AUTOMATED TESTING MOCKING

A SHORT HISTORY OF AUTOMATED TESTING

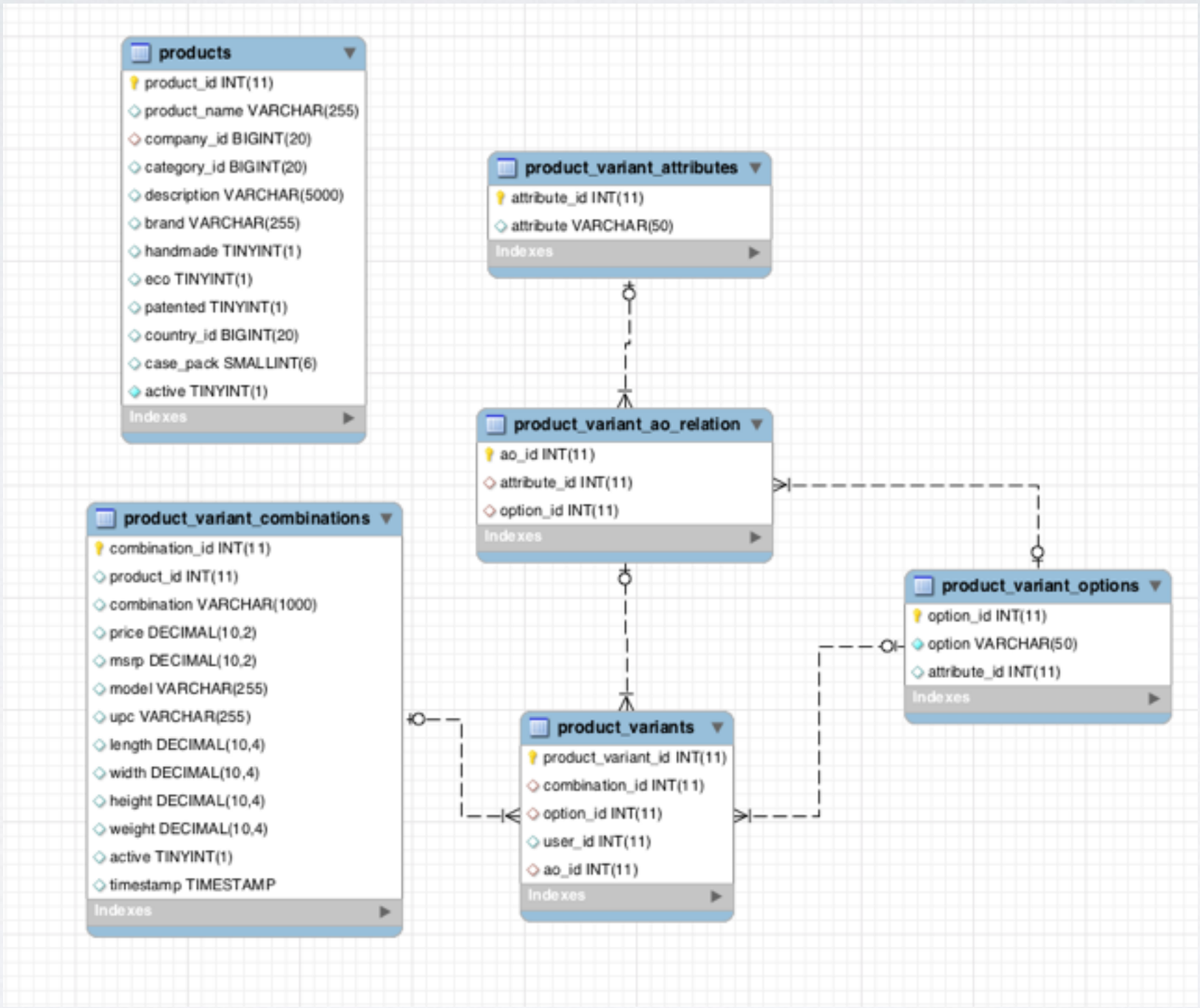
at  LOTARIS

LOTARIS LICENSED MOBILE ENVIRONMENT (LME)

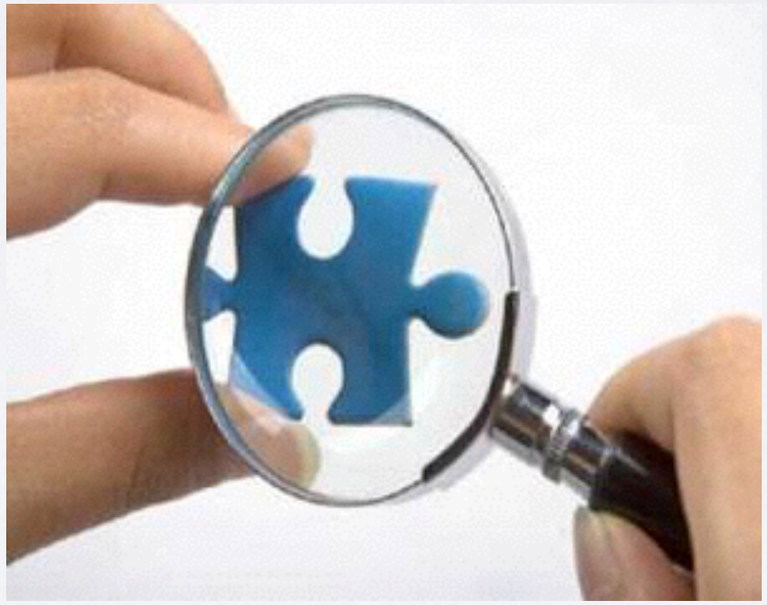
- Allow app developers to monetize their app by including our client library

- Flexible licensing strategy (trial, monthly, one-time)

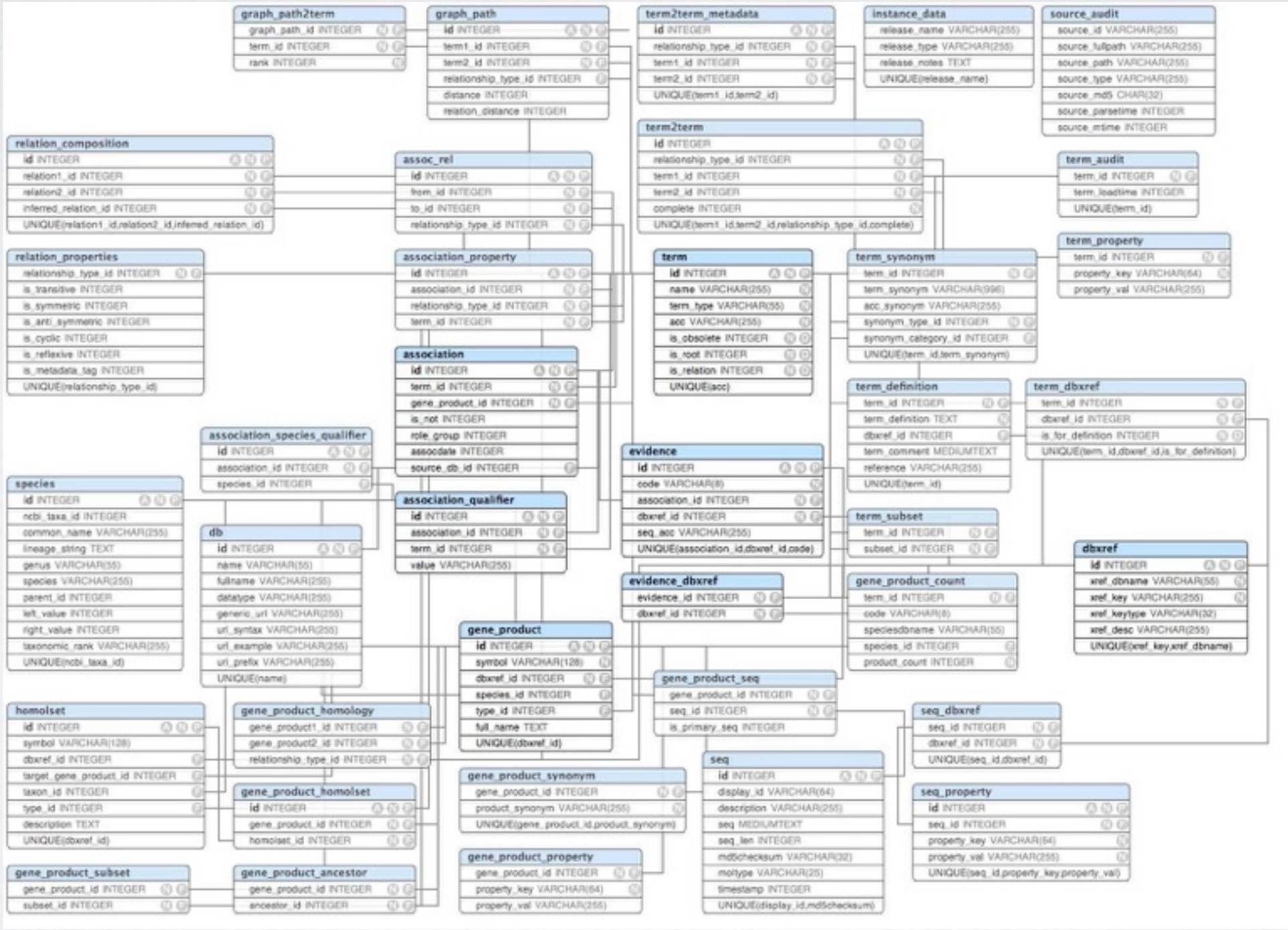
COMPLEXITY


products
product_id INT(11)
product_name VARCHAR(255)
company_id BIGINT(20)
category_id BIGINT(20)
description VARCHAR(5000)
brand VARCHAR(255)
handmade TINYINT(1)
eco TINYINT(1)
patented TINYINT(1)
country_id BIGINT(20)
case_pack SMALLINT(6)
active TINYINT(1)
Indexes
product_variant_attributes
attribute_id INT(11)
attribute VARCHAR(50)
Indexes
product_variant_ao_relation
ao_id INT(11)
attribute_id INT(11)
option_id INT(11)
Indexes
product_variant_combinations
combination_id INT(11)
product_id INT(11)
combination VARCHAR(1000)
price DECIMAL(10,2)
msrp DECIMAL(10,2)
model VARCHAR(255)
upc VARCHAR(255)
length DECIMAL(10,4)
width DECIMAL(10,4)
height DECIMAL(10,4)
weight DECIMAL(10,4)
active TINYINT(1)
timestamp TIMESTAMP
Indexes
product_variants
product_variant_id INT(11)
combination_id INT(11)
option_id INT(11)
user_id INT(11)
ao_id INT(11)
Indexes
product_variant_options
option_id INT(11)
option VARCHAR(50)
attribute_id INT(11)
Indexes

MANUAL TESTING

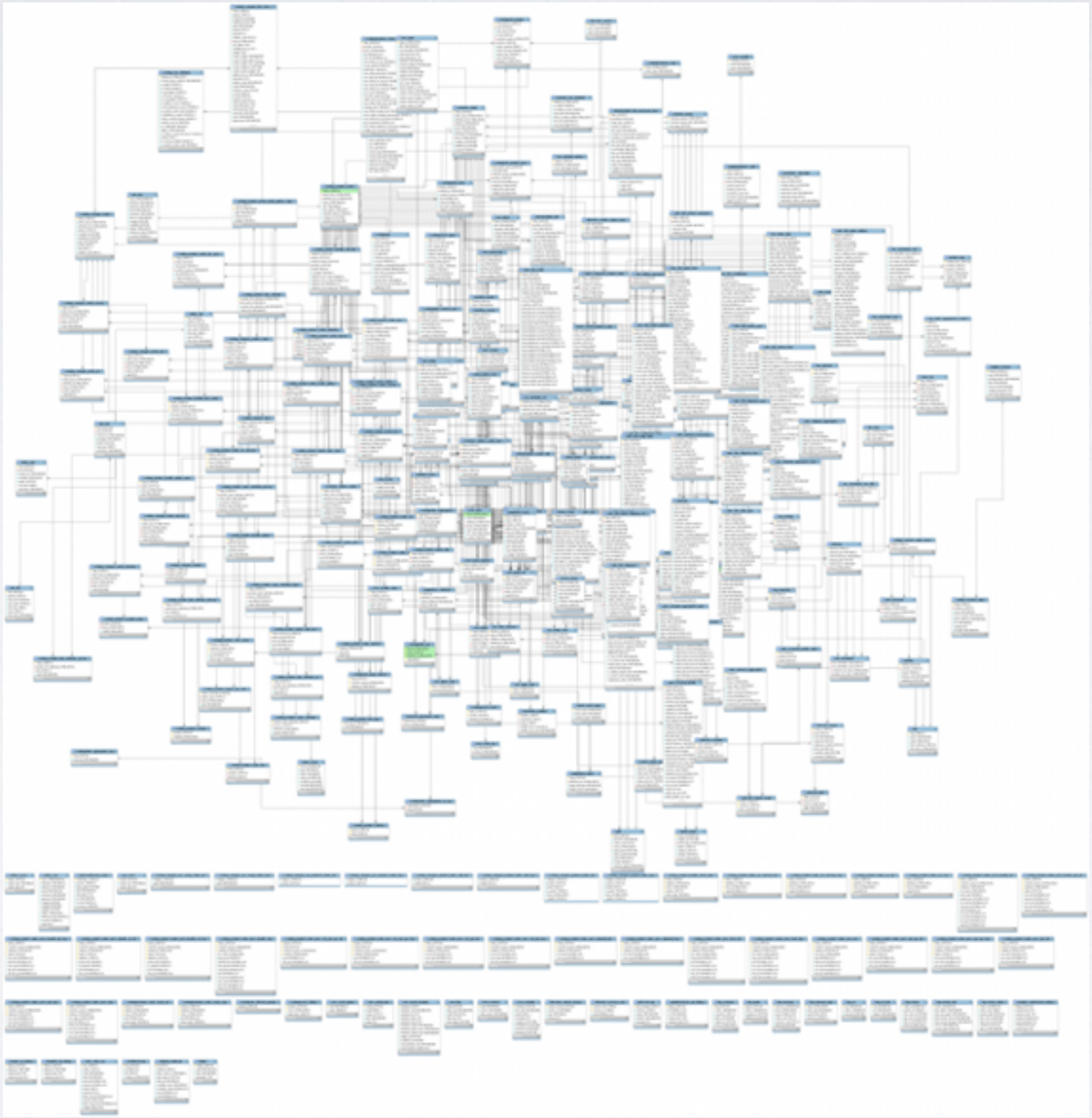


INCREASING COMPLEXITY



QUALITY ASSURANCE

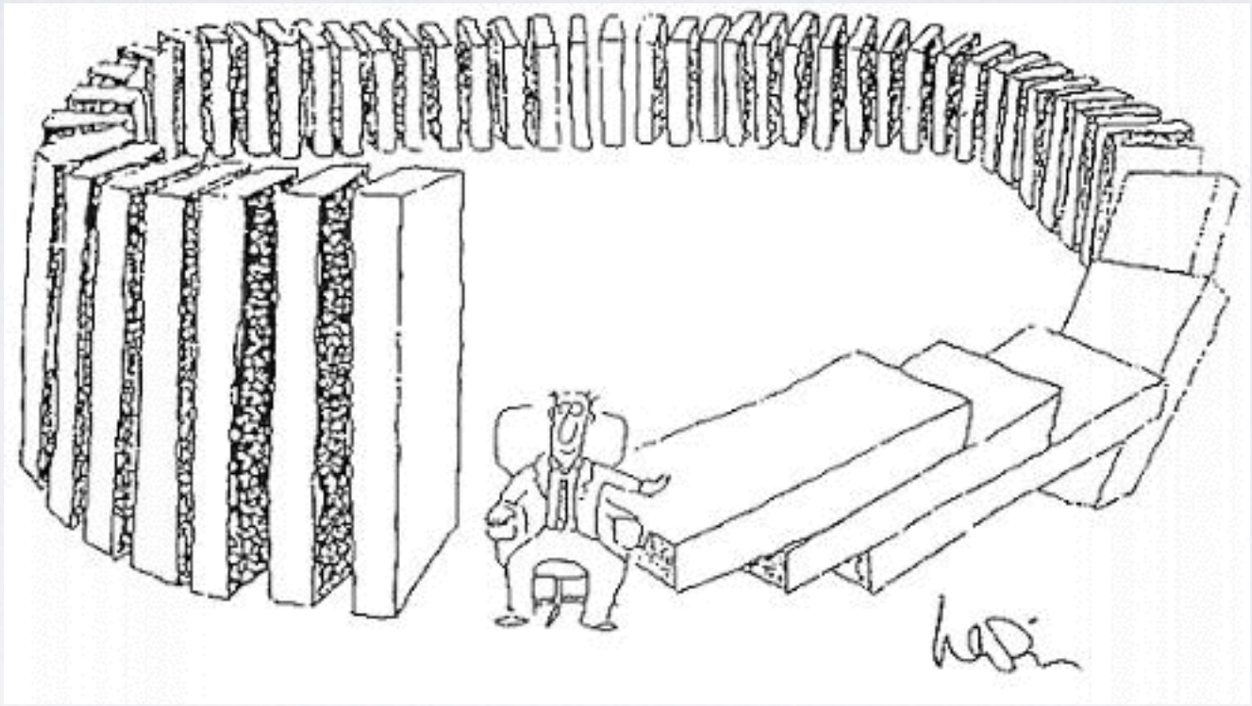
LUDICROUS COMPLEXITY



QA ANALYST TESTING A LUDICROUSLY COMPLEX SYSTEM


KEEP
CALM
AND
CODING

UNINTENDED CONSEQUENCES



Dev A: "Why did the payments page stop working?"

Dev B: "I don't know, I only fixed a bug in the login page..."

TYPICAL DEPLOYMENT


KEEP
CALM
AND
REFACTOR
THIS MESS

QUALITY IS ALSO THE JOB OF THE DEVELOPER

WHAT KIND OF TESTS?

Java EE Backend

Jersey Resource

- Constructor
- Method
- Method
- Method

Business Service

- Method
- Method
- Method
- Method

DAOs, Serialization, ...

AngularJS Frontend

Controller

- Constructor
- Method
- Method
- Method

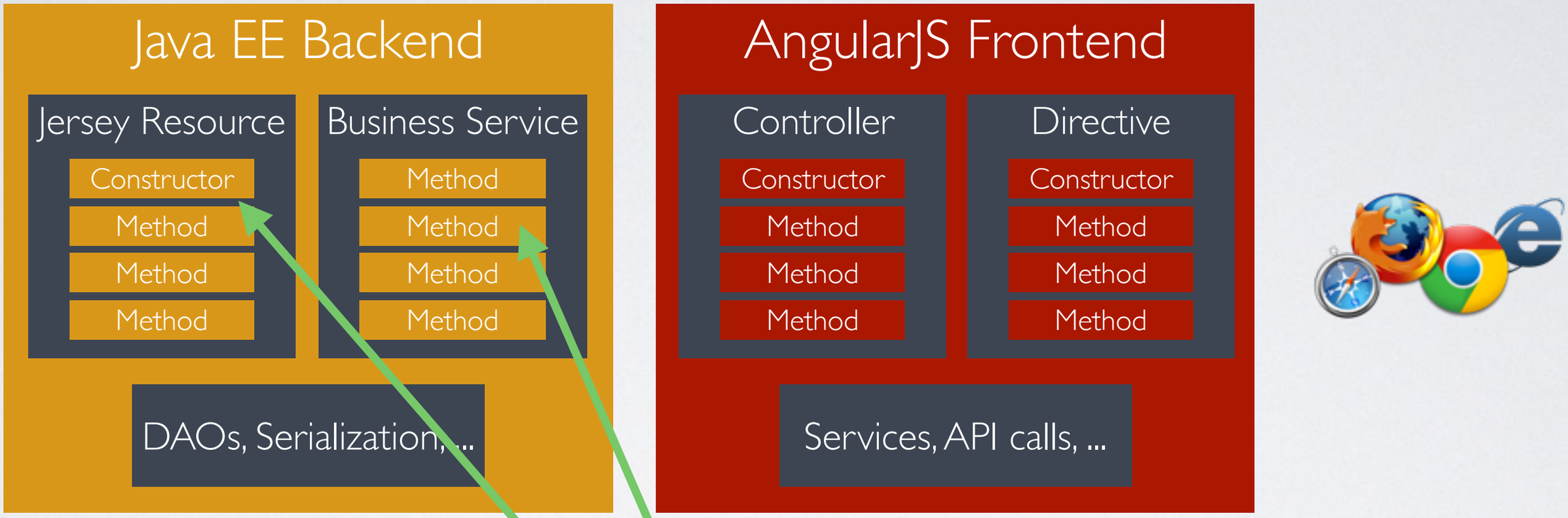
Directive

- Constructor
- Method
- Method
- Method

Services, API calls, ...

MySQL

WHAT KIND OF TESTS?

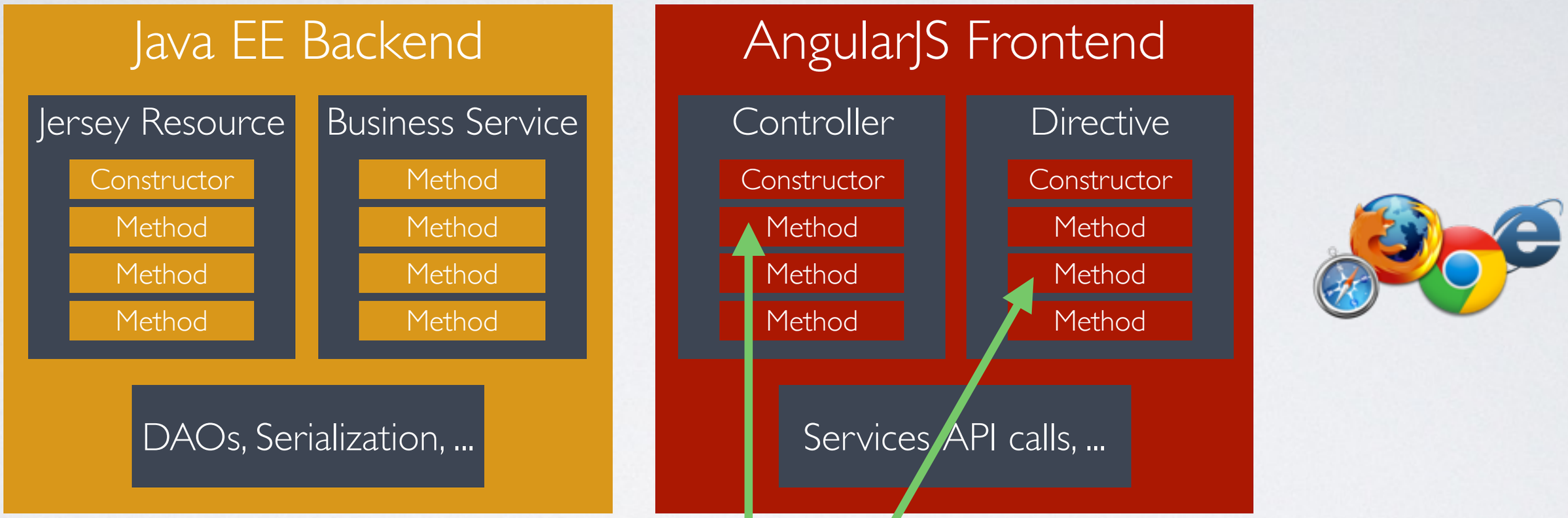


MySQL

JUnit

Unit tests are meant to test the smallest possible units of code. In our case, we used **JUnit** to test the individual methods of the business classes in our Java EE backend.

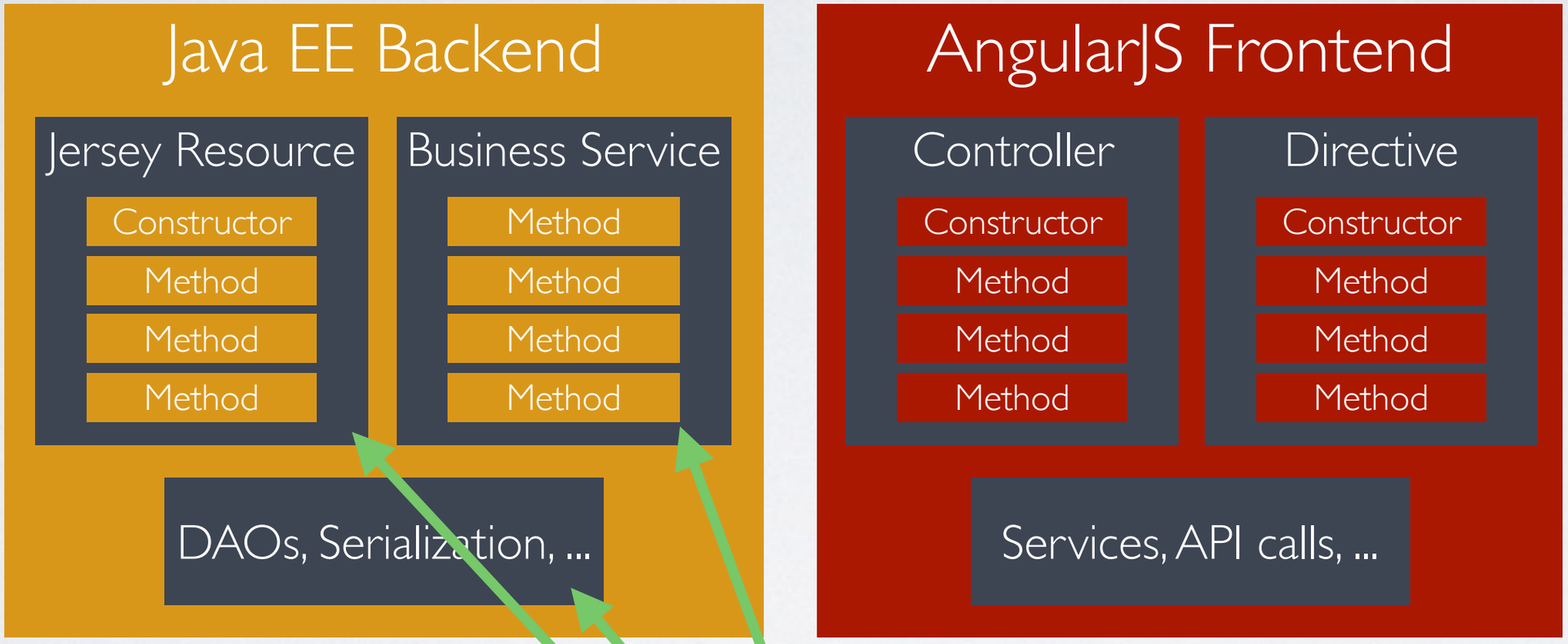
WHAT KIND OF TESTS?



MySQL

Jasmine is a Javascript testing framework. We used it to write the **unit tests** for the components of our AngularJS frontend.

WHAT KIND OF TESTS?



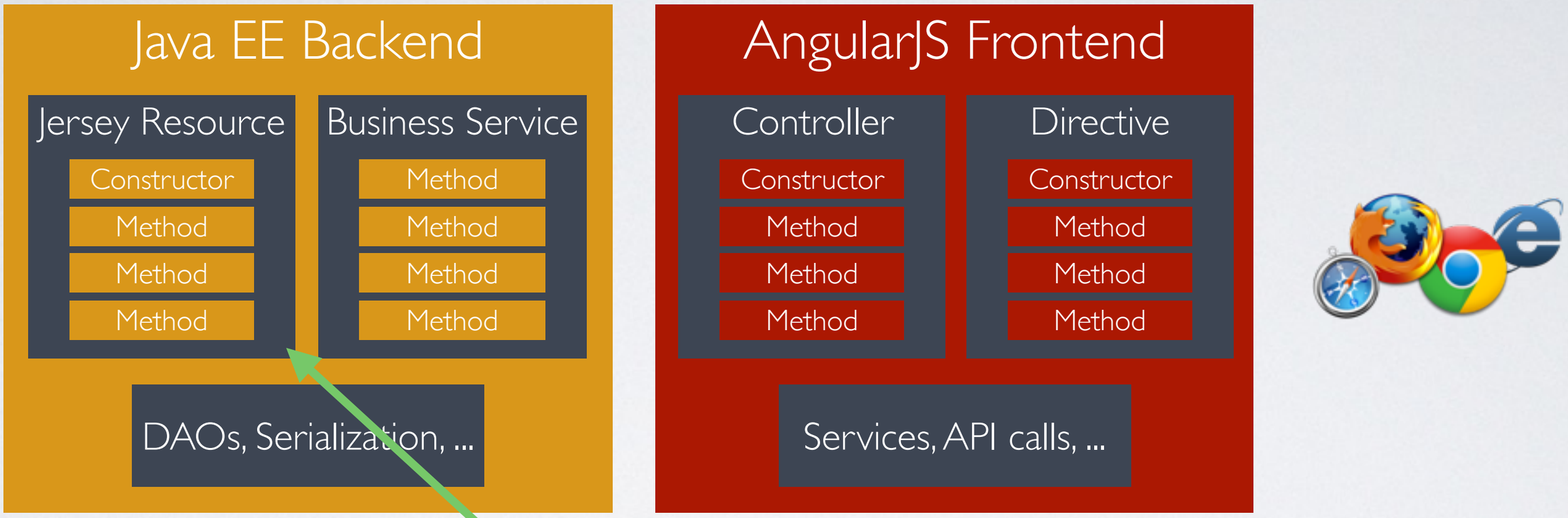
MySQL

Lotaris ITF

(Integration Test Framework)

Integration tests are used to test the components of your application when they are integrated together. We wrote our own framework to help with managing data and transactions in Java EE tests.

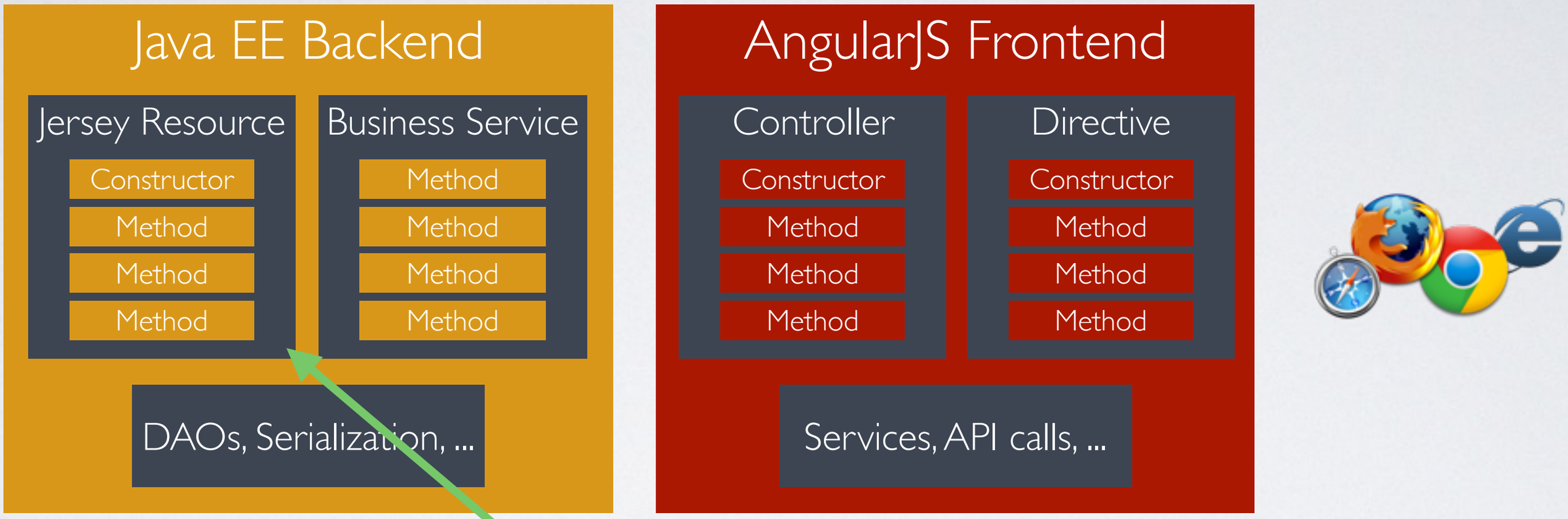
WHAT KIND OF TESTS?



MySQL

SoapUI is a functional testing framework for API tests. We started to use it to test our JSON APIs, but after 100 tests it became very slow, leaked memory, and was very hard to maintain for developers.

WHAT KIND OF TESTS?



MySQL

Lotaris
Java API Test

We wrote our own lightweight testing library, using the **Apache HTTP client** to call the API, and the **json-path** library to make assertions on the JSON responses.

WHAT KIND OF TESTS?

Java EE Backend

Jersey Resource
- Constructor
- Method
- Method
- Method

Business Service
- Method
- Method
- Method
- Method

DAOs, Serialization, ...

AngularJS Frontend

Controller
- Constructor
- Method
- Method
- Method

Directive
- Constructor
- Method
- Method
- Method

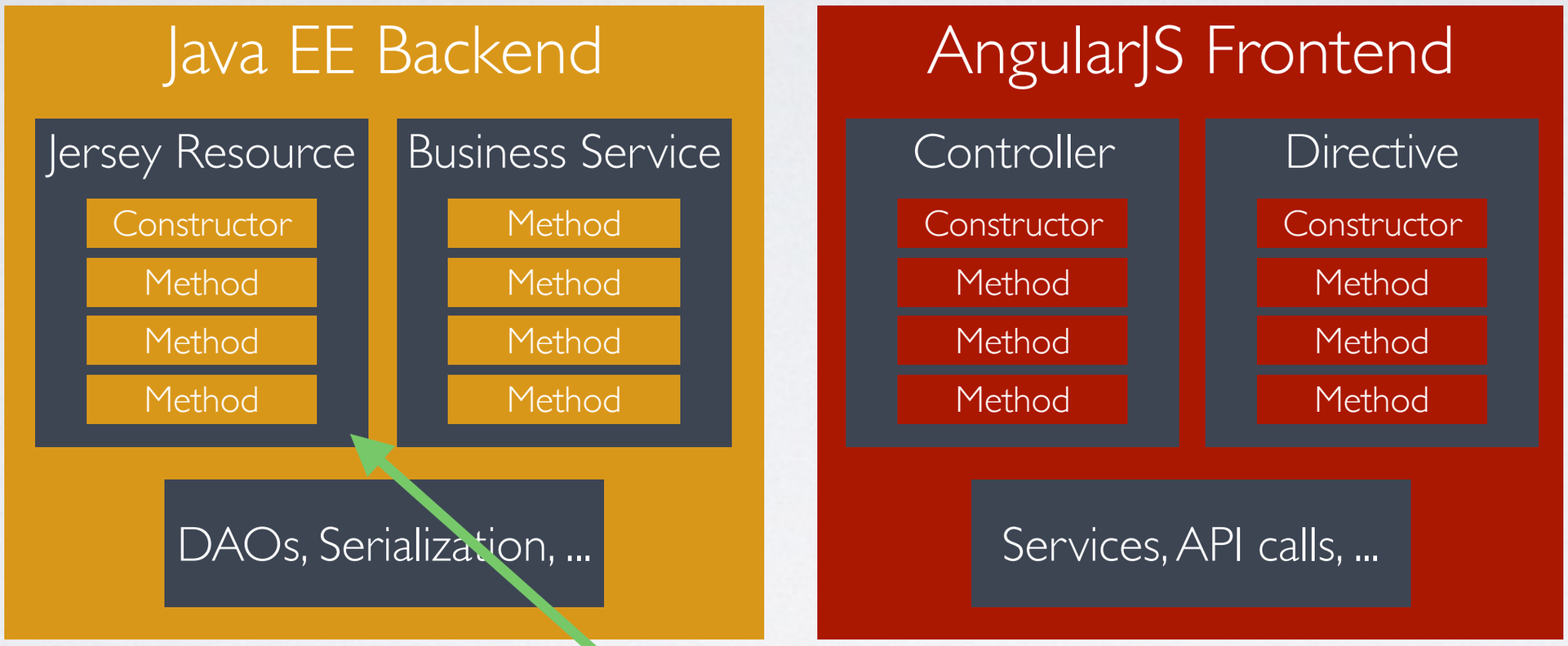
Services, API calls, ...

MySQL

+ JUnit

Selenium is a browser automation tool. We used it, combined with **JUnit**, to write tests simulating real users using our web frontend.

WHAT KIND OF TESTS?



MySQL

JMeter is a load testing tool. We used it to write tests simulating thousands of users using our API and frontend simultaneously.

MOCKING

Simulating complex systems in unit tests

MOCK OBJECTS

Mock objects are simulated objects that ***mimic the behavior of real objects*** *in controlled ways.*

WHAT TO MOCK?

If an actual object has any of the following characteristics, it may be useful to use a mock object in its place:

- The object supplies non-deterministic results (e.g., the current time or the current temperature).
- It has states that are difficult to create or reproduce (e.g., a network error).
- It is slow (e.g., a complete database, which would have to be initialized before the test).
- It does not yet exist or may change behavior.
- It would have to include information and methods exclusively for testing purposes (and not for its actual task).

Tasty mocking framework for unit tests in Java.

MOCKITO

MOCK OBJECTS

CREATE A MOCK OBJECT

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);

mockedList.add("one");
mockedList.clear();
mockedList.add("two");

mockedList.get(0);      // => null
mockedList.size();      // => 0
mockedList.isEmpty();   // => false
```

Create a mock object that looks like a List.

You can use the mock object. Note that **nothing happens** when you call these methods. Since it's a mock object, it doesn't do anything.

Methods of Mockito mock objects always return the **default value** for a Java type. False for booleans, 0 for integers, null for objects. etc.

MOCKITO

WHEN => THEN

MAKE IT DO SOMETHING

You define what happens when a mock's methods are called.

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);

when(mockedList.get(0)).thenReturn("yeehaw");
when(mockedList.size()).thenReturn(1);
when(mockedList.isEmpty()).thenReturn(false);
```

This mock now looks like a list with one element.

```
mockedList.get(0);      // => "yeehaw"
mockedList.size();      // => 1
mockedList.isEmpty();   // => false
```

But it's **still not a real List!**

```
mockedList.clear();
mockedList.size();   // => 1
```

WHY USE WHEN?

Sometimes you have non-deterministic component. When you write tests using these components, it can be difficult to predict the exact values that will be used.

```
public interface TimeService {
  Date getCurrentDate();
}
```

```
import static org.mockito.Mockito.*;

TimeService mockedTimeService = mock(TimeService.class);

Date fixedDate = new Date(-446774400000L); // 5 nov 1955

when(mockedTimeService.getCurrentDate()).thenReturn(fixedDate);
```

Define what it's going to be for the purposes of your test.

MOCKING EXCEPTIONS

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);
```

You can simulate an error.

```
when(mockedList.get(3)).thenThrow(new IndexOutOfBoundsException());
```

Note that the syntax is different for **methods that return void**, like List#clear.

```
doThrow(new RuntimeException()).when(mockedList).clear();

mockedList.get(3);   // EXCEPTION THROWN!
mockedList.clear();  // EXCEPTION THROWN!
```

MOCKITO MATCHERS

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);
```

Mockito provides useful **argument matchers** to make more flexible mock objects.

```
when(mockedList.get(anyInt())).thenReturn("yeehaw");

mockedList.get(0);   // => "yeehaw"
mockedList.get(1);   // => "yeehaw"
mockedList.get(42);  // => "yeehaw"
```

Now any integer will match our mocked method.

MOCKITO

VERIFY

VERIFY WHAT HAPPENS

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);

mockedList.add("one");
mockedList.clear();

verify(mockedList).add("one");  // OK
verify(mockedList).clear();     // OK
verify(mockedList).add("two");  // EXCEPTION THROWN!
```

Make your calls.

Verify what was called.

This will throw an exception because add was never called with "two".

VERIFY NUMBER OF INVOCATIONS

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);

mockedList.add("twice");
mockedList.add("twice");

verify(mockedList, times(2)).add("twice");  // OK
```

Check that a method was called exactly 2 times.

```
verify(mockedList, never()).clear();  // OK
```

Check that a method was never called.

MOCKITO

INJECTIONS

DEPENDENCY INJECTION

Mockito can perform dependency injection of mock objects for you!

Take this class for example.

```
public class BankingService {

  @EJB private CustomerAccountManager accountManager;
  @EJB private TimeService timeService;

  // ...
}
```

It has two external services that you need to mock and inject.

MOCKITO @INJECTMOCKS

```
public class BankingServiceUnitTest {

  @Mock private TimeService timeService;
  @Mock private CustomerAccountManager accountManager;

  @InjectMocks private BankingService bankingService;

  @Before
  public void setUp() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void testBankingService() {

    when(timeService.getCurrentDate()).thenReturn(someDate);
    when(accountManager.findOperationsSince(...)).thenReturn(someOperations);

    assertEquals(25.0, bankingService.getLastMonthBalance(someCustomer);
```

Simple annotate fields in your test class with **@Mock** and Mockito will create mock objects for you.

Annotate the class you want to test with **@InjectMocks**. Mockito will inject the mocks into it.

Don't forget to tell Mockito to set up this test class.

Then just use the mock objects.

MOCKITO

TEST EXAMPLES

We will write tests for a **BankingService** class. It has a method that computes the balance of a customer's banking operations over the last month.

```java
public class BankingService {
  @EJB private CustomerAccountManager accountManager;
  @EJB private TimeService timeService;

  public double getLastMonthBalance(Customer customer) {

    // get last month date
    Calendar cal = Calendar.getInstance();
    cal.setTime(timeService.getCurrentDate());
    cal.add(Calendar.MONTH, -1);
    Date lastMonth = cal.getTime();

    // fetch operations during the last month
    List<Operation> operations;

    try {
      operations = accountManager.findOperationsSince(customer, lastMonth);
    } catch (DatabaseException de) {
      throw new CustomerAccountException("Could not compute balance", de);
    }

    double balance = 0;

    // compute the balance from the operations
    for (Operation operation : operations) {
      if ("credit".equals(operation.getType()) {
        balance = balance + operation.getAmount();
      } else if ("debit".equals(operation.getType()) {
        balance = balance - operation.getAmount();
      }
    }

    return balance;
  }
}
```

This method returns the balance of a customer's banking account for the last month.

It uses a **TimeService** to get the current date and determine when last month was.

It uses a **CustomerAccountManager** to fetch banking operations from the database.

If a database exception is thrown, it is wrapped into a **CustomerAccountException**.

It iterates over the list and computes the balance from credit/debit operations.

TEST SETUP

Set up mocks.

Inject them.

Write tests.

```
public class BankingServiceUnitTest {

  @Mock
  private TimeService timeService;
  @Mock
  private CustomerAccountManager accountManager;
  @InjectMocks
  private BankingService bankingService;

  @Before
  public void setUp() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void testSomething() {
    // ...
  }

  @Test
  public void testSomethingElse() {
    // ...
  }
}
```

Note that the mocks are re-created and re-injected **before each test**, so you have a clean slate to start with.

FIRST TEST

```
@Test
public void testPositiveBalance() {

  when(timeService.getCurrentDate()).thenReturn(getDate("2000-01-01"));

  Customer customer = new Customer("Peter Gibbons");

  List<Operation> operations = new ArrayList<>();
  operations.add(new Operation("credit", 20));
  operations.add(new Operation("debit", 30));
  operations.add(new Operation("credit", 5));
  operations.add(new Operation("credit", 50));

  when(accountManager.findOperationsSince(any(Customer.class), any(Date.class)))
    .thenReturn(operations);


  double balance = bankingService.getLastMonthBalance(customer);

  verify(accountManager).findOperationsSince(customer, getDate("1999-12-01"));

  assertEquals(45.0, balance, 0);
}
```

Mock the current date.

Create a sample customer.

Define a list of last month's credit and debit operations.

Mock the account manager to return that list.

Compute the balance.

Check that the account manager was called with the correct arguments (the customer and last month's date).

Check that the computed balance is correct.

TESTS ARE CODE TOO

```
// ...

List<Operation> operations = new ArrayList<>();
operations.add(new Operation("credit", 20));
operations.add(new Operation("debit", 30));
operations.add(new Operation("credit", 5));
operations.add(new Operation("credit", 50));

when(accountManager.findOperationsSince(any(Customer.class), any(Date.class)))
  .thenReturn(operations);

// ...

assertEquals(45.0, balance, 0);
```

Note that we do not repeat the calculation in the test. We write the assertion **with the expected end result**. This reduces the likelihood of introducing a **bug in the test**.

TEST: EDGE CASE

When you test a piece of code, don't stop after writing a passing test for the most common use case. Try to test all the edge cases: minima, maxima, empty result sets, etc. This is where bugs are most likely to occur.

Here, we test what happens when no operation was performed during the last month.

```
@Test
public void testZeroBalanceWithNoOperations() {

  when(timeService.getCurrentDate()).thenReturn(getDate("2010-05-05"));

  Customer customer = new Customer("Peter Gibbons");

  List<Operation> noOperations = new ArrayList<>();
  when(accountManager.findOperationsSince(any(Customer.class), any(Date.class)))
    .thenReturn(noOperations);

  double balance = bankingService.getLastMonthBalance(customer);

  verify(accountManager).findOperationsSince(customer, getDate("2010-04-05"));

  assertEquals(0.0, balance, 0);
}
```

Mock the account manager to return an empty list of operations.

Check that the resulting balance is 0.

TEST ERRORS

Do not forget to test that your code behave as expected when errors occur:

```
@Test
public void testDatabaseError() {

  when(timeService.getCurrentDate()).thenReturn(getDate("2000-01-01"));

  Customer customer = new Customer("Peter Gibbons");

  DatabaseException databaseException = new DatabaseException("bug");
  when(accountManager.findOperationsSince(any(Customer.class), any(Date.class)))
    .thenThrow(databaseException);

  try {
    bankingService.getLastMonthBalance(customer);
    fail("Expected CustomerAccountException to be thrown");
  } catch (CustomerAccountException e) {
    assertEquals("Could not compute balance", e.getMessage());
    assertEquals(databaseException, e.getCause());
  }



  verify(accountManager).findOperationsSince(customer, getDate("1999-12-01"));
}
```

Make the account manager throw an exception.

Check that the error is handled as expected. In this case, we expect the DatabaseException to be wrapped in a CustomerAccountException.

REFERENCES

- JUnit: http://junit.org
- Mockito: http://site.mockito.org
- Selenium: http://www.seleniumhq.org
- Jasmine: http://jasmine.github.io
- SoapUI: http://www.soapui.org
- JMeter: http://jmeter.apache.org
- Maven project with all the examples in this presentation: https://github.com/SoftEng-HEIGVD/Teaching-AutomatedTestsMocking