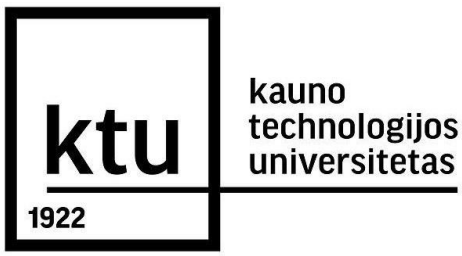


FACULTY OF INFORMATICS

T120B162 Programų sistemų testavimas

Lab 2. Automated Testing

Students:

Martynas Šimkus

Povilas Sakalauskas

Mantas Liutkus

Marius Varna

Lecturers:

asist. BARISAS Dominykas

doc. PACKEVIČIUS Šarūnas

Kaunas, 2024

1. Generate software unit tests in order to evaluate software quality for chosen software. The JTest, C++ Test or dot Tests programs could be used. Create some unit tests manually.

Generate tests for a whole software application (100% code coverage):

In the our Express.js web server project, we didn't have any automated tests generated, it was due to that Javascript has some automated testing tools, however they didn‘t cover our scenarios. All testing was done manually, and we wrote unit tests for every project-specific function (we mean excluding 3rd party dependencies) in our application.

Example:

This test file checks if the getLakes method in lakeController correctly returns a list of lakes with an isLiked property, based on the user's likes.

lake.test.ts

```
it("should return lakes with a matching `isLiked` tag based on the user's
likes", async () => {
      const mockLakes = [
        {
          id: 1,
          name: "Ežeras Vienas",
          area: 250.75,
          depth: 20.5,
          description: "Test 1 ežeras",
          location: { type: "Point", coordinates: [50.0, 30.0] },
          caughtFishes: [],
          likes: [],
        },
        {
          id: 2,
          name: "Ežeras Du",
          area: 300.5,
          depth: 25.0,
          description: "Test 2 ežeras",
          location: { type: "Point", coordinates: [52.0, 32.0] },
          caughtFishes: [],
          likes: [],
        },
        {
          id: 3,
          name: "Ežeras Trys",
          area: 200.25,
          depth: 18.0,
          description: "Test 3 ežeras",
```

```
        location: { type: "Point", coordinates: [48.0, 28.0] },
        caughtFishes: [],
        likes: [],
      },
    ];

    const mockLikes = [
      {
        lakeId: 1,
      },
      {
        lakeId: 3,
      },
    ];

    const expectedResult = [
      {
        ...mockLakes[0],
        isLiked: true,
      },
      {
        ...mockLakes[1],
        isLiked: false,
      },
      {
        ...mockLakes[2],
        isLiked: true,
      },
    ];

    (mockQueryRunner.manager.getRepository as jest.Mock).mockReturnValue({
      find: jest.fn().mockResolvedValue(mockLakes),
    });
    (mockQueryRunner.manager.query as jest.Mock).mockResolvedValue(mockLikes);

    const res = mockResponse();
    await lakeController.getLakes(mockRequest as Request, res as Response);
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith(expectedResult);
  });
```

2. Research mocks, stubs, drivers, use them while creating unit tests where appropriate.

In short:

Stubs:

- Provide fixed responses during tests.

Mocks:

- Pre-defined objects that expect certain calls.

In our project:

Stub:

In the createCaughtFishEntry test named "should validate request body and session", a stub is created for the res object to simulate a response:

```
describe("fishController", () => {
  const mockEmptyRequest = {} as Request;
  const mockResponse = () => {
    const res: any = {};
    res.status = jest.fn().mockReturnValue(res);
    res.json = jest.fn().mockReturnValue(res);
    return res;
  };
```

Mock

In the createCaughtFishEntry test named "should successfully create a caught fish entry", a mock is used to simulate the behavior of the getRepository method from the database manager:

```
(mockQueryRunner.manager.getRepository as jest.Mock)
        .mockReturnValueOnce({
          findOneBy: jest.fn().mockResolvedValue(mockUser),
        })
        .mockReturnValueOnce({
          findOneBy: jest.fn().mockResolvedValue(mockFish),
        })
        .mockReturnValueOnce({
          findOneBy: jest.fn().mockResolvedValue(mockLake),
        })
        .mockReturnValueOnce({
          save: jest.fn().mockResolvedValue(mockCaughtFish),
        });
```

4. Research parametrized tests, use them while creating unit tests where appropriate.

Parametrized tests let you use one test for lots of different data. Instead of making separate tests for each piece of data, you write one test and give it different values to check different situations.

We used parametrized tests to test invalid login format validation:

```
describe("User creation validation", () => {
    const username = "testVartotojas";
    const email = "testVartotojas@gmail.com";
    const password = "testVartotojasSlaptazodis";

    const testCases = [
      {
        input: { email, password },
        expectedMessage: "Reikalingas vartotojo vardas",
      },
      {
        input: { username, password },
        expectedMessage: "El. paštas reikalingas",
      },
      {
        input: { username, email },
        expectedMessage: "Slaptažodis reikalingas",
      },
```

```
      {
        input: { username: "a", email, password },
        expectedMessage: "Vartotojo vardas turi būti bent 5 simbolių",
      },
      {
        input: { username, email: "neteising_formatas", password },
        expectedMessage: "El. paštas yra neteisingas",
      },
      {
        input: { username, email, password: "abc" },
        expectedMessage: "Slaptažodis turi būti bent 8 simbolių",
      },
    ];

    it.each(testCases)(
      "should fail user creation with invalid data",
      async ({ input, expectedMessage }) => {
        const response = await request(testServer)
          .post("/api/users")
          .send(input);

        expect(response.statusCode).toEqual(400);
        expect(response.body.message).toEqual(expectedMessage);
      }
    );
  });
```

5. Research tests set-up, tear-down phases, use them while creating unit tests where appropriate.

Set-up and tear-down phases are done because tests often need some setup some data before running and might need to clean up after they finish.

In our project:

In Jest, beforeEach is used to set things up before each test. Example:

```
beforeEach(() => {
  mockRequest = {
    params: {
      lakeId: "1",
    },
  };

  (AppDataSource.createQueryRunner as jest.Mock).mockReturnValue(
    mockQueryRunner
  );
});
```

In the example beforeEach:

- We set up a mockRequest object with a lakeId parameter.
- We mock the createQueryRunner method to return mockQueryRunner.

Now, every test that runs will have these set up and ready to use.

Use of more complex set-up and tear-down phases will be mentioned in Integrated testing part

6. Create unit tests in code for one chosen class to test.

We used the controller pattern in our Express.js project, so there is no classes, however testing fully controller is a good substitute because it contains all logic related to specific entity.

Here are the unit tests for lakeController, which test the logic related to lakes:

