UNIT TESTS PASSING
NO INTEGRATION TESTS
imgflip.com

Module 2-8

Integration Testing

Objectives

- What is an integration test?
- DAO Integration testing

Integration Testing

- Broad category of tests that validate integration between
  - Units of code
  - Outside dependencies such as databases or network resources

Integration Testing

- Use same tools as unit tests (i.e. Junit)
- Usually slower than unit tests (but still measured in ms)
- More complex to write and debug
- Can have dependencies on outside resources like files or a database

DAO Integration Testing

DAOs exist solely to interact with database
Best tested with integration tests

Rules of testing:

- DRY – production code should be DRY – don’t repeat yourself
- WET – testing code should be WET – write everything twice

DAO Integration Testing

Integration tests with a database should ensure that the DAO code functions correctly:

- SELECT statements are tested by inserting dummy data before the test
- INSERT statements are tested by searching for the data
- UPDATE statements are tested by verifying dummy data has been changed
- DELETE statements are tested by seeing if dummy data is missing

DAO Integration Testing

Tests should be:

- Repeatable – If test passes/fails on first execution, it should pass/fail on second execution if no code has changed
- Independent – A test should be able to run on its own, independently of other tests, OR together with other tests and have the same result either way
- Obvious – When a test fails, it should be as obvious as possible as to why it failed

How to manage test data

- Remotely Hosted Shared Test Database
  - Advantages:
    - Easy setup
    - Production-like software and (possibly) hardware
  - Disadvantages
    - Lack of test isolation
    - Temptation to rely on existing data (which can change)

How to manage test data

- Locally Hosted Test Database
  - Advantages
    - Production-like software
    - Reliable (local control)
    - Isolation
  - Disadvantages
    - Requires local hardware resources
    - RDBMS needs to be installed and managed

Mocking

- Make a replica or imitation
- Creating objects that simulate the behavior of real objects
- Typically used in unit testing, but we need to create fake data in order to test CRUD statements

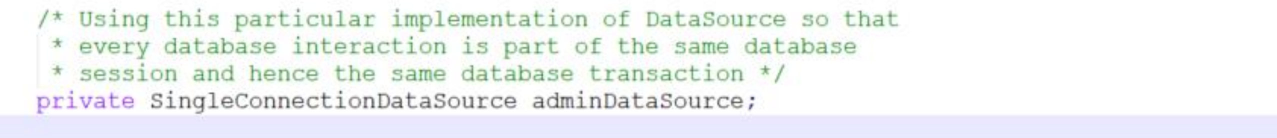
Database considerations

- When testing, we create “test data”
  - Insert new data, update data, or remove rows of data
- Do not want these to be permanent changes
  - Need to roll back changes when done

SingleConnectionDataSource class

- We have used BasicDataSource for our production code
- For integration testing, we use SingleConnectionDataSource
  - Preferred implementation for testing

- Both BasicDataSource and SingleConnectionDataSource are implementations of DataSource



```
/* Using this particular implementation of DataSource so that
 * every database interaction is part of the same database
 * session and hence the same database transaction */
private SingleConnectionDataSource adminDataSource;
```

@PostConstruct method

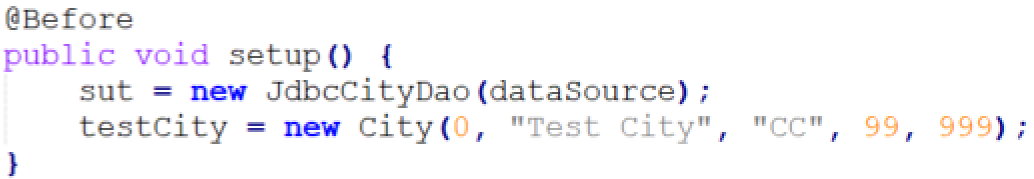
- Generally set up the data source in a @PostConstruct method:

```
/* This method creates the temporary database to be used for the tests. */
   @PostConstruct
   public void setup() {
       if (System.getenv("DB_HOST") == null) {
           adminDataSource = new SingleConnectionDataSource();
           adminDataSource.setUrl("jdbc:postgresql://localhost:5432/postgres");
           adminDataSource.setUsername("postgres");
           adminDataSource.setPassword("postgres1");
           adminJdbcTemplate = new JdbcTemplate(adminDataSource);
           adminJdbcTemplate.update("DROP DATABASE IF EXISTS \"" + DB_NAME + "\";");
           adminJdbcTemplate.update("CREATE DATABASE \"" + DB_NAME + "\";");
       }
   }
```

https://www.baeldung.com/spring-postconstruct-predestroy

@Before method

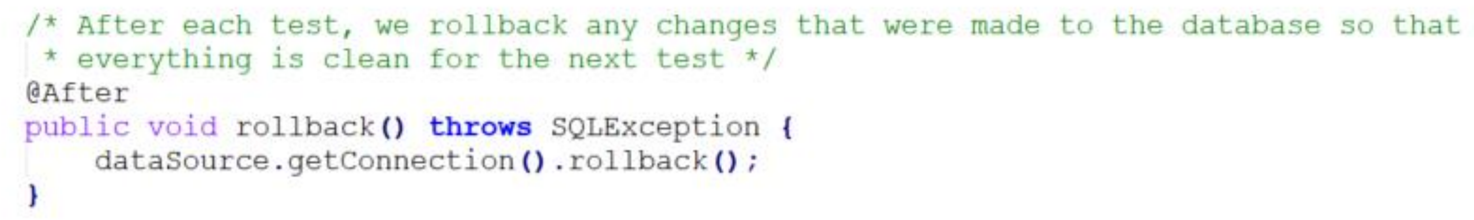
- Where we would insert mocked data into the database:



```
@Before
public void setup() {
    sut = new JdbcCityDao(dataSource);
    testCity = new City(0, "Test City", "CC", 99, 999);
}
```

@After method

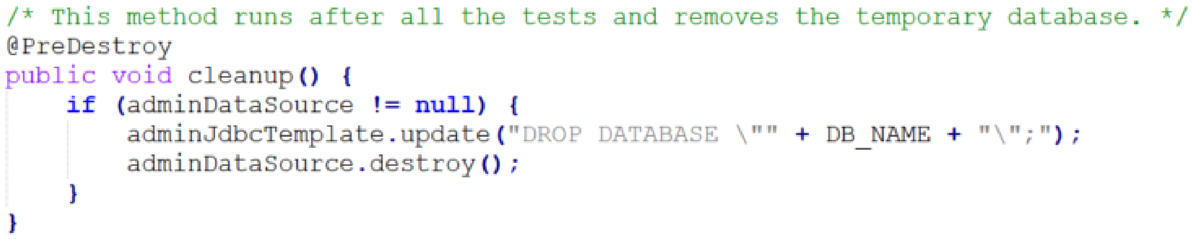
- Want to rollback after each test method runs using the @After annotation:



```java
/* After each test, we rollback any changes that were made to the database so that
 * everything is clean for the next test */
@After
public void rollback() throws SQLException {
    dataSource.getConnection().rollback();
}
```

@PreDestroy method

- Destroy the data source when done with all the tests using the @PreDestroy annotation



```
/* This method runs after all the tests and removes the temporary database. */
@PreDestroy
public void cleanup() {
    if (adminDataSource != null) {
        adminJdbcTemplate.update("DROP DATABASE \"" + DB_NAME + "\";");
        adminDataSource.destroy();
    }
}
```

Module 2-8

Another Look

Integration Testing

Integration Testing is a broad category of tests that validate the integration between units of code or code and outside dependencies such as databases or network resources.

Integration tests in Java

- Use the same tools as unit tests (i.e. JUnit)
- Usually slower than unit tests
- More complex to write and debug
- Can have dependencies on outside resources like files or a database

Test Database Approaches - Shared Database

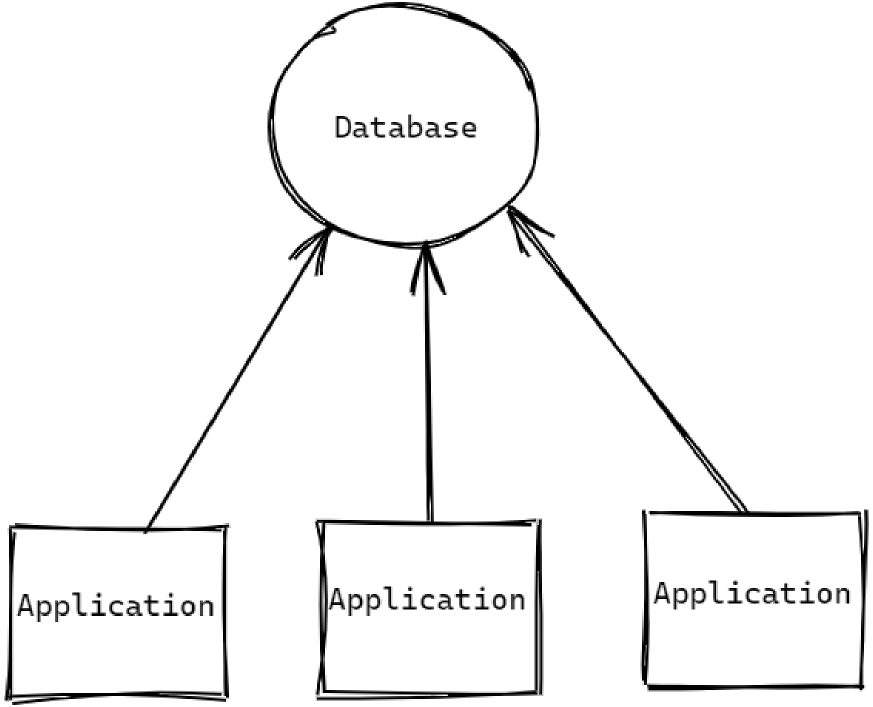
All Developers share a remote test database on the network.

