Clase 9/4

#PAW

NOTA: Postgres ➜ 9.3 (o 9.x) [NO 10.x]

Repaso

Test correcto: valida un único comportamiento. Agnóstico de orden.

1. **Setup**: precondiciones (estado conocido) (evitar depender de otros métodos).
2. **Ejercitar** mi clase (*una* llamada al método particular en *un* caso particular)
3. **Meaningful asserts**: solo cosas que tengan sentido, un buen testing tiene poco asserts. Intentar poder dar resultado unívocos al developer. No voy a validar todo. (1, 2 o 3 assert).

   assert .. == false no me da información ➜ ser semántico

   Cuidado con el *orden* al hacer asserts.

   Todos los métodos tiene un *overload* que da información.

   Escribir un buen *unit test* **lleva tiempo**.

   Test Driven Development: defino comportamiento. Escribo test contra interfaz que no tiene implementación (va a fallar). Y ahí empiezo a implementar. Me obliga a hacer **Top Down**. Yo como cliente espero que me exponga un método; no importa como se resuelve mañana.
   Pienso desde la perspectiva de quién usará mi API.

- **Top Down**: desde funciones a comportamiento. Mejores resultados, salvo cuando quiero cómo por sobre qué.
- **Bottom Up**: de lo atómico a lo general. Más fácil.

Spring antes de initializer tests de JUnit arranca contexto, para tener inversion of control.

DataSource de prueba. Versión en memoria (usamos esta porque prefiero velocidad, no me interesa cómo estaba antes) o en archivo.

JdbcTestUtils.deleteFromTables(jdbcTemplate, “users”);

```
@Test
public void testCreate() {
  final User user = userDao.create(USERNAME);

  assertEquals(USERNAME, user.getUsername());

}
```

SERIAL es de posgres ➜ rompe en JUnits ➜ debo abstraerme de qué motor uso, entonces:

Tengo schema.sql en src*main*resources (cuando corre la app) y src*test*resources para test .

Recursos de test *por convención de maven* tienen prioridad.

posgres por definición tiene estado persistente.

```
@Sql(“classpath:schema.sql”) <– que busque este archivo que está en src/test/
resources
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = TestConfig.class)
public class UserJdbcTest {
…
```

```
 @Bean
public DataSourceInitializer dsInitializer(final DataSource ds) {

  final DataSourceInitializer dsi = new DataSourceInitializer();

  dsi.setDataSource(ds);
  dsi.setDataPopulator(databasePopulator());

  return dsi;
}

private DatabasePopulator databasePopulator() {
  final ResourceDatabasePopulator dbp = new ResourceDatabasePopulator();

  dbp.addScript(schemaSql);
  return dbp;

}



...

en WebConfig:

@Value("classpath:schema.sql")
private Resource
```

ver: *Introducción a Spring JDBC y JUnit.md.pdf*

mvn eclipse:clean eclipse:eclipse

en services:

agrego dependencia junit en service/pom

```
public class UserServiceImplTest {

  private static final String USERNAME = "username";

  private UserServiceImpl us;

  @Mock
  private UserDao dao;

  @Mock
  private User userMock;

  @Before
  public void setUp() {
      MockitoAnnotations.initMocks(this);
      us = new UserServiceImpl();
      us.setUserDao(dao); /ver A1 abajo/
  }

  @Test
  public void testCreateUser() {
      Mockito.when(dao.create(Mockito.any())).thenReturn(userMock);
      User user = us.create(USERNAME);

      assertSame(userMock, user);
      /** mal porque testeo la implementación de create de mi servicio, yo
quiero comportamiento. sí tendría sentido para mailing **/
      Mockito.verify(dao.create(Mockito.eq(USERNAME)));
      Mockito.verifyNoMoreInteractions(dao);
      /** end **/
  }

}
```

Normalmente lo anterior no se testea porque es un pasamanos, pero qué quiero testear? Lo que sea responsabilidad de este servicio y no otra cosa. Quiero aislarme del DAO ➜ le doy un Dao de juguete que me garantiza cierto comportamiento. ➜ Entonces vamos a usar **Mockito**. ➜ devuelve valores defaults ➜ reduce overhead (evito usar Sprint).

A1
en UserServiceImpl

```
/* default */ void setUserDao(UserDao dao) {
  this.userDao = dao;
}
```

Nota: aclarar con * *default* * la intención.

verifyNoMoreInteractions ⬅ es mala idea