Period 2: Learning goals

Note: This description is too big for a single exam-question. It will be divided up into separate questions for the exam

Explain Pros & Cons in using Node.js + Express to implement your Backend compared to a strategy using, for example, Java/JAX-RS/Tomcat

Pro:

- Det hele skrevet i samme sprog (TypeScript)
- Letlæselig kode (udvikler skal ikke sætte sig ind i et andre sprog løbende)
- Hurtigere at sætte en basis server op
- NPM simplificerer overblikket over hvilke pakker der bruges og muliggør nemmere download af andre pakker

Cons:

- Hvis man ikke benytter et typestærkt sprog (TS) så kan der opstå uventede fejl
- Skal selv implemterere meget debugging

Explain the difference between *Debug outputs* and *ApplicationLogging*. What's wrong with console.log(..) statements in our backend code?

Applicationslogging : Prod miljø, gemmer i en seperat fil et sted i src

Debug : Dev miljø,

- Console.log() kan ses af slutbrugeren på en prod server
- Debug() kan kun ses i dev miljø

Package.json og middleware logger fil

Demonstrate a system using application logging and environment controlled debug statements.

Under middleware logger.ts

Nedenstående logger bliver bestemt af env variblerne. Hvis env er sat til production vil den logge og ellers vil den debug i dev.

```
import winston from 'winston';
import path from "path"

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json()
});

const format = winston.format.combine(
  winston.format.timestamp({ format: 'DD-MM-YYYY HH:mm:ss:ms' }),
  winston.format.colorize({ all: true }),
  winston.format.printf(
    (info) => `${info.timestamp} : '${info.level}' ${info.message}`,
  ),
)

if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    handleExceptions: true,
    format
  }));
} else {
  logger.add(new winston.transports.File({ filename: path.join('logs', 'error.log'), level: 'error', handleExceptions: true }))
```

```
  logger.add(new winston.transports.File({ filename: path.join('logs', 'combined.log') }))
}

export const stream = {
  write: (message: string) => {
    logger.info(message)
  },
};

export default logger;
```

App.ts

```
import logger, { stream } from "./middleware/logger";
const morganFormat = process.env.NODE_ENV == "production" ? "combined" : "dev";
app.use(require("morgan")(morganFormat, { stream }));
app.set("logger", logger);
```

Explain, using relevant examples, concepts related to testing a REST-API using Node/JavaScript/Typescript + relevant packages

Relevant testing packages:

- Mocha
- Chai
- Nock
- Chai as promised

Explain a setup for Express/Node/Test/Mongo-DB/GraphQL development with Typescript, and how it handles "secret values", debug, debug-outputs, application logging and testing.

- Debugging og logging winston/morgan
- Secret values gemmes i ENV filen
- Testing Mocha, Chai, Nock

Explain a setup for Express/Node/Test/Mongo-DB/GraphQL development with Typescript. Focus on how it uses Mongo-DB (how secret values are handled, how connections (production or test) are passed on to relevant places in code, and if use, how authentication and authorization is handled

- Secret values i env filen (connection string)
- Key/value pair som sættes til prod eller dev i db connection
- DB connection gemmes i config folder, db sættes der.
- Facade tager en DB med i constructor som parameter, ved prod bruges db med som paramterer og ved test gives in-memory db med.
- friendsRouteAuth tjekker i env om "SKIP_AUTHENTICATION" er sat

```
// ALL ENDPOINTS BELOW REQUIRES AUTHENTICATION

import authMiddleware from "../middleware/basic-auth";
const USE_AUTHENTICATION = !process.env["SKIP_AUTHENTICATION"];

if (USE_AUTHENTICATION) {
  router.use(authMiddleware);
}
```

Explain, preferably using an example, how you have deployed your node/Express applications, and which of the Express Production best practices you have followed.

?

Explain possible steps to deploy many node/Express servers on the same droplet, how to deploy the code and how to ensure servers will continue to operate, even after a droplet restart.

?

Explain, your chosen strategy to deploy a Node/Express application including how to solve the following deployment problems:

- Ensure that you Node-process restarts after a (potential) exception that closed the application
- Ensure that you Node-process restarts after a server (Ubuntu) restart
- Ensure that you can run “many” node-applications on a single droplet on the same port (80)

??

Explain, using relevant examples, the Express concept; middleware.

