# JUnit + Mockito

Unit Testing and Integration Testing

## What is Unit Testing?

- Practice of testing small pieces of code, typically individual functions, alone and isolated. Testing external resource eg: the network or a database **is not Unit test**
- Preventing regressions – bugs that occur repeatedly. Many times there’s been a particularly troublesome piece of code which just keeps breaking no matter how many times I fix it. By adding unit tests to check for those specific bugs, you can easily prevent situations like that.

## What is Integration Testing?

- As the name suggests, in integration testing the idea is to test how parts of the system work together – the integration of the parts.
- If
    - Test uses the database
    - Test uses the network
    - Test uses an external system (e.g. a queue or a mail server)
    - Test reads/writes files or performs other I/O

  **Then it's not a Unit test , its an Integration test.**

## Brief About JUnit and Mockito

- JUnit is a unit testing framework for the Java programming language.
- Supports Unit testing + Integration Testing

- Mockito is a mocking framework.
- Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing

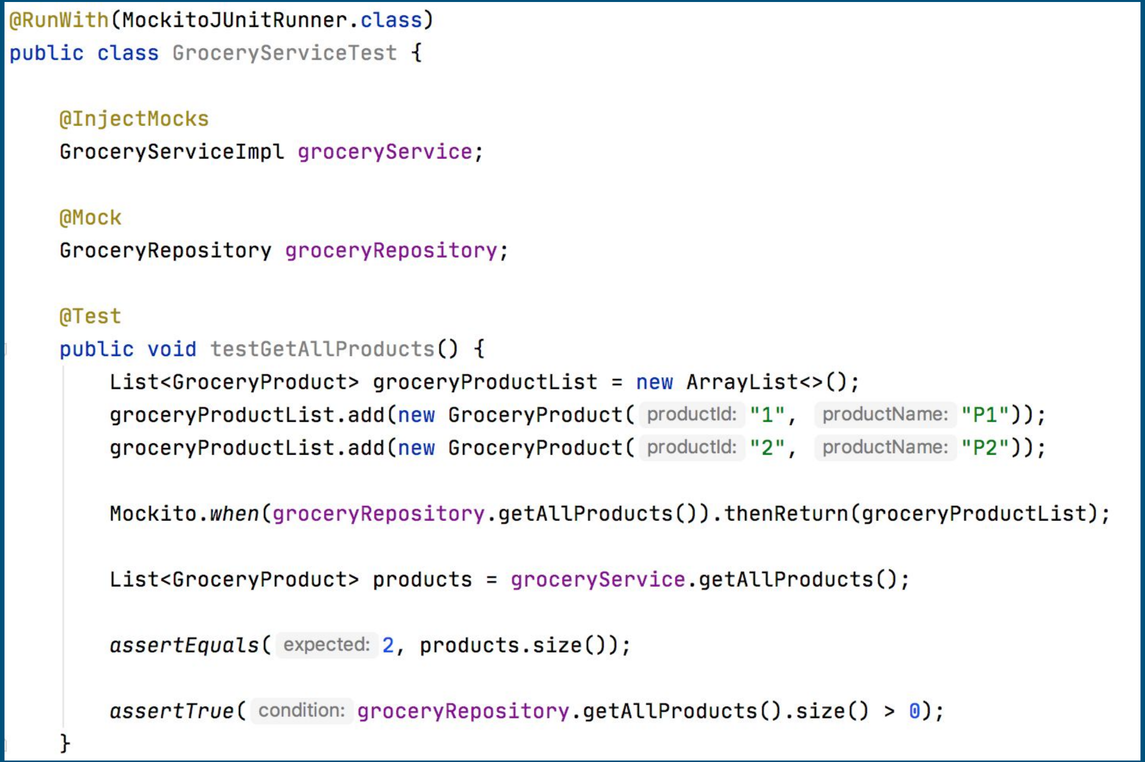
## Why to Mock?

- Object has dependencies that are not yet implemented or in progress
  - For eg:  REST API endpoint which will be available later at some point in time, but you have consumed it in the code via a dependency.
  - Now as the real implementation is still not available,  you really know most of the time what is the expected response of that API. Mocks allow you to test those kinds of integration.
- Component updates the state in the system.
  - DB calls – you would not want to update your DB with data that is just for testing purposes. This might result in corrupting the data, moreover, the availability of DB is another challenge when the test is executed.

# Junit - Terminology

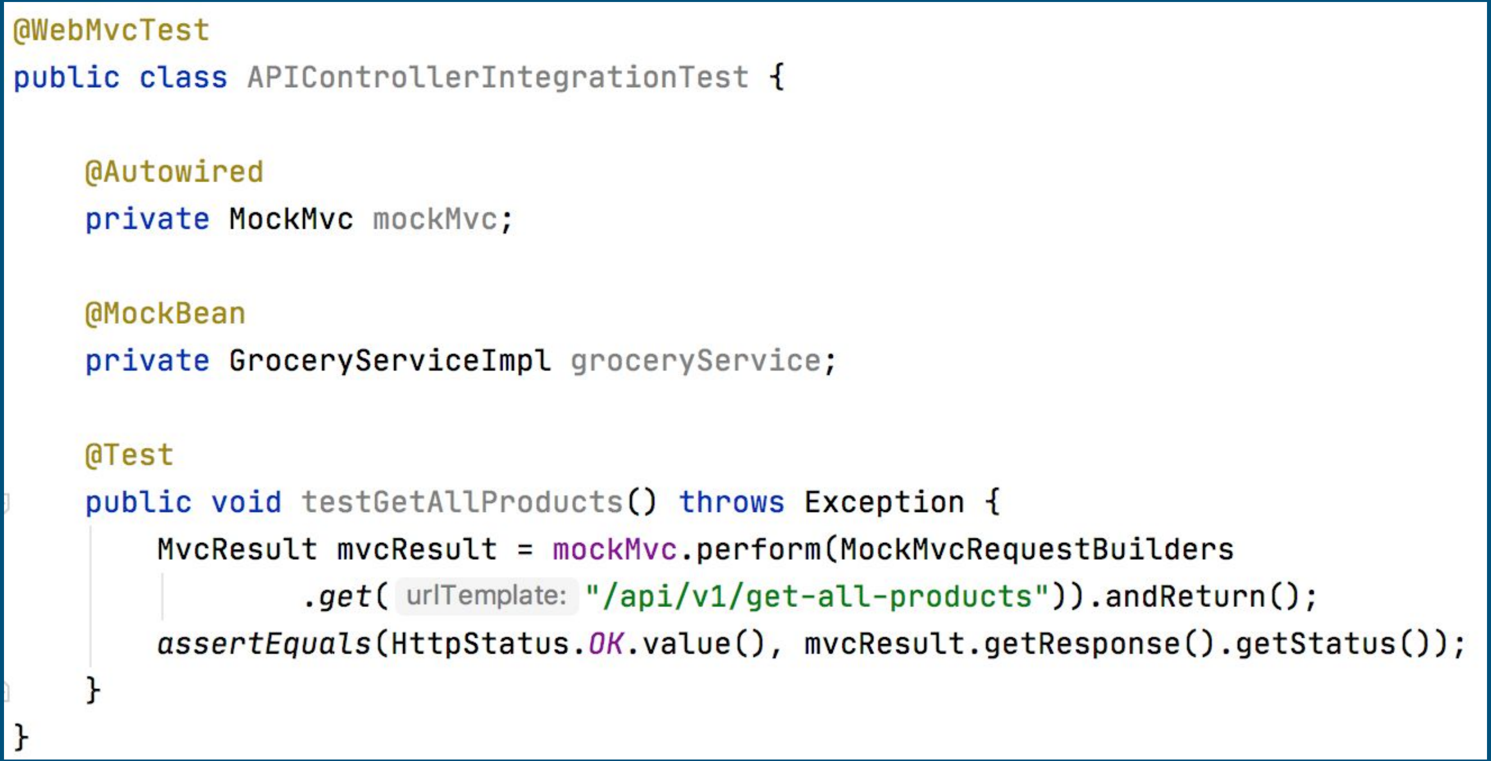
- @RunWith(MockitoJUnitRunner.class)
- @InjectMocks
- @Mock
- @Test
- Mockito.when().thenReturn()
- AssertEquals
- AssertTrue
- Verify
- Argument Captor
- @WebMvcTest
- MockMvc (@Autowired)
- @MockBean
- @DataJpaTest
- TestingEntityManager

## Unit Testing - Service Layer



```java
@RunWith(MockitoJUnitRunner.class)
public class GroceryServiceTest {

    @InjectMocks
    GroceryServiceImpl groceryService;

    @Mock
    GroceryRepository groceryRepository;

    @Test
    public void testGetAllProducts() {
        List<GroceryProduct> groceryProductList = new ArrayList<>();
        groceryProductList.add(new GroceryProduct( productId: "1",  productName: "P1"));
        groceryProductList.add(new GroceryProduct( productId: "2",  productName: "P2"));

        Mockito.when(groceryRepository.getAllProducts()).thenReturn(groceryProductList);

        List<GroceryProduct> products = groceryService.getAllProducts();

        assertEquals( expected: 2, products.size());

        assertTrue( condition: groceryRepository.getAllProducts().size() > 0);
    }
```

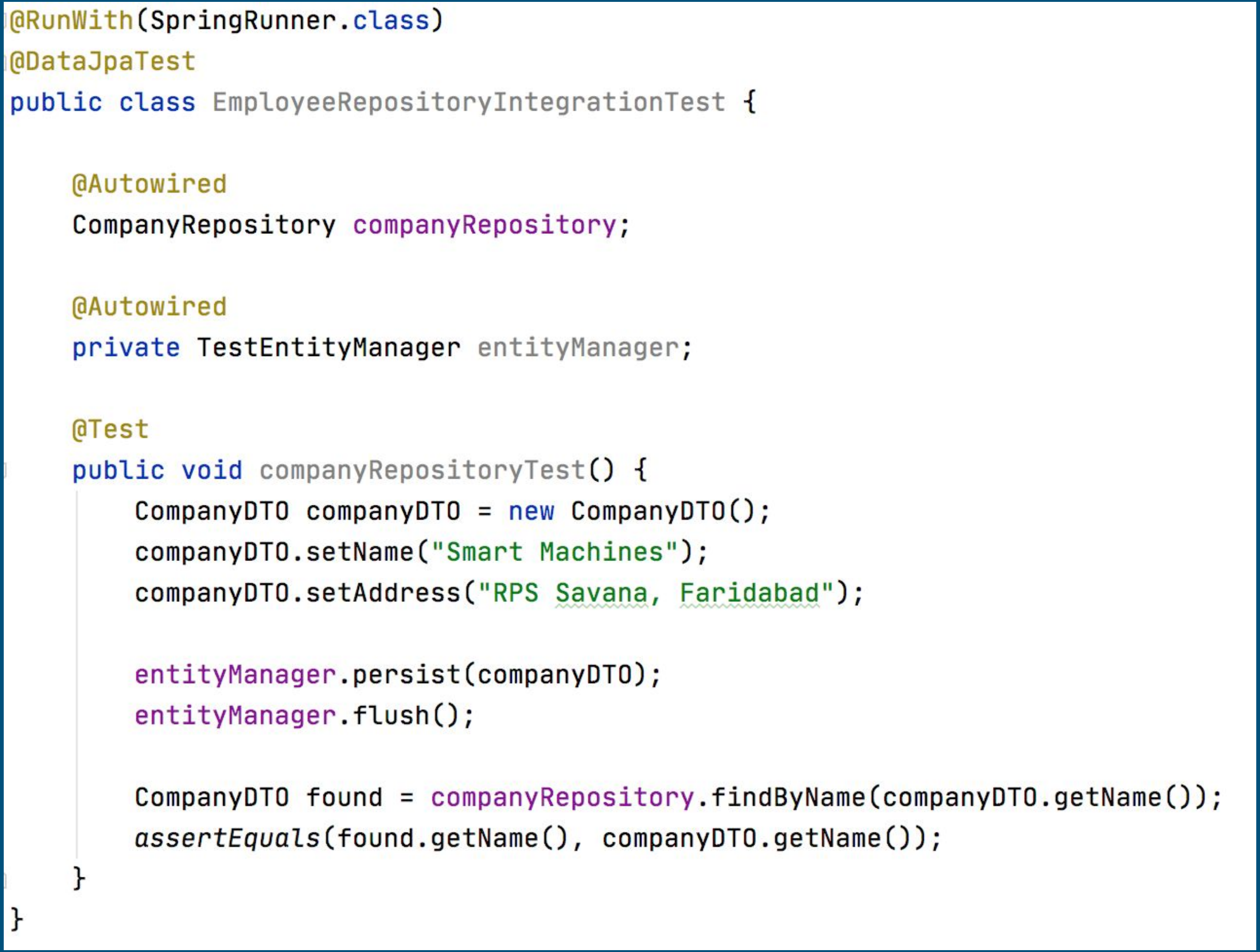
## Integration Testing - Web Layer



```
@WebMvcTest
public class APIControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private GroceryServiceImpl groceryService;

    @Test
    public void testGetAllProducts() throws Exception {
        MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders
                .get( urlTemplate: "/api/v1/get-all-products")).andReturn();
        assertEquals(HttpStatus.OK.value(), mvcResult.getResponse().getStatus());
    }
}
```

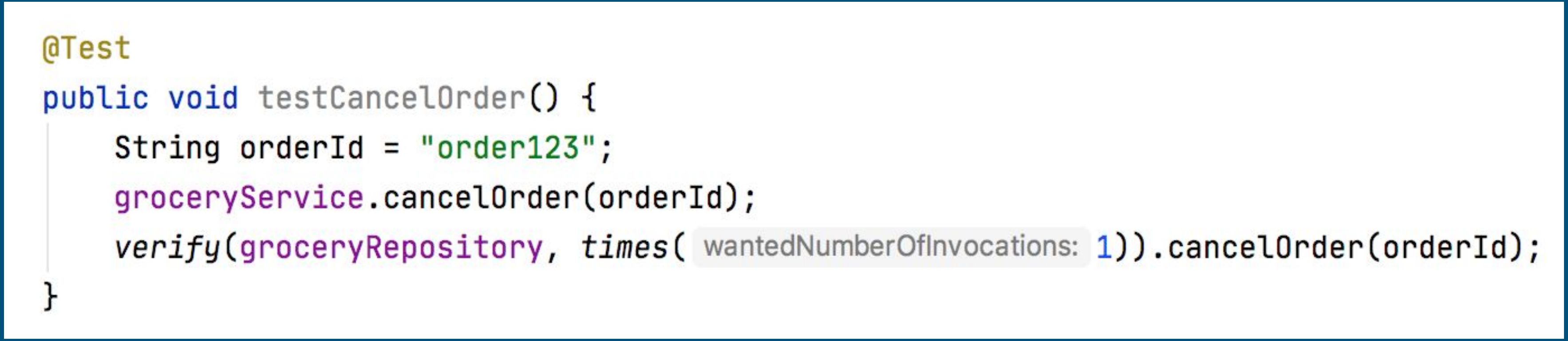
## Integration Testing - Persistence Layer

- @DataJpaTest which requires:
  - Configuring H2, in-memory database
  - Setting Hibernate, Spring Data, and the DataSource
  - Performing an @EntityScan
- TestEntityManager
  - Alternative to the standard JPA EntityManager that provides methods commonly used when writing tests.

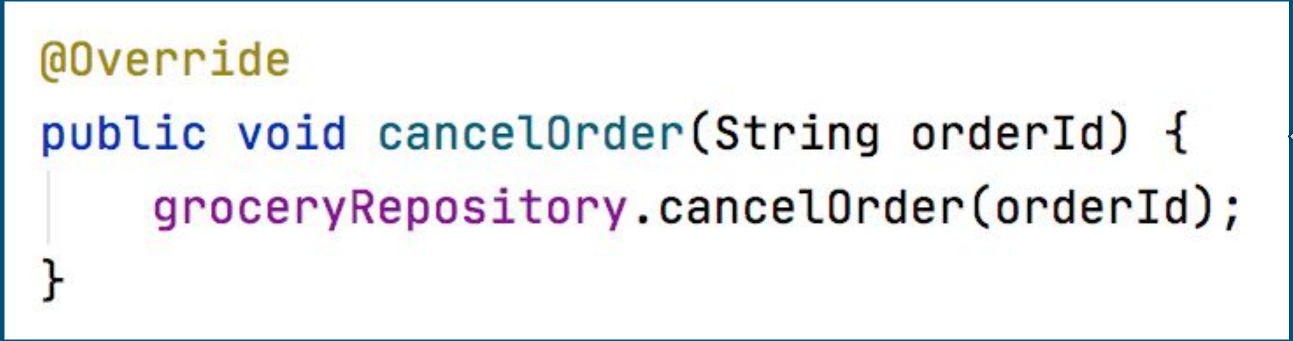


```java
@RunWith(SpringRunner.class)
@DataJpaTest
public class EmployeeRepositoryIntegrationTest {

    @Autowired
    CompanyRepository companyRepository;

    @Autowired
    private TestEntityManager entityManager;

    @Test
    public void companyRepositoryTest() {
        CompanyDTO companyDTO = new CompanyDTO();
        companyDTO.setName("Smart Machines");
        companyDTO.setAddress("RPS Savana, Faridabad");

        entityManager.persist(companyDTO);
        entityManager.flush();

        CompanyDTO found = companyRepository.findByName(companyDTO.getName());
        assertEquals(found.getName(), companyDTO.getName());
    }
}
```

