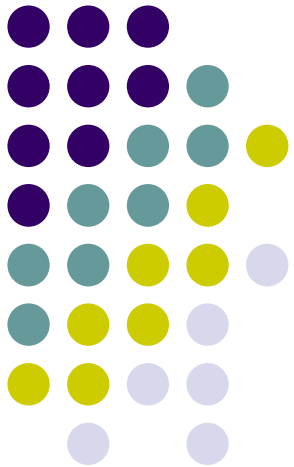
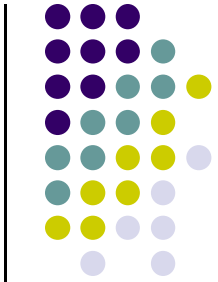
Unit testing

Code @ https://github.com/koen-serneels/unit-testing.git



Koen Serneels

Why bother writing tests?

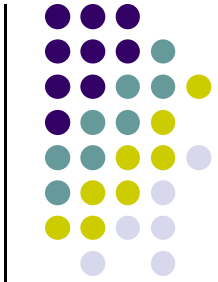


- Simple example: this snippet determines if a given number is odd or even

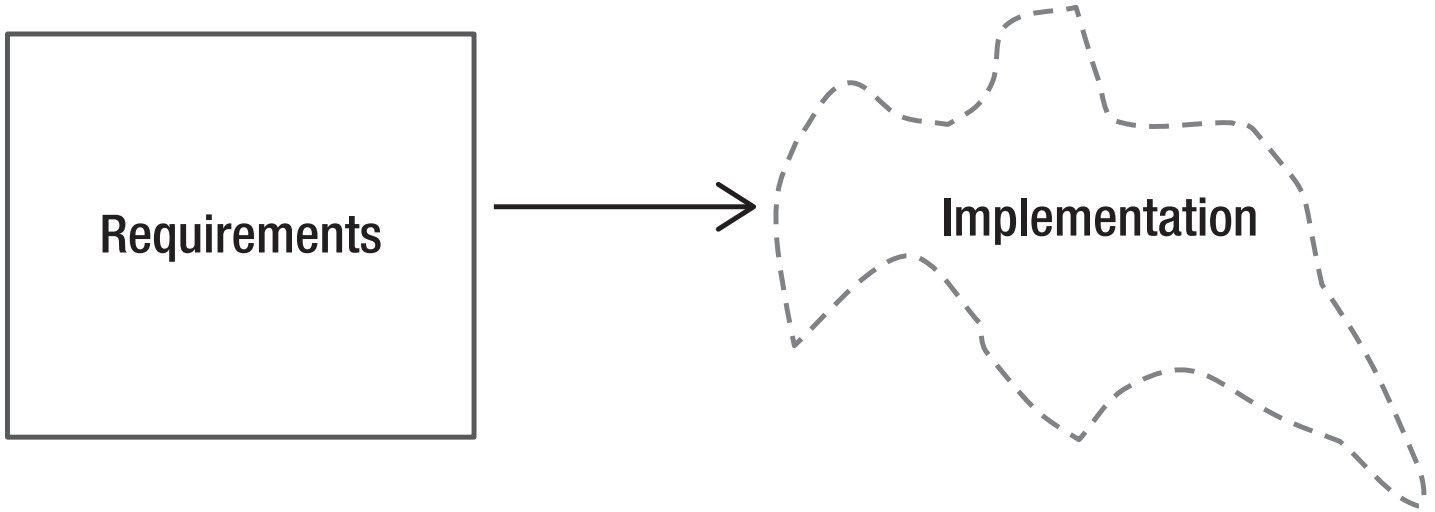
```
public boolean isOdd(int i) {
   return i % 2 == 1;
}
```

- It is one line of code, how hard can it be? Will it work? Do I even need to test this?

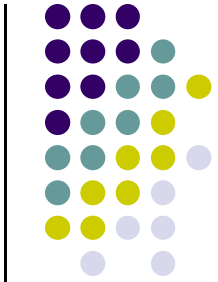
Why bother writing tests?



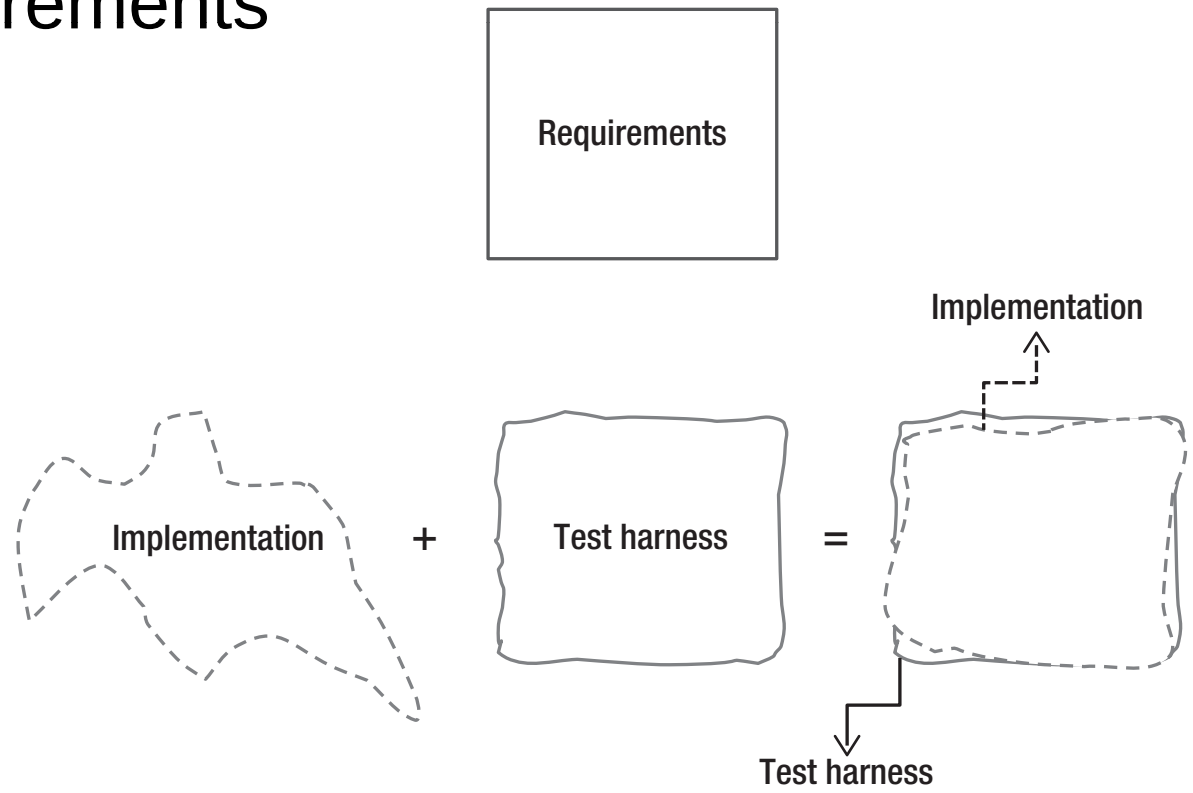
- Without tests there is no indication that your code adheres to the requirements. Deformation occurs instantly



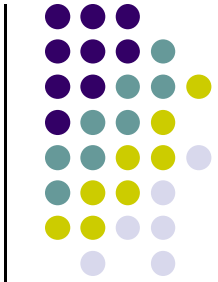
Why bother writing tests?



- By creating tests you build a harness that guards the compliance of your code in regard to its requirements

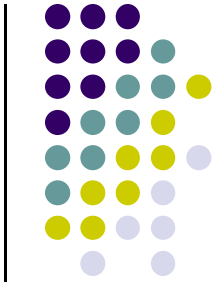


Why bother writing tests?



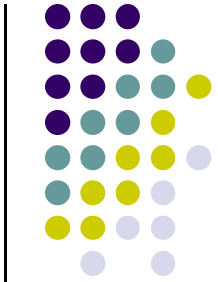
- Your test set creates freedom
  - Enables you to refactor code without being scared
    - No more: “don’t touch that code! It needs to be in the next release, and we really don’t want to break it!”
- Your test set helps others (and yourself) verify that their changes or additions didn’t break anything
  - In a long running project, you become “the other” pretty fast as you forget about the code you have written
- Your test set protects you from regression

Why bother writing tests?



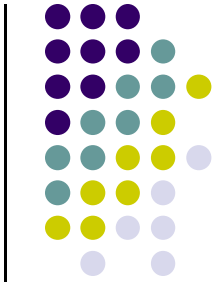
- There is never a good reason not a write a test
  - No, the reason you are thinking about right now is not a reason, it’s an excuse
- Threat your tests as production code
  - Tests require design too
- Make your tests live in a solid environment
  - Build or re-use infrastructure to make your life easy

Why bother writing tests?



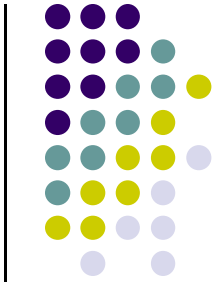
- Sometimes testing is hard. Really hard. It can be harder then writing the actual production code
  - Again, this is no reason not to do it, it’s an excuse. Find a solution for it, write the test and cary on

Different types of testing



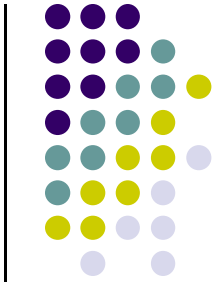
- There are different types of testing
- To name a few
  - Integration testing
  - System testing
  - User acceptance testing
  - Performance testing
  - Stress testing

Different types of testing



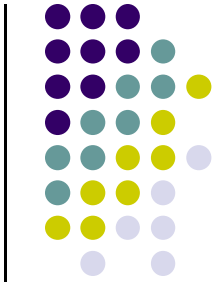
- We will limit ourselves to automatic testing of our code base with unit testing
- First ,we have our basic “unit” test
  - Tests the individual methods of a class
  - Can test different layers
  - Can use different strategies to overcome resource dependencies

Automated unit testing



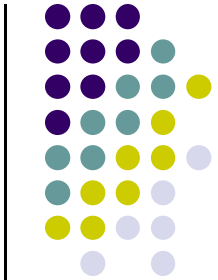
- Review what we are going to look at
  - Basic unit test
    - Using TestNG as the testing framework
    - Using Mockito as mocking framework
    - Using Spring (MVC) test support
  - Unit resource test
    - Using all of the above
    - Adding in memory resources into the picture
  - Unit frontend test (aka Selenium test)
    - Using all of the above
    - Adding selenium for driving our browser

A good unit test (on technical level)



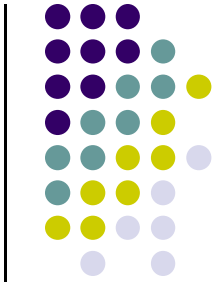
- Runs fast
- Has no **external** configuration
  - Should be self containing
- Runs independed of other tests
  - Tests do never share state!
- Has no external resource dependencies
- Runs in a pre defined context
  - Think about moving targets such as time or dates
- Does not leave traces

Resource dependencies



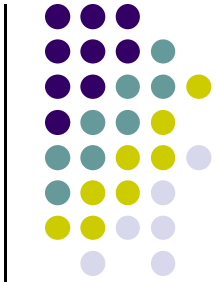
- Resource dependencies start to make testing more difficult
  - DAO’s executing SQL/HQL/JPQL queries
  - Business logic depending on a webservice response
  - Code publising results to a Q
  - Application controller requiring http session access
  - Behavioral logic inside views
  - ...

Resource dependencies



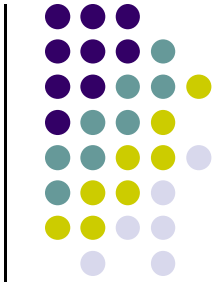
- In the context of our automated testing we will divide tests requiring such resources under the category ‘unit resource tests’
  - Tests that cannot be executed without having a resource dependency

Unit resource test



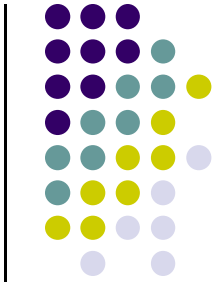
- Before a unit test needs to be promoted to a unit resource test you have some alternatives
  - Dummies
  - Stubs
  - Mocks
- Outweigh the different options
  - Don’t forget, this is not black and white and there ain’t no silver bullet

Unit frontend test



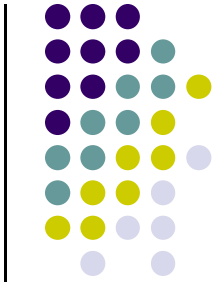
- Finally we have a specialization of a unit resource test; the frontend test
- It is a resource test, but with special requirements

Running tests



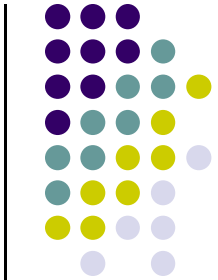
- All of these test will run
  - In our IDE (Eclipse in my case)
  - Via our CLI using Maven
  - Since its a Maven project and we don’t have any external dependencies: can be plugged in hudson/jenkins in about “5 seconds”

The demo app



- I will proceed by explaining how we address the different test types using the “testing” demo application
- Code can be found on GitHub https://github.com/koen-serneels/unit-testing.git
- Suggestions, ideas, bugs are always welcome via email

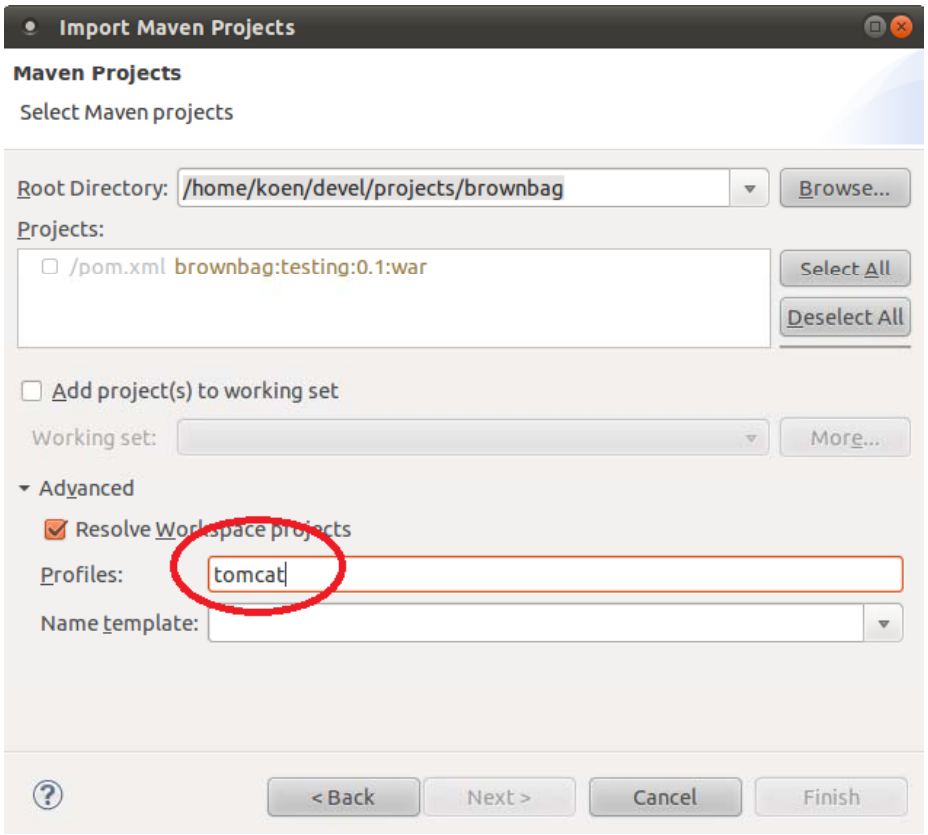
Using the demo app



- **First** check README on Github
- Clone/download project
  - “mvn clean install” will run all tests, including selenium tests
    - Uses firefox as default
  - “mvn clean install -DheadlessIT”
    - Will run selenium tests in VFB
      - sudo apt-get install xvfb
  - “mvn clean install -DskipITs”
    - Skips integration tests

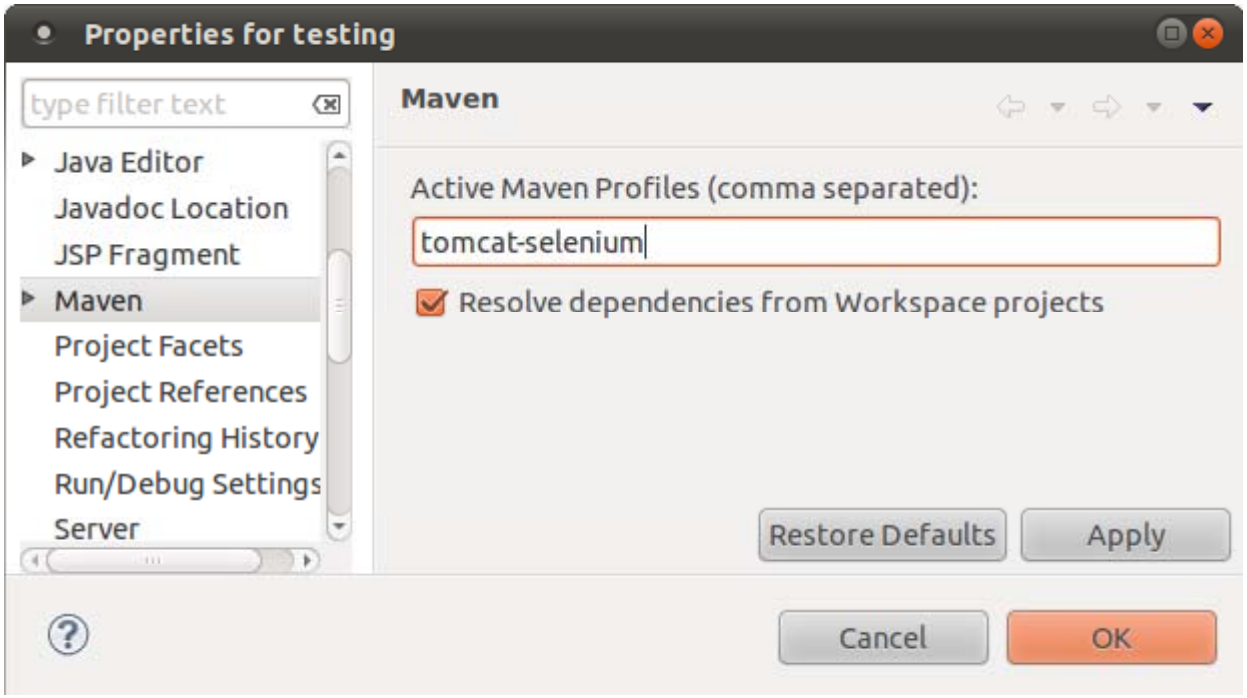
Using the demo app

- Import in Eclipse as maven project
  - Make sure to select the “tomcat” as profile

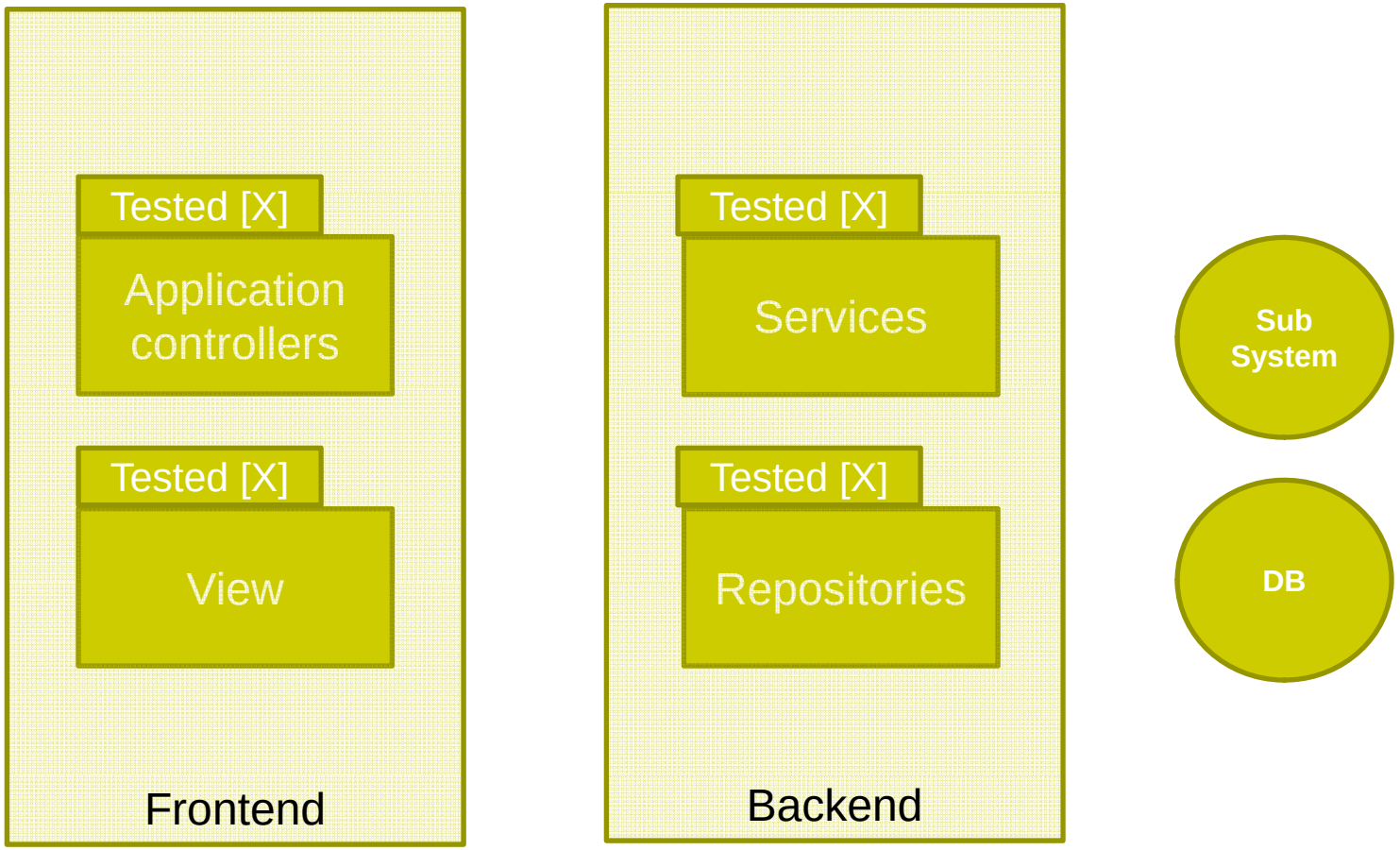


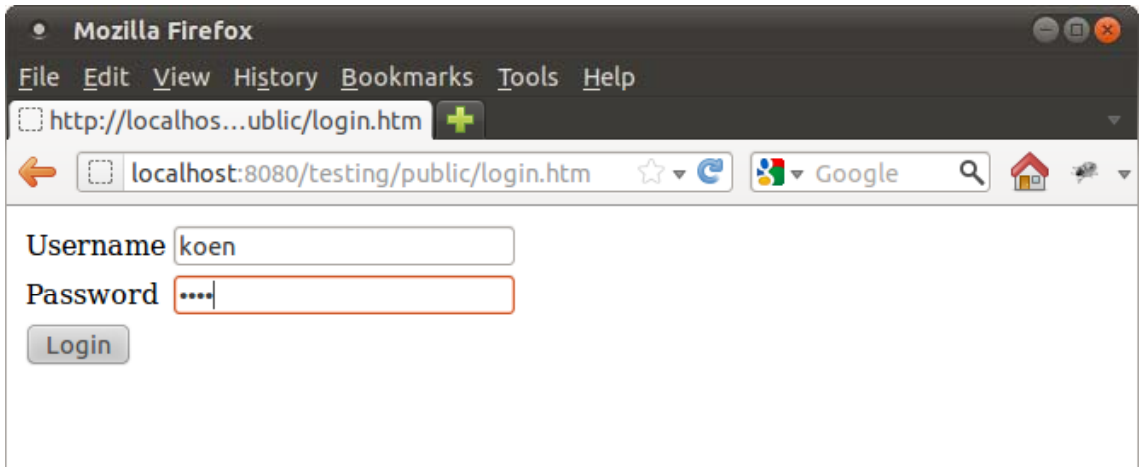
Using the demo app

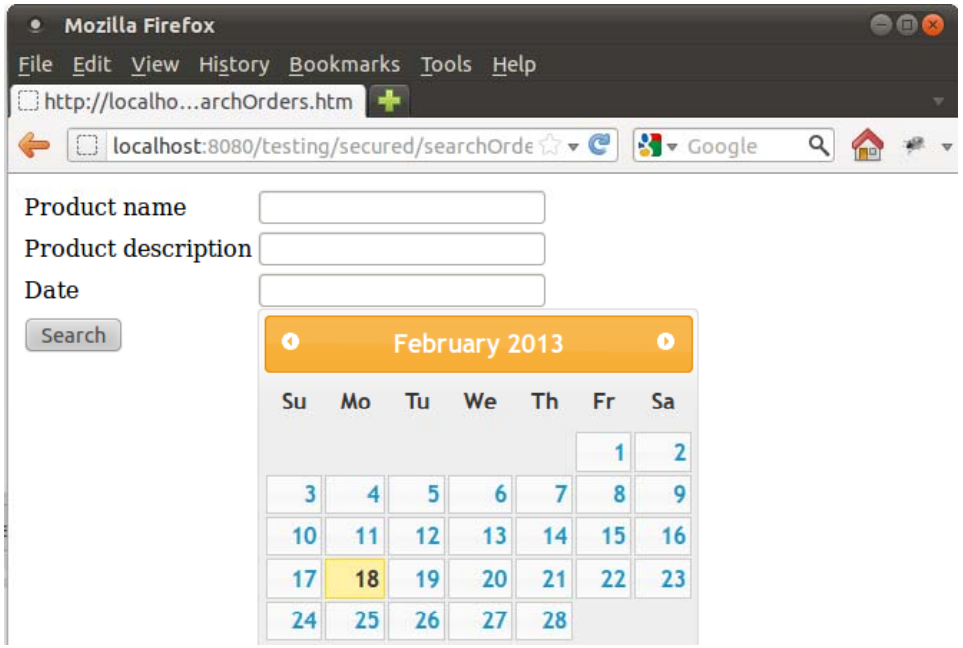
- If you want to run a selenium test against tomcat in eclipse, select the “tomcat-selenium” profile
  - Once the project is imported, you can switch profiles by right clicking on the project -> properties -> Maven

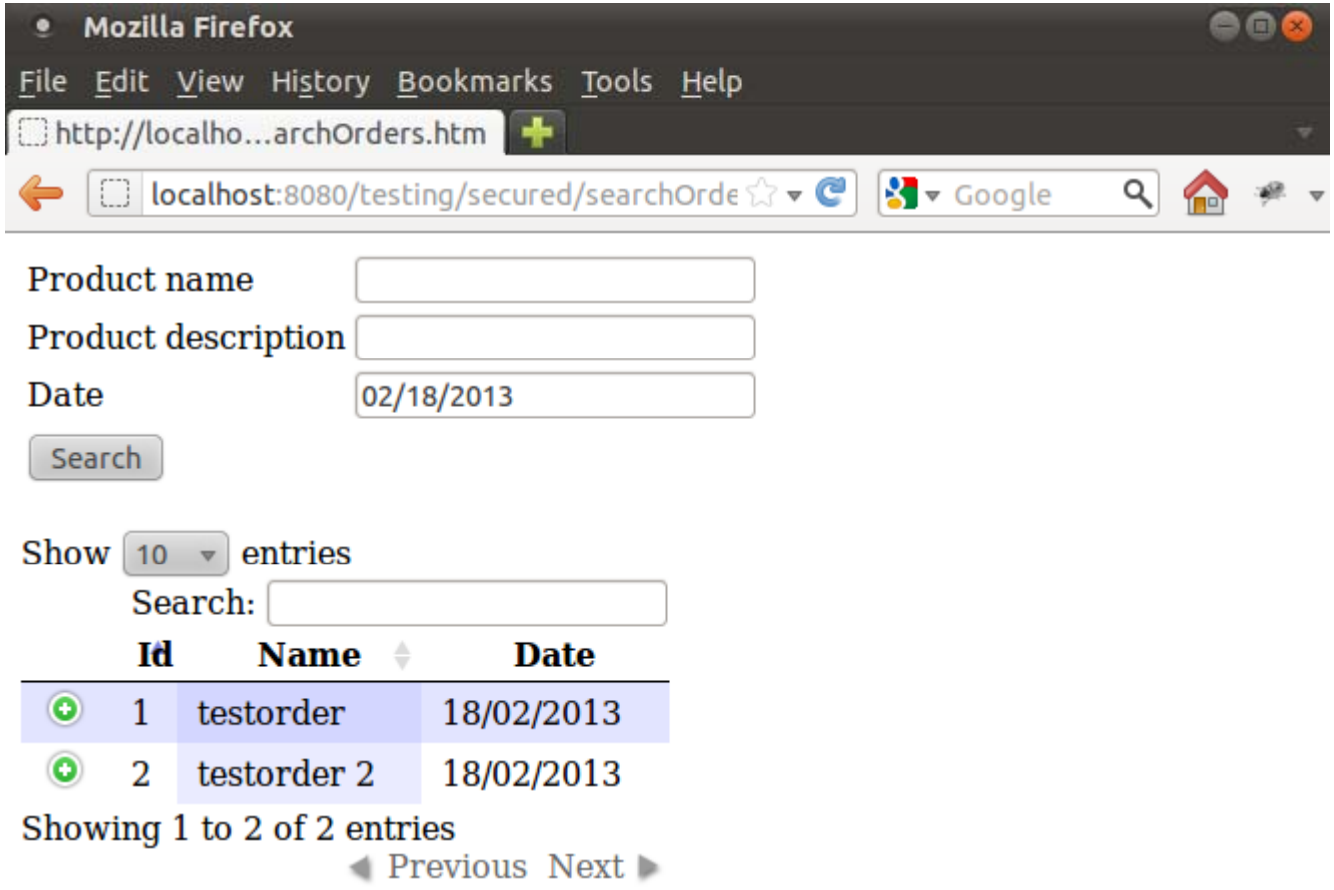


The demo app

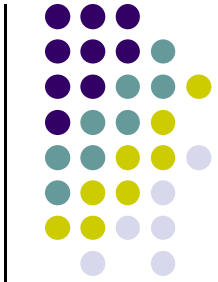

Tested [X]
Application controllers
Tested [X]
View
Frontend
Tested [X]
Services
Tested [X]
Repositories
Backend
Sub System
DB


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhos...ublic/login.htm
localhost:8080/testing/public/login.htm
Google
Username koen
Password
Login


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localho...archOrders.htm
localhost:8080/testing/secured/searchOrde
Google
Product name
Product description
Date
Search
February 2013
Su Mo Tu We Th Fr Sa
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28

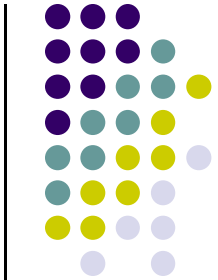

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localho...archOrders.htm
localhost:8080/testing/secured/searchOrde
Google
Product name
Product description
Date
02/18/2013
Search
Show 10 entries
Search:
Id
Name
Date
1 testorder 18/02/2013
2 testorder 2 18/02/2013
Showing 1 to 2 of 2 entries
Previous Next

The demo app



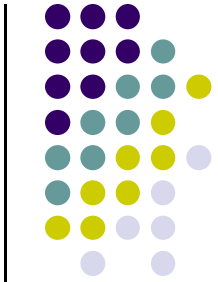
- Repositories for accessing our resources
- Services accessing our repositories, driving BL and acting as transactional boundary
- Application controllers talking to our services
- A view to be able to interact with our app
- Everything glued together with JPA, Hibernate, JMX, Spring, Spring MVC, JQuery and JQuery datatables

Unit test



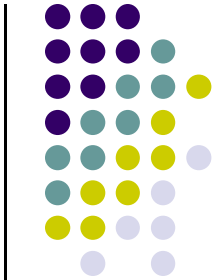
- Using TestNG (@Test)
  - Enables the test to be identified as a test which can be run by the Eclipse TestNG plugin or surefire (Maven)
  - Has some attributes such as the group to which the test belongs, timeout, dependencies, ...
  - Can be defined on the class or on individual methods: there is no need to define this annotation on the individual methods if no custom configuration is required. Each public method becomes automatically part of the test

Unit test



- Lifecycle annotations such as
  - Before/AfterSuite
  - Before/AfterGroups
  - Before/AfterTest
  - Before/AfterMethod

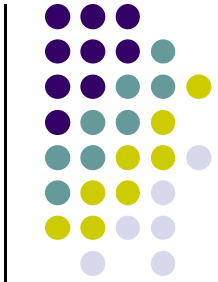
Unit test



- Suite xml file
  - Let's surefire identify which tests to run in group. Can also be picked up by the Eclipse plugin

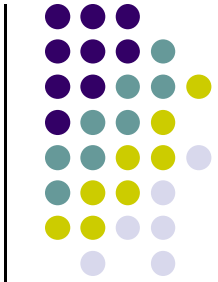
```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="testng-unit">
        <test verbose="0" name="unit tests" annotations="JDK">
                <groups>
                        <run>
                                <exclude name="selenium" />
                        </run>
                </groups>
                <packages>
                        <package name="be.testing.*" />
                </packages>
        </test>
</suite>
```

Unit test



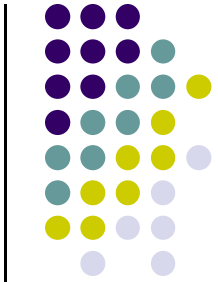
- Beware of dependsOnMethods and dependsOnGroups
  - Without them tests and test methods are ran in “random” order
  - Which is a good thing!
- We have about 2000 test clases, there is nowhere any dependency made between test classes or test methods

Spring test framework



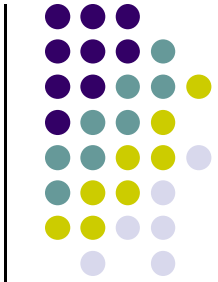
- Integrates with TestNG and Junit
  - Behavior can be extended by using @TestExecutionListeners
- Manages the application context
  - caches the application context
  - Uses the resources/annotations as key
  - You can mark an application context to be closed/refreshed using @DirtiesContext
    - This is nothing you typically want to do. If so, you might want to consider using StaticApplicationContext fo that given test (we'll cover that later)

Spring test framework



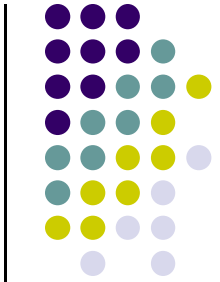
- Can manage transactions for our test cases
- Provides a wide range of stubs/mocks
  - StaticApplicationContext
  - MockMvc
    - MockServletContext
    - MockHttpServletRequest/Response
      - builders
    - MockHttpSession

Spring test framework



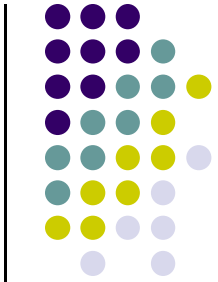
- @IfProfileValue
  - @IfProfileValue(name="java.vendor", value="Sun Microsystems Inc.")
  - @IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"})
- @ProfileValueSourceConfiguration
  - @ProfileValueSourceConfiguration(CustomProfileValueSource.class)
- @DirtiesContext

Spring test framework



- @ExpectedException
  - expectedExceptions on @Test
- @Timed
  - timeOut on @Test
- @Repeat
  - InvocationCount and invocationTimeout on @Test
- @Rollback
- @NotTransactional

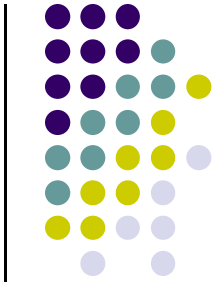
Spring test framework



- TestNG test example with the Spring test framework

```
@Test
@ContextConfiguration
public class SandboxTest extends AbstractTestNGSpringContextTests {

    @Autowired
    private String string;

    public void test() {
        Assert.assertEquals(string, "SGS3");
    }

    @Configuration
    static class SandboxTestConfiguration {
        @Bean
        public String imASimpleString() {
            return "SGS3";
        }
    }
}
```

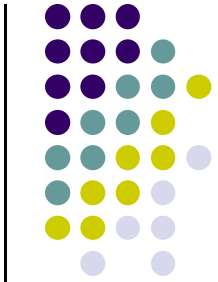
Profiles in our app



- Profiles
  - Production.java
  - Tomcat.java
  - TomcatSelenium.java
  - UnitResourceTest.java
  - UnitSeleniumTest.java

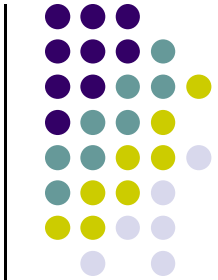
```
/**
 * @author Koen Serneels
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile(Production.name)
public @interface Production {

    public static String name = "production";
}
```

Unit test application controllers



- Our controller tests require some extra setup
- Annotations that contain some information that needs to be checked
- Parameter mapping
- Result checking
  - In our case JSON, but could also be a view

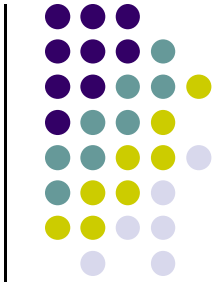
Unit test application controllers



- Spring based unit test with MockMvc to simulate the Spring MVC environment

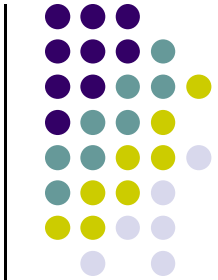
```
/**
 * @author Koen Serneels
 */
@Test
public class LoginControllerTest {

    public void testShowLogin() throws Exception {
        LoginController loginController = new LoginController();
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(loginController).build();
        mockMvc.perform(get("/public/login.htm")).andExpect(status()
                        .isOk()).andExpect(view().name("public/login"));
    }
}
```

Unit test application controllers



- MockMvc can be ran in two modes
  - StandAlone
  - Based on an initialized WebApplicationContext
- Creating a WebApplicationContext can be done in two ways
  - Using the mock/stub approach: StaticWebApplicationContext
  - Using @WebAppConfiguration in combination with @Configuration

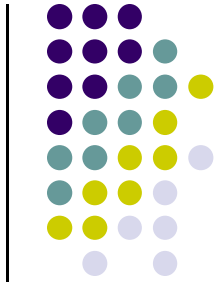
Unit test application controllers



- How to test this? (searchOrders)
  - Requires dependency
  - Requires parameters and conversion (date)
  - Returns JSON

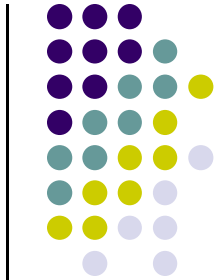
```
@Controller
public class OrderSearchController {

    @Autowired
    private OrderService orderService;

    @RequestMapping(value = "/secured/searchOrders.json",
                    method = RequestMethod.GET)
    public @ResponseBody
    Collection<Order> searchOrders(OrderSearchCriteria orderSearchCriteria) {
        return orderService.findOrders(orderSearchCriteria);
    }
}
```

Unit test application controllers



- Let’s check out the code...

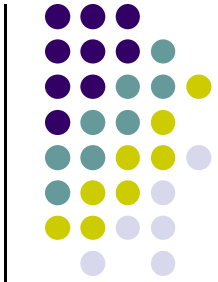
Unit test maven integration



- Maven integration
  - Surefire

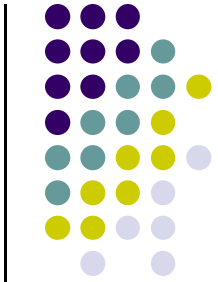
```
<plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.13</version>
        <configuration>
                <suiteXmlFiles>
                        <suiteXmlFile>src/test/resources/${testng.unit.file}</suiteXmlFile>
                </suiteXmlFiles>
        </configuration>
</plugin>
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
        <version>2.13</version>
        <configuration>
                <suiteXmlFiles>
                        <suiteXmlFile>src/test/resources/${testng.unit.file}</suiteXmlFile>
                </suiteXmlFiles>
        </configuration>
</plugin>
```

Unit resource test



- Launches the embedded H2 database
  - JPA properties profile configured to create our tables
- Drives the transaction for the test method
  - Starts before the test method begins, rolls back after the test method ends
    - Can be influenced by using @Rollback(false) for debugging purposes
- Injects our EntityManager and our OrderRepository to test

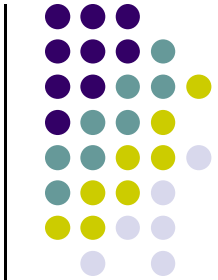
Unit resource test



- Re-uses the basic unit principles
- Uses extra infrastructure to be able to use resources
- Loading Spring together with our test

```
/**
 * @author Koen Serneels
 */
@Test
@ActiveProfiles(UnitResourceTest.name)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class OrderRepositoryTest extends AbstractTransactionalTestNGSpringContextTests {

    @Autowired
    private OrderRepository orderRepository;
```

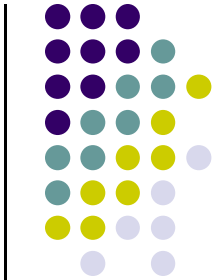
Unit resource test



- Database setup is conditional thanks to our Spring (not Maven) profiles. This returns actually a javax.sql.DataSource, the database is started by the driver

```
@Configuration
@Profile({ "!" + Production.name })
static class H2IsolationLevelInitializer {
    @Autowired
    private DataSource dataSource;

    @Bean
    public H2IsolationLevelInitializerBean h2IsolationLevelInitializerBean() {
        return new H2IsolationLevelInitializerBean(dataSource);
    }
}
```

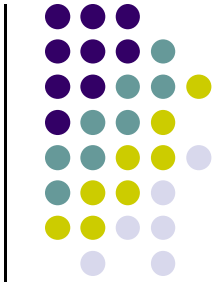
Unit resource test



- Our entity manager is constructed via the standard JPA persistence.xml
  - However, our “dynamic” configuration is pulled out of the file

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/p
    version="2.0">

    <persistence-unit name="testing" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
    </persistence-unit>
</persistence>
```

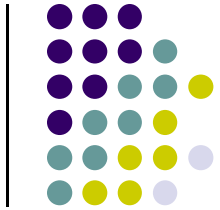
Unit resource test



```
@Configuration
public class JpaConfiguration {

    @Autowired
    @Qualifier("jpa.provider.properties")
    private Properties properties;

    @Autowired
    private DataSource dataSource;

@Bean
public FactoryBean<EntityManagerFactory> entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean localContainerEntityManagerFactoryBean = new LocalCon
    localContainerEntityManagerFactoryBean.setJpaProperties(properties);
    localContainerEntityManagerFactoryBean.setDataSource(dataSource);
    localContainerEntityManagerFactoryBean.setJpaVendorAdapter(jpaVendorAdapter());
    return localContainerEntityManagerFactoryBean;
}
@Configuration
@Profile({ Tomcat.name, UnitResourceTest.name, UnitSeleniumTest.name, TomcatSelenium.name })
static class JpaProviderH2Properties {
    @Bean(name = "jpa.provider.properties")
    public Properties properties() {
        Properties properties = new Properties();
        properties.setProperty("hibernate.dialect", H2Dialect.class.getName());
        properties.setProperty("hibernate.hbm2ddl.auto", "update");
        return properties;
    }
}
```

- Creating the EntityManager using specific properties based upon the active profile

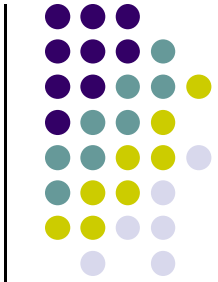
Data setup



- Builder pattern
- Anonymous initializer prevents from code being re-formatted
- Instances is saved after being “build”
- Very, very simple implementation

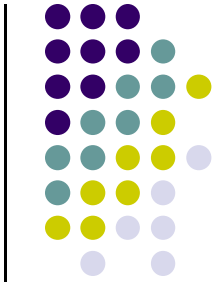
```
@BeforeMethod
public void dataSetup() {
    TestDataBuilderManager.init(entityManager);
    new OrderBuilder() {
        {
            name("testorder");
            date(today);

            addProduct(new ProductBuilder() {
                {
                    name("SGS3");
                    description("Better smartphone");
                    price(new BigDecimal("500.99"));

                }
            }.build());

            addProduct(new ProductBuilder() {
                {
                    name("SGS2");
                    description("Good smartphone");
                    price(new BigDecimal("400.99"));

                }
            }.build());
        }
    }.build();
}
```

Unit resource test



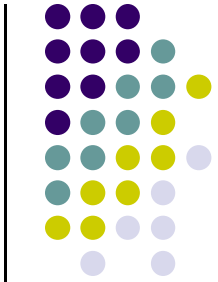
- Thanks to profiles we run against our production configuration with altered resource configuration
  - All machinery involved is working as would be in production
    - We are using our real production repo’s and services
- Using simple data builders to setup our data
  - Depending on the entity model, no XML files with SQL
  - Clean and readable
  - Can be re-used, even in production code

Unit resource test



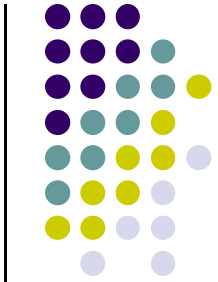
- Our repository is fully tested including all queries it produces
  - Tables created by JPA provider
  - General data setup was provided for the test class, additions made in the test method, but the state is method scoped
  - Nothing was commited to database (remains empty)
- The only configuration for the test itself are a couple of annotations

Unit frontend test selenium



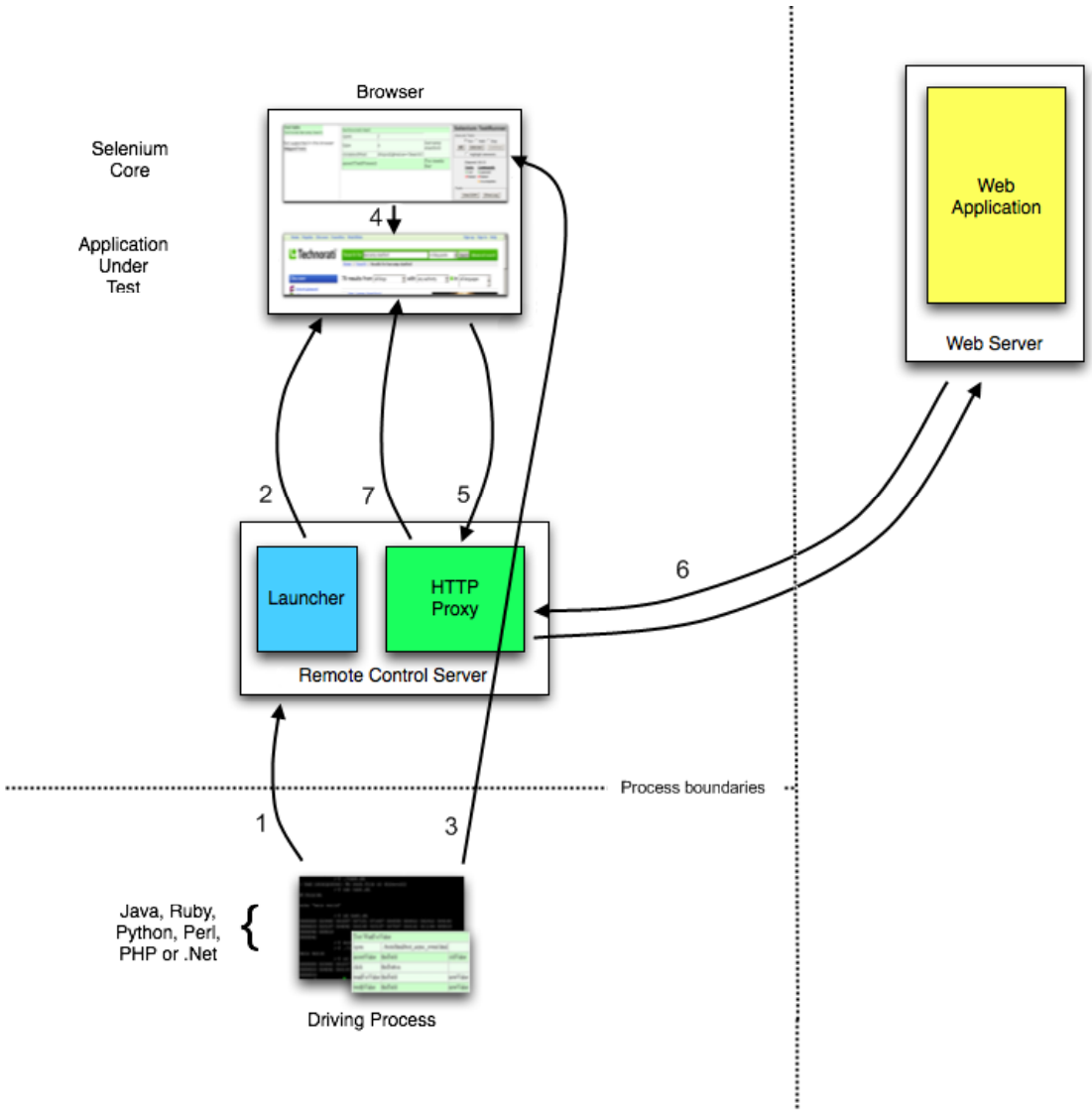
- We use the same infrastructure as for the other resource tests
- The unit frontend tests are build around Selenium
- Selenium offers a client API to interact with the browser
  - So called “WebDriver” API

Unit frontend test selenium

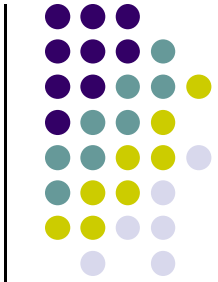


- Selenium 1 required a separate server to be running, aka selenium RC acting as a proxy between your test, brower and application
  - Injecting itself to control the browser via Javascript
  - Cool, but not realy reliable

Unit frontend test selenium

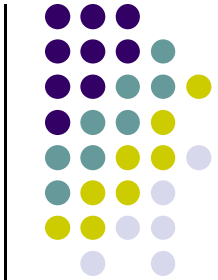


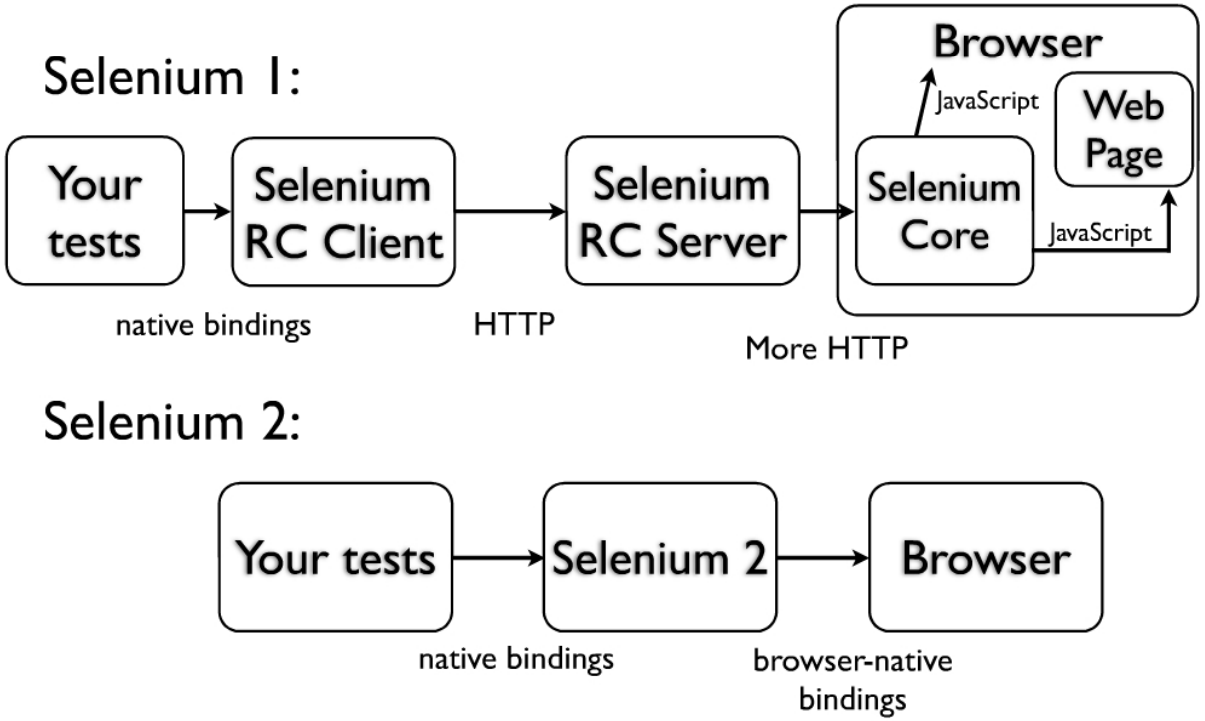
Unit frontend test selenium



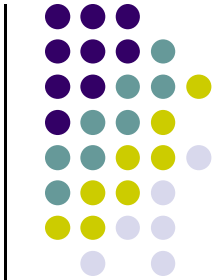
- Selenium 2 with WebDriver does not require a separate server (aka Selenium RC server)
  - Uses a “standard” interface to communicatie with the browser directly
  - The server can still be used, but not as a javascript injecting proxy, but merely a remote WebDriver

Unit frontend test selenium





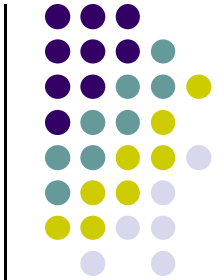
Unit frontend test selenium



- Setup our test
  - Both a UnitResourceTest as a UnitSeleniumTest

```
/**
 * @author Koen Serneels
 */
@Test(groups = "selenium")
@ActiveProfiles({ UnitResourceTest.name, UnitSeleniumTest.name })
@ContextConfiguration(classes = ApplicationConfiguration.class)
public abstract class AbstractSeleniumTest
        extends AbstractTransactionalTestNGSpringContextTests {
```

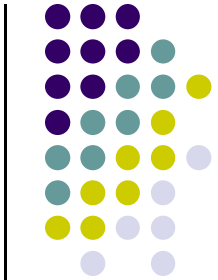
Unit frontend test selenium



- Data setup is identical as in the unit resource tests
  - test will connect to the database running inside tomcat instead of start an embedded database itself

```
@Configuration
@Profile({ UnitSeleniumTest.name })
static class H2RemoteConnection {
    @Bean
    @DependsOn("embeddedDatabase")
    public DataSource dataSource() {
        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setDriverClassName(Driver.class.getName());
        basicDataSource.setUrl("jdbc:h2:tcp://localhost/mem:db");
        basicDataSource.setUsername("sa");
        basicDataSource.setPassword("");
        return basicDataSource;
    }
}
```

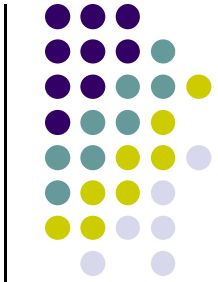
Unit frontend test selenium



- Hostname, port and browser are defaulted in the configuration
- Are overriden when we run from maven
  - system properties overrule our custom properties for this case

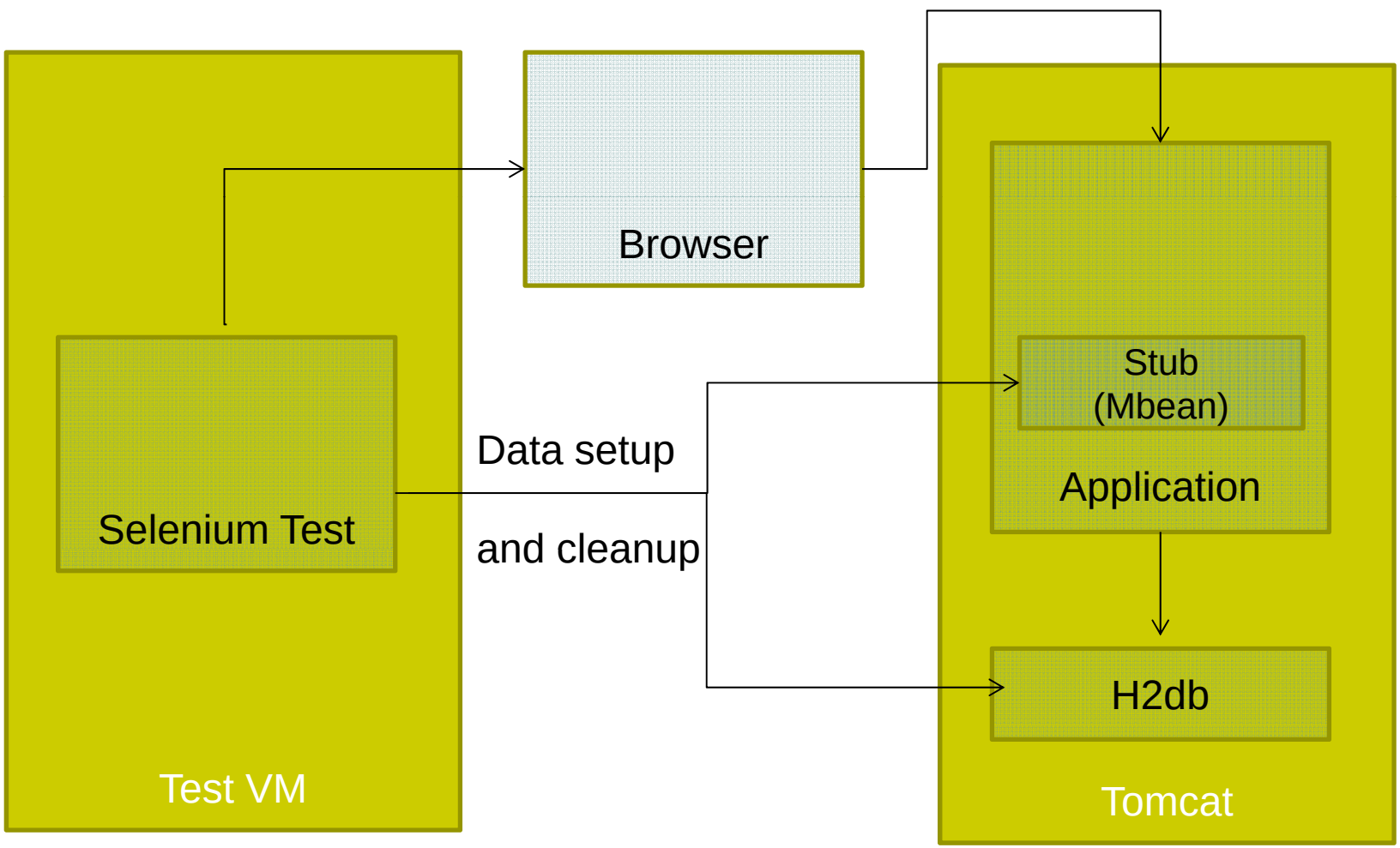
```
/**
 * These are the default properties indicating which browser Selenium should start and to which
 * server (host/port) it should connect to. They are mainly here for Selenium tests ran from the
 * IDE. In case the Selenium tests are ran by Maven, Maven will set these properties as system
 * properties which will (see {@link PropertySourcesPlaceholderConfigurer} override the ones
 * configured here.
 */
@Bean
public PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {
    PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer = new PropertySourc
    Properties properties = new Properties();
    properties.setProperty("selenium.server.port", "8080");
    properties.setProperty("selenium.server.name", "localhost");
    properties.setProperty("selenium.browser.name", "firefox");
    propertySourcesPlaceholderConfigurer.setProperties(properties);
    return propertySourcesPlaceholderConfigurer;
}
```

Unit frontend test selenium

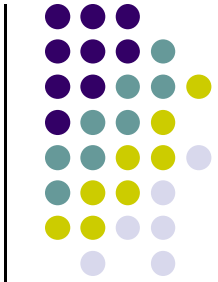


- Start tomcat
  - Using the TomcatSelenium profile we let it start an embedded database with tcp connector
  - Exposes our stubs via JMX
- Container can be started in eclipse
  - Also using the TomcatSelenium profile
- Container can be started in Maven with Cargo
  - More on that in a minute

Unit frontend test selenium

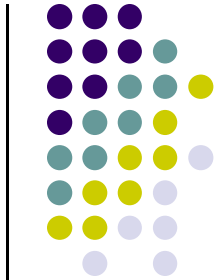


Unit frontend test selenium



- CustomerRepository depending on an external sub system does not have a in memory replacement
- Using a Stub implementation in unit frontend testing mode which can be controled via JMX
- The test can, just like it connects to the database, connect to the stub inside the tomcat VM and setup/clear data

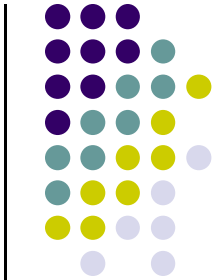
Unit frontend test selenium



- Setting up the Mbean server exposed via RMI

```
public static String JMX_URL = "service:jmx:rmi:///jndi/rmi://localhost:13000/jmxrmi";

@Bean
public RmiRegistryFactoryBean rmiRegistry() {
    RmiRegistryFactoryBean rmiRegistryFactoryBean = new RmiRegistryFactoryBean();
    rmiRegistryFactoryBean.setPort(13000);
    return rmiRegistryFactoryBean;
}

@Bean
@DependsOn("rmiRegistry")
public ConnectorServerFactoryBean connectorServerFactoryBean() {
    ConnectorServerFactoryBean connectorServerFactoryBean = new ConnectorServerFactoryBe
    connectorServerFactoryBean.setServiceUrl(JMX_URL);
    return connectorServerFactoryBean;
}
```

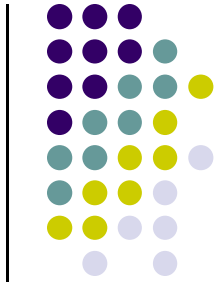
Unit frontend test selenium



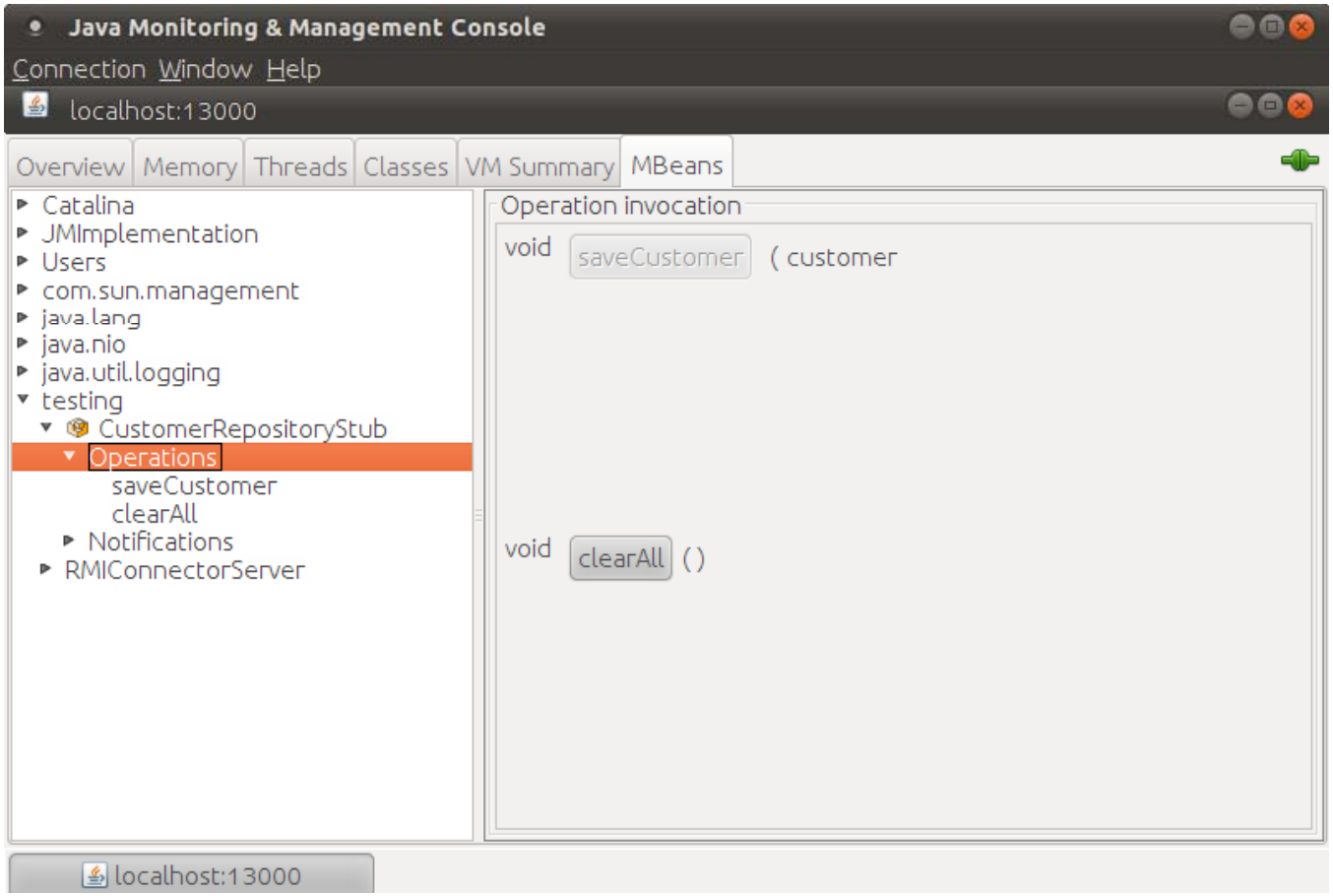
- Client side for the test

```
@Autowired
@Qualifier("mBeanServerConnectionFactoryBean")
private MBeanServerConnection mBeanServerConnection;

@Bean
public MBeanServerConnectionFactoryBean mBeanServerConnectionFactoryBean()
        throws MalformedURLException {
    MBeanServerConnectionFactoryBean mBeanServerConnectionFactoryBean = new MBeanServerC
    mBeanServerConnectionFactoryBean.setServiceUrl(TomcatSeleniumConfiguration.JMX_URL);
    return mBeanServerConnectionFactoryBean;
}

@Bean
public MBeanProxyFactoryBean mBeanProxyFactoryBean() throws MalformedObjectNameExceptior
    MBeanProxyFactoryBean mBeanProxyFactoryBean = new MBeanProxyFactoryBean();
    mBeanProxyFactoryBean.setObjectName("testing:name="
            + CustomerRepositoryStub.class.getSimpleName());
    mBeanProxyFactoryBean.setProxyInterface(CustomerRepositoryMBean.class);
    mBeanProxyFactoryBean.setServer(mBeanServerConnection);
    return mBeanProxyFactoryBean;
}
```

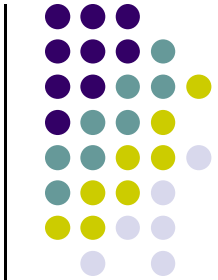
Unit frontend test selenium



- Connecting with jconsole to verify

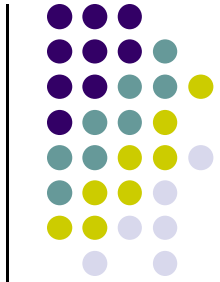


Unit frontend test selenium



- We use the maven failsafe plugin to run the frontend tests
  - Same plugin as surefire actually, but allows to perform cleanup even after tests failed
- We already saw how we setup surefire to use the suite file for the unit (resoure) tests
  - Basically we do the same with the failsafe plugin but let it use the selenium suite
- The frontend tests belong to a different group then normal tests
  - See AbstractSeleniumTest, its has @Test(groups="selenium")

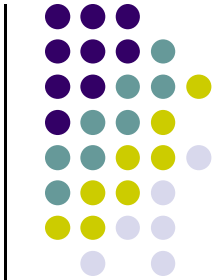
Unit frontend test selenium



- Maven config

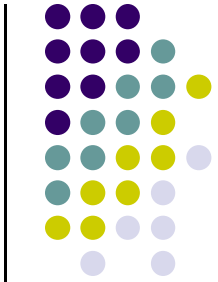
```
<plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.13</version>
        <configuration>
                <systemProperties>
                        <selenium.server.port>${selenium.server.port}</selenium.server.port>
                        <selenium.server.name>${selenium.server.name}</selenium.server.name>
                        <selenium.browser.name>firefox</selenium.browser.name>
                </systemProperties>

                <runOrder>alphabetical</runOrder>
                <suiteXmlFiles>
                        <suiteXmlFile>src/test/resources/${selenium.unit.file}</suiteXmlFile>
                </suiteXmlFiles>
        </configuration>
        <executions>
                <execution>
```

Unit frontend test selenium

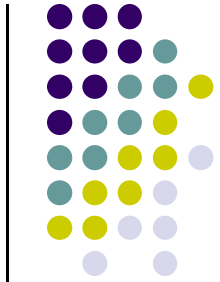


- Before the frontend tests run, we need to start our container: Maven cargo plugin
- Downloads tomcat (once) if not already present in ${base.dir}/software dir
- Can configure tomcat really extensively if required
- We also set system properties for the container host, port and the browser we desire
  - They are picked up by the test as discussed before

Unit frontend test selenium



- Let’s see the configuration



That’s it. Thank you.