Testing in Spring

Philippe Collet

pictures: sxc.hu, pixabay

Basic Testing (default Junit support)

Specific container and setup for testing

Injecting the interface under test (and component implementing it thanks to Spring)

```java
@SpringBootTest
class CatalogTest {

    @Autowired
    CatalogExplorator catalog;

    @Test
    void listPreMadeRecipesTest() {
        Set<Cookies> premade = catalog.listPreMadeRecipes();
        assertEquals( expected: 3, premade.size());
    }

    @Test
    void exploreCatalogueTest() {
        assertEquals( expected: 0, catalog.exploreCatalogue( regexp: "unknown").size());
        assertEquals( expected: 2, catalog.exploreCatalogue( regexp: ".*CHOCO.*").size());
        assertEquals( expected: 1, catalog.exploreCatalogue(Cookies.DARK_TEMPTATION.name()).size());
    }
```

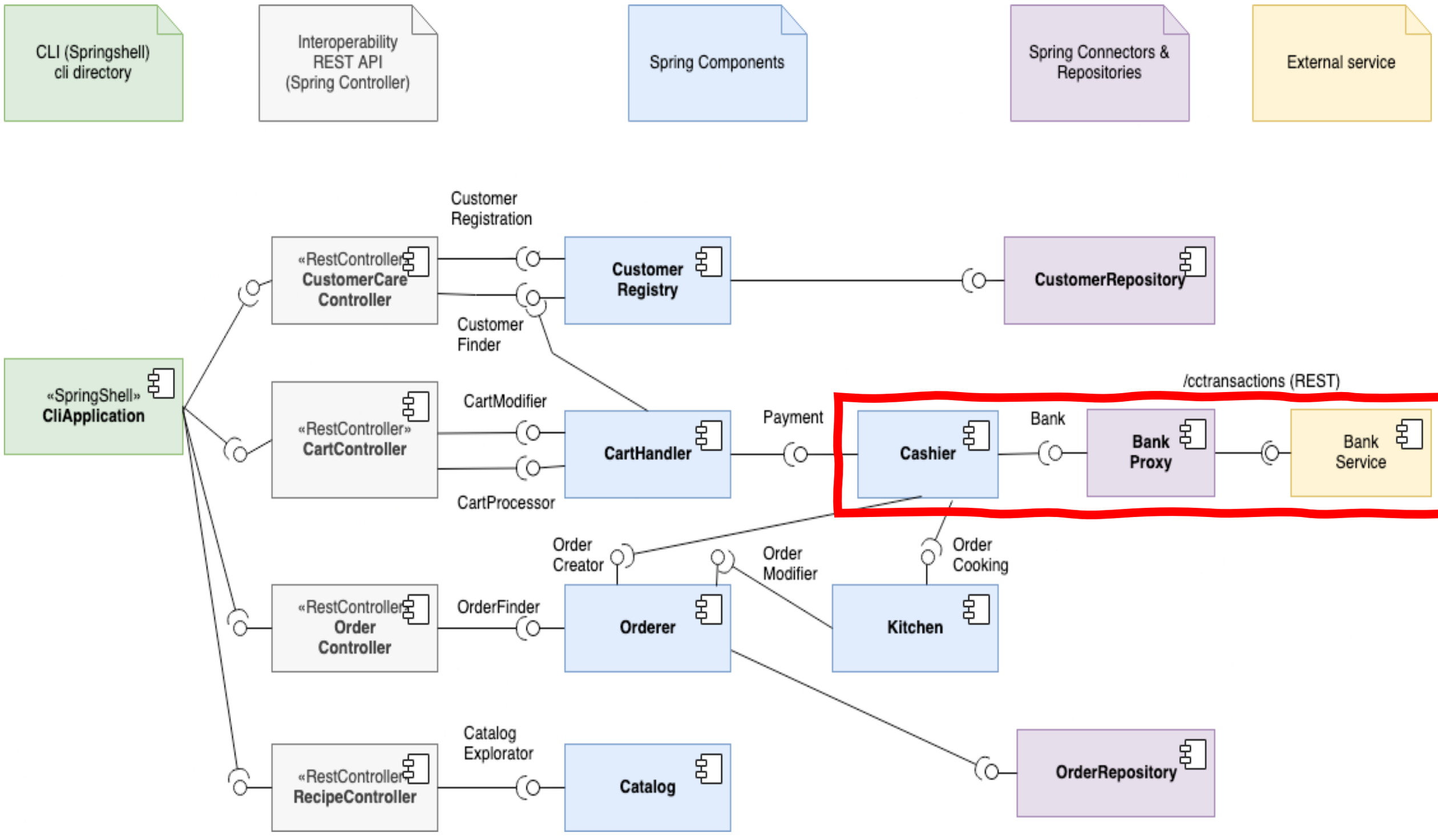
@BeforeEach / @AfterEach (in CartHandlerTest)

```
class CartHandlerTest {

    @Autowired
    private CartModifier cartModifier;

    @Autowired
    private CartProcessor cartProcessor;

    @Autowired
    private CustomerRegistration customerRegistration;

    @Autowired
    private CustomerRepository customerRepository;

    private Long johnId;

    @BeforeEach
    void setUp() throws AlreadyExistingCustomerException {
        johnId = customerRegistration.register("John", "1234567890").getId();
    }

    @AfterEach
    void cleaningUp()  {
        Optional<Customer> toDispose = customerRepository.findCustomerByName("John");
        toDispose.ifPresent(customer -> customerRepository.delete(customer));
        johnId = null;
    }
```

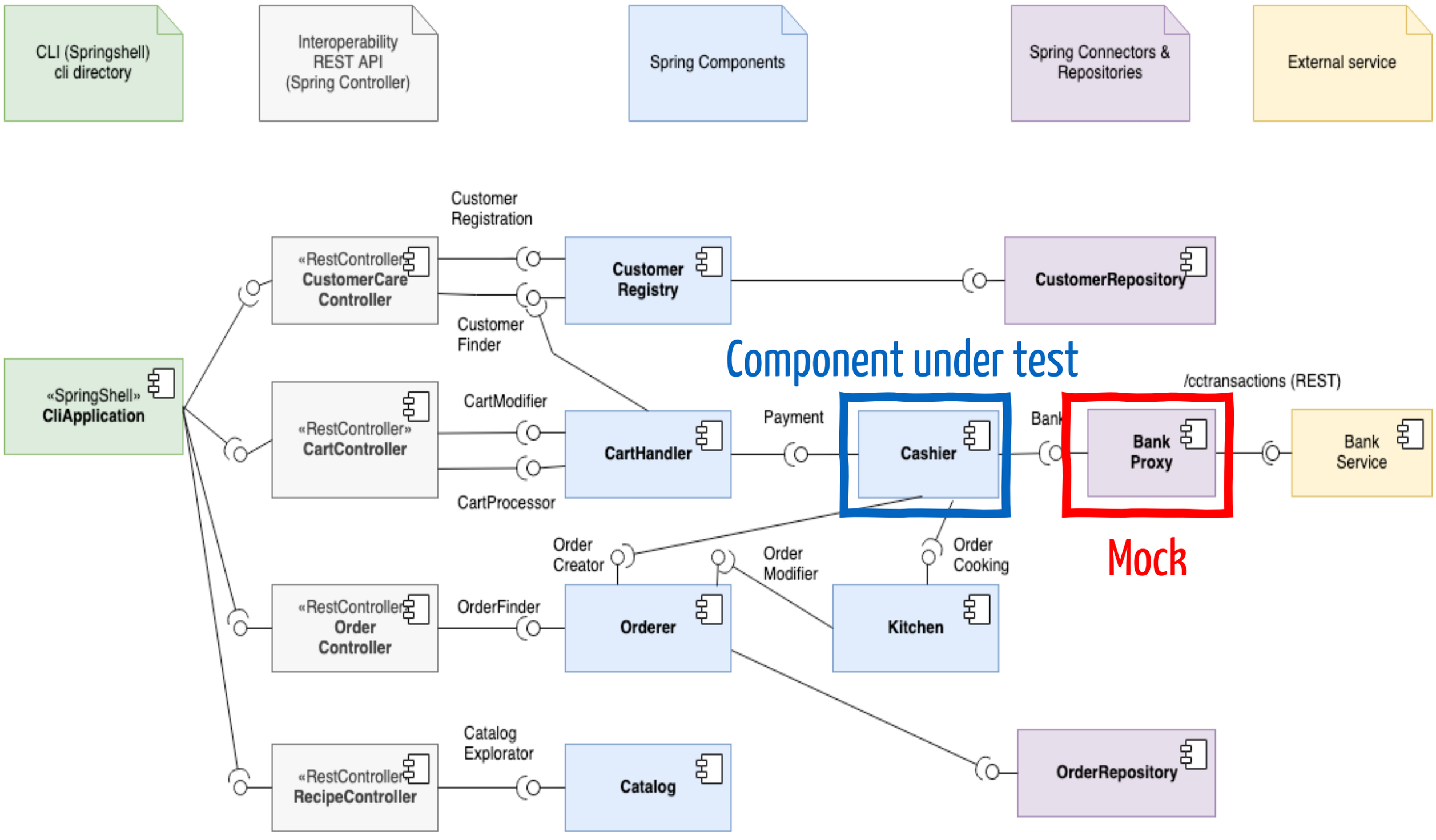
Interfaces needed to make the setup

Interfaces needed to clean up

Mocking?


CLI (Springshell)
cli directory
Interoperability
REST API
(Spring Controller)
Spring Components
Spring Connectors &
Repositories
External service
Customer
Registration
«RestController»
CustomerCare
Controller
Customer
Registry
CustomerRepository
Customer
Finder
«SpringShell»
CliApplication
«RestController»
CartController
CartModifier
CartHandler
Payment
Cashier
Bank
Bank
Proxy
Bank
Service
/cctransactions (REST)
CartProcessor
Order
Creator
Order
Modifier
Order
Cooking
«RestController»
Order
Controller
OrderFinder
Orderer
Kitchen
Catalog
Explorator
«RestController»
RecipeController
Catalog
OrderRepository

Mocking?


CLI (Springshell)
cli directory
Interoperability
REST API
(Spring Controller)
Spring Components
Spring Connectors &
Repositories
External service
Customer
Registration
«RestController»
CustomerCare
Controller
Customer
Registry
CustomerRepository
Customer
Finder
Component under test
«SpringShell»
CliApplication
«RestController»
CartController
CartModifier
CartHandler
Payment
Cashier
Bank
Bank
Proxy
/cctransactions (REST)
Bank
Service
CartProcessor
Order
Creator
Order
Modifier
Order
Cooking
Mock
«RestController»
Order
Controller
OrderFinder
Orderer
Kitchen
Catalog
Explorator
«RestController»
RecipeController
Catalog
OrderRepository

Mocking (the Bank Proxy) – default Mockito support

```
@Test
void processToPayment() throws Exception {
    double price = (3 * Cookies.CHOCOLALALA.getPrice()) + (2 * Cookies.DARK_TEMPTATION.getPrice());
    // paying order
    Order order = cashier.payOrderFromCart(john, price);
    assertNotNull(order);
    assertEquals(john, order.getCustomer());
    assertEquals(items, order.getItems());
    assertEquals(price, order.getPrice(), 0.0);
    assertEquals(2,order.getItems().size());
    assertEquals(OrderStatus.IN_PROGRESS, order.getStatus());
    Set<Order> johnOrders = john.getOrders();
    assertEquals(1, johnOrders.size());
    assertEquals(order, johnOrders.iterator().next());
}

@Test
void identifyPaymentError() {
    Assertions.assertThrows( PaymentException.class, () -> cashier.payOrderFromCart(pat, 44.2));
}
```

Mocking (the Bank Proxy)

```
class CashierTest {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private Payment cashier;

    @MockitoBean
    private Bank bankMock;

    // Test context
    private Set<Item> items;
    private Customer john;
    private Customer pat;
```

@MockitoBean on the interface

Mocking (the Bank Proxy)

```
@BeforeEach
void setUpContext() {
    items = new HashSet<>();
    items.add(new Item(Cookies.CHOCOLALALA, 3));
    items.add(new Item(Cookies.DARK_TEMPTATION, 2));
    // Customers
    john = new Customer("john", "1234896983");  // ends with the secret YES Card number
    john.setCart(items);
    customerRepository.save(john);
    pat  = new Customer("pat", "1234567890");   // should be rejected by the payment service
    pat.setCart(items);
    customerRepository.save(pat);
    // Mocking the bank proxy
    when(bankMock.pay(eq(john),  anyDouble())).thenReturn(Optional.of("playReceiptOKId"));
    when(bankMock.pay(eq(pat),  anyDouble())).thenReturn(Optional.empty());
}
```

Classic mockito when/thenReturn

Set up with 2 databases (dev, test)

From persistence lecture

src/main/resources/application.yaml

```yaml
# postgres DB # IN DOCKER COMPOSE SHOULD BE OVERRIDEN BY ENV VARIABLES

spring:
  datasource:
    # POSTGRES_USER
    username: postgresuser
    # POSTGRES_PASSWORD
    password: postgrespass
    url: jdbc:postgresql://${POSTGRES_HOST}/tcf-db
    driver-class-name: org.postgresql.Driver

jpa:
  properties:
    hibernate:
      dialect: org.hibernate.dialect.PostgreSQLDialect
      format_sql: true
  show-sql: true
  generate-ddl: true
  open-in-view: false
```

Ensures Controllers do not start a default transaction
(bad conceptually and bad for performance)

src/test/resources/application.yaml

```yaml
spring:
  main:
    banner-mode: off
  jpa:
    properties:
      hibernate:
        format_sql: true
    show-sql: true
    open-in-view: false

logging:
  level:
    org:
      springframework: ERROR
```

Test DB setup omitted
(default is H2, a in-memory java DB)

Only configuration changes are done (e.g. display generated SQL statements)

Testing with DBs, repositories and transactions

@Transactional : test runs in a transaction that is automatically rollback at the end

```
@SpringBootTest
@Transactional // default behavior : rollback DB operations after each test (even if it fails)
@Commit // test-specific annotation to change default behaviour to Commit on all tests (could be applied on a me
        // This annotation obliges us to clean the DB (removing the 2 customers) but it is only here for illustr
        // The "rollback" policy should be privileged unless some specific testing context appears
class CashierTest {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private Payment cashier;

    @MockitoBean
    private Bank bankMock;
```

@Commit : Transaction is not rollback but COMMITTED at the end of each test

@Commit -> Need for a proper clean up

```
@BeforeEach
void setUpContext() {
    items = new HashSet<>();
    items.add(new Item(Cookies.CHOCOLALALA,  quantity: 3));
    items.add(new Item(Cookies.DARK_TEMPTATION,  quantity: 2));
    // Customers
    john = new Customer( n: "john",  c: "1234896983");  // ends with the secret YES Card number
    john.setCart(items);
    customerRepository.save(john);
    pat  = new Customer( n: "pat",  c: "1234567890");   // should be rejected by the payment service
    pat.setCart(items);
    customerRepository.save(pat);
    // Mocking the bank proxy
    when(bankMock.pay(eq(john),  anyDouble())).thenReturn(Optional.of( value: "playReceiptOKId"));
    when(bankMock.pay(eq(pat),  anyDouble())).thenReturn( t: Optional.empty());
}

@AfterEach
void cleanUpContext() {
    orderRepository.deleteAll();
    customerRepository.deleteAll();
}
```

Remove @Commit ->
No need for @AfterEach

@Transactional : the classic test

```java
@SpringBootTest
@Transactional
class KitchenTest {

    @Autowired
    private CustomerRegistration registry;

    @Autowired
    private OrderFinder orderFinder;

    @Autowired
    private Kitchen kitchen;

    @Autowired
    private OrderRepository orderRepository;

    private Order order;

    @BeforeEach
    void setUpContext() throws Exception {
        Set<Item> items = new HashSet<>();
        items.add(new Item(Cookies.CHOCOLALALA, quantity: 3));
        items.add(new Item(Cookies.DARK_TEMPTATION, quantity: 2));
        Customer john = registry.register( name: "john", creditCard: "1234-896983");
        Order newOrder = new Order(john, items, price: (3 * Cookies.CHOCOLALALA.getPrice()) + (2 * Cookies.DARK_TEMPTATION.getPrice()), payReceiptId: "
        order = orderRepository.save(newOrder);
        john.addOrder(order);
    }

    @Test
    void processCommand() throws Exception {
        assertEquals(OrderStatus.VALIDATED, order.getStatus());
        kitchen.processInKitchen(order);
        assertEquals(OrderStatus.IN_PROGRESS, orderFinder.retrieveOrderStatus(order.getId()));
    }
```

@BeforeEach to setup and that's all

Test of a component without @Transactional in the test

```java
@Service
public class CartHandler implements CartModifier, CartProcessor {

    @Transactional
    public Item update(Long customerId, Item item) throws NegativeQuantityException, CustomerIdNotFoundException {
```

```java
@SpringBootTest // you can make test non transactional to be sure that transactions are properly handled in
        // controller methods (if you are actually testing controller methods!)
// @Transactional
// @Commit // default @Transactional is ROLLBACK (no need for the @AfterEach
class CartHandlerTest {

    private Long johnId;

    @BeforeEach
    void setUp() throws AlreadyExistingCustomerException {
        johnId = customerRegistration.register( name: "John",  creditCard: "1234567890").getId();
    }

    @AfterEach
    void cleaningUp()  {
        Optional<Customer> toDispose = customerRepository.findCustomerByName("John");
        toDispose.ifPresent(customer -> customerRepository.delete(customer));
        johnId = null;
    }
```

Test of a component without @Transactional in the test

Method @Transactional

```java
@Test
void addItems() throws NegativeQuantityException, CustomerIdNotFoundException {
    Item itemResult = cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA, quantity: 2));
    assertEquals(new Item(Cookies.CHOCOLALALA, quantity: 2), itemResult);
    cartModifier.update(johnId, new Item(Cookies.DARK_TEMPTATION, quantity: 3));
    Set<Item> oracle = Set.of(new Item(Cookies.CHOCOLALALA, quantity: 2), new Item(Cookies.DARK_TEMPTATION, quantity: 3));
    assertEquals(oracle, cartModifier.cartContent(johnId));
}
```

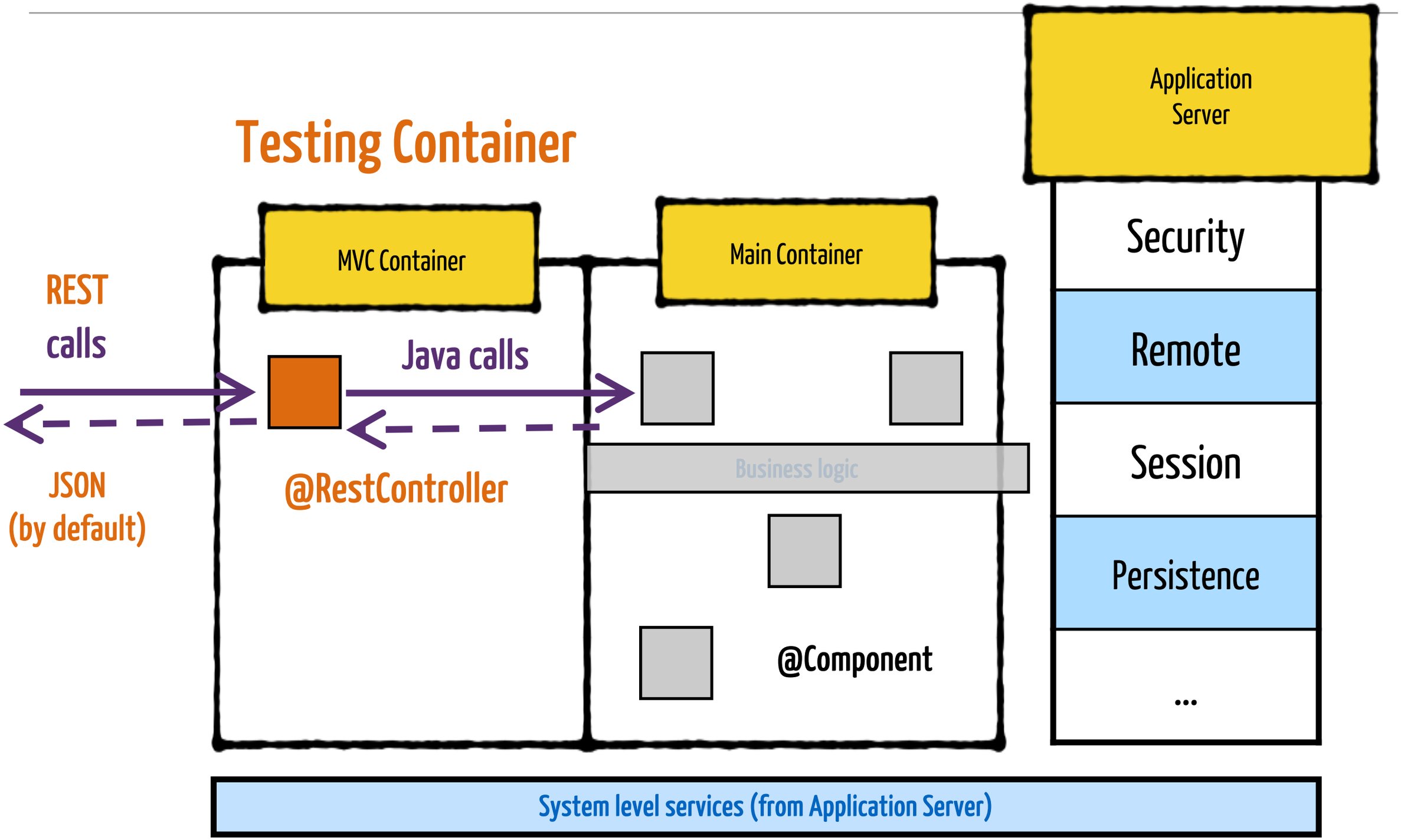
Method @Transactional

```java
@Test
void removeItems() throws NegativeQuantityException, CustomerIdNotFoundException {
    cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA, quantity: 2));
    Item itemResult = cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA, quantity: -2));
    assertEquals(new Item(Cookies.CHOCOLALALA, quantity: 0), itemResult);
    assertEquals( expected: 0, cartModifier.cartContent(johnId).size());
    cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA, quantity: 6));
    cartModifier.update(johnId, new Item(Cookies.CHOCOLALALA, quantity: -5));
    Set<Item> oracle = Set.of(new Item(Cookies.CHOCOLALALA, quantity: 1));
    assertEquals(oracle, cartModifier.cartContent(johnId));
}
```