## Testing methods with **void** return type?



```java
@Test
public void testCancelOrder() {
    String orderId = "order123";
    groceryService.cancelOrder(orderId);
    verify(groceryRepository, times( wantedNumberOfInvocations: 1)).cancelOrder(orderId);
}
```

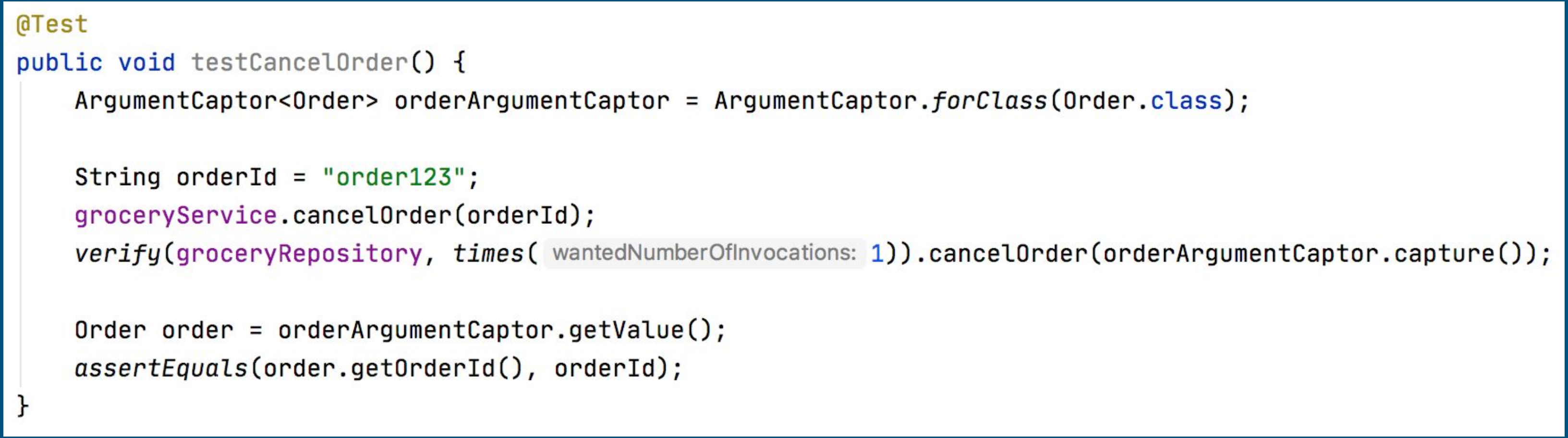


```java
@Override
public void cancelOrder(String orderId) {
    groceryRepository.cancelOrder(orderId);
}
```

GroceryServiceImpl

# Argument Captor

Mockito ArgumentCaptor is used to capture arguments for mocked methods. ArgumentCaptor is used with Mockito verify() methods to get the arguments passed when any method is called.



```java
@Test
public void testCancelOrder() {
    ArgumentCaptor<Order> orderArgumentCaptor = ArgumentCaptor.forClass(Order.class);

    String orderId = "order123";
    groceryService.cancelOrder(orderId);
    verify(groceryRepository, times( wantedNumberOfInvocations: 1)).cancelOrder(orderArgumentCaptor.capture());

    Order order = orderArgumentCaptor.getValue();
    assertEquals(order.getOrderId(), orderId);
}
```

## Can we mock private and static methods?

- For Mockito, there is no direct support to mock private and static methods. In order to test private methods, you will need to refactor the code to change the access to protected (or package) and you will have to avoid static/final methods.
- Powermock extends capabilities of other frameworks like EasyMock and Mockito and provides the capability to mock static and private methods.

## Rules For Good Testable Code

- Reduced no of dependencies or tight coupling
- Separation between logic and presentation
- Code that adheres to SRP (Single Responsibility Principle)
- Less / Minimal usage of static methods and final classes