lake.test.ts

```
import { Request, Response } from "express";
import lakeController from "../../src/controllers/lake/lakeController";
import { AppDataSource } from "../../src/data-source";
import { User } from "../../src/entities/user";
import { Lake } from "../../src/entities/lake";
import { Like } from "../../src/entities/like";


// kai tikrasis kodas importuoja data-source iš nurodytos direktorijos, tai
grąžina mock'intą objektą.
jest.mock("../../src/data-source", () => ({
  AppDataSource: {
    createQueryRunner: jest.fn(),
  },
}));

describe("lakeController", () => {
  // Šitą naudojame kaip konstantą, kad gražiai pamock'inti Response objektą,
nuoroda, based on: https://codewithhugo.com/express-request-response-mocking/
  const mockResponse = () => {
    const res: any = {};
    res.status = jest.fn().mockReturnValue(res);
    res.json = jest.fn().mockReturnValue(res);
    return res;
  };
```

```
  //Mockinsim visur query runneri irgi
  const mockQueryRunner = {
    manager: {
      query: jest.fn(),
      getRepository: jest.fn(),
    },
    release: jest.fn(),
  };

  describe("getLakeInfo", () => {
    // mockRequest darome kaip kintamąjį ir kiekvieną kartą iš naujo nustatome,
nes kai kuriuose unit testuose mes objekto savybes ištriname (pvz.: L37).
    let mockRequest: Partial<Request>;

    beforeEach(() => {
      mockRequest = {
        params: {
          lakeId: "1",
        },
      };

      (AppDataSource.createQueryRunner as jest.Mock).mockReturnValue(
        mockQueryRunner
      );
    });

    it("should return 400 if lakeId is not provided", async () => {
      delete mockRequest.params.lakeId;
      const res = mockResponse();
      await lakeController.getLakeInfo(mockRequest as Request, res as Response);
      expect(res.status).toHaveBeenCalledWith(400);
      expect(res.json).toHaveBeenCalledWith({ message: "Lake ID is required" });
    });

    it("should fetch and return lake info", async () => {
      const mockFishes = [
        { name: "Fish1", username: "User1", count: 1, id: 1 },
      ];
      const res = mockResponse();
      (mockQueryRunner.manager.query as jest.Mock).mockResolvedValue(
        mockFishes
      );
      await lakeController.getLakeInfo(mockRequest as Request, res as Response);
      expect(mockQueryRunner.manager.query).toHaveBeenCalledWith(
        expect.any(String),
        [mockRequest.params.lakeId]
      );
      expect(res.status).toHaveBeenCalledWith(200);
```

```
    expect(res.json).toHaveBeenCalledWith(mockFishes);
  });

  it("should handle errors", async () => {
    const mockError = new Error("Test error");
    const res = mockResponse();
    (mockQueryRunner.manager.query as jest.Mock).mockRejectedValue(mockError);
    await lakeController.getLakeInfo(mockRequest as Request, res as Response);
    expect(mockQueryRunner.manager.query).toHaveBeenCalledWith(
      expect.any(String),
      [mockRequest.params.lakeId]
    );
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({
      message: "Failed to fetch lake info",
      error: mockError.message,
    });
  });
});

describe("likeLake", () => {
  let mockRequest;

  beforeEach(() => {
    mockRequest = {
      body: {
        like: true,
        lakeId: "1",
      },
      session: {
        passport: {
          user: "userId",
        },
      },
    };

    (AppDataSource.createQueryRunner as jest.Mock).mockReturnValue(
      mockQueryRunner
    );
  });

  it("should return 400 if lakeId is not provided", async () => {
    delete mockRequest.body.lakeId;
    const res = mockResponse();
    await lakeController.likeLake(mockRequest as Request, res as Response);
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ message: "Lake ID is required" });
  });
```

```
    it("should like a lake", async () => {
      const mockUser = new User();
      const mockLake = new Lake();
      const mockLike = new Like();
      mockLike.user = mockUser;
      mockLike.lake = mockLake;

      const res = mockResponse();
      (mockQueryRunner.manager.getRepository as jest.Mock)
        .mockReturnValueOnce({
          findOneOrFail: jest.fn().mockResolvedValue(mockUser),
        }) // userRepository
        .mockReturnValueOnce({
          findOneOrFail: jest.fn().mockResolvedValue(mockLake),
        }) // lakeRepository
        .mockReturnValueOnce({ save: jest.fn().mockResolvedValue(mockLike) });
// likeRepository

      await lakeController.likeLake(mockRequest as Request, res as Response);
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith({
        success: true,
        message: "Lake liked",
      });
    });

    it("should unlike a lake", async () => {
      mockRequest.body.like = false;
      const mockUser = new User();
      const mockLake = new Lake();

      const res = mockResponse();
      (mockQueryRunner.manager.getRepository as jest.Mock)
        .mockReturnValueOnce({
          findOneOrFail: jest.fn().mockResolvedValue(mockUser),
        }) // userRepository
        .mockReturnValueOnce({
          findOneOrFail: jest.fn().mockResolvedValue(mockLake),
        }) // lakeRepository
        .mockReturnValueOnce({ delete: jest.fn().mockResolvedValue({}) }); //
likeRepository

      await lakeController.likeLake(mockRequest as Request, res as Response);
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith({
        success: true,
        message: "Lake unliked",
```

```
      });
    });

    it("should handle errors", async () => {
      const mockUser = new User();
      const mockLake = new Lake();
      const mockError = new Error("Test error");
      const res = mockResponse();

      (mockQueryRunner.manager.getRepository as jest.Mock)
        .mockReturnValueOnce({
          findOneOrFail: jest.fn().mockResolvedValue(mockUser),
        }) // userRepository
        .mockReturnValueOnce({
          findOneOrFail: jest.fn().mockResolvedValue(mockLake),
        }) // lakeRepository
        .mockReturnValueOnce({ save: jest.fn().mockRejectedValue(mockError) });
// likeRepository

      await lakeController.likeLake(mockRequest as Request, res as Response);

      expect(res.status).toHaveBeenCalledWith(500);
      expect(res.json).toHaveBeenCalledWith({
        message: "Failed to fetch lake info",
        error: mockError.message,
      });
    });
  });

  describe("getLakes", () => {
    const mockRequest = {};

    beforeEach(() => {
      (AppDataSource.createQueryRunner as jest.Mock).mockReturnValue(
        mockQueryRunner
      );
    });

    it("should return lakes with a matching `isLiked` tag based on the user's
likes", async () => {
      const mockLakes = [
        {
          id: 1,
          name: "Ežeras Vienas",
          area: 250.75,
          depth: 20.5,
          description: "Test 1 ežeras",
          location: { type: "Point", coordinates: [50.0, 30.0] },
```

```
    caughtFishes: [],
    likes: [],
  },
  {
    id: 2,
    name: "Ežeras Du",
    area: 300.5,
    depth: 25.0,
    description: "Test 2 ežeras",
    location: { type: "Point", coordinates: [52.0, 32.0] },
    caughtFishes: [],
    likes: [],
  },
  {
    id: 3,
    name: "Ežeras Trys",
    area: 200.25,
    depth: 18.0,
    description: "Test 3 ežeras",
    location: { type: "Point", coordinates: [48.0, 28.0] },
    caughtFishes: [],
    likes: [],
  },
];

const mockLikes = [
  {
    lakeId: 1,
  },
  {
    lakeId: 3,
  },
];

const expectedResult = [
  {
    ...mockLakes[0],
    isLiked: true,
  },
  {
    ...mockLakes[1],
    isLiked: false,
  },
  {
    ...mockLakes[2],
    isLiked: true,
  },
];
```

```
    (mockQueryRunner.manager.getRepository as jest.Mock).mockReturnValue({
      find: jest.fn().mockResolvedValue(mockLakes),
    });
    (mockQueryRunner.manager.query as jest.Mock).mockResolvedValue(mockLikes);

    const res = mockResponse();
    await lakeController.getLakes(mockRequest as Request, res as Response);
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith(expectedResult);
  });

  it("should handle errors", async () => {
    const mockError = new Error("Test error");
    (mockQueryRunner.manager.getRepository as jest.Mock).mockReturnValue({
      find: jest.fn().mockRejectedValue(mockError),
    });

    const res = mockResponse();
    await lakeController.getLakes(mockRequest as Request, res as Response);
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({
      message: "Failed to fetch lakes info",
      error: mockError.message,
    });
  });
});

describe("getLikesByLakeId", () => {
  let mockRequest;

  beforeEach(() => {
    mockRequest = {
      params: {
        lakeId: "1",
      },
      session: {
        passport: {
          user: "userId",
        },
      },
    };

    (AppDataSource.createQueryRunner as jest.Mock).mockReturnValue(
      mockQueryRunner
    );
  });
```

```
it("should return 400 if lakeId is not provided", async () => {
  delete mockRequest.params.lakeId;
  const res = mockResponse();
  await lakeController.getLikesByLakeId(
    mockRequest as Request,
    res as Response
  );
  expect(res.status).toHaveBeenCalledWith(400);
  expect(res.json).toHaveBeenCalledWith({ message: "Lake ID is required" });
});

it("should fetch and return likes for a lake", async () => {
  const mockLikes = [
    {
      id: 1,
      user: {
        id: "userId",
        username: "user1",
        imageBlob: Buffer.from("image"),
      },
      lake: {
        id: 1,
      },
    },
    {
      id: 2,
      user: {
        id: "userId",
        username: "user2",
        imageBlob: Buffer.from("image"),
      },
      lake: {
        id: 1,
      },
    },
  ];

  const expectedResult = {
    likedUsers: [
      {
        name: "user1",
        avatar: "aW1hZ2U=",
      },
      {
        name: "user2",
        avatar: "aW1hZ2U=",
      },
    ],
```

```
        hasUserLiked: true,
      };

      (mockQueryRunner.manager.getRepository as jest.Mock).mockReturnValueOnce({
        find: jest.fn().mockResolvedValue(mockLikes),
        findOne: jest.fn().mockResolvedValue({ id: "userId" }),
      }); // likeRepository

      const res = mockResponse();
      await lakeController.getLikesByLakeId(
        mockRequest as Request,
        res as Response
      );
      expect(res.status).toHaveBeenCalledWith(200);
      expect(res.json).toHaveBeenCalledWith(expectedResult);
    });

    it("should handle errors", async () => {
      const mockError = new Error("Test error");
      const res = mockResponse();

      (mockQueryRunner.manager.getRepository as jest.Mock).mockReturnValueOnce({
        find: jest.fn().mockRejectedValue(mockError),
      }); // likeRepository

      await lakeController.getLikesByLakeId(
        mockRequest as Request,
        res as Response
      );
      expect(res.status).toHaveBeenCalledWith(500);
      expect(res.json).toHaveBeenCalledWith({
        message: "Failed to fetch like",
        error: mockError.message,
      });
    });
  });
});
```

7. Create integration tests between several components (classes, remote services).

Our integration tests validate the interaction between the user routes, database operations, and error handling.We specifically performed integration tests for auht functionalities (Login methods etc.)

For integration tests we used jest for assertions and supertest for HTTP requests

Tests set-up, tear-down phases:

• The beforeAll hook initialized the test server using setupServer(), the setupServer() initiliazes our seperate test database

• The afterAll hook cleaned up the database using teardownDatabase() and closed the test server.

• The afterEach hook reset the database state using resetDatabase() after each individual test.

Test Cases:

• Successful User Creation: Checked if the system can successfully create a user with valid credentials.

• Invalid Request Body: We tested if system handles an invalid request body as expected.

• Invalid User Data: We tested if system correctly handles invalid user data, such as an incorrect email, during user creation.

• Duplicate User Data: We tested if the system denies attempts to create a user with a username or email that already exists in the database

Code for integration tests:

testsHelpers.ts (Used as for set-up, tear-down phases)

```
import type { Server } from "http";
import request from "supertest";
import type { Express } from "express";

import createServer from "../../src/config/server";
import { AppDataSource } from "../../src/data-source";
import type { TestUserAttributes } from "./userHelpers";
import { registerTestUser } from "./userHelpers";

interface OverrideExpressOptions {
  logout?: (cb: any) => unknown;
```

```
  logIn?: (user: any, cb: any) => unknown;
}

const setupExpressOverrides = (
  server: Express,
  overrideExpressOptions: OverrideExpressOptions
) => {
  if (overrideExpressOptions.logout) {
    server.request.logOut = overrideExpressOptions.logout;
  }
  if (overrideExpressOptions.logIn) {
    server.request.logIn = overrideExpressOptions.logIn;
  }
  return server;
};

//SET UP PHASE FOR INTEGRATION TESTS
export const setupServer = async (
  port = 7777,
  preventDatabaseConnection = false,
  overrideExpressOptions?: OverrideExpressOptions
) => {
  const server = createServer();
  if (overrideExpressOptions) {
    setupExpressOverrides(server, overrideExpressOptions);
  }

  if (!preventDatabaseConnection) {
    await AppDataSource.initialize();
  }

  return server.listen(port);
};

export const teardownDatabase = async () => {
  await AppDataSource.destroy();
};

//TEARN DOWN PHASE FOR ITNEGRATION TESTS
export const resetDatabase = async () => {
  const tables = AppDataSource.entityMetadatas
    .map((entity) => `"${entity.tableName}"`)
    .join(", ");
  await AppDataSource.query(`TRUNCATE ${tables} CASCADE;`);
};

//SET UP PHASE (SETUP AUTHENTICATED USER)
export const createAuthenticatedAgent = async (
```

```
  server: Server,
  testUser?: TestUserAttributes
) => {
  const userAgent = request.agent(server);
  const user = await registerTestUser(testUser);
  await userAgent
    .post("/api/auth/login")
    .send({ login: user.username, password: testUser?.password || "password"
});

  return { agent: userAgent, user };
};
```

userHelpers.ts (used as for set-up of user))

```
import { AppDataSource } from "../../src/data-source";
import { User } from "../../src/entities/user";

export interface TestUserAttributes {
  name?: string;
  emailAddress?: string;
  password?: string;
}

export const registerTestUser = async (userDetails?: TestUserAttributes) =>
{
  const userRepository = AppDataSource.getRepository(User);

  const newUser = new User();
  newUser.username = userDetails?.name || "testinis_vartotojas";
  newUser.email = userDetails?.emailAddress ||
"testinis_vartotojas@gmail.com";
  newUser.setPassword(userDetails?.password || "slaptazodis");

  await userRepository.save(newUser);
  return newUser;
};
```

users.test.ts (integration tests)

