Unit testing

Code for today - https://github.com/100xdevs-cohort-2/week-25-integ-e2e-tests

Recap of Unit tests

The following is a great example of a unit test - https://github.com/100xdevs-cohort-2/week-24-testing/tree/main/5-express-vitest-prisma

We have used concepts like

1. Mocking
2. mockingResolvedValue
3. Spying

to create unit tests for our simple express app.

Code

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

    if (a > 1000000 || b > 1000000) {
        return res.status(422).json({
            message: "Sorry we dont support big numbers"
        })
    }
    const result = a + b;

    const request = await prismaClient.request.create({
        data: {
            a: a,
            b: b,
            answer: result,
            type: "Sum"
        }
    })
    res.json({ answer: result, id: request.id });
})
```

Test

Copy

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

// mockReturnValue
vi.mock("../db");

describe("Tests the sum function", () => {
    it("Should return 3 when 1 + 2", async () => {
        prismaClient.request.create.mockResolvedValue({
            id: 1,
            answer: 3,
            type: "Sum",
            a: 1,
            b: 2
        })

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

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

        expect(prismaClient.request.create).toHaveBeenCalledWith({
            data: {
                a: 1,
                b: 2,
                type: "Sum",
                answer: 3
            }
        })

        expect(res.body.answer).toBe(3);
        expect(res.body.id).toBe(1);
        expect(res.statusCode).toBe(200);
    })

    it("Should fail when a number is too big", async () => {
        const res = await request(app).post("/sum").send({
            a: 10000000000000,
            b: 2
        })

        expect(res.body.message).toBe("Sorry we dont support big numbers
        expect(res.statusCode).toBe(422);
    })
})
```

Integration tests

While `unit tests` are great, they mock out a lot of external services (DB, cache, message queues ...). This is great for testing the functionality of a function in isolation.

Integration tests are used to test how all `integrated components` work together.

This means you have to start all auxilary services before running your tests and you `DONT` mock out any external service calls

Downsides

1. Slower to execute
2. Add complexity
3. Local development setup if required for a developer (things like docker)

\

Pre-requisites of writing integration tests

Before we write an integration test, we should write the code that

1. Brings up the external services
2. Seeds data in there
3. Brings down the service when the test suite succeeds/fails

Express + prisma app

- Initialize project

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

- Update rootDir and outDir

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

- Install dependencies

```
npm i express @types/express prisma
```

- Initialize prisma

```
npx prisma init
```

- Update schema

```
model Request {
  id          Int      @id @default(autoincrement())
  a           Int
  b           Int
  answer      Int
  type        Type
}

enum Type {
  ADD
  MUL
}
```

- Generate the `prisma client`

```
npx prisma generate
```

- Add a `db.ts` file to export the prisma client

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

export const prismaClient = new PrismaClient();
```

- Write the express logic (index.ts)

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

export const app = express();

app.use(express.json());

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

    if (a > 1000000 || b > 1000000) {
        return res.status(422).json({
            message: "Sorry we dont support big numbers"
        })
    }
    const result = a + b;

    const request = await prismaClient.request.create({
        data: {
            a: a,
            b: b,
            answer: result,
            type: "ADD"
        }
    })

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

- Create `bin.ts` to listen on a port while starting the server

```
import { app } from "./index";

app.listen(3000);
```

- Try running the app locally

```
tsc -b
node dist/bin.js
```

You will notice the request fails because we've not yet started the DB locally

Starting the DB

Until now, we've used one of the following ways to start a DB

1. Start one on https://neon.tech/ / aieven
2. Start it locally using docker

```
docker run -p 5432:5432 -e POSTGRES_PASSWORD=mysecretpassword  -d postgr
```

Let's use the second one to start a database and then hit our backend

- Make sure docker is running
- Start a DB locally

```
docker run -p 5432:5432 -e POSTGRES_PASSWORD=mysecretpassword  -d postgr
```

- Update .env