Pros:

- Easy Developer setup
- 1 Setup for all developers
- Production-like software and hardware
- Can be managed by DBAs

Cons:

- Unreliable
- Brittle
- No Isolation
- Temptation to rely on existing data



Test Database Approaches - Local Database

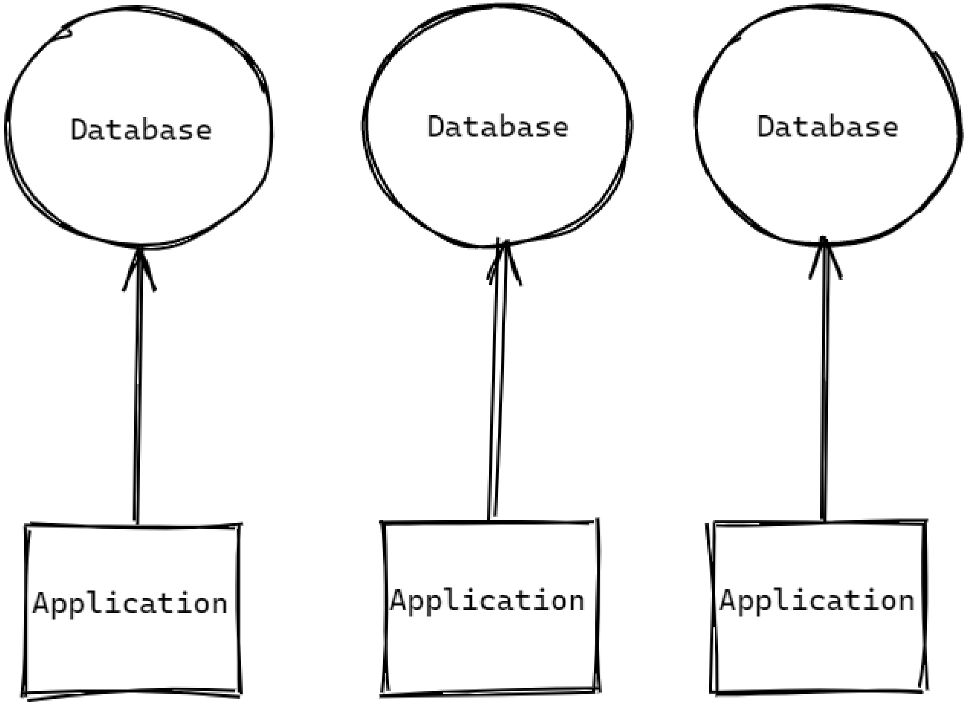
Each developer has their own copy of the database on their computer.

Pros:

- Production-like software
- Reliable
- Isolation

Cons:

- Requires developer to act as DBA
- RDBMS needs to be installed locally, requiring additional licences
- Hardware is not production like
- Production like data can be difficult
- Inconsistent across machines



Test Database Approaches - Embedded Database

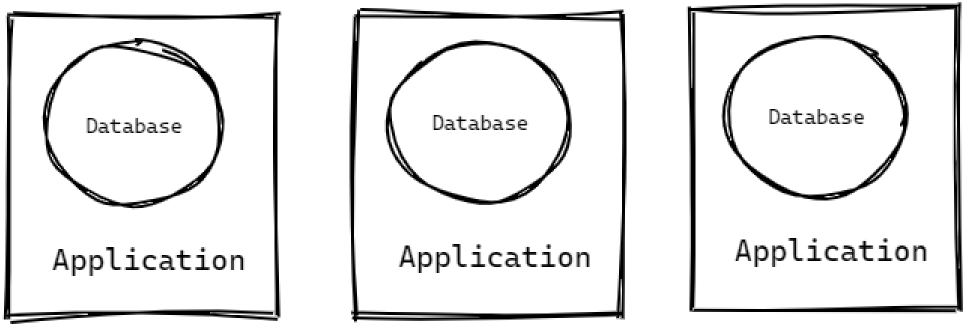
An in-memory database server is started and managed by test code and run inside the application