Test @JpaTest

```java
@DataJpaTest // Only run a test container with the JPA layer (only repositories are up)
// @DataJpaTest is "transactional rollback by default
class CustomerRepositoryTest {

    @Autowired
    private CustomerRepository customerRepository;

    @Test
    void testIdGenerationAndUnicity() {
        Customer john = new Customer( n: "john",  c: "1234567890");
        Assertions.assertNull(john.getId());
        customerRepository.saveAndFlush(john); // save in the persistent context and force saving in the DB (thus ensuring validation by Hibernate
        Assertions.assertNotNull(john.getId());
        Assertions.assertThrows(DataIntegrityViolationException.class, () -> customerRepository.saveAndFlush(new Customer( n: "john",  c: "12345678
        Assertions.assertThrows(DataIntegrityViolationException.class, () -> customerRepository.saveAndFlush(new Customer( n: "john",  c: "12345678
    }

    @Test
    void testFindCustomerByName() {
        Customer john = new Customer( n: "john",  c: "1234567890");
        customerRepository.saveAndFlush(john);
        Assertions.assertEquals(customerRepository.findCustomerByName("john").get(),john);
    }
```

Testing a RestController in isolation



Testing a RestController in isolation

Disables full auto-configuration

RecipeController is the only Controller created in the test envt

```
@WebMvcTest(RecipeController.class)
// start only the specified MVC front controller and no other Spring components nor the server -> Unit test of the co
class RecipeWebMvcTest {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
    private CatalogExplorator mockedCat; // the real Catalog component is not created, we have to mock it
```

MockMvc : to perform call on the controller

Classic @MockitoBean

Testing a RestController in isolation

Mocking the business component

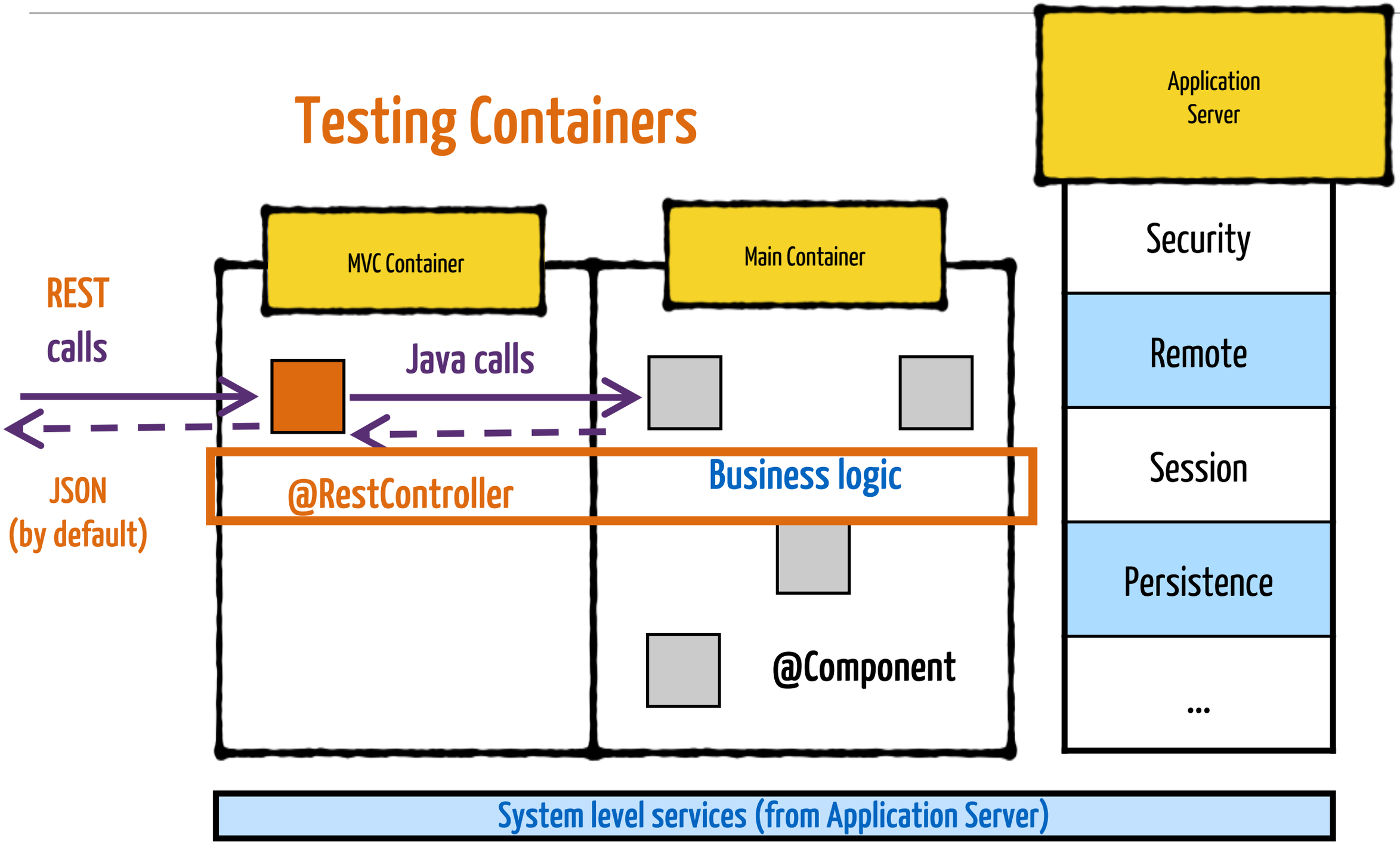
```
@Test
void recipesRestTest() throws Exception {
    when(mockedCat.listPreMadeRecipes())
            .thenReturn(Set.of(Cookies.CHOCOLALALA, Cookies.DARK_TEMPTATION)); // only 2 of the 3 enum values

    mockMvc.perform(get(RecipeController.BASE_URI)
                    .contentType(APPLICATION_JSON))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$").isArray())
            .andExpect(jsonPath( expression: "$", hasSize(2)))
            .andExpect(jsonPath( expression: "$", hasItem("CHOCOLALALA")))
            .andExpect(jsonPath( expression: "$", hasItem("DARK_TEMPTATION")));
}
```

Usage of Builders, Matchers, Handlers

Hitting the API on the route

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
```

Testing a RestController with the full backend



With the full backend

SpringBootTest

Again MockMvc

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc // start the FULL Web+Back stack (all controllers + all components) in an embedded server -> Integration test
class RecipeWebAutoConfigureIT {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void recipesFullStackTest() throws Exception {
        mockMvc.perform(get(RecipeController.BASE_URI)
                        .contentType(APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$").isArray())
                .andExpect(jsonPath( expression: "$", hasSize(3)))
                .andExpect(jsonPath( expression: "$", hasItem("CHOCOLALALA")))
                .andExpect(jsonPath( expression: "$", hasItem("DARK_TEMPTATION")))
                .andExpect(jsonPath( expression: "$", hasItem("SOO_CHOCOLATE")));
    }
```

Setup the full server in testing containers

With the full backend

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
```

WebEnvironment.RANDOM_PORT
starts an embedded server with a random port, which is useful to avoir conflict in test environments while being closer to the real application deployment.

WebEnvironment.DEFINED_PORT
starts an embedded server with a fixed port, usually used in some specific constraints are to be applied on ports.

WebEnvironment.MOCK is the default.
It loads a web application context and provides a mock web environment. It does not load a real http server, just mocks the entire web server behavior. You gain isolation but it is weaker in terms of integration.

WebEnvironment.NONE
loads the business part but does not provide any web environment (mocked or not).

Wait!?

```
public class RecipeWebAutoConfigureIT {
```

https://github.com/CookieFactoryInSpring/simpleTCFS/blob/main/chapters/Testing.md

---

On Testing

- Author: Philippe Collet

We focus here on several kinds of tests that can be done in the Spring stack. It must be noted that some of them can be used to implement integration testing or end to end testing depending on which components are assembled, mocked, and even deployed.

Running different types of test with maven

By default, maven use its *surefire* plugin to run tests. This plugin is especially built for running unit tests, as it will diretly fail if any test fails. This is a good property for preventing the build to be made (the goal *package* will typically fail). However, when you implement integration tests, you usually want to:

- isolate them from unit tests (e.g. to run them only on a CI server),
- use built packages (that have passed unit tests) to put some of them together to setup a context for some integration tests, and cleaning up this context if some tests fail.

The *failsafe* plugin is made for that! From the FAQ:

- *maven-surefire-plugin* is designed for running unit tests and if any of the tests fail then it will fail the build immediately.
- *maven-failsafe-plugin* is designed for running integration tests, and decouples failing the build if there are test failures from actually running the tests.

https://github.com/CookieFactoryInSpring/simpleTCFS/blob/main/chapters/Testing.md

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
...
                <executions>
                    <execution>
                        <goals>
                            <goal>integration-test</goal>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
```

It must be noted that *surefire* will, by default, find tests with the following names and run them during the `test` phase (i.e. just before `package`):

- `"**/Test*.java"` - includes all of its subdirectories and all Java filenames that start with "Test".
- `"**/*Test.java"` - includes all of its subdirectories and all Java filenames that end with "Test".
- `"**/*Tests.java"` - includes all of its subdirectories and all Java filenames that end with "Tests".
- `"**/*TestCase.java"` - includes all of its subdirectories and all Java filenames that end with "TestCase".`

On its side, *failsafe* is integrated in the `verify` phase, and will run integration tests that follow, by default, the following patterns:

- `"**/IT*.java"` - includes all of its subdirectories and all Java filenames that start with "IT".
- `"**/*IT.java"` - includes all of its subdirectories and all Java filenames that end with "IT".
- `"**/*ITCase.java"` - includes all of its subdirectories and all Java filenames that end with "ITCase".

https://github.com/CookieFactoryInSpring/simpleTCFS/blob/main/chapters/Testing.md

---

With this setup, a classic packaging command:

```
mvn clean package
```

will run unit tests, while a `verify` command:

```
mvn clean verify
```

will first run unit tests through *surefire*, and then the integration tests through *failsafe*. In our case, it will run a test of the full backend through a controller and a set of Cucumber tests (see below for details).

If one wants to separate integration tests (e.g., in a CI) the following command will only run them:

```
mvn clean verify '-Dtest=!*' -DfailIfNoTests=false
```

Testing a REST RestClient using MockRestServiceServer

(example from the backend)

```
@RestClientTest(Bank.class)
class BankProxyTest {

    private static final String CC_ROUTE = "/cctransactions";

    @Autowired
    private ObjectMapper objectMapper;

    @Autowired
    private BankProxy bankProxy;

    @Autowired
    private MockRestServiceServer mockServer;

    @Value("${bank.host.baseurl}")
    String bankHostandPort;
```

RestClientTest (with the tested client class as parameter)

Mapper JSON <-> object if needed

Wiring the tested client

mockRestServiceServer

Testing a REST RestClient using MockRestServiceServer

(example from the backend)

```
@Test
void payWithSuccess() throws Exception {
    // Given
    mockServer.expect(requestTo(bankHostandPort + CC_ROUTE))
            .andExpect(method(HttpMethod.POST))
            .andRespond(withStatus(HttpStatus.CREATED)
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(objectMapper.writeValueAsString(new PaymentReceiptDTO("654321", 100.0))));

    // When
    Optional<String> payReceiptId = bankProxy.pay(new Customer("nameIsNotImporta                    100.0);
    // Then
    assertTrue(payReceiptId.isPresent());
    assertEquals("654321", payReceiptId.get());
                                                  expectations were met when calling the mocked url
}
```

Add a first response in the queue

Mocked response

Calling and Asserting

Additional verification if needed (like Spying in Mockito)

Sharing the RestClient to be tested

(example from the cli)

```
@RestClientTest(RecipeCommands.class)
@Import(RestTestConfig.class)
class RecipeCommandsTest {

    @Autowired
    private RecipeCommands client;

    @Autowired
    private MockRestServiceServer mockServer;
```

RestClientTest (with the tested client class as parameter)

Importing the shared test configuration

Defining a restClient that is going to be shared in all tests that import the config

```
@TestConfiguration
class RestTestConfig {

    public static final String TEST_BASE_URL = "http://test";

    @Bean
    RestClient restClient(RestClient.Builder builder) {
        return builder
                .baseUrl(TEST_BASE_URL) // dummy base URL
                .build();
    }
}
```

BDD with cucumber

simpleTCFS / backend / src / test / resources / features / ordering / **OrderingCookies.feature**

```
Feature: Ordering Cookies

  This feature supports the way a Customer can order cookies through the Cookie Factory system

  Background:
    Given a customer named "Maurice" with credit card "1234896983"

  Scenario: The cart is empty by default
    When "Maurice" asks for his cart contents
    Then there is 0 item inside the cart

  Scenario: adding cookies to a cart
    When "Maurice" orders 1 x "CHOCOLALALA"
    And "Maurice" asks for his cart contents
    Then there is 1 item inside the cart
    And the cart contains the following item: 1 x "CHOCOLALALA"

  Scenario: Ordering multiple cookies
    When "Maurice" orders 1 x "CHOCOLALALA"
    And "Maurice" orders 1 x "SOO_CHOCOLATE"
    And "Maurice" asks for his cart contents
    Then there are 2 items inside the cart
    And the cart contains the following item: 1 x "CHOCOLALALA"
    And the cart contains the following item: 1 x "SOO_CHOCOLATE"
```

Implementation steps

simpleTCFS / backend / src / test / java / fr / univcotedazur / simpletcfs / cucumber / ordering / **OrderingCookies.java**

```
@Transactional
public class OrderingCookies {

    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private CartModifier cartModifier;

    @Autowired
    private CartProcessor cartProcessor;

    @Autowired
    private CustomerRegistration customerRegistration;

    @Autowired
    private CustomerFinder customerFinder;
```

No specific Cucumber/Spring configuration except @Transactional (each tests is run inside a transaction and rollback at the end)

Implementation steps

Bug fixing in the autowiring of mocked components

```
@Autowired // Spring/Cucumber bug workaround: autowired the mock declared in the Config class
private Bank bankMock;

private Long customerId;
private Long orderId;
```

```
@Before
public void settingUpContext() throws PaymentException {
    customerRepository.deleteAll();
    orderRepository.deleteAll();
    when(bankMock.pay(any(Customer.class), anyDouble())).thenReturn(Optional.of("payReceiptIdOK"));
}
```

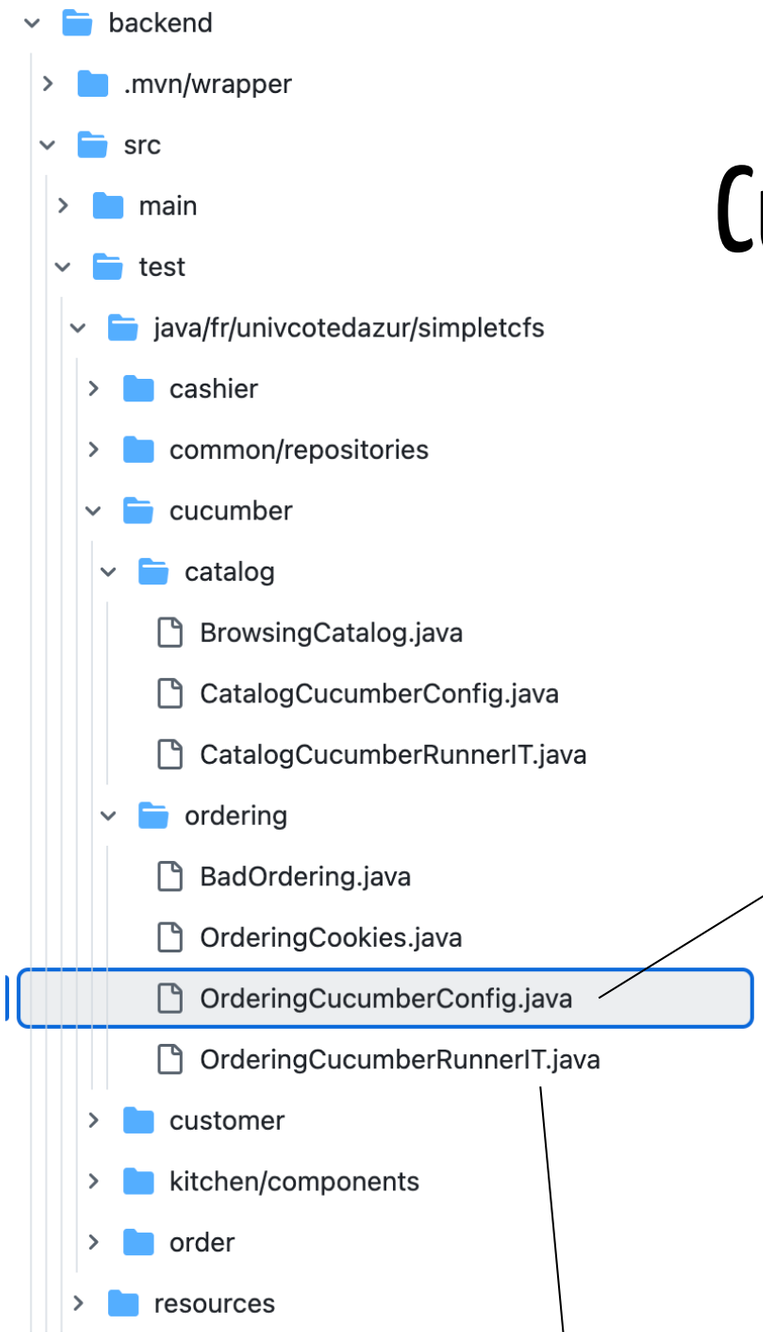
Cucumber specific @Before

Implementation steps

Cucumber « expression »

```
@Then("^there (?:is|are) (\\d+) items? inside the cart$") // Regular Expressions, not Cucumber expression
// Note that you cannot mix Cucumber expression such as {int} with regular expressions
public void thereAreItemsInsideTheCart(int nbItems) throws CustomerIdNotFoundException {
    assertEquals(nbItems, cartModifier.cartContent(customerId).size());
}

@When("{string} orders {int} x {string}")
public void customerOrders(String customerName, int howMany, String recipe) throws NegativeQuantityException,
    this.customerId = customerFinder.findByName(customerName).get().getId();
    Cookies cookie = Cookies.valueOf(recipe);
    cartModifier.update(customerId, new Item(cookie, howMany));
}
```

Note that you cannot mix Cucumber expression such as {int} with regular expressions such as (\\d+)



Cucumber configuration in code

One ContextConfiguration per directory

```java
@CucumberContextConfiguration
@SpringBootTest
public class OrderingCucumberConfig {

    @MockitoBean // Spring/Cucumber setup: declare here the mocks to be used (an
    private Bank bankMock;
}
```

One runner per directory

```java
@Suite
@IncludeEngines("cucumber")
@SelectPackages("features.ordering")
@ConfigurationParameter(key = PLUGIN_PROPERTY_NAME, value = "pretty")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "fr.univcotedazur.simpletcfs.cucumber.ordering")
public class OrderingCucumberRunnerIT { // IT suffix on test classes make them "Integration Test" run by "verify"
}
```

Partial view of dependencies

Cucumber setup 7.31.0 + Junit 5.14.0 + Mockito 5.20.0 (all compatible with SpringBoot 3.5.9)

```
<maven.surefire.plugin.version>3.5.4</maven.surefire.plugin.version>
<junit.jupiter.version>5.14.0</junit.jupiter.version>
<mockito.version>5.20.0</mockito.version>
<cucumber.version>7.31.0</cucumber.version> <!-- Last cucumber versi
```

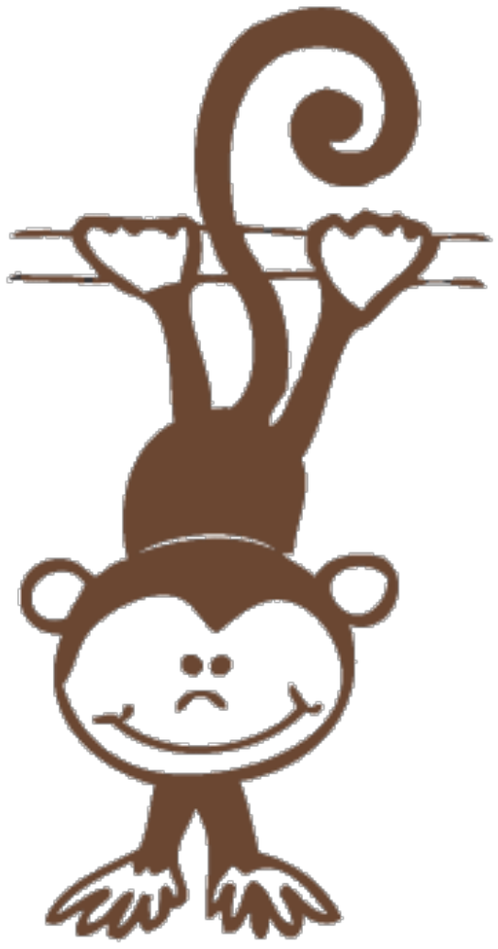
Junit 5 Suite support

```
@Suite
@IncludeEngines("cucumber")
@SelectPackages("features.ordering")
@ConfigurationParameter(key = PLUGIN_PROPERTY_NAME, value = "prett
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "fr.univ
public class OrderingCucumberRunnerIT { // IT suffix on test class
}
```

```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
</dependency>
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit-platform-engine</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-spring</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-junit-jupiter</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

https://github.com/CookieFactoryInSpring/simpleTCFS

monkey see



monkey do