```
DATABASE_URL="postgresql://postgres:mysecretpassword@localhost:5432/post
```

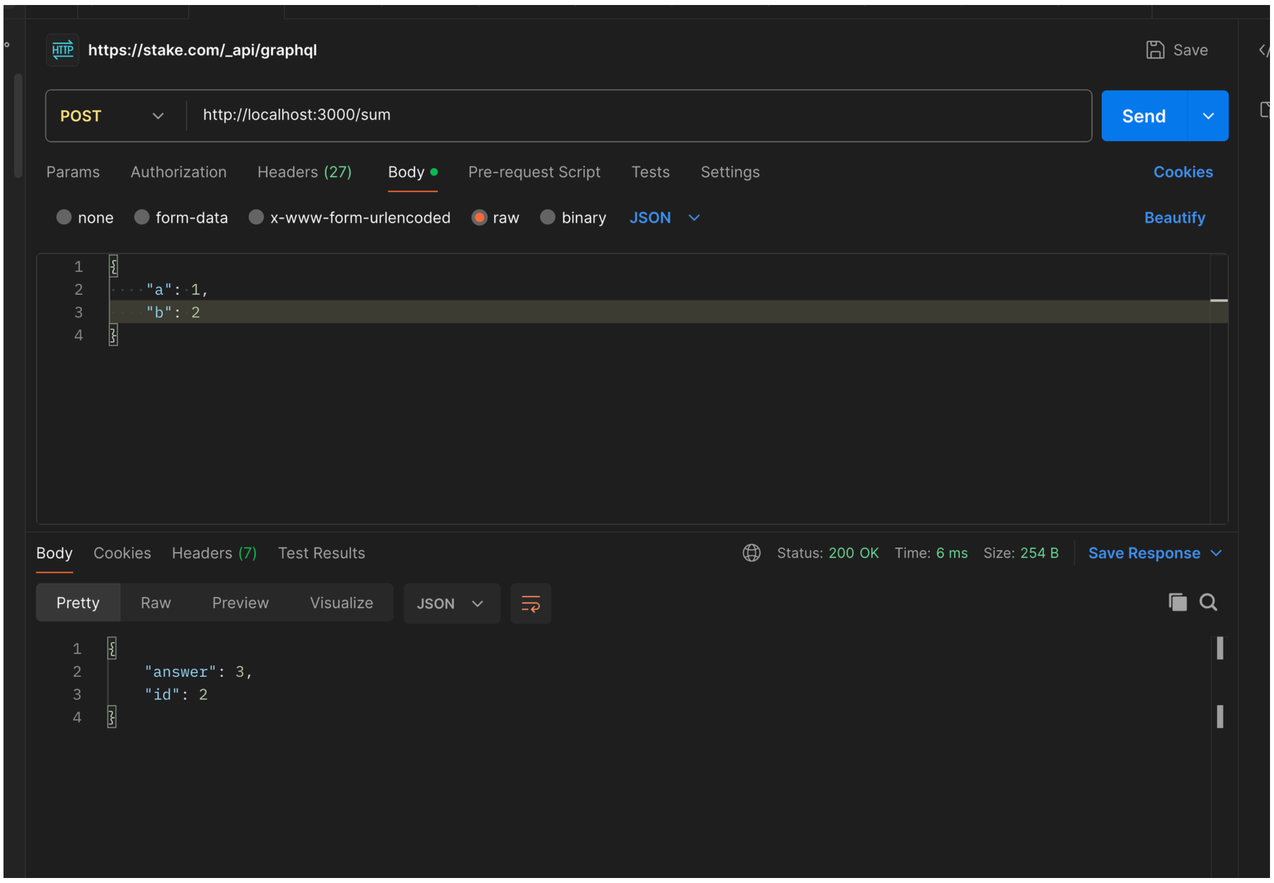
- Migrate the DB

```
npx prisma migrate dev
```

- Generate the client

```
npx prisma generate
```

- Send a request from POSTMAN


https://stake.com/_api/graphql

POST http://localhost:3000/sum

```
{
    "a": 1,
    "b": 2
}
```

Status: 200 OK Time: 6 ms Size: 254 B

```
{
    "answer": 3,
    "id": 2
}
```

- Check the DB and ensure data is going in

```
npx prisma studio
```

What we did right now is a `manual integration test`

We now need to automate this thing and do the same programatically

Let’s take down the database for now -

```
docker ps
docker kill container_id
```

Bootstraping Integration tests in vitest

- Add vitest as a dependency

```
npm i vitest
```

- Add a docker-compose with all your external services

```
version: '3.8'
services:
  db:
    image: postgres
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=mysecretpassword
    ports:
      - '5432:5432'
```

- Crate `src/tests/helpers/reset-db.ts`

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

const prisma = new PrismaClient()

export default async () => {
  await prisma.$transaction([
    prisma.request.deleteMany(),
  ])
}
```

- Create a new script `scripts/run-integration.sh`

```
docker-compose up
```

- Bring in `wait-for-it.sh` locally in `scripts/wait-for-it.sh`

```
curl https://raw.githubusercontent.com/vishnubob/wait-for-it/master
```

> 💡 On a mac, you might need this to run the following command -
>
> `brew install coreutils && alias timeout=gtimeout`
>
> Ref - https://github.com/vishnubob/wait-for-it/issues/108

- Make the scripts executable

```
chmod +x scrip
```

- Update `run-integration.sh`

```
docker-compose up -d
echo '🟡 - Waiting for database to be ready...'
./wait-for-it.sh "postgresql://postgres:mysecretpassword@localhost:5432/
npx prisma migrate dev --name init
npm run test
docker-compose down
```

- Update `package.json`

```
"scripts": {
    "test": "vitest",
  "test:integration": "./scripts/run-integration.sh"
},
```

Adding integration tests

- Install supertest

```
npm i -D supertest @types/supertest
```

- Add `src/tests/sum.test.ts`

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

describe("POST /sum", () => {
    it("should sum add 2 numbers", async () => {
        const { status, body } = await request(app).post('/sum').send({
            a: 1,
            b: 2
        })
        expect(status).toBe(200);
        expect(body).toEqual({ answer: 3, id: expect.any(Number) });
    });
})
```

- Try running the tests

```
npm run test
```

beforeEach and beforeAll function

beforeEach

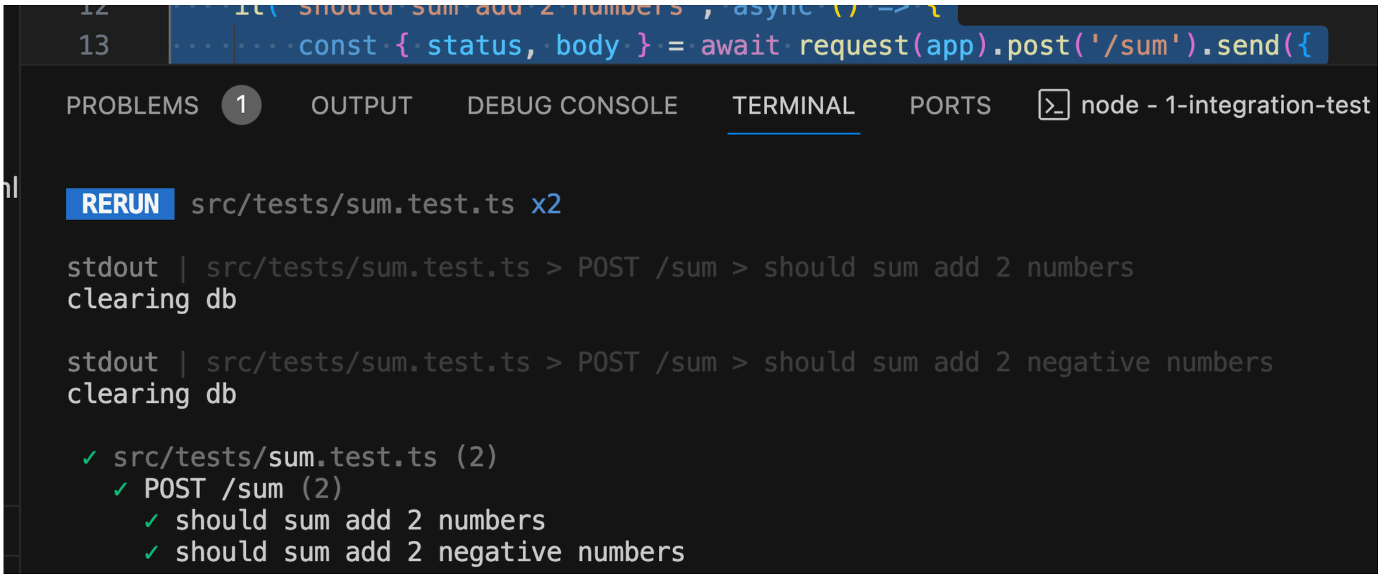
If you want to clear the DB between tests/descibe blocks, you can use the `beforeEach` function

```
import { beforeEach, describe, expect, it } from "vitest";
import { app } from "..";
import request from "supertest";
import resetDb from "./helpers/reset-db";

describe("POST /sum", () => {
    beforeEach(async () => {
        console.log("clearing db");
        await resetDb();
    });

    it("should sum add 2 numbers", async () => {
        const { status, body } = await request(app).post('/sum').send({
            a: 1,
            b: 2
        })
        expect(status).toBe(200);
        expect(body).toEqual({ answer: 3, id: expect.any(Number) });
    });

    it("should sum add 2 negative numbers", async () => {
        const { status, body } = await request(app).post('/sum').send({
            a: -1,
            b: -2
        })
        expect(status).toBe(200);
        expect(body).toEqual({ answer: -3, id: expect.any(Number) });
    });
})
```



```
const { status, body } = await request(app).post('/sum').send({
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS node - 1-integration-test

```
 RERUN  src/tests/sum.test.ts x2

stdout | src/tests/sum.test.ts > POST /sum > should sum add 2 numbers
clearing db

stdout | src/tests/sum.test.ts > POST /sum > should sum add 2 negative numbers
clearing db

 ✓ src/tests/sum.test.ts (2)
   ✓ POST /sum (2)
     ✓ should sum add 2 numbers
     ✓ should sum add 2 negative numbers
```

beforeAll

If you want certain code to run before all the tests (but not before every individual test), you can use the `beforeAll` function