Middleware er en fællesbetegnelse for computerprogrammer, der ikke er en del af computerens styresystem, men dog stiller muligheder til rådighed for andre programmer. Software, der fungerer som en bro mellem et operativsystem eller database og applikationer, især på et netværk. Hvad der er middleware er ret flydende. Et krav er dog, at programmet stiller et API til rådighed, så systemet kan tilpasses et konkret formål. Mange programmer, der afvikles på en server kan betragtes som middleware.

Middleware er fx: en cors.ts i startkoden.

```
import express from "express";
const app = express();

export default app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept"
  );
  next();
});
```

Explain, conceptually and preferably also with some code, how middleware can be used to handle problems like logging, authentication, cors and more.

A middleware is basically a function that will the receive the Request and Response objects, just like your route Handlers do. As a third argument you have another function which you should call once your middleware code completed.

https://expressjs.com/en/guide/using-middleware.html

Se middleware mappe i src folder

```
import auth from "basic-auth";
import { Request, Response } from "express";
import FriendFacade from "../facades/friendFacade";

let facade: FriendFacade;

const authMiddleware = async function (
  req: Request,
  res: Response,
  next: Function
) {
  if (!facade) {
    facade = new FriendFacade(req.app.get("db")); //Observe how you have access to the global app-object via the request object
```

```
  }
  var credentials = auth(req);
  if (credentials && (await check(credentials.name, credentials.pass, req))) {
    next();
  } else {
    res.statusCode = 401;
    res.setHeader("WWW-Authenticate", 'Basic realm="example"');
    res.end("Access denied");
  }
};

async function check(userName: string, pass: string, req: any) {
  //if (user && compare(pass, user.password)) {
  const verifiedUser = await facade.getVerifiedUser(userName, pass);
  if (verifiedUser) {
    req.credentials = { userName: verifiedUser.email, role: verifiedUser.role };
    //req.credentials = {userName:user.email,role:"user"}
    return true;
  }
  return false;
}
export default authMiddleware;
```

Importeres i app.ts og kaldes så der hvor den skal bruges.

Explain, using relevant examples, your strategy for implementing a REST-API with Node/Express + TypeScript and demonstrate how you have tested the API.

Testing af API med Chai og supertest

▼ FacadeEndpointTest

```
import path from "path";
import { expect } from "chai";
import app from "../src/app";

import supertest from "supertest";
const request = supertest(app);

import bcryptjs from "bcryptjs";
import * as mongo from "mongodb";
import { InMemoryDbConnector } from "../src/config/dbConnector";
let friendCollection: mongo.Collection;

describe("### Describe the Friend Endpoints (/api/friends) ###", function () {
  let URL: string;

  before(async function () {
    //Connect to IN-MEMORY test database
    //Get the database and set it on the app-object to make it availabe for the friendRoutes
    //(See bin/www.ts if you are in doubt relateded to the part above)
    //Initialize friendCollection, to operate on the database without the facade
    const client = await InMemoryDbConnector.connect(); //Connect to inmemory test database
    const db = client.db();
    app.set("db", db);
    app.set("db-type", "TEST-DB");
    friendCollection = db.collection("friends");
  });

  beforeEach(async function () {
    const hashedPW = await bcryptjs.hash("secret", 8);
    await friendCollection.deleteMany({});
    //Last friend below is only necessary if you have added authentications
    await friendCollection.insertMany([
      {
        firstName: "Peter",
        lastName: "Pan",
        email: "pp@b.dk",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Donald",
        lastName: "Duck",
        email: "dd@b.dk",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Ad",
```

```
        lastName: "Admin",
        email: "aa@a.dk",
        password: hashedPW,
        role: "admin",
      },
    ]);
  });

  //In this, and all the following REMOVE tests that requires authentication if you are using the simple version of friendRoutes
  describe("While attempting to get all users", function () {
    it("it should get two users when authenticated", async () => {
      const response = await request
        .get("/api/friends/all")
        .auth("pp@b.dk", "secret");
      expect(response.status).to.equal(200);
      expect(response.body.length).to.equal(3);
    });

    it("it should get a 401 when NOT authenticated", async () => {
      const response = await request.get("/api/friends/all");
      expect(response.status).to.equal(401);
    });
  });

  describe("While attempting to add a user", function () {
    it("it should Add the user Jan Olsen", async () => {
      const newFriend = {
        firstName: "Jan",
        lastName: "Olsen",
        email: "jan@b.dk",
        password: "secret",
      };
      const response = await request.post("/api/friends").send(newFriend);
      expect(response.status).to.equal(200);
      expect(response.body.id).to.be.not.null;
    });

    xit("It should fail to Add user due to wrong password length", async () => {});
  });
```

Vi bruger router i friendRoutes til at lave endpoints.

Router() skal ses som en "mini" udgave af app.ts og heri kan der sættes relevante endpoints.

```
router.get("/all", async (req: any, res) => {
  const friends = await facade.getAllFriends();
  const friendsDTO = friends.map((friend) => {
    const { firstName, lastName, email } = friend;
    return { firstName, lastName, email };
  });
  res.json(friendsDTO);
});
```

Explain, using relevant examples, how to test JavaScript/Typescript Backend Code, relevant packages (Mocha, Chai etc.) and how to test asynchronous code.

▼ FacadeTest

```
import * as mongo from "mongodb";
import FriendFacade from "../src/facades/friendFacade";

import chai from "chai";
const expect = chai.expect;

//use these two lines for more streamlined tests of promise operations
import chaiAsPromised from "chai-as-promised";
chai.use(chaiAsPromised);

import bcryptjs, { hash } from "bcryptjs";
import { InMemoryDbConnector } from "../src/config/dbConnector";
import { ApiError } from "../src/errors/errors";

let friendCollection: mongo.Collection;
let facade: FriendFacade;

describe("## Verify the Friends Facade ##", () => {
  before(async function () {
    const client = await InMemoryDbConnector.connect(); //Connect to inmemory test database
    const db = client.db();
```

```
  friendCollection = db.collection("friends"); //Initialize friendCollection, to operate on the database without the facade
  facade = new FriendFacade(db); //Get the database and initialize the facade
});

beforeEach(async () => {
  const hashedPW = await bcryptjs.hash("secret", 4);
  await friendCollection.deleteMany({});
  await friendCollection.insertMany([
    {
      firstName: "Donald",
      lastName: "Trump",
      email: "trump@tower.com",
      password: hashedPW,
      role: "user",
    },
    {
      firstName: "Joe",
      lastName: "Biden",
      email: "biden@cnn.com",
      password: hashedPW,
      role: "user",
    },
    {
      firstName: "Peter",
      lastName: "Pan",
      email: "pp@b.com",
      password: hashedPW,
      role: "user",
    },
  ]);
});

describe("Verify the addFriend method", () => {
  it("It should Add the user Jan", async () => {
    const newFriend = {
      firstName: "Jan",
      lastName: "Olsen",
      email: "jan@b.dk",
      password: "secret",
    };
    const status = await facade.addFriend(newFriend);
    expect(status).to.be.not.null;
    const jan = await friendCollection.findOne({ email: "jan@b.dk" });
    expect(jan.firstName).to.be.equal("Jan");
  });

  it("It should not add a user with a role (validation fails)", async () => {
    const newFriend = {
      firstName: "Jan",
      lastName: "Olsen",
      email: "jan@b.dk",
      password: "secret",
      role: "admin",
    };
    try {
      await facade.addFriend(newFriend);
      expect(false).to.be.true("Should never get here");
    } catch (err) {
      expect(err instanceof ApiError).to.be.true;
    }
  });
});

describe("Verify the editFriend method", () => {
  it("It should change lastName to XXXX", async () => {
    const newLastName = {
      firstName: "Joe",
      lastName: "XXXX",
      email: "biden@cnn.com",
      password: "secret",
    };
    const status = await facade.editFriend("biden@cnn.com", newLastName);
    expect(status.modifiedCount).to.equal(1);
    const editedFriend = await friendCollection.findOne({
      email: "biden@cnn.com",
    });
    expect(editedFriend.lastName).to.be.equal("XXXX");
  });
});

describe("Verify the deleteFriend method", () => {
  it("It should remove the user Peter", async () => {
    const status = await facade.deleteFriend("pp@b.com");
    expect(status).to.be.true;
  });
  it("It should return false, for a user that does not exist", async () => {
```

```
      const status = await facade.deleteFriend("fake@fake.dk");
      expect(status).to.be.false;
    });
  });

  describe("Verify the getAllFriends method", () => {
    it("It should get three friends", async () => {
      const friends = await facade.getAllFriends();
      expect(friends.length).equal(3);
    });
  });

  describe("Verify the getFriend method", () => {
    it("It should find Donald Trump", async () => {
      const friend = await facade.getFriend("trump@tower.com");
      expect(friend.firstName + " " + friend.lastName).to.be.equal(
        "Donald Trump"
      );
    });
    it("It should not find xxx.@.b.dk", async () => {
      try {
        await facade.getFriend("xxx.@.b.dk");
      } catch (err) {
        expect(err instanceof ApiError).to.be.true;
      }
    });
  });

  describe("Verify the getVerifiedUser method", () => {
    it("It should correctly validate Peter Pan's credential,s", async () => {
      const veriefiedPeter = await facade.getVerifiedUser("pp@b.com", "secret");
      expect(veriefiedPeter).to.be.not.null;
    });

    it("It should NOT validate Peter Pan's credential,s", async () => {
      const veriefiedPeter = await facade.getVerifiedUser("pp@b.com", "XXXXX");
      expect(veriefiedPeter).to.be.null;
    });

    it("It should NOT validate a non-existing users credentials", async () => {
      const notVerifed = await facade.getVerifiedUser("lol@fake.com", "fake");
      expect(notVerifed).to.be.null;
    });
  });
});
```

For at teste async kode benyttes async og await hvilket giver async kode sync opførsel

NoSQL and MongoDB

Explain, generally, what is meant by a NoSQL database.

***Explain* Pros & Cons in using a NoSQL database like MongoDB as your data store, compared to a traditional Relational SQL Database like MySQL.**

SQL

- Bruger data schemes
- Relations
- Data gemmes i forskellige tabeller
- Limits for tusind read and write queries pr second

NoSQL

- Schema-less (data mangler måske)
- Ingen eller få relationer (ulempe i write-request der skal påvirke flere collections)
- Data merges / nested i få collections
- Kan både scales horizontal og vertical
- God performance for simple read og write requests

Explain about indexes in MongoDB, how to create them, and *demonstrate* how you have used them.

??

Explain, *using your own code* examples, how you have used some of MongoDB's "special" indexes like *TTL* and *2dsphere and perhaps also the Unique Index*.

??

***Demonstrate*, using your own code samples, how to perform all CRUD operations on a MongoDB**

```
router.get("/all", async (req: any, res) => {
  const friends = await facade.getAllFriends();
  const friendsDTO = friends.map((friend) => {
    const { firstName, lastName, email } = friend;
    return { firstName, lastName, email };
  });
  res.json(friendsDTO);
});

router.put("/:email", async function (req: any, res, next) {
  try {
    const email = req.params.userid;
    let newFriend = req.body;
    facade.editFriend(email, newFriend);
  } catch (err) {
    debug(err);
    if (err instanceof ApiError) {
      return next(err);
    }
    next(new ApiError(err.message, 400));
  }
});

router.post("/", async function (req, res, next) {
  try {
    let newFriend = req.body;
    facade.addFriend(newFriend);
  } catch (err) {
    debug(err);
    if (err instanceof ApiError) {
      next(err);
    } else {
      next(new ApiError(err.message, 400));
    }
  }
});
```

Demonstrate how you have set up sample data for your application testing

```
await friendCollection.insertMany([
      {
        firstName: "Donald",
        lastName: "Trump",
        email: "trump@tower.com",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Joe",
        lastName: "Biden",
        email: "biden@cnn.com",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Peter",
        lastName: "Pan",
        email: "pp@b.com",
        password: hashedPW,
        role: "user",
      },
    ]);
  });
```

Explain the purpose of mocha, chai, supertest and nock, which you should have used for your testing

Mocha : Måde at gøre asynkrone test simple og hurtige. Bruges til unit og integration test. Minder om JUNIT. Godt til BDD (behaviour-driven-development)

Eksempel på mocha test

- **Describe** = Den funktionalitet man tester
- **It** = et testcase

```
var expect    = require("chai").expect;
var converter = require("../app/converter");

describe("RGB to Hex conversion", function() {
    it("converts the basic colors", function() {
      var redHex   = converter.rgbToHex(255, 0, 0);
      var greenHex = converter.rgbToHex(0, 255, 0);
      var blueHex  = converter.rgbToHex(0, 0, 255);

      expect(redHex).to.equal("ff0000");
      expect(greenHex).to.equal("00ff00");
      expect(blueHex).to.equal("0000ff");
    });
  });
```

Chai : BDD (behaviour-driven-development) / tdd (test-driven-development). Bruges til at lave assertions i test, men på en let læsbar måde.

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Supertest : Giver mulighed for at sende HTTP request (post, get, osv) og kan bruges til at teste resultater fra ens HTTP requests.

Nock : At "stjæle" et request til et endpoint og returnere et mock-dataset i stedet så man kan teste på et rigtigt endpoint, men med forbehold for at data/databasen kan ændre sig.

Explain, using a sufficient example, how to mock and test endpoints that relies on data fetched from external endpoints

▼ Test using nock

```
const expect = require("chai").expect;
import app from "./whattodo";
const request = require("supertest")(app);
import nock from "nock";

describe("What to do endpoint", function () {
  before(() => {
    const scope = nock("https://www.boredapi.com")
      .get("/api/activity")
      .reply(200, {
        activity: "drink a single beer",
      });
  });

  it("Should eventually provide 'drink a single beer'", async function () {
    const response = await request.get("/whattodo");
    expect(response.body.activity).to.be.equal("drink a single beer");
  });
});
```

Se vedlagt kodeblok

Fra projektet usingNock

Her bruges nock til at "intercepte" response og ændre i reply ved at tilføje noget specifikt data som vi så kan lave assertions på.

Explain, using a sufficient example a strategy for how to test a REST API. If your system includes authentication and roles explain how you test this part.

▼ FriendEndpointsTest

```
describe("### Describe the Friend Endpoints (/api/friends) ###", function () {
  let URL: string;

  before(async function () {
    //Connect to IN-MEMORY test database
    //Get the database and set it on the app-object to make it availabe for the friendRoutes
    //(See bin/www.ts if you are in doubt relateded to the part above)
    //Initialize friendCollection, to operate on the database without the facade
    const client = await InMemoryDbConnector.connect(); //Connect to inmemory test database
    const db = client.db();
    app.set("db", db);
    app.set("db-type", "TEST-DB");
    friendCollection = db.collection("friends");
  });

  beforeEach(async function () {
    const hashedPW = await bcryptjs.hash("secret", 4);
    await friendCollection.deleteMany({});
    //Last friend below is only necessary if you have added authentications
    await friendCollection.insertMany([
      {
        firstName: "Peter",
        lastName: "Pan",
        email: "pp@b.dk",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Donald",
        lastName: "Duck",
        email: "dd@b.dk",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Ad",
        lastName: "Admin",
        email: "aa@a.dk",
        password: hashedPW,
        role: "admin",
      },
    ]);
  });

  //In this, and all the following REMOVE tests that requires authentication if you are using the simple version of friendRoutes
  describe("While attempting to get all users", function () {
    it("it should get two users when authenticated", async () => {
      const response = await request
        .get("/api/friends/all")
        .auth("pp@b.dk", "secret");
      expect(response.status).to.equal(200);
      expect(response.body.length).to.equal(3);
    });

    it("it should get a 401 when NOT authenticated", async () => {
      const response = await request.get("/api/friends/all");
      expect(response.status).to.equal(401);
    });
  });
```

Se vedlagt kodeblok taget fra FriendEndpointsTest i startkoden.

Der opsættes en before som køres inden testene startes hvori der laves en forbindelse til vores in-memory-database, en beforeEach som tilføjer data til databasen.

I hver enkelt test tilgås et endpoint og authentication tilføjes således at man får adgang til data på endpointet. Se linje: 62 i kodeeksemplet (filen FriendEndpointTest.ts i startkoden)

Explain, using a *relevant example*, a full JavaScript backend including relevant test cases to test the REST-API (not on the production database)

