Testing in Node.js + Express

One of the core things to do while writing your code is testing it.

It's highly ignored in most codebases but we're going to try to get close to how testing happens in MERN stack codebases

Goal would be to understand

1. How to test an express backend
2. Mocking, spying, jest, vitest
3. Unit tests vs integration tests vs end to end tests
4. How to integrate testing and coverage in CI/CD

Code for today - https://github.com/100xdevs-cohort-2/week-24-testing/

Testing a simple app

Final code - https://github.com/100xdevs-cohort-2/week-24-testing/tree/main/1-simple-test

Jest is one of many famous testing frameworks in Typescript

- Initialize a simple TS project

```
npm init -y
npx tsc --init
```

- Change rootDir and srcDir

```
"rootDir": "./src",
"outDir": "./dist",
```

- Create `src/index.ts`

```
export function sum(a: number, b: number) {
  return a + b
}
```

- Add ts-jest as a dependency

```
npm install --save-dev ts-jest @jest/globals
```

- Initialize jest.config.ts

```
npx ts-jest config:init
```

- Update package.json

```
"scripts": {
  "test": "jest"
},
```

- Add tests (index.test.ts)

```
import {describe, expect, test} from '@jest/globals';
import {sum} from '../index';

describe('sum module', () => {
  test('adds 1 + 2 to equal 3', () => {
    expect(sum(1, 2)).toBe(3);
  });
});
```

- Run `npm run test`

Testing an express app

Code - https://github.com/100xdevs-cohort-2/week-24-testing/tree/main/2-simple-express-app

Let's say we have an express app that doesnt have any DB connections

- Initialize a simple TS project

```
npm init -y
npx tsc --init
```

- Change rootDir and srcDir

```
"rootDir": "./src",
"outDir": "./dist",
```

- Add dependencies

```
npm install --save-dev ts-jest  @jest/globals @types/express
npm i supertest @types/supertest
npm install express
```

- Initialize jest.config.ts

```
npx ts-jest config:init
```

- Create `src/index.ts`

```
import express from "express";

export const app = express();
app.use(express.json());

app.post("/sum", (req, res) => {
   const a = req.body.a;
   const b = req.body.b;
   const answer = a + b;
```

```
    res.json({
      answer
    })
  });
```

- Update package.json scripts

```
"test": "jest"
```

- Add tests/sum.test.ts

```
import {describe, expect, test, it} from '@jest/globals';
import request from "supertest";
import { app } from "../index"

describe("POST /sum", () => {
  it("should return the sum of two numbers", async () => {
    const res = await request(app).post("/sum").send({
      a: 1,
      b: 2
    });
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(3);
  });

  it("should return the sum of two negative numbers", async () => {
    const res = await request(app).post("/sum").send({
      a: -1,
      b: -2
    });
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(-3);
  });

  it("should return the sum of two zero number", async () => {
    const res = await request(app).post("/sum").send({
      a: 0,
      b: 0
    });
```

```
      expect(res.statusCode).toBe(200);
      expect(res.body.answer).toBe(0);
    });
});
```

- Update jest.config.js

```
/** @type {import('ts-jest').JestConfigWithTsJest} */
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  testMatch: ["<rootDir>/src/tests/**/*.ts"]
};
```

Slightly more complex endpoint

Code - https://github.com/100xdevs-cohort-2/week-24-testing/tree/main/3-express-with-zod

Lets add zod to add solid input validation and return erroneous status codes if the input is incorrect

- Install zod

```
npm install zod
```

- Update `index.ts`

```
import express from "express";
import { z } from "zod";

export const app = express();
app.use(express.json());

const sumInput = z.object({
    a: z.number(),
    b: z.number()
})

app.post("/sum", (req, res) => {
    const parsedResponse = sumInput.safeParse(req.body)

    if (!parsedResponse.success) {
        return res.status(411).json({
            message: "Incorrect inputs"
        })
    }
```

```
    const answer = parsedResponse.data.a + parsedResponse.data.b;

    res.json({
      answer
    })
  });

  app.get("/sum", (req, res) => {
    const parsedResponse = sumInput.safeParse({
      a: Number(req.headers["a"]),
      b: Number(req.headers["b"])
    })

    if (!parsedResponse.success) {
      return res.status(411).json({
        message: "Incorrect inputs"
      })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    res.json({
      answer
    })
  });
```

- Update `sum.test.ts`

```
import {describe, expect, test, it} from '@jest/globals';
import request from "supertest";
import { app } from "../index"

describe("POST /sum", () => {
 it("should return the sum of two numbers", async () => {
    const res = await request(app).post("/sum").send({
      a: 1,
      b: 2
    });
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(3);
```

```
  });

  it("should return 411 if no inputs are provided", async () => {
    const res = await request(app).post("/sum").send({});
    expect(res.statusCode).toBe(411);
    expect(res.body.message).toBe("Incorrect inputs");
  });

});


describe("GET /sum", () => {
 it("should return the sum of two numbers", async () => {
    const res = await request(app)
      .get("/sum")
      .set({
        a: "1",
        b: "2"
      })
      .send();
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(3);
 });

 it("should return 411 if no inputs are provided", async () => {
   const res = await request(app)
     .get("/sum").send();
   expect(res.statusCode).toBe(411);
 });

});
```

Moving from jest to vitest

https://vitest.dev/ is the mildly recent entrant in the testing framework market.

It has a bunch of benefits over jest, specially has great support for TS.

So we'll be moving to vitest for all future tests

It is highly compatable with jest

Link to why vitest - https://vitest.dev/guide/why.html

Simple express project with vitest

Code - https://github.com/100xdevs-cohort-2/week-24-testing/tree/main/4-express-with-vitest

- Init express app

```
npm init -y
npx tsc --init
npm install express @types/express zod
```

- Update tsconfig

```
"rootDir": "./src",
"outDir": "./dist"
```

- Write a simple `src/index.ts` file

```
import express from "express";
import { z } from "zod";

export const app = express();
app.use(express.json());

const sumInput = z.object({
    a: z.number(),
    b: z.number()
})

app.post("/sum", (req, res) => {
    const parsedResponse = sumInput.safeParse(req.body)

    if (!parsedResponse.success) {
        return res.status(411).json({
```

```
            message: "Incorrect inputs"
        })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    res.json({
        answer
    })
});

app.get("/sum", (req, res) => {
    const parsedResponse = sumInput.safeParse({
        a: Number(req.headers["a"]),
        b: Number(req.headers["b"])
    })

    if (!parsedResponse.success) {
        return res.status(411).json({
            message: "Incorrect inputs"
        })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    res.json({
        answer
    })
});
```

We're not doing an app.listen here. This is because we dont want the app to actually start when the tests are running.
Usually you create a `bin.ts` file or `main.ts` file that imports app and actually listens on a port

- Install vitest

```
npm i -D vitest
```

- Add a simple `test/index.test.ts` file

```
import { expect, test } from 'vitest'

test('true === true', () => {
  expect(true).toBe(true)
})
```

- Add a script to test in package.json

```
"test": "vitest"
```

- Add supertest

```
npm i supertest @types/supertest
```

- Update test - Notice all we had to do was update the imports. `vitest` is highly compatible with the jest api

```
import {describe, expect, test, it} from 'vitest';
import request from "supertest";
import { app } from "../index"

describe("POST /sum", () => {
  it("should return the sum of two numbers", async () => {
      const res = await request(app).post("/sum").send({
        a: 1,
        b: 2
      });
      expect(res.statusCode).toBe(200);
```

```
    expect(res.body.answer).toBe(3);
  });

  it("should return 411 if no inputs are provided", async () => {
    const res = await request(app).post("/sum").send({});
    expect(res.statusCode).toBe(411);
    expect(res.body.message).toBe("Incorrect inputs");
  });

});


describe("GET /sum", () => {
 it("should return the sum of two numbers", async () => {
    const res = await request(app)
     .get("/sum")
     .set({
       a: "1",
       b: "2"
     })
     .send();
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(3);
 });

 it("should return 411 if no inputs are provided", async () => {
   const res = await request(app)
    .get("/sum").send();
   expect(res.statusCode).toBe(411);
 });

});
```

Adding a database

There are two approaches to take when you add external services to your backend.

You can

1. Mock out the external service calls (unit tests).
2. Start the external services when the tests are running and stop them after the tests end (integration/end to end tests)

- Add prisma to your codebase

```
npm i prisma
npx prisma init
```

- Add a basic schema in `schema.prisma`

```
model Sum {
  id      Int @id   @default(autoincrement())
  a       Int
  b       Int
  result    Int
}
```

- Generate the client (notice we don't need to migrate since we wont actually need a DB)

```
npx prisma generate
```

- Create `src/db.ts` which exports the prisma client. This is needed because we will be mocking this file out eventually

```
import { PrismaClient } from "@prisma/client";
export const prismaClient = new PrismaClient();
```

- Update `src/index.ts` to store the requests in the db

```
import express from "express";
import { z } from "zod";
import { prismaClient } from "./db";

export const app = express();
app.use(express.json());

const sumInput = z.object({
    a: z.number(),
    b: z.number()
})

app.post("/sum", async (req, res) => {
    const parsedResponse = sumInput.safeParse(req.body)

    if (!parsedResponse.success) {
        return res.status(411).json({
            message: "Incorrect inputs"
        })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    await prismaClient.sum.create({
        data: {
            a: parsedResponse.data.a,
            b: parsedResponse.data.b,
            result: answer
        }
    })

    res.json({
        answer
    })
});

app.get("/sum", (req, res) => {
    const parsedResponse = sumInput.safeParse({
```

```
        a: Number(req.headers["a"]),
        b: Number(req.headers["b"])
    })

    if (!parsedResponse.success) {
        return res.status(411).json({
            message: "Incorrect inputs"
        })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    res.json({
        answer
    })
});
```

- Notice how the tests begin to error out now

Mocking dependencies

Ref - https://vitest.dev/guide/mocking.html

When writing unit tests , you mock out all external service calls.

This means you test the core of your logic, and assume the database calls would succeed.

This is done so tests can run without starting a database / external services

Mocking

Mocking, as the name suggests, means mocking the behaviour of a file/class/variable when tests are running.

Creating a mock

Mocking our prismaClient

To mock out the `prismaClient`, you can add the following code to the top of `index.test.ts`

```
vi.mock('../db', () => ({
  prismaClient: { sum: { create: vi.fn() }}
}));
```

Since we know we are only calling

```
prismaClient.sum.create
```

I have `mocked` the implementation of that function. A `mock` does nothing and returns `undefined` when the function call succeeds.

Try running `npm run test` now. It should succeed

Problems

Can you guess the two problems that exist here?

1. What if I want to use the value that the database call returns? Right now, it will return `undefined` while a real DB call would return some real data
2. I have to constantly keep upgrading the `mock` since in the future I might use the `findOne` function, then might add a new table called `users` ...

Deep mocking

Another way to `mock` variables is to let `vitest` figure out the types and mock out all the attributes of the object being mocked.

For example, the `prismaClient` object has a lot of functions -

```
console.log(Object.keys(prismaClient))
```

What if we could mock out all these keys in a single function call?

Deep mocking

- Install vitest-mock-extended

  ```
  npm i -D vitest-mock-extended
  ```

- Create `__mocks__/db.ts`

  ```
  import { PrismaClient } from '@prisma/client'
  import { beforeEach } from 'vitest'
  import { mockDeep, mockReset } from 'vitest-mock-extended'

  export const prismaClient = mockDeep<PrismaClient>()
  ```

- Remove the `mock` we added in `index.test.ts` , simply add a `vi.mock("../db")`

```
// vi.mock('../db', () => ({
//   prismaClient: { sum: { create: vi.fn() }}
// }));
vi.mock('../db');
```

- Try running the tests

```
npm run test
```

Problem

What if we are using the return value from the database call?

```
import express from "express";
import { z } from "zod";
import { prismaClient } from "./db";

export const app = express();
app.use(express.json());

const sumInput = z.object({
  a: z.number(),
  b: z.number()
})

app.post("/sum", async (req, res) => {
  const parsedResponse = sumInput.safeParse(req.body)

  if (!parsedResponse.success) {
    return res.status(411).json({
```

```
            message: "Incorrect inputs"
        })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    const response = await prismaClient.sum.create({
        data: {
            a: parsedResponse.data.a,
            b: parsedResponse.data.b,
            result: answer
        }
    })

    res.json({
        answer,
        id: response.id
    })
});

app.get("/sum", async (req, res) => {
    const parsedResponse = sumInput.safeParse({
        a: Number(req.headers["a"]),
        b: Number(req.headers["b"])
    })

    if (!parsedResponse.success) {
        return res.status(411).json({
            message: "Incorrect inputs"
        })
    }

    const answer = parsedResponse.data.a + parsedResponse.data.b;

    const response = await prismaClient.sum.create({
        data: {
            a: parsedResponse.data.a,
            b: parsedResponse.data.b,
            result: answer
        }
```

```
    })

    res.json({
        answer,
        id: response.id
    })
});
```

Mocking return values

You can mock the values returned from a `mock` by using `mockResolvedValue`

Update index.test.ts

```
import { prismaClient } from '../__mocks__/db'

prismaClient.sum.create.mockResolvedValue({
  id: 1,
  a: 1,
  b: 1,
  result: 3
});
```

Final index.test.ts

```
import {describe, expect, test, it, vi} from 'vitest';
import request from "supertest";
import { app } from "../index"
import { prismaClient } from '../__mocks__/db'

vi.mock('../db');

describe("POST /sum", () => {
  it("should return the sum of two numbers", async () => {
    prismaClient.sum.create.mockResolvedValue({
      id: 1,
      a: 1,
      b: 1,
      result: 3
    });
```

```
    const res = await request(app).post("/sum").send({
      a: 1,
      b: 2
    });
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(3);
  });

  it("should return 411 if no inputs are provided", async () => {
    const res = await request(app).post("/sum").send({});
    expect(res.statusCode).toBe(411);
    expect(res.body.message).toBe("Incorrect inputs");
  });

});


describe("GET /sum", () => {
 it("should return the sum of two numbers", async () => {
    prismaClient.sum.create.mockResolvedValue({
      id: 1,
      a: 1,
      b: 1,
      result: 3
    });

    const res = await request(app)
      .get("/sum")
      .set({
        a: "1",
        b: "2"
      })
      .send();
    expect(res.statusCode).toBe(200);
    expect(res.body.answer).toBe(3);
 });

 it("should return 411 if no inputs are provided", async () => {
   const res = await request(app)
    .get("/sum").send();
```

```
    expect(res.statusCode).toBe(411);
  });

});
```

We only need to mock in one of the tests because in the second one, control never reaches the place where `id` is needed

Spys vs Mocks

While `mocks` let you `mock` the functionality of a function call, spies let you `spy` on function calls.

Right now, we've mocked out the database call. Which means even if I pass in wrong inputs to the `prismaClient.user.create` function, tests would still pass

Problem

Try flipping the a and b inputs

```
const response = await prismaClient.sum.create({
  data: {
    a: parsedResponse.data.b,
    b: parsedResponse.data.a,
    result: answer
  }
})
```

Try running the tests, they would still work

```
npm run test
```

This means our tests are flaky. They succeed even when the code is incorrect.

Solution

Let's put a `spy` on the `prismaClient.sum.create` function which ensures that the db call inputs are correct

Change the first test to be

```
it("should return the sum of two numbers", async () => {
  prismaClient.sum.create.mockResolvedValue({
    id: 1,
    a: 1,
    b: 1,
    result: 3
  });

  vi.spyOn(prismaClient.sum, "create");

  const res = await request(app).post("/sum").send({
    a: 1,
    b: 2
  });

  expect(prismaClient.sum.create).toHaveBeenCalledWith({
    data: {
      a: 1,
      b: 2,
      result: 3
      }
  })

  expect(res.statusCode).toBe(200);
  expect(res.body.answer).toBe(3);
});
```

Notice that the tests begin to fail

Revert the application logic

Make the application logic right again

```
const response = await prismaClient.sum.create({
  data: {
    a: parsedResponse.data.a,
    b: parsedResponse.data.b,
    result: answer
  }
})
```

Adding a CI/CD pipeline

- Create a CI/CD pipeline that runs `npm run test`
- Create `.github/workflows/test.yml`

```yaml
name: CI/CD Pipeline

on:
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: 20

      - name: Install dependencies
        working-directory: 5-express-vitest-prisma
        run: npm install && npx prisma generate

      - name: Run tests
        working-directory: 5-express-vitest-prisma
        run: npm run test
```

Final code - https://github.com/100xdevs-cohort-2/week-24-testing

PR #1 - https://github.com/100xdevs-cohort-2/week-24-testing/pull/2

PR #2 - https://github.com/100xdevs-cohort-2/week-24-testing/pull/3

Unit tests vs integration tests vs end to end tests

Unit tests

If you mock out external services (DBs, kafka, redist), then you're testing just the functionality of the method. These are called unit tests

Integration tests

If you don't mock out these services but actually start them locally, then it is considered an integration test

End to end tests

If you have a full stack app and you actually open a browser and test things, it's called an end to end test