Copy

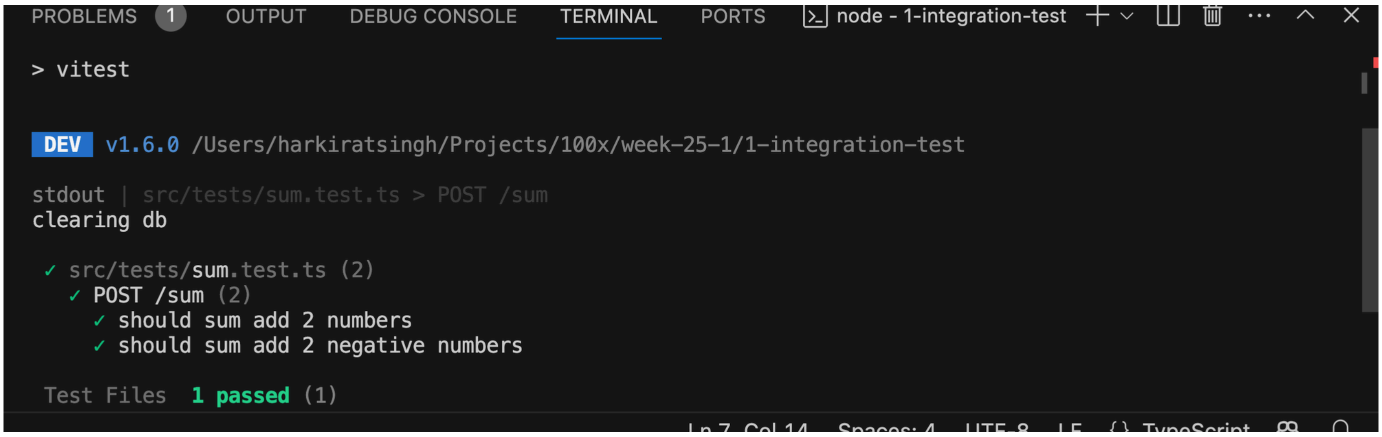
```
import { beforeAll, beforeEach, describe, expect, it } from "vitest";
import { app } from "..";
import request from "supertest";
import resetDb from "./helpers/reset-db";

describe("POST /sum", () => {
    beforeAll(async () => {
        console.log("clearing db");
        await resetDb();
    });

    it("should sum add 2 numbers", async () => {
        const { status, body } = await request(app).post('/sum').send({
            a: 1,
            b: 2
        })
        expect(status).toBe(200);
        expect(body).toEqual({ answer: 3, id: expect.any(Number) });
    });

    it("should sum add 2 negative numbers", async () => {
        const { status, body } = await request(app).post('/sum').send({
            a: -1,
            b: -2
        })
```

```
            expect(status).toBe(200);
            expect(body).toEqual({ answer: -3, id: expect.any(Number) });
        });
    })
```



```
> vitest

 DEV  v1.6.0 /Users/harkiratsingh/Projects/100x/week-25-1/1-integration-test

stdout | src/tests/sum.test.ts > POST /sum
clearing db

 ✓ src/tests/sum.test.ts (2)
   ✓ POST /sum (2)
     ✓ should sum add 2 numbers
     ✓ should sum add 2 negative numbers

 Test Files  1 passed (1)
```

CI/CD pipeline

Final code - https://github.com/100xdevs-cohort-2/week-25-integ-e2e-tests

- Add a .env.example

```
DATABASE_URL="postgresql://postgres:mysecretpassword@localhost:5432/post
```

- Add `.github/workflows/test.yml`

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Set up Docker Compose
      uses: docker/setup-qemu-action@v2

    - name: Ensure Docker Compose is available
      run: docker-compose version

    - name: Copy .env.example to .env
      run: cp ./1-integration-test/.env.example ./1-integration-test/.en

    - name: Run integration script
      run: cd 1-integration-test && npm run test:integration
```

End to end tests

Until now, we're not tested our frontend + backend together.

End to end tests let you `spin up a browser` and test things like an end user.

Good reference video - https://www.cypress.io/

There are many frameworks that let u do browser based testing

1. Cypress
2. Playwright
3. nightwatchjs

We'll be using `cypress`

Cypress

Ref - https://www.cypress.io/

Let's create a simpe test for https://app.100xdevs.com/

- Init ts project

```
npm init -y
npx tsc --init
mkdir src
```

- Change rootDir, outDir

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

- Install cypress (You might face issues here if u dont have a browser)
  Linux pre-requisites here - https://docs.cypress.io/guides/getting-started/installing-cypress

Copy
```
npm install cypress --save-dev
```

- Bootstrap cypress

Copy
```
npx cypress open
```

- Select default example to start with
- Delete `2-advanced-examples`
- Try running the `todo` test

Copy
```
npx cypress run --browser chrome --headed
```

- Update the todo test

Copy
```
describe('Testing app', () => {
  beforeEach(() => {
    cy.visit('https://app.100xdevs.com')
  })

  it('is able to log in', () => {
    cy.contains('Login').should('exist')
    cy.contains('Login').click()
    cy.contains('Signin to your Account').should('exist', { timeout: 100
    cy.get('#email').type('harkirat.iitr@gmail.com');

    // Fill in the password field
    cy.get('#password').type('123random');

    cy.get('button').eq(4).click()

    cy.contains('View Content').should("exist", {timeout: 10000})
  })
})
```