests/user.test.ts
  GET /api/users
    √ should return an array of users (133 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.935 s, estimated 2 s
Ran all test suites.
```

Mocking Dependencies with Jest

How to handle this test without affecting the database directly? There is the way to handle this issue called as mocking dependencies. In this case we would like to mocking dependencies using Jest.

Mocking Dependencies with Jest

Mocking dependencies is a common practice in testing, and there are several reasons why it is beneficial:

- **Isolation**: By mocking dependencies, you can isolate the code you are testing from the behavior of external components or services. This allows you to focus solely on the functionality you want to test, without worrying about the correctness or availability of the actual dependencies.
- **Control**: Mocking dependencies gives you full control over their behavior during testing. You can define specific responses, simulate error conditions, or manipulate data to create different scenarios that are difficult or impossible to reproduce with the real dependencies. This enables you to thoroughly test various edge cases and ensure your code handles them correctly.
- **Performance**: Some dependencies might have performance implications, such as external services or databases. By mocking these dependencies, you can eliminate the need to make actual network or database calls during testing, resulting in faster test execution and reducing external dependencies in the testing environment.

Mocking Dependencies with Jest

- **Reproducibility**: Dependencies can change over time or have unpredictable behavior, making it challenging to have consistent test results. By using mocks, you can reproduce specific scenarios consistently, making it easier to identify and debug issues.
- **Test Coverage**: Mocking dependencies allows you to test different code paths and error conditions that may be difficult to trigger with real dependencies. For example, you can simulate network errors, invalid responses, or exceptional scenarios to ensure your code handles them correctly.
- **Parallel Development**: Mocking dependencies enables parallel development by allowing developers to work independently on different parts of the codebase. They can create and test their components using mocked dependencies, reducing the need for constant coordination and avoiding conflicts during development.
- **External Service Interaction**: Mocking external services during testing avoids unnecessary interactions and potential side effects, such as making actual purchases or sending real emails. This helps maintain the integrity of the external service and prevents unwanted consequences during testing.

Mocking Dependencies with Jest

To ensure your unit tests are isolated from external factors you can mock Prisma Client, this means you get the benefits of being able to use your schema (*type-safety*), without having to make actual calls to your database when your tests are run.

This guide will cover two approaches to mocking Prisma Client, a singleton instance and dependency injection. Both have their merits depending on your use cases. To help with mocking Prisma Client the jest-mock-extended package will be used.

```
npm i Jest ts-jest jest-mock-extended --save-dev
```

Mocking Dependencies with Jest

The following steps guide you through mocking Prisma Client using a singleton pattern.

1. Create a folder at your projects root called `setup_test`
2. Create a file at your `setup_test` called `client.ts` and add the following code.

   This will instantiate a Prisma Client instance.

setup_test/client.ts

```ts
import { PrismaClient } from '@prisma/client'

const prisma = new PrismaClient()

export default prisma
```

Mocking Dependencies with Jest

3. Next create a file named `singleton.ts` at your `setup_test` and add the following:

```
                                setup_test/singleton.ts

import { PrismaClient } from '@prisma/client'
import { mockDeep, mockReset, DeepMockProxy } from 'jest-mock-extended'

import prisma from './client'

jest.mock('./client', () => ({
  __esModule: true,
  default: mockDeep<PrismaClient>(),
}))

beforeEach(() => {
  mockReset(prismaMock)
})

export const prismaMock = prisma as unknown as DeepMockProxy<PrismaClient>
```
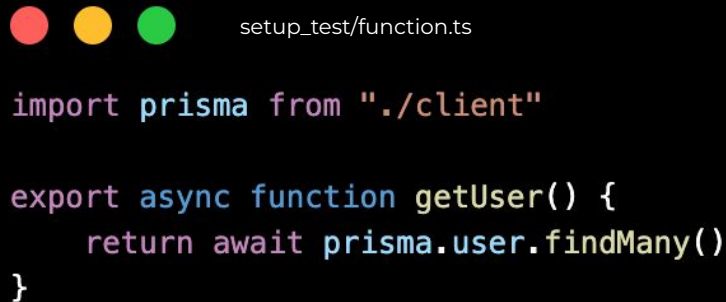
Mocking Dependencies with Jest

4. Next create a file named `function.ts` at your `setup_test` and add the following:

```
setup_test/function.ts

import prisma from "./client"

export async function getUser() {
    return await prisma.user.findMany()
}
```

Mocking Dependencies with Jest

Now we are setup the test.

create a file named `mock.user.test.ts`
at folder `test`

And then, Write this code bellow.

Find out more about test and expect

```
                    test/mock.user.test.ts

import { prismaMock } from '../setup_test/singleton'
import { getUser } from "../setup_test/function"

test('should return an array of users', async () => {
    prismaMock.user.findMany.mockResolvedValue([
        { firstName: "John", lastName: "Doe", email: "john@example.com" },
        { firstName: "Jane", lastName: "Smith", email: "jane@example.com" }
    ])

    await expect(getUser()).resolves.toEqual([
        { firstName: "John", lastName: "Doe", email: "john@example.com" },
        { firstName: "Jane", lastName: "Smith", email: "jane@example.com" }
    ])
})
```

Nock to Intercept Network Call

What if our application server is integrated with another service or application through network?

In this session, you will learn how to create a unit test that routes are integrated to other services through network.
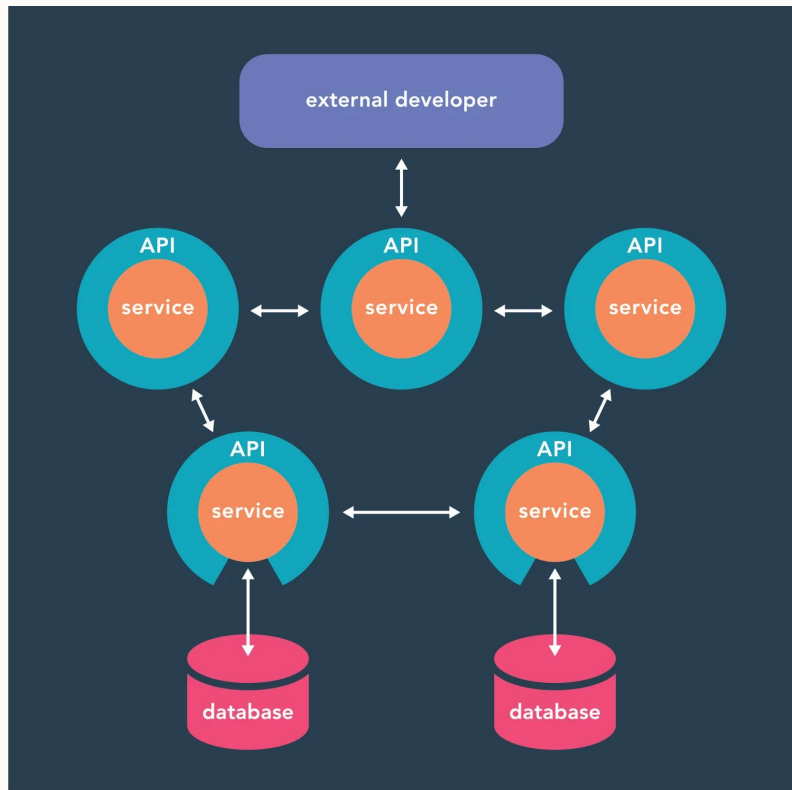
Nock to Intercept Network Call

Lets add axios package in order to call network in you projects.

In this example we would use pokemon api in order to retrieve list of pokemon data using get method

Find out more about pokemon api here

Nock to Intercept Network Call

Lets create another file on routes/pokemon.js

In this file, we would like to create a routes with **GET** method to retrieve pokemon data using "https://pokeapi.co/api/v2/pokemon" this api through axios.

```ts
import express, { Router, Request, Response } from "express";
import axios from 'axios';

const router: Router = express.Router();

router.get('/', async (req: Request, res: Response) => {
    try {
        const { data } = await axios.get("https://pokeapi.co/api/v2/pokemon");
        res.json(data.results);
    } catch (err) {
        console.error("Error fetching pokemons", err);
        res.status(400).send(err);
    }
});

export default router;
```
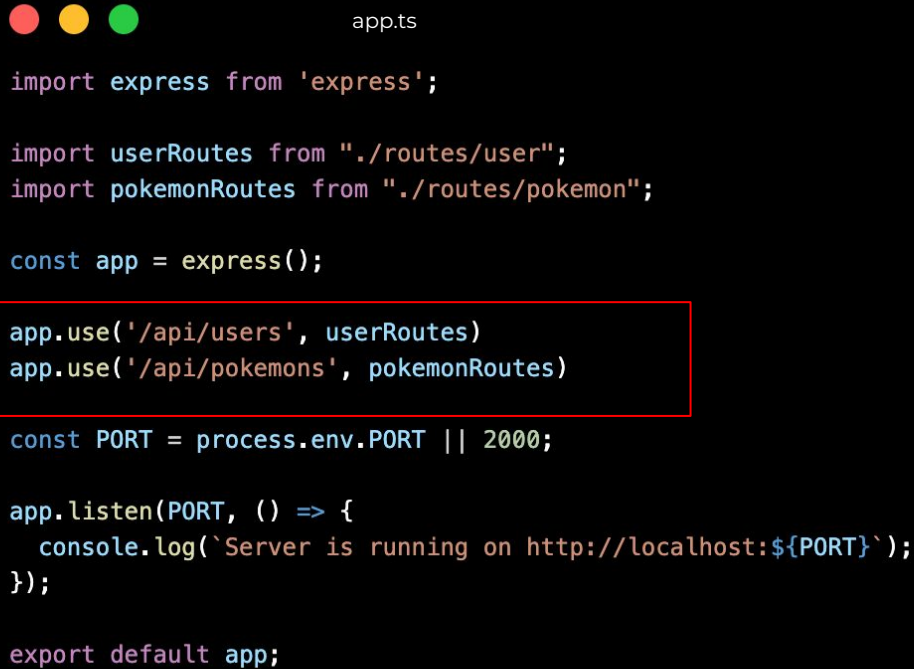
routes/pokemon.ts

Nock to Intercept Network Call

Modify your app.js add routes for pokemons. And now we are ready to create the unit test.

```ts
app.ts

import express from 'express';

import userRoutes from "./routes/user";
import pokemonRoutes from "./routes/pokemon";

const app = express();

app.use('/api/users', userRoutes)
app.use('/api/pokemons', pokemonRoutes)

const PORT = process.env.PORT || 2000;

app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});

export default app;
```

Nock to Intercept Network Call

In this part we would like to create a unit test for routes /api/pokemons with GET method.

We expect this test would give the response status to 200.

Try to run and check the result of the test!

```ts
                                    test/pokemon.test.ts

import request from 'supertest';
import app from '../src/app';

describe("GET /api/pokemons", () => {
    it("should return an array of pokemons", async () => {
        const response = await request(app).get("/api/pokemons")

        expect(response.status).toBe(200)
    })
})
```

Nock to Intercept Network Call

Congratulation the test become a green one. But if you check the result on the terminal, this test performing a direct network call into pokemon api. What if someday this api goes down or having an issue that throws some error? Is your application would keep waiting for response? Or returning the same error as the error message from pokemon api? What if every network call through pokemon api there is charge fee?

Nock to Intercept Network Call

From several question before, direct network call from the unit test is not necessary. The best practice is, every unit test that integrate with other network call, we should mock or intercept the network call.

In this session, you will learn to intercept the network call using nock.

Nock to Intercept Network Call

Nock is HTTP server mocking and expectations library for Node.js

Nock can be used to test modules that perform HTTP requests in isolation.

Nock works by overriding Node's http.request function. Also, it overrides http.ClientRequest too to cover for modules that use it directly.

Now, lets install nock into our project


```
npm install --save-dev nock
```

Nock to Intercept Network Call

Lets modify our tests/pokemon.test.api and
add nock to intercept the network call.

Import nock at the top of the file.

```
test/pokemon.test.ts

import request from 'supertest';
import app from '../src/app';
import nock from 'nock'

describe("GET /api/pokemons", () => {
    it("should return an array of pokemons", async () => {
        const mockResponse = {
            results: [
                { name: 'bulbasaur', url: 'https://pokeapi.co/api/v2/pokemon/1/' },
                { name: 'ivysaur', url: 'https://pokeapi.co/api/v2/pokemon/2/' }
            ]
        }

        nock("https://pokeapi.co").get("/api/v2/pokemon").reply(200, mockResponse)
        const response = await request(app).get("/api/pokemons")

        expect(response.status).toBe(200)
        expect(response.body).toEqual(mockResponse.results)
    })
})
```

Nock to Intercept Network Call

Define variable that would we expect as a mock value from the network call. Lets follow the code on the right side.

```
test/pokemon.test.ts

import request from 'supertest';
import app from '../src/app';
import nock from 'nock'

describe("GET /api/pokemons", () => {
    it("should return an array of pokemons", async () => {
        const mockResponse = {
            results: [
                { name: 'bulbasaur', url: 'https://pokeapi.co/api/v2/pokemon/1/' },
                { name: 'ivysaur', url: 'https://pokeapi.co/api/v2/pokemon/2/' }
            ]
        }

        nock("https://pokeapi.co").get("/api/v2/pokemon").reply(200, mockResponse)
        const response = await request(app).get("/api/pokemons")

        expect(response.status).toBe(200)
        expect(response.body).toEqual(mockResponse.results)
    })
})
```

Nock to Intercept Network Call

Intercept the network call through this code.

Inside nock method add string of url domain and then select what kind of method that would be used following with the specific path. At the end add reply to mock the response.

```ts
                                                    test/pokemon.test.ts
import request from 'supertest';
import app from '../src/app';
import nock from 'nock'

describe("GET /api/pokemons", () => {
    it("should return an array of pokemons", async () => {
        const mockResponse = {
            results: [
                { name: 'bulbasaur', url: 'https://pokeapi.co/api/v2/pokemon/1/' },
                { name: 'ivysaur', url: 'https://pokeapi.co/api/v2/pokemon/2/' }
            ]
        }

        nock("https://pokeapi.co").get("/api/v2/pokemon").reply(200, mockResponse)
        const response = await request(app).get("/api/pokemons")

        expect(response.status).toBe(200)
        expect(response.body).toEqual(mockResponse.results)
    })
})
```

Nock to Intercept Network Call

Try to run the test again. And check out the result. Now your project is successfully intercept the network call.

Thank you