



6/18/2024

# Tests Implementation

Individual: FestivalConnect

Name: Lucas Jacobs

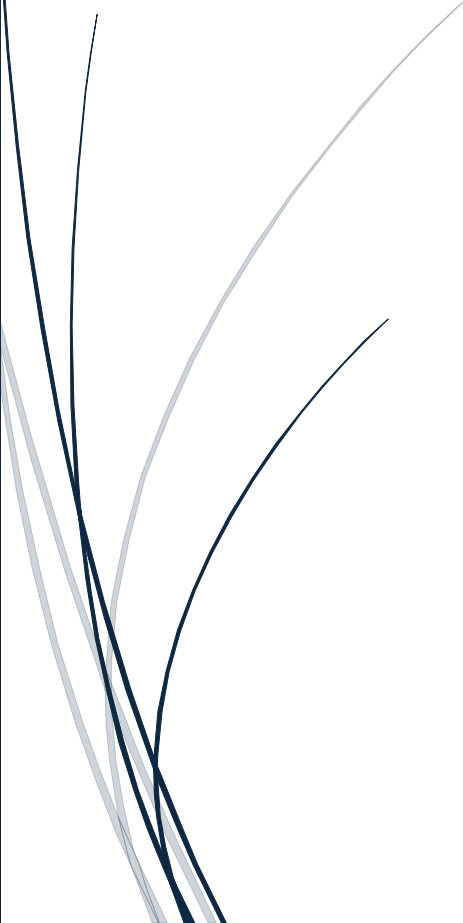
Class: S-A-RB06

PCN: 490692

Student number: 4607368

Technical teachers: Felipe Ebert, Bartosz Paszkowski

Semester coach: Gerard Elbers



## Table of Contents

Introduction.....	1
Unit Tests.....	2
Integration Tests.....	4
E2E Tests .....	5
References .....	7

## Introduction

I will show some results and examples of my tests. For user acceptance tests and why I used these tests, look at (Jacobs, Acceptance Testplan) and the evidence of doing load testing is in the implementation document of (Jacobs, Monitoring Azure Deployment). Also, code coverage and the results of the different tests are provided inside (Jacobs, 2024 06 07 CICD implementation).

# Unit Tests

To show first some of my passing tests for the identity and user service.

Test	Duration	Group Summary
IdentityApiTests (5)	886 ms	IdentityApiTests
IdentityApiTests.Services (5)	886 ms	Tests in group : 5
IdentityLogicTests (5)	886 ms	⌚ Total Duration : 886 ms
LoginUserAsync_InvalidP...	469 ms	Outcomes
LoginUserAsync_UserNo...	1 ms	✓ 5 Passed
LoginUserAsync_ValidCr...	289 ms	
RegisterUserAsync_Email...	1 ms	
RegisterUserAsync_Regis...	126 ms	

Figure 1: Unit Test Results Identity Service

Test	Duration	Group Summary
UserApiTests (7)	344 ms	UserApiTests
UserApiTests.Services (7)	344 ms	Tests in group : 7
UserLogicTests (7)	344 ms	⌚ Total Duration : 344 ms
DeleteUser_DeletionFails...	326 ms	Outcomes
DeleteUser_ExistingUser...	8 ms	✓ 7 Passed
DeleteUser_NonExisting...	1 ms	
RegisterUser_FailedRegis...	4 ms	
UpdateUser_ExistingUse...	3 ms	
UpdateUser_NonExisting...	1 ms	
UpdateUser_UpdateFails...	1 ms	

Figure 2: Unit Test Results User Service

To give a exmample of how I made tests, with the usage of the AAA pattern, let me give a example of a happy flow and an alternative flow.

## Test with a happy flow

```
[TestMethod]
0 references
public async Task UpdateUser_ExistingUser_SuccessfullyUpdated()
{
    // Arrange
    var request = new UpdateUserRequest
    {
        IdentityId = 1,
        Username = "newUsername"
    };
    var existingUser = new UserModel { IdentityId = 1, Username = "oldUsername" };
    _userRepository.Setup(repo => repo.GetByIdentityId(request.IdentityId))
        .Returns(existingUser);
    _userRepository.Setup(repo => repo.Update(It.IsAny<UserModel>()))
        .Returns(true);

    // Act
    var result = await _userLogic.UpdateUser(request);

    // Assert
    Assert.IsTrue(result);
    Assert.AreEqual(request.Username, existingUser.Username);
    _cache.Verify(c => c.RemoveAsync(It.Is<string>(k => k == $"User_{request.IdentityId}"), default), Times.Once);
}
```

Figure 3: Test example of updating a user correctly

## Test with an alternative flow

```
[TestMethod]
0 references
public async Task RegisterUserAsync_EmailAlreadyExists_ReturnsErrorResponse()
{
    // Arrange
    var request = new CreateRegisterUserRequest
    {
        Email = "existing@example.com",
        Password = "password",
        RoleId = 1,
        Username = "username"
    };
    _identityRepository.Setup(repo => repo.GetIdentityByEmail(It.IsAny<string>()))
        .ReturnsAsync(new IdentityModel());

    // Act
    var result = await _identityLogic.RegisterUserAsync(request);

    // Assert
    Assert.IsFalse(result.Flag);
    Assert.AreEqual("Email Already Exist.", result.Message);
}
```

Figure 4: Test example of registering a user alternative flow

As you can notice, the tests are set up with the AAA pattern, by first setting up the data and mocking the database, then calling the method that needs to be tested, and finally checking if the outcome of the method meets the required outcome.

# Integration Tests

For integration tests, I used Postman to setup the request and wrote down a simple test to verify that the response was valid, ensuring that the components' connections were working correctly.

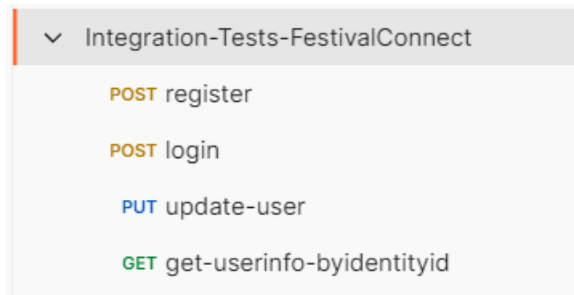


Figure 5: Integration tests in postman

For login, this test is set as follows

```
1 pm.test("Successful POST request login", function () {
2   pm.expect(pm.response.code).to.be.oneOf([200, 201]);
3   let json = pm.response.json();
4   let token = json.token;
5
6   pm.environment.set('bearerToken', token);
7 });
```

Figure 6: Integration test of login

This will check if the response code is successful. After that it will set the JWT token as an environment variable, making it possible for the other tests to use this and execute the test with the JWT token. See the following example of for example the update of a user request.

```
pm.request.headers.add({
  key: 'Authorization',
  value: `Bearer ${pm.environment.get('bearerToken')}`
});
```

Figure 7: pre-requisite before performing the test

This is executed before testing the request, making sure the request has a valid JWT token. This is mainly used when applying the tests in the pipeline since this is an automated process.

# E2E Tests

For E2E tests I used Cypress to test the flow of the application and if it all works. See the following results of the register user tests.

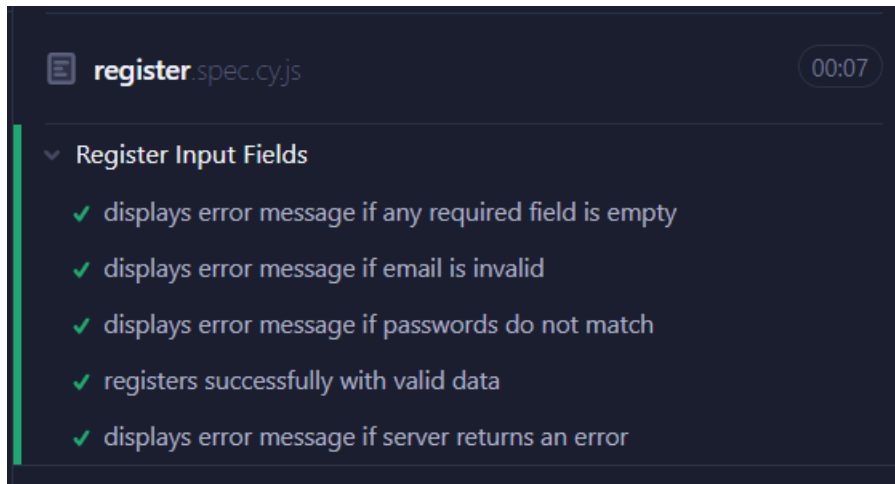


Figure 8: E2E Tests Registering a user

Also see the (Jacobs, 2024 06 07 CICD implementation) for the results of the tests in the pipeline.

To give an example of how I wrote down the tests, I will give one happy flow and two alternative flows.

```
it('displays error message if email is invalid', () => {  
  cy.get('input[name="username"]').type('testuser');  
  cy.get('input[name="email"]').type('invalid-email');  
  cy.get('input[name="password"]').type('password123');  
  cy.get('input[name="confirmPassword"]').type('password123');  
  cy.get('button[type="submit"]').click();  
  cy.get('#error').should('have.text', 'Please enter a valid email');  
});
```

Figure 9: alternative flow E2E test

```
it('registers successfully with valid data', () => {  
  cy.intercept('POST', 'http://festivalconnectapi.germanywestcentral.cloudapp.azure.com:8080/Identity/register', {  
    statusCode: 200,  
    body: { flag: true },  
  }).as('registerRequest');  
  
  cy.get('input[name="username"]').type('testuser');  
  cy.get('input[name="email"]').type('test@example.com');  
  cy.get('input[name="password"]').type('password123');  
  cy.get('input[name="confirmPassword"]').type('password123');  
  cy.get('button[type="submit"]').click();  
  
  cy.wait('@registerRequest').its('response.statusCode').should('eq', 200);  
  cy.window().its('location.href').should('eq', 'http://4.182.171.179:8080/register');  
});
```

Figure 10: Happy Flow E2E on Deployment With Intercept

```

it('registers email exists with valid data on database', () => {

  cy.get('input[name="username"]').type('testuser');
  cy.get('input[name="email"]').type('test@example.com');
  cy.get('input[name="password"]').type('test');
  cy.get('input[name="confirmPassword"]').type('test');
  cy.get('button[type="submit"]').click();

  cy.contains('Email Allready Exist.').should('be.visible')
  cy.window().its('location.href').should('eq', 'http://4.182.171.232/register');
});

```

Figure 11: Alternative Flow With Service Call

As you can see the tests will get the type of input/button with the appropriate name, fill in the inputs, and then click on the button which should result in an action that needs to be performed. Note that I made tests for when calling the services and also made tests using intercept to mock responses. When for example creating a register, the user can for example already exist and a unique user can exist with a certain email.

These tests were also performed on my deployment. Note that I used an IP address, this was because due to the security settings of Cypress (HTTP), I could not reach the URL there.

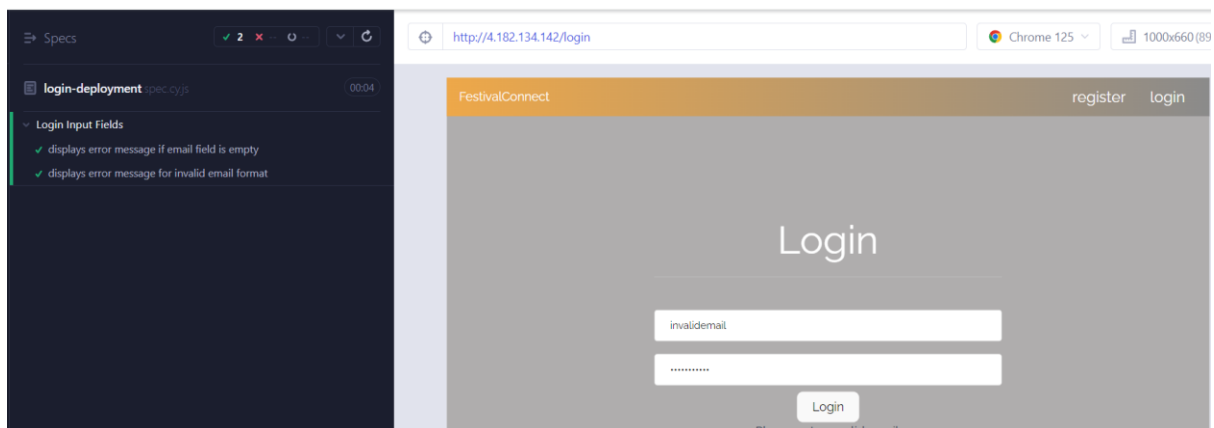


Figure 12: Test Results of Login

Chrome is being controlled by automated test software.

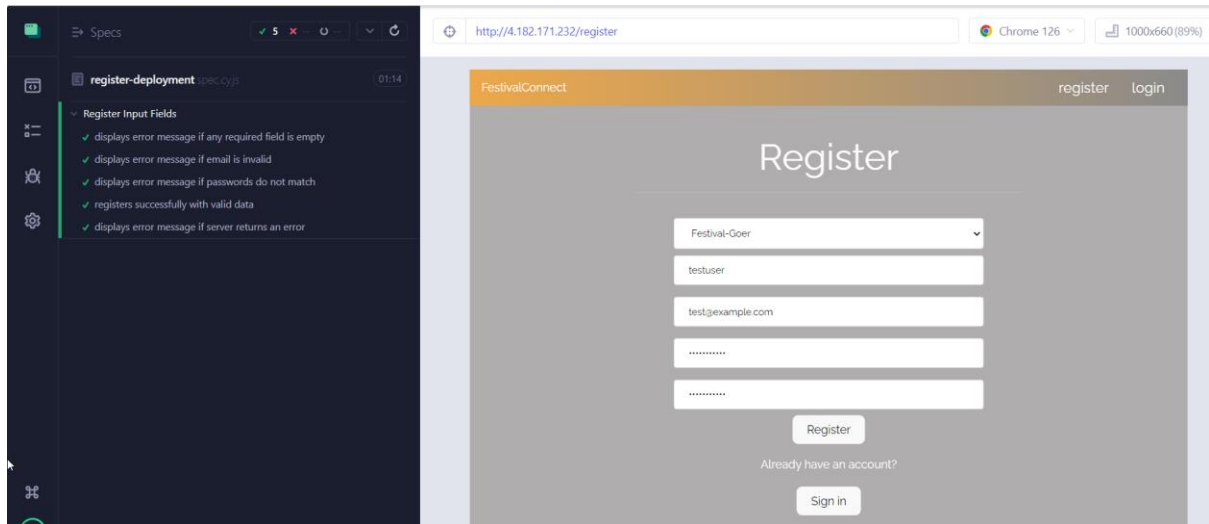


Figure 13: Test Results Deployment Register

## References

- Jacobs, L. (2024). Acceptance Testplan, FontysICT.
- Jacobs, L. (2024). Monitoring Azure Deployment (Unpublished manuscript), FontysICT.
- Jacobs, L. (2024). 2024 06 07 CI/CD implementation, FontysICT.