Pros:

- Very reliable
- Consistent across machines
- Lightweight
- Supports Continuous Integration

Cons:

- Software and hardware is not production like
- Can not use proprietary features of an RDBMS
- Production like data can be difficult



DAO Testing

Integration tests should be:

- *Repeatable:* If the test passes/fails on first execution, it should pass/fail on second execution if no code has changed.
- *Independent:* A test should be able to be run on it's own, independently of other tests, **OR** together with other tests and have the same result either way.
- *Obvious:* When a test fails, it should be as obvious as possible why it failed.

Integration Test should *never use existing data.*

They should always provide their own data.

Transaction Scope

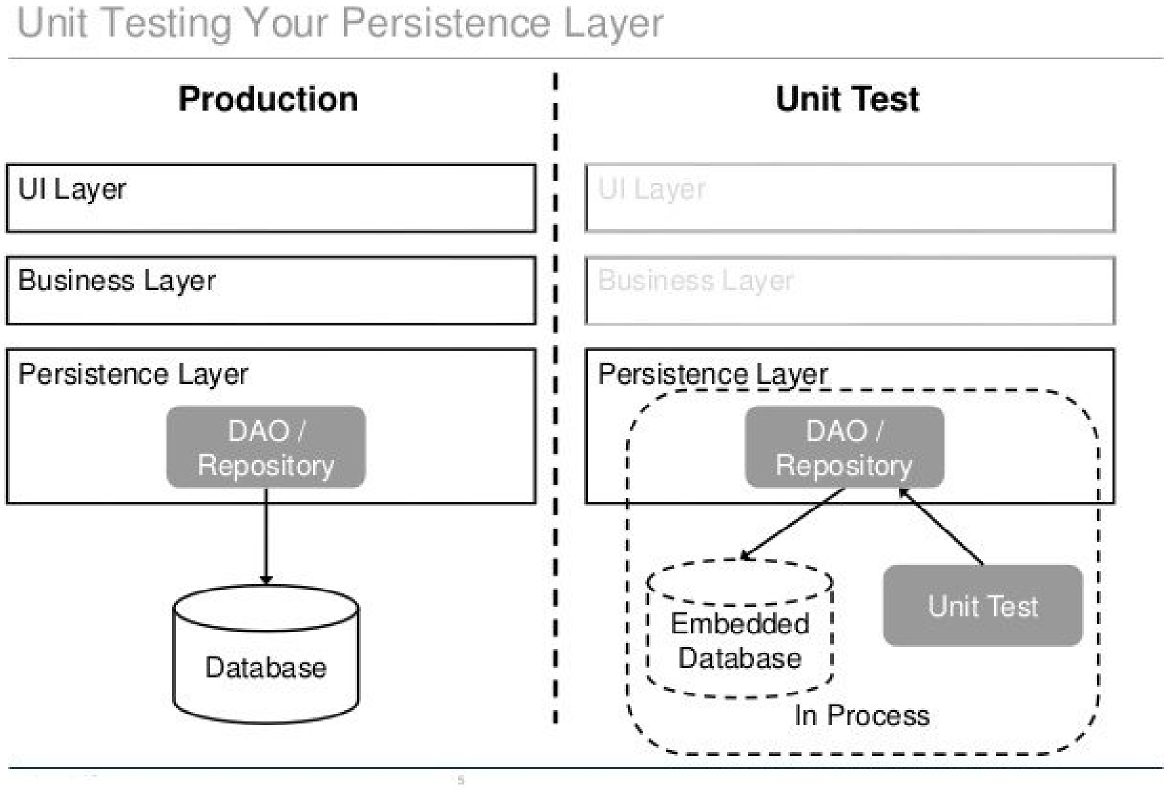
After the test is run the database and data should be in the same state as before the test was run.

Transactions will be used to create an automatic transaction scope that will start a transaction before each test is run and *rollback* after each test has completed. This will prevent the database from being permanently changed during testing.

Our DAOs used the *BasicDataSource* from Apache’s DBCP2 library, which provided a *connection pool*. Since we need to create a *Transaction scope*, a connection pool will not allow steps in our tests to see the changes made by other steps.

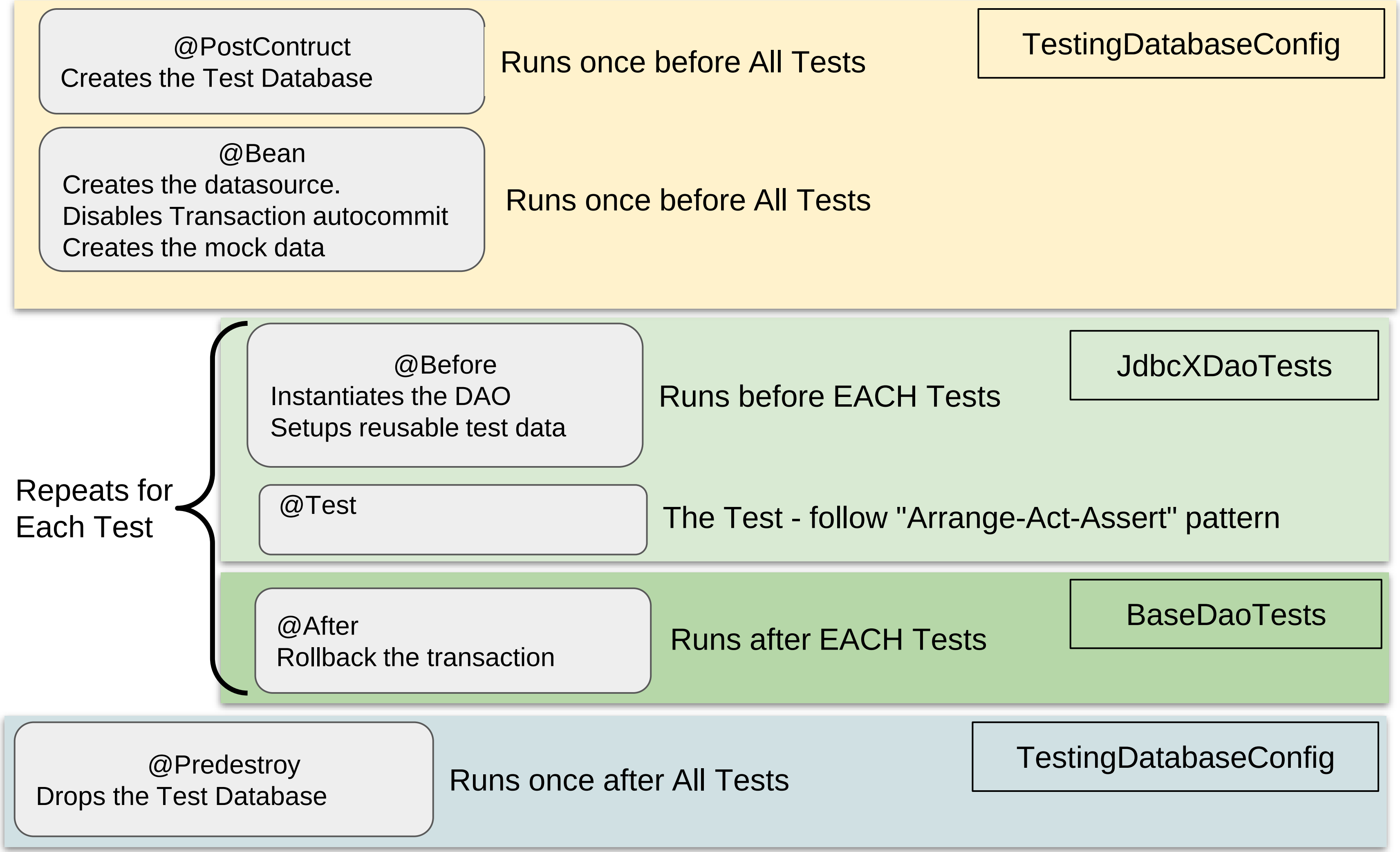
For testing we will use the ***SingleConnectionDataSource***, which will create a direct connection *without a connection pool*, allowing steps to share the connection, and see changes being made by other steps.

Mock Database and Data



Testing is done in a Mock database with Mock data so there is no risk to the actual database.

Integration Test Life Cycle



TestingDatabaseConfig

@PostContruct
Creates the Test Database

Runs once before All Tests

@Bean
Creates the datasource.
Disables Transaction autocommit
Creates the mock data

Runs once before All Tests

JdbcXDaoTests

@Before
Instantiates the DAO
Setups reusable test data

Runs before EACH Tests

Repeats for
Each Test

@Test

The Test - follow "Arrange-Act-Assert" pattern

BaseDaoTests

@After
Rollback the transaction

Runs after EACH Tests

TestingDatabaseConfig

@Predestroy
Drops the Test Database

Runs once after All Tests

Let’s Code!