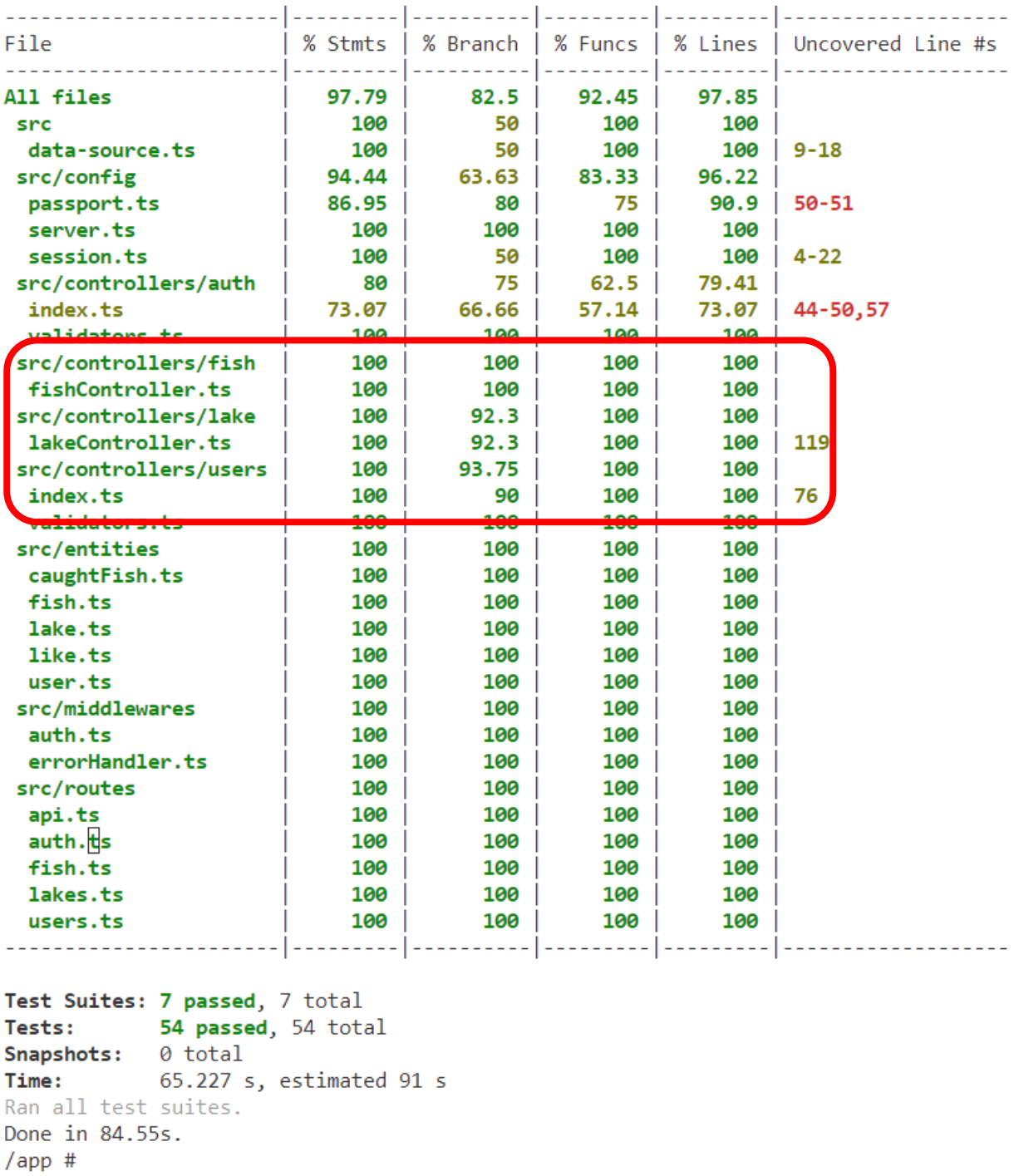
```
import type { Server } from "http";
import request from "supertest";

import { AppDataSource } from "../../src/data-source";
import { User } from "../../src/entities/user";
import {
```

```
  resetDatabase,
  teardownDatabase,
  setupServer,
} from "../utils/testsHelpers";
import { registerTestUser } from "../utils/userHelpers";

let testServer: Server;

beforeAll(async () => {
  testServer = await setupServer();
});

afterAll(async () => {
  await teardownDatabase();
  testServer.close();
});

describe("User Routes", () => {
  afterEach(async () => {
    await resetDatabase();
  });

  test("Should successfully create a user", async () => {
    const username = "test_vartotojas";
    const email = "test_vartotojas@gmail.com";
    const password = "test_vartotojo_slaptazodis";

    const response = await request(testServer)
      .post("/api/users")
      .send({ username, email, password });

    const userRepo = AppDataSource.getRepository(User);
    const user = await userRepo.findOneOrFail({
      where: { username: username },
    });

    expect(response.statusCode).toEqual(200);
    expect(response.text).toEqual(user.id);
  });

  test("Should fail user creation with invalid request body", async () => {
    const response = await request(testServer).post("/api/users").send({});

    expect(response.statusCode).toEqual(400);
    expect(response.body.message).toEqual("Reikalingas vartotojo vardas");
  });
```

```
  test("Should fail user creation if username or email already exists",
async () => {
    const { username, email } = await registerTestUser();

    const testCases = [
      {
        input: {
          username,
          email: "testinis_email@gmail.com",
          password: "slaptazodis",
        },
        expectedMessage: "Vartotojo vardas jau užimtas",
      },
      {
        input: { username: "kitasVardas", email, password: "slaptazodis" },
        expectedMessage: "Elektroninis paštas jau užimtas",
      },
    ];

    for (const { input, expectedMessage } of testCases) {
      const response = await
request(testServer).post("/api/users").send(input);

      expect(response.statusCode).toEqual(409);
      expect(response.body.message).toEqual(expectedMessage);
    }
  });

  //Satisfies: 4. Research parametrized tests, use them while creating unit
tests where appropriate.
  //PARAMETRIZED TESTS
  describe("User creation validation", () => {
    const username = "testVartotojas";
    const email = "testVartotojas@gmail.com";
    const password = "testVartotojasSlaptazodis";

    const testCases = [
      {
        input: { email, password },
        expectedMessage: "Reikalingas vartotojo vardas",
      },
      {
        input: { username, password },
        expectedMessage: "El. paštas reikalingas",
      },
      {
        input: { username, email },
        expectedMessage: "Slaptažodis reikalingas",
```

```
      },
      {
        input: { username: "a", email, password },
        expectedMessage: "Vartotojo vardas turi būti bent 5 simbolių",
      },
      {
        input: { username, email: "neteising_formatas", password },
        expectedMessage: "El. paštas yra neteisingas",
      },
      {
        input: { username, email, password: "abc" },
        expectedMessage: "Slaptažodis turi būti bent 8 simbolių",
      },
    ];

    it.each(testCases)(
      "should fail user creation with invalid data",
      async ({ input, expectedMessage }) => {
        const response = await request(testServer)
          .post("/api/users")
          .send(input);

        expect(response.statusCode).toEqual(400);
        expect(response.body.message).toEqual(expectedMessage);
      }
    );
  });
});
```