▼ FriendFacadeTest

```
import * as mongo from "mongodb";
import FriendFacade from "../src/facades/friendFacade";
```

```
import chai from "chai";
const expect = chai.expect;

//use these two lines for more streamlined tests of promise operations
import chaiAsPromised from "chai-as-promised";
chai.use(chaiAsPromised);

import bcryptjs, { hash } from "bcryptjs";
import { InMemoryDbConnector } from "../src/config/dbConnector";
import { ApiError } from "../src/errors/apiError";

let friendCollection: mongo.Collection;
let facade: FriendFacade;

describe("## Verify the Friends Facade ##", () => {
  before(async function () {
    const client = await InMemoryDbConnector.connect(); //Connect to inmemory test database
    const db = client.db();
    friendCollection = db.collection("friends"); //Initialize friendCollection, to operate on the database without the facade
    facade = new FriendFacade(db); //Get the database and initialize the facade
  });

  beforeEach(async () => {
    const hashedPW = await bcryptjs.hash("secret", 4);
    await friendCollection.deleteMany({});
    await friendCollection.insertMany([
      {
        firstName: "Donald",
        lastName: "Trump",
        email: "trump@tower.com",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Joe",
        lastName: "Biden",
        email: "biden@cnn.com",
        password: hashedPW,
        role: "user",
      },
      {
        firstName: "Peter",
        lastName: "Pan",
        email: "pp@b.com",
        password: hashedPW,
        role: "user",
      },
    ]);
  });

  describe("Verify the addFriend method", () => {
    it("It should Add the user Jan", async () => {
      const newFriend = {
        firstName: "Jan",
        lastName: "Olsen",
        email: "jan@b.dk",
        password: "secret",
      };
      const status = await facade.addFriend(newFriend);
      expect(status).to.be.not.null;
      const jan = await friendCollection.findOne({ email: "jan@b.dk" });
      expect(jan.firstName).to.be.equal("Jan");
    });

    it("It should not add a user with a role (validation fails)", async () => {
      const newFriend = {
        firstName: "Jan",
        lastName: "Olsen",
        email: "jan@b.dk",
        password: "secret",
        role: "admin",
      };
      try {
        await facade.addFriend(newFriend);
        expect(false).to.be.true("Should never get here");
      } catch (err) {
        expect(err instanceof ApiError).to.be.true;
      }
    });
  });

  describe("Verify the editFriend method", () => {
    it("It should change lastName to XXXX", async () => {
      const newLastName = {
        firstName: "Joe",
        lastName: "XXXX",
```

```
        email: "biden@cnn.com",
        password: "secret",
      };
      const status = await facade.editFriend("biden@cnn.com", newLastName);
      expect(status.modifiedCount).to.equal(1);
      const editedFriend = await friendCollection.findOne({
        email: "biden@cnn.com",
      });
      expect(editedFriend.lastName).to.be.equal("XXXX");
    });
  });

  describe("Verify the deleteFriend method", () => {
    it("It should remove the user Peter", async () => {
      const status = await facade.deleteFriend("pp@b.com");
      expect(status).to.be.true;
    });
    it("It should return false, for a user that does not exist", async () => {
      const status = await facade.deleteFriend("nouser@notauser.com");
      expect(status).to.be.false;
    });
  });

  describe("Verify the getAllFriends method", () => {
    it("It should get three friends", async () => {
      const friends = await facade.getAllFriends();
      expect(friends.length).equal(3);
    });
  });

  describe("Verify the getFriend method", () => {
    it("It should find Donald Trump", async () => {
      const friend = await facade.getFrind("trump@tower.com");
      if (friend != null) {
        expect(friend.firstName + " " + friend.lastName).to.be.equal(
          "Donald Trump"
        );
      }
    });
    it("It should not find xxx.@.b.dk", async () => {
      try {
        await facade.getFrind("xxx.@.b.dk");
      } catch (err) {
        expect(err instanceof ApiError).to.be.true;
      }
    });
  });

  describe("Verify the getVerifiedUser method", () => {
    it("It should correctly validate Peter Pan's credential,s", async () => {
      const veriefiedPeter = await facade.getVerifiedUser("pp@b.com", "secret");
      expect(veriefiedPeter).to.be.not.null;
    });

    it("It should NOT validate Peter Pan's false credential,s", async () => {
      const veriefiedPeter = await facade.getVerifiedUser("pp@b.com", "XXXXX");
      expect(veriefiedPeter).to.be.null;
    });

    it("It should NOT validate a non-existing users credentials", async () => {
      const notVerifed = await facade.getVerifiedUser(
        "false@false.false",
        "false"
      );
      expect(notVerifed).to.be.null;
    });
  });
});
```

Se vedlagt kodeblok taget fra FriendfacadeTest i startkoden

Der laves en before som opsætter en in-memory-database, en beforeeach som tilføjer data til databasen og så testes alle metoder som kan udføre CRUD-operationer.