Schedule

Week	Topics
Week 0	Module 2 Orientation / PostgreSQL
Week 1	Intro to databases / Ordering, limiting, and grouping
Week 2	SQL joins / Insert, update, and delete / Database design
Week 3	Data Access / Data security
Week 4	DAO testing
Week 5	**Mid-module project**
Week 6	Postman / NPM / Networking and HTTP / Consuming RESTful APIs
Week 7	Server-side APIs
Week 8	Securing APIs
Week 9	**End-of-module project**
Week 10	**Assessment**

Objectives

- **Understand the difference between unit tests and integration tests**
- Arrange a minimal data set to use in a DAO integration test
- Write, execute, and interpret results of DAO Integration tests

Integration Testing

- A good unit test isolates the code that it tests from dependencies on outside resources. This is desirable because it usually makes the test more reliable and also easier to debug when it fails.
- However, interactions with outside resources are a potential source of bugs and bad assumptions and should definitely be tested.
- There are classes whose primary function is to interact with outside resources. Data Access Objects are an example of this.
- In order to validate that a JDBC DAO is functioning correctly, we really need to test it against a database. This is an example of an "integration test".

Integration Testing

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

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

Integration Tests should be:

- Repeatable: pass or fail, a test should always have the same result if no code has changed
- Independent: should be able to run with or without 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

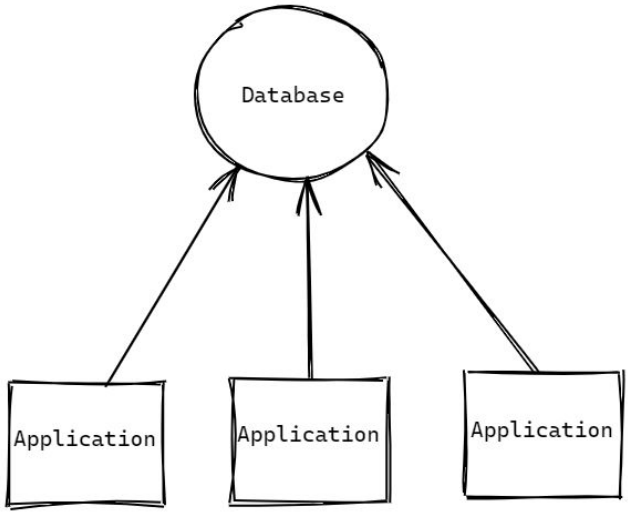
Integration tests with a database should ensure that the DAO code functions correctly:

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

Test Data

Remotely Hosted Shared Test Database

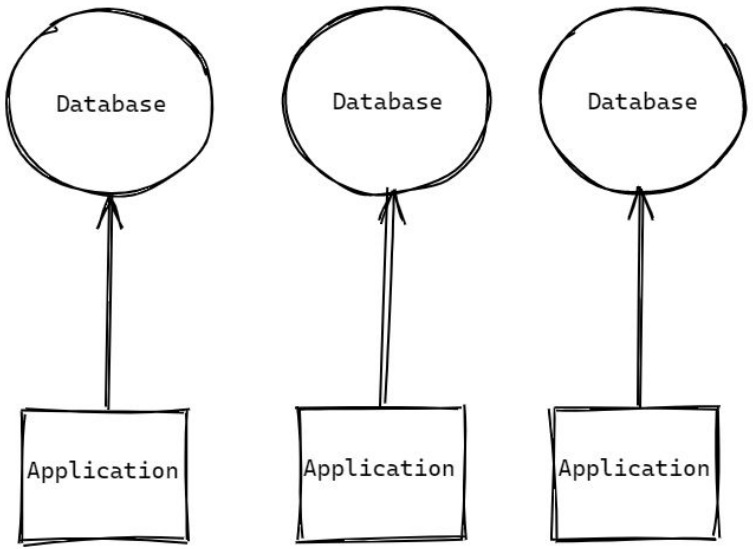
An RDBMS is installed on a remote server and shared by all developers on the team for testing.



Test Data

Locally Hosted Test Database

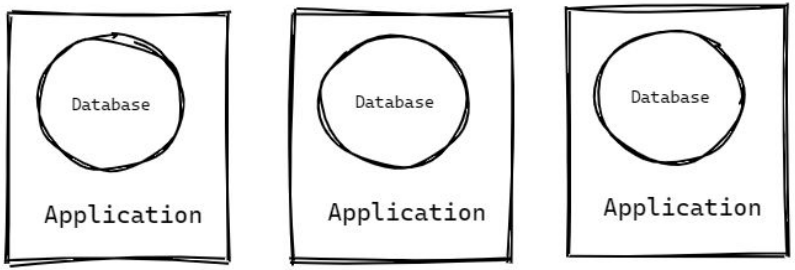
An RDBMS is installed and hosted locally on each developer's machine.
This is the approach we will use.



Test Data

Embedded, In-memory Database

An in-memory, embedded database server is started and managed by test code while running integration tests. For example, H2 is an embedded, open-source, and in-memory database. It is a relational database management system written in Java.



Mocks

- Mocking is a process typically used in unit testing when the unit being tested has external dependencies.
- A mock is a replica or imitation.
- In mock testing, we replace the dependent objects with mock objects that simulate the behavior of the real dependent objects.
- In our DAO testing we are testing the integration between our DAOs and the database so we want to use a real RDBMS database, but with ‘dummy’ data.
- Packages like Mockito and JMock

Understand difference between unit and integration tests: Objective 1

An **integration test is a test that interacts with another system**, such as a file store or database. A **unit test is a self-contained test that has no external dependencies**.

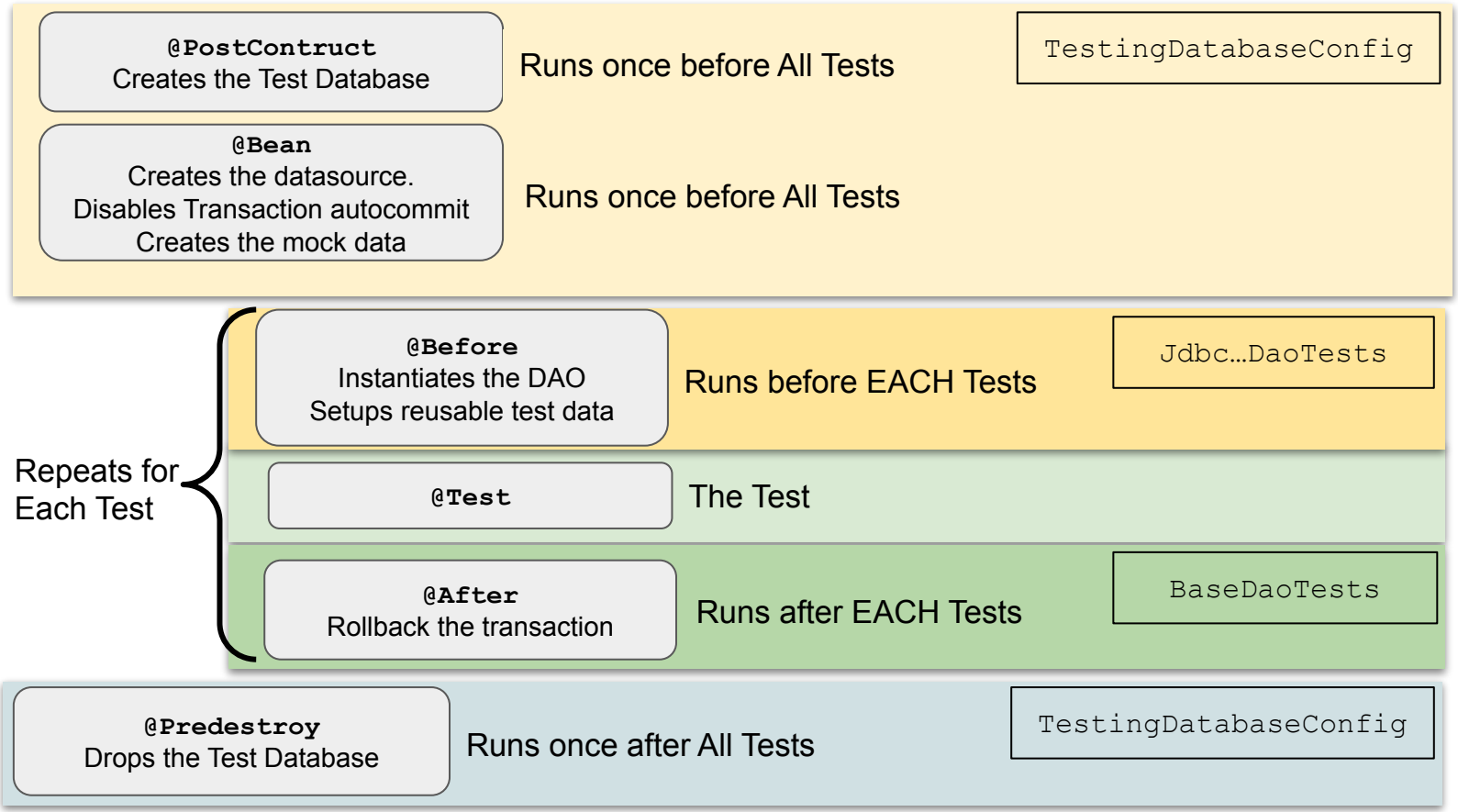
The key difference is that unit tests do not have dependencies on outside data or systems.

Unit	Integration
Single class/unit tested in isolation	Multiple components are tested at once
Dependencies are mocked	Tests interaction with actual dependencies

Objectives

- Understand the difference between unit tests and integration tests
- **Arrange a minimal data set to use in a DAO integration test**
- Write, execute, and interpret results of DAO Integration tests

Testing Life Cycle



Transactions

In order to allow us to rollback transactions, we use `setAutoCommit(false)` when creating the test datasource. This prevents the database from automatically committing after each SQL statement executes. Since queries are not being auto-committed, we can roll them back when our tests complete to avoid permanently affecting data in the database.

```
@Bean
public DataSource dataSource() throws SQLException {
     SingleConnectionDataSource dataSource = new SingleConnectionDataSource();
          dataSource.setUrl("jdbc:postgresql://localhost:5432/UnitedStates");
     dataSource.setUsername("postgres");
     dataSource.setPassword("postgres1");
     dataSource.setAutoCommit(false);
     return dataSource;
}
```

We call **`setAutoCommit(false)`** here.

Transactions

After each test we use the `SingleConnectionDataSource` connection to roll back all changes.

The **`@After`** annotation denotes code to be called after each test completes.

```
@After
public void rollback() throws SQLException {
     dataSource.getConnection().rollback();
}
```

We use the connection of the **`SingleConnectionDataSource`** to roll back any database changes caused by the test.

Transactions

`SingleConnectionDataSource` is useful because it does not close connections automatically after it used. This allows us to roll back changes after each test but it also requires us to destroy the connection manually when we are done using the test class.

```
@PreDestroy
public void doCleanup() {
     dataSource.destroy();
}
```

In the code that gets called when the tests are complete, we call the **`destroy()`** method of the **`SingleConnectionDataSource`** to make sure it is cleaned up properly.

Objectives

- Understand the difference between unit tests and integration tests
- Arrange a minimal data set to use in a DAO integration test
- **Write, execute, and interpret results of DAO Integration tests**

Write, execute, and interpret results of DAO tests: Objective 3

- What should we assert for the test?
- When should we break tests into multiple methods?
- Understand why a test has failed.
- Determine whether the test setup or the code caused a test to fail.

Next Week– Mid-module project!