8. Evaluate software tests coverage.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.79	82.5	92.45	97.85	
src	100	50	100	100	
data-source.ts	100	50	100	100	9-18
src/config	94.44	63.63	83.33	96.22	
passport.ts	86.95	80	75	90.9	50-51
server.ts	100	100	100	100	
session.ts	100	50	100	100	4-22
src/controllers/auth	80	75	62.5	79.41	
index.ts	73.07	66.66	57.14	73.07	44-50,57
validators.ts	100	100	100	100	
src/controllers/fish	100	100	100	100	
fishController.ts	100	100	100	100	
src/controllers/lake	100	92.3	100	100	
lakeController.ts	100	92.3	100	100	119
src/controllers/users	100	93.75	100	100	
index.ts	100	90	100	100	76
validators.ts	100	100	100	100	
src/entities	100	100	100	100	
caughtFish.ts	100	100	100	100	
fish.ts	100	100	100	100	
lake.ts	100	100	100	100	
like.ts	100	100	100	100	
user.ts	100	100	100	100	
src/middlewares	100	100	100	100	
auth.ts	100	100	100	100	
errorHandler.ts	100	100	100	100	
src/routes	100	100	100	100	
api.ts	100	100	100	100	
auth.ts	100	100	100	100	
fish.ts	100	100	100	100	
lakes.ts	100	100	100	100	
users.ts	100	100	100	100	

```
Test Suites: 7 passed, 7 total
Tests:       54 passed, 54 total
Snapshots:   0 total
Time:        65.227 s, estimated 91 s
Ran all test suites.
Done in 84.55s.
/app #
```

We evaluated our software test coverage using the Jest coverage tool. Our project functionalities have achieved 100% coverage for project related functions. While auth/index.ts setups aren't covered in unit tests (because they cover 3rd libraries setup), lakeController, fishController, and usersController are fully covered with 100% unit test coverage.