Testing Enterprise Applications

Ryan Cuprak

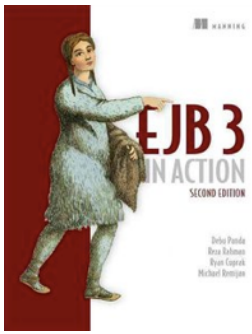
About

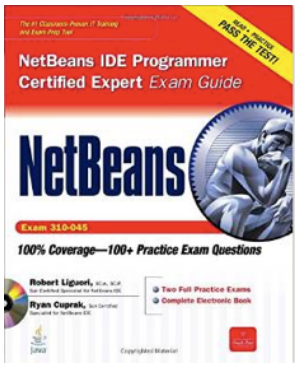
Twitter: @ctjava

Email: rcuprak@gmail.com / r5k@3ds.com

Blog: cuprak.info

Linkedin: www.linkedin.com/in/rcuprak





Testing EE Challenges

- Container Control
- Test Configuration/Deployment
- Infrastructure (JMS / Databases)
- Mocking / Simulation
- JavaScript Library / User Interface Testing
- Web Services / Microservices

Development Environment

- Maven or Gradle Build
- Docker
- Java EE 7+
- Java 8
- Node.js (npm)

Components of EE Testing

- Arquillian + Extensions
- Arquillian Cube
- Gradle Docker Plugins
- Gradle npm integration

Testing Types

Unit Tests

- Fine-grained
- Building Blocks
- Test methods

Integration Tests

- Course-grained
- Functional Units
- Test external APIs
- Full-stack
- Container Based

J2D

Introducing Arquillian

- Testing framework leveraging JUnit or TestNG to test a code running a Java container
- Framework is composed of three parts:
  - Test Runners (JUnit or TestNG)
  - Containers
  - Test Enrichers
- Leverages ShrinkWrap – utility for defining deployments.

http://arquillian.org

What can Arquillian Test?

- CDI / EJB3
- JPA
- JAX-RS / JAX-WS / WebSockets
- JSF / JSPs / Servlets
- Security

Full Stack – Running in Application Container

J2D

Arquillian Extensions

Drone

- Testing of web based interfaces
- Leverages WebDriver

Graphene

- Extensions for Selenium WebDriver
- Integrates with Arquillian Drone

Warp

- Enables server-side tests to be executed on an HTTP invocation

Persistence

- Database seeding and database comparison

Performance

- Performance regression testing

Container Types

Container interaction styles:

- Remote – container resides in a separate JVM
- Managed – container is remove but start/stop controlled
- Embedded – resides inside same JVM

Container capabilities:

- Java EE application server (JBoss, Wildfly, Payara, etc.)
- Servlet container (Tomcat/Jetty)
- Standalone bean container (OpenEJB, Weld SE)
- OSGi container

Supported Containers

Container	Support
GlassFish / Payara	Embedded, Managed, Remote
WildFly, JBoss	Embedded, Managed, Remote
TomEE	Embedded, Managed, Remote
WebSphere, Liberty	Embedded, Managed, Remote
WebLogic	Managed, Remote

Run Modes

In-container

- Test application internals
- Default mode

Client

- Test how application is used by clients
- Test web services, remote EJBs, etc.

Test Enrichment

- Injection
  - *@Resource* – reference to any JNDI entry
  - *@EJB* – Injects EJB, Local/Remote
  - *@Inject* – CDI Beans
  - *@PersistenceContext* – JPA persistence context
- Contexts
  - Request and conversation – test method
  - Session – test class
  - Application – test class
- Interceptors and decorators

Adding Arquillian

```
<dependencyManagement>
    <!-- Arquillian BOM used to reduce collision between testing dependecies. -->
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>
            <artifactId>arquillian-bom</artifactId>
            <version>1.1.10.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Adding Arquillian & Container

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</
    <version>4.12</vers
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.jboss.
    <artifactId>arquil
    <scope>test</scope>
</dependency>
```

```
<profile>
    <id>glassfish</id>
    <activation>
        <activeByDefault>true</activeByDefault>
    </activation>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian.container</groupId>
            <artifactId>arquillian-glassfish-remote-3.1</artifactId>
            <version>1.0.0.CR4</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
```

J2D

Dependency Summary

- Arquillian BOM
- JUnit and Arquillian JUnit Container
- Testing Profile for GlassFish

Simple Example

```
@RunWith(Arquillian.class)
public class MeetingBeanTests {

    @EJB
    private MeetingBean meetingBean;

    @Deployment
    public static Archive<?> createDeployment() {
        return ShrinkWrap.create(WebArchive.class, archiveName: "test.war")
                .addPackage(TestMeetingBean.class.getPackage())
                .addClass(MeetingDao.class)
                .addClass(MemberDao.class)
                .addClass(MeetingBean.class)
                .addClass(TestingSupportBean.class)
                .addClass(EntityProducer.class)
                .addAsResource( resourceName: "beans.xml", target: "META-INF/beans.xml")
                .addAsResource( resourceName: "glassfish-resources.xml", target: "META-INF/glassfish-resources.xml")
                .addAsResource( resourceName: "test-persistence.xml", target: "META-INF/persistence.xml");
    }

    @Test
    public void testCurrentMeeting() {
        Meeting currentMeeting = meetingBean.getCurrentMeeting();
        Assert.assertNotNull(currentMeeting);
    }
}
```

Use Arquillian Test Runner

Inject an EJB

Package Application

Test Application

J2D

Persistence Example

Arquillian Persistence Extension

```
<dependency>
    <groupId>org.jboss.arquillian.extension</groupId>
    <artifactId>arquillian-persistence-dbunit</artifactId>
    <version>${version.arquillian.persistence}</version>
    <scope>test</scope>
</dependency>
```

Persistence Example

```
@RunWith(Arquillian.class)
public class UserPersistenceTest {

    @Deployment
    public static WebArchive createDeploymentPackage() {...3 lines }

    @EJB
    UserService userService;

    @Resource
    javax.transaction.UserTransaction userTransaction;

    @Test
    @UsingDataSet("datasets/users.yml")
    public void shouldFindAllUsers() {}

    @Test
    @ApplyScriptBefore("scripts/drop-referential-integrity.sql")
    @ShouldMatchDataSet("datasets/expected-users.yml")
    public void shouldInsertUSers() {}
}
```

J2D

ShrinkWrap

ShrinkWrap

- API for programmatically building artifacts
- JBoss project, leveraged from Arquillian
- Used internally by the JBoss application container
- Supported archives types:
  - JAR
  - WAR
  - EAR
  - RAR

ShrinkWrap Terminology

- ***Archive*** – virtual file system.
- ***File*** – Entry in an archive – content or folder.
- ***Path*** – Location within an archive where a node lives.
- ***Asset*** – Byte based content with a node.

J2D

ShrinkWrap API

- *ShrinkWrap* class is the main entry point
- Call ShrinkWrap.create() with one of the following:
  - **GenericArchive** – simