TESTES E PERFORMANCE WEB

TECNOLOGIAS E SISTEMAS DE INFORMAÇÃO PARA A WEB

POLITÉCNICO DO PORTO
ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

M10 – INTEGRATION TESTS

TSIW 2023/2024

AGENDA

1. Why Integration testing?
2. What is an integration test?
3. Out-of-process dependencies;
4. Integration testing;
5. Testing the database;
6. JEST – JavaScript Testing Framework.

WHY INTEGRATION TESTING

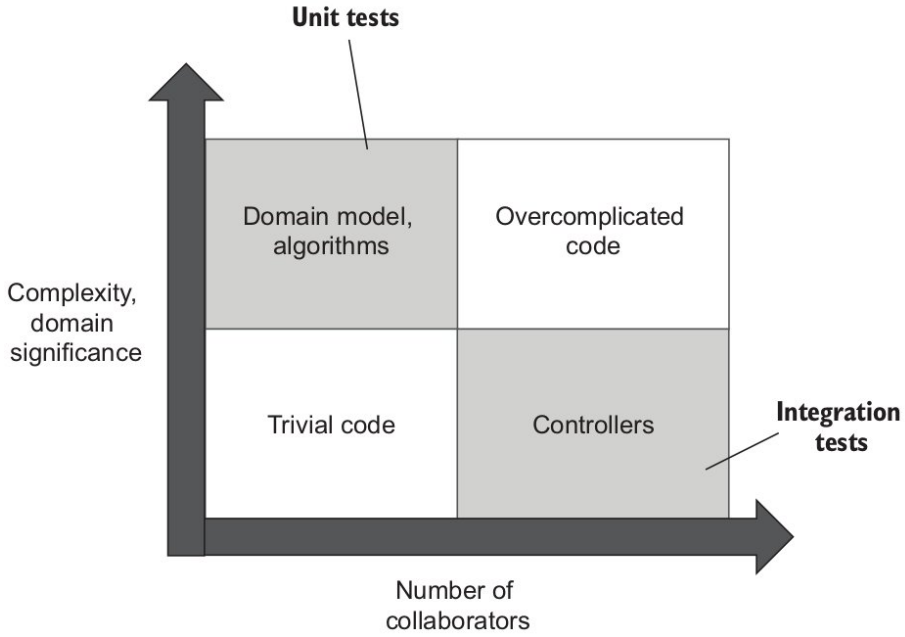
1. WHY INTEGRATION TESTING?

- Although all the unit tests pass, sometimes the application still doesn't work;
- You can never be sure your system works as a whole if you rely on unit tests exclusively;
- Unit tests are great at verifying business logic, but it is not enough to check that logic in a vacuum;
- Validating software components in isolation from each other is important, but it is equally important to check how those components work in integration with external systems such as the databases, web services, messages bus, and so on.

WHAT IS AN INTEGRATION TEST

2. WHAT IS AN INTEGRATION TEST?

- Integration test play an important role in your test suite;
- It is crucial to balance the number of unit and integration tests;
- An integration test is any test that **is not** a unit test;
- Integration tests almost always verify how your system works in integration with out-of-process dependencies;
- These test cover the code from the controllers quadrant while unit tests cover the domain model, which means that integration tests check the code that glues that domain model with out-of-process dependencies.



2. WHAT IS AN INTEGRATION TEST?

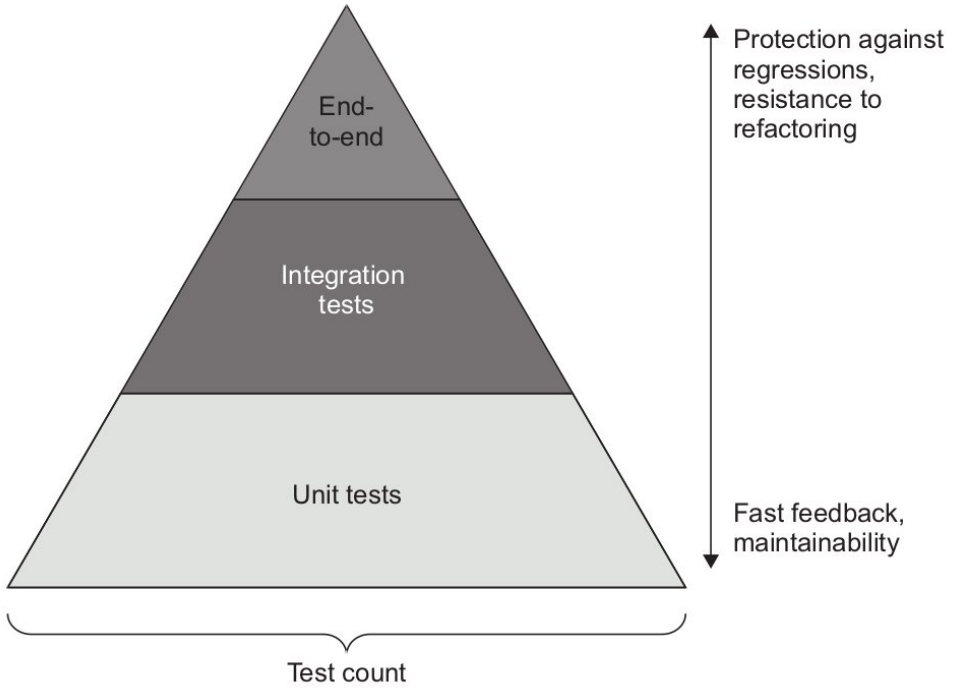
- That tests covering the controllers quadrant can sometimes be unit tests to. If all out-of-process dependencies are replaced with mocks, there will be no dependencies shared between tests;
- Most applications do have an out-of-process dependency that can’t be replaced with a mock. It is usually a database;
- It is important to maintain a balance between unit and integration tests;
- Working directly with out-of-process dependencies makes integration test slows;
- Such tests are also more expensive to maintain:
  - The necessity to keep the out-of-process dependencies operational;
  - The greater number of collaborations involved, which inflates the test’s size.

2. WHAT IS AN INTEGRATION TEST?

- Integration tests go through a larger amount of code (both your code and the code of the libraries used by the application);
- This makes them better than unit tests at protecting against regressions;
- They are also more detached from the production code and therefore have better resistance to refactoring;
- The ration between unit tests and integration tests can differ depending on the project's specific but the general rule is to check as many of the business scenario's edge cases as possible with unit tests and use integration tests to cover one happy path, as well as any edge case that can't be covered by unit tests;
- A happy path is a successful execution of a business scenario;
- An edge case is when the business scenario execution results in an error.

2. WHAT IS AN INTEGRATION TEST?

- Shifting the majority of the workload to unit tests helps keep maintenance costs low;
- At the same time, having one or two overarching integration tests per business scenario ensures the correctness of your system as a whole.



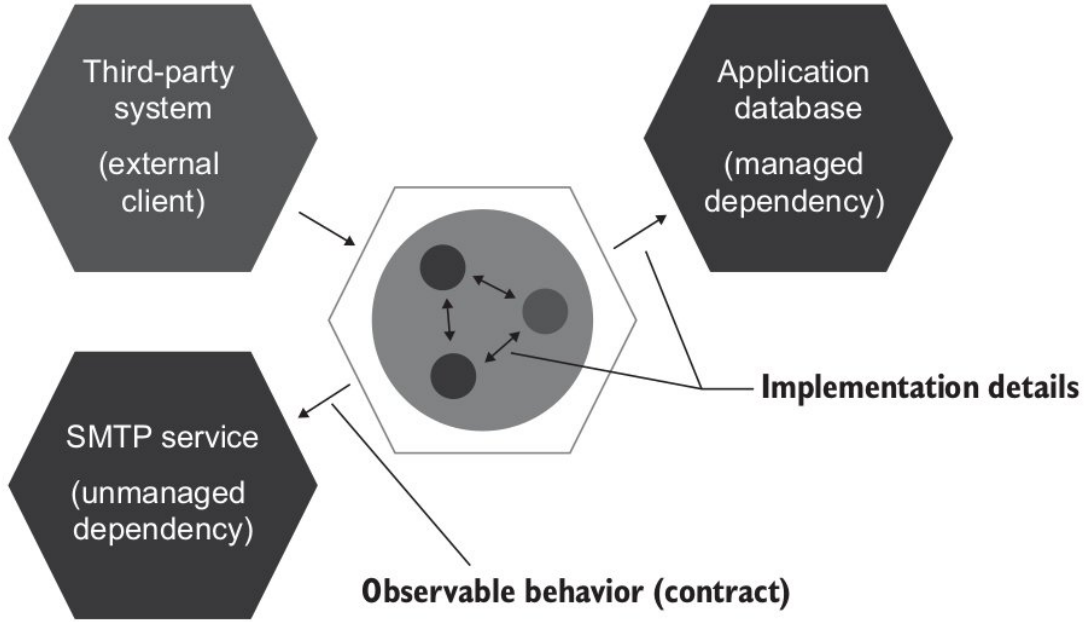
OUT-OF-PROCESS DEPENDENCIES

3. OUT-OF-PROCESS DEPENDENCIES

- Integration tests verify how your system integrates with out-of-process dependencies;
- There are two ways to implement such verification:
  - Use the real out-of-process dependency;
  - Or replace that dependency with a mock.
- All out-of-process dependencies fall into two categories:
  - Managed dependencies (out-of-process dependencies you have full control over) - these dependencies are only accessible through your applications; interactions with them aren't visible to the external world such as, for example, database interactions;
  - Unmanaged dependencies (out-of-process dependencies that you don't have full control over) – interactions with such dependencies are observable externally, such as, for example, SMTP server and message bus.

3. OUT-OF-PROCESS DEPENDENCIES

- Communication with managed dependencies are implementation details;
- Communication with unmanaged dependencies are part of your system's observable behaviour;
- This distinction leads to the difference in treatment of out-of-process dependencies in integration tests;
- Use real instances of managed dependencies;
- Replace unmanaged dependencies with mocks;
- Using real instances of managed dependencies helps you verify the final state.



3. OUT-OF-PROCESS DEPENDENCIES

- Sometimes, for reasons outside of your control, you just can’t use a real version of a managed dependency in integration test, such as, for example a database;
- If you can’t test the database as-is, don’t write integration tests at all, and instead, focus exclusively on unit testing of the domain model.

INTEGRATION TESTING

4. INTEGRATION TESTING

- There are some general guidelines that can help you get the most out of your integration tests:
  - Making domain model boundaries explicit;
  - Reducing the number of layers in the application;
  - Eliminating circular dependencies.
- As usual, best practices that are beneficial for tests also tend to improve the health of your code base in general.

4. INTEGRATION TESTING

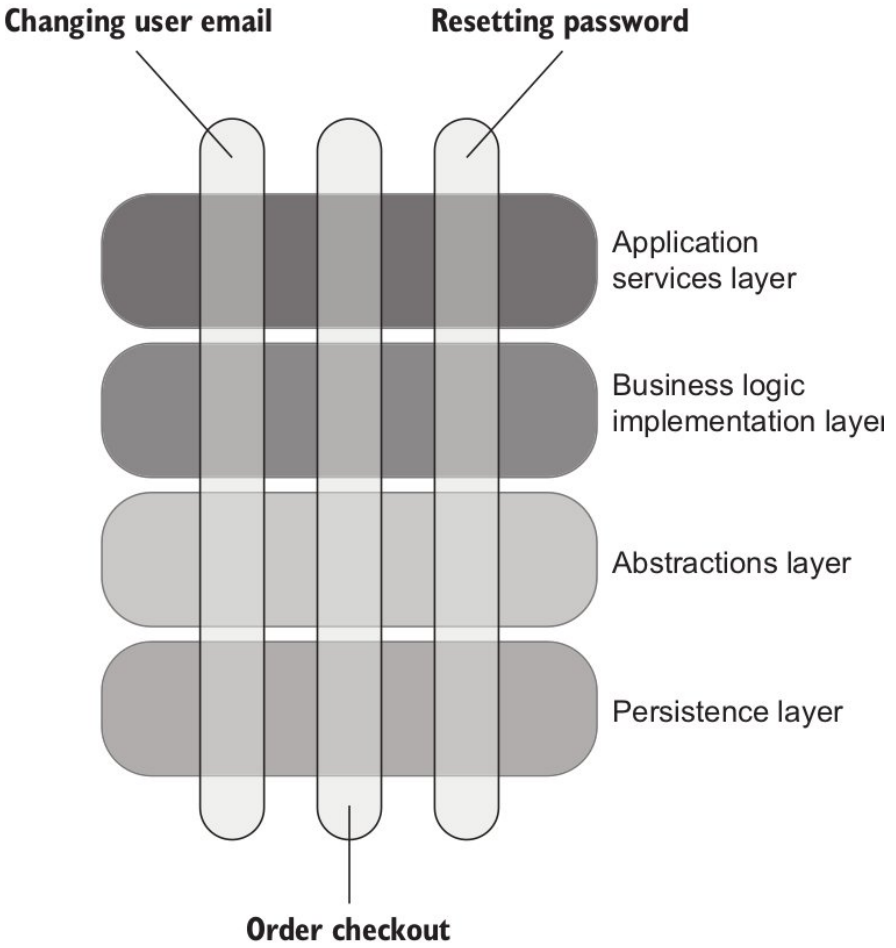
Making domain model boundaries explicit

- Try to always have an explicit, well-known place for the domain model in your code base;
- The domain model is the collection of domain knowledge about the problem your project mean to solve;
- Assigning the domain model an explicit boundary helps you better visualize and reason about that part of your code;
- This practice also helps with testing. Unit tests target the domain model and algorithms, while integration tests target controllers;
- The explicit boundary between domain classes and controllers makes it easier to tell the difference between unit and integration tests.

4. INTEGRATION TESTING

Reducing the number of layers

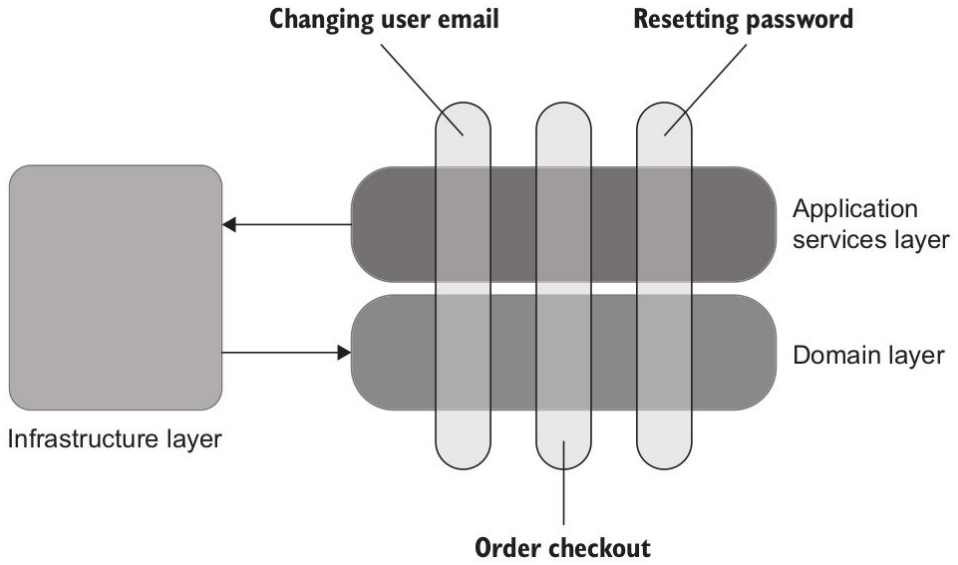
- Most programmers naturally gravitate toward abstracting and generalizing the code by introducing additional layers of indirection;
- In extreme cases, an application gets so many abstraction layers that it becomes to hard to navigate the code base and understand the logic behind;
- Layers of indirection negatively affect your ability to reason about the code;
- An excessive number of abstraction doesn't help unit or integration testing either.



4. INTEGRATION TESTING

Reducing the number of layers

- Try to have as few layers of indirection as possible;
- In most backend systems, you can get away with just three: the domain model, application services layer (controllers) , and infrastructure layer (algorithms that doesn't belong to the domain model, as well as code that enables access to out-of-process dependencies).



4. INTEGRATION TESTING

Eliminating circular dependencies

- Another practice that can drastically improve the maintainability of your code base and make testing easier is eliminating circular dependencies;
- A circular dependency is two or more classes that directly or indirectly depend each other to function properly;
- A better approach to handle circular dependencies is to get rid of them;
- It is rarely possible to eliminate all circular dependencies in your code base.

TESTING THE DATABASE

5. TESTING THE DATABASE

- The last piece of the puzzle in integration testing is managed out-of-process dependencies;
- The most common example of a managed dependency is an application database – a database no other application has access to;
- Running tests against a real database provides bulletproof protection against regressions, but those tests aren't easy to set up.

5. TESTING THE DATABASE

Prerequisites for testing the database

- Managed dependencies should be included as-is in integration tests;
- That makes working with those dependencies more laborious than unmanaged ones because using a mock is out of the question;
- But even before you starting writing tests, you must take preparatory steps to enable integration testing.

5. TESTING THE DATABASE

Prerequisites for testing the database

1. Keeping the database in the source control system

- The first step is treating the database schema as regular code;
- Just as with regular code, a database schema is best stored in a source control system such as Git. The benefits are:
  - Change history – trace the database schema back to some point in the past, which might be important when reproducing bugs in production;
  - Single source of truth – there are only one single source of truth, the Git repository.
- No modifications to the database structure should be made outside the source control.

5. TESTING THE DATABASE

Prerequisites for testing the database

2. Reference data is part of the database schema

- When it comes to the database schema, the usual suspects are: tables, views, indexes, store procedures, and anything else that forms a blueprint of how database is constructed;
- However, there's another part of the database that belongs to the database schema but is rarely viewed as such: reference data;
- Reference data is data that must be prepopulated in order for the application to operate properly (example: userTypes table);
- There's a simple way to differentiate reference data from regular data. If your application can modify the data, it's regular data; if not, it's reference data.

5. TESTING THE DATABASE

Prerequisites for testing the database

2. Reference data is part of the database schema

- Because reference data is essential for your application, you should keep it in the source control system along with tables, views, and others parts of the database schema in the form of SQL insert statements.

5. TESTING THE DATABASE

Prerequisites for testing the database

3. Separate instance for every developer

- It is difficult enough to run tests against a real database;
- It becomes even more difficult if you have to share that database with other developers because:
  - Test runs by different developers interfere with each other;
  - Non-backward-compatible changes can block the work of other developers.
- Keep a separate database instance for every developer, preferable on that developer's own machine in order to maximize test execution speed.

5. TESTING THE DATABASE

Prerequisites for testing the database

4. State-based vs migration-based database delivery

- There are two major approaches to database delivery: state-based and migration-based;
- The migration-based approach is more difficult to implement and maintain initially, but it works much better than the state-based approach in the long run.
- **State-based approach**
  - In the state-based approach you have a model database that you maintain throughout development;
  - During developments, a compassion tool generates scripts for the production database to bring it up to date with the model database;
  - You have SQL scripts store in source control that you can use to create the database.

5. TESTING THE DATABASE

Prerequisites for testing the database

4. State-based vs migration-based database delivery

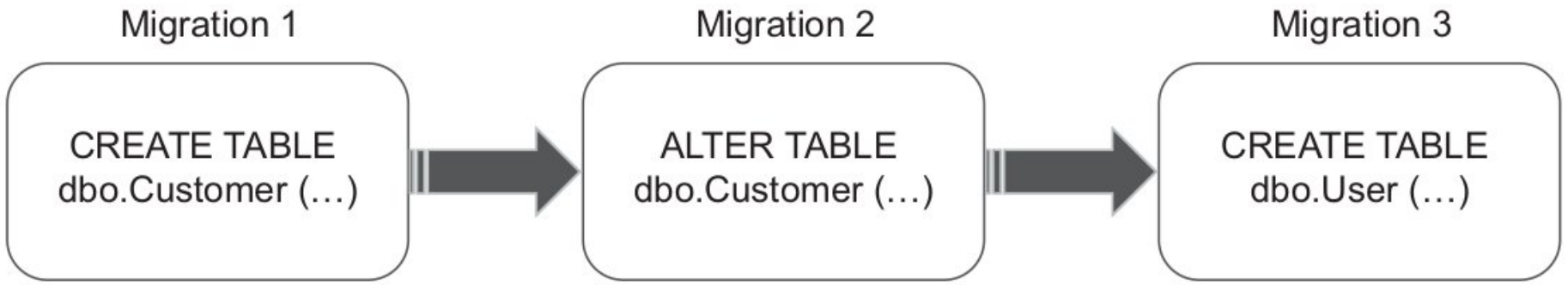
- **State-based approach**
  - In the state-based approach, the comparison tool does everything needed to get it in sync with the model database: delete unnecessary tables, creates new ones, rename columns, and so on.
- **Migration-based approach**
  - Emphasizes the use of explicit migrations that transition the database from one version to another;
  - You don’t use tools to automatically synchronize the production and development databases.

5. TESTING THE DATABASE

Prerequisites for testing the database

4. State-based vs migration-based database delivery

- **Migration-based approach**
  - You come up with upgrade scripts yourself;



5. TESTING THE DATABASE

Prerequisites for testing the database

4. State-based vs migration-based database delivery

- **Prefer the migration-based approach over the state-based approach**
  - The state-based approach makes the state explicit (stored in source control) and lets the comparison tool implicit control the migrations;
  - The migration-based approach makes the migration explicit but leaves the state implicit. It is impossible to view the database state directly. You have to assemble it from the migrations.

	State of the database	Migration mechanism
State-based approach	✓ Explicit	✗ Implicit
Migration-based approach	✗ Implicit	✓ Explicit

5. TESTING THE DATABASE

Database transaction management

- Data transaction is important for both production and test code;
- Proper transaction management in production code helps avoid data inconsistency;
- In tests helps you verify integration with the database in a close-to-production setting;
- To avoid potential inconsistencies, you need to introduce a separation between two types of decisions:
  - What data to update;
  - Whether to keep the updates or roll them back.
- A transaction is a class that either commits or roll back data updates in full;
- A unit of work maintains a list of objects affected by a business operation.

5. TESTING THE DATABASE

Database transaction management

- When it comes to managing database transaction in integration tests, adhere to the following guideline: don’t reuse database transactions or units of work between sections of the test;
- To avoid the risk of inconsistent behaviour, integration tests should replicate the production environment as closely as possible, which means the act section should not share created instances with anymore;
- In the same way, the arrange and assert sections must get their own instances, because, it is important to check the state of the database independently of the data used as input parameters;
- Use at least three transactions or units of work in an integration test: one per each arrange, act, and assert section.

5. TESTING THE DATABASE

Database transaction management

- The shared databases raises the problem of isolating integration tests from each other;
- To solve this you need to:
  - Execute integration tests sequentially;
  - Remove leftover data between test runs.
- Overall, your tests should not depend on the state of you database;
- Your tests should bring that state to the required condition on their own.
- **Parallel vs Sequential test execution**
  - Parallel execution of integration tests involves significant effort. You have to ensure that all test data is unique so no database constrains are violated and test don’t accidentally pick up input data after each other.

5. TESTING THE DATABASE

Database transaction management

- **Parallel vs Sequential test execution**
  - Cleaning up leftover data also becomes trickier;
  - It is more practical to run integration tests sequentially rather than spend time trying to squeeze additional performance out of them.
- **Clearing data between test runs**
  - Restoring a database backup before each call (slower approach);
  - Cleaning up data at the end of a test (fast approach but susceptible of errors);
  - Wrapping each test in a database transaction and never committing it (automatically roll up);
  - Cleaning up data at the beginning of a test (best option).

5. TESTING THE DATABASE

Reusing code in test sections

- Integration tests can quickly grow too large and thus lose ground on the maintainability metric;
- It is important to keep integration test as short as possible but without coupling them to each other or affecting readability;
- They also should preserve the full context of the test scenario and should not require you to examine different parts of the test class to understand what's going on;
- The best way to shorten integration is by extracting technical, non-business-related bits into private methods or helper classes.

JEST – JAVASCRIPT TESTING FRAMEWORK

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Testing Asynchronous Code

- It is common in JavaScript for code to run asynchronous;
- When you have code that runs asynchronously, Jest needs to know when the code it is testing has completed, before it can move on to another test;
- Jest has several ways to handle this.
- **Promises**

```
test('the data is peanut butter', () => {
  return fetchData().then(data => {
    expect(data).toBe('peanut butter');
  });
});
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Testing Asynchronous Code

- **Async/Await**

```
test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});

test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
    await fetchData();
  } catch (e) {
    expect(e).toMatch('error');
  }
});
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Testing Asynchronous Code

- **Callbacks**

```
test('the data is peanut butter', done => {
  function callback(error, data) {
    if (error) {
      done(error);
      return;
    }
    try {
      expect(data).toBe('peanut butter');
      done();
    } catch (error) {
      done(error);
    }
  }

  fetchData(callback);
});
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Testing Asynchronous Code

- **Resolves/Rejects**

```
test('the data is peanut butter', () => {
  return expect(fetchData()).resolves.toBe('peanut butter');
});
```

```
test('the fetch fails with an error', () => {
  return expect(fetchData()).rejects.toMatch('error');
});
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- Often while writing tests you have some setup work that needs to happen before test runs;
- Often you have also some finishing work that needs to happen after test runs;
- **Setup can be repeated for many times:**
  - **beforeEach()** – repeats inner content before each test being executed;
  - **afterEach()** – repeats inner content after each test has been executed.

```
beforeEach(() => {
  initializeCityDatabase();
});

afterEach(() => {
  clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **One-time setup:**
  - In some cases, you only need to do setup once, at the beginning of a file;
  - **beforeAll()** – executes its inner content only once, in the beginning of the execution of a test suite before all tests;
  - **afterAll()** - executes its inner content only once, in the end of the execution of a test suite after all test being executed.

```
beforeAll(() => {
  return initializeCityDatabase();
});

afterAll(() => {
  return clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **Scoping:**
  - By default, the beforeAll() and afterAll() blocks apply to every test in a file;
  - You can also group tests together using a describe block;
  - When they are inside a describe block, the beforeAll() and afterAll() only apply to the tests within that block.

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **Scoping:**

```
// Applies to all tests in this file
beforeEach(() => {
  return initializeCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});

describe('matching cities to foods', () => {
  // Applies only to tests in this describe block
  beforeEach(() => {
    return initializeFoodDatabase();
  });

  test('Vienna <3 veal', () => {
    expect(isValidCityFoodPair('Vienna', 'Wiener Schnitzel')).toBe(true);
  });

  test('San Juan <3 plantains', () => {
    expect(isValidCityFoodPair('San Juan', 'Mofongo')).toBe(true);
  });
});
```

```
beforeAll(() => console.log('1 - beforeAll'));
afterAll(() => console.log('1 - afterAll'));
beforeEach(() => console.log('1 - beforeEach'));
afterEach(() => console.log('1 - afterEach'));
test('', () => console.log('1 - test'));
describe('Scoped / Nested block', () => {
  beforeAll(() => console.log('2 - beforeAll'));
  afterAll(() => console.log('2 - afterAll'));
  beforeEach(() => console.log('2 - beforeEach'));
  afterEach(() => console.log('2 - afterEach'));
  test('', () => console.log('2 - test'));
});

// 1 - beforeAll
// 1 - beforeEach
// 1 - test
// 1 - afterEach
// 2 - beforeAll
// 1 - beforeEach
// 2 - beforeEach
// 2 - test
// 2 - afterEach
// 1 - afterEach
// 2 - afterAll
// 1 - afterAll
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- **Order of execution of describe and test blocks:**
  - Jest executes all describe handlers in a test before it executes any of the actual tests;
  - Once the describe blocks are complete, by default Jest runs all the tests serially in the order they where encountered in the collection phase, waiting for each to finisish and be tidied up before moving on.

```
describe('outer', () => {
  console.log('describe outer-a');

  describe('describe inner 1', () => {
    console.log('describe inner 1');
    test('test 1', () => {
      console.log('test for describe inner 1');
      expect(true).toEqual(true);
    });
  });

  console.log('describe outer-b');

  test('test 1', () => {
    console.log('test for describe outer');
    expect(true).toEqual(true);
  });

  describe('describe inner 2', () => {
    console.log('describe inner 2');
    test('test for describe inner 2', () => {
      console.log('test for describe inner 2');
      expect(false).toEqual(false);
    });
  });

  console.log('describe outer-c');
});

// describe outer-a
// describe inner 1
// describe outer-b
// describe inner 2
// describe outer-c
// test for describe inner 1
// test for describe outer
// test for describe inner 2
```

6. JEST - JAVASCRIPT TESTING FRAMEWORK

Setup and Teardown

- If a test is failing, one of the first things to check should be whether the test is failing when it is the only test that runs;
- To run only one test with jest, temporarily change that test command to a test.only.

```
test.only('this will be the only test that runs', () => {
  expect(true).toBe(false);
});

test('this test will not run', () => {
  expect('A').toBe('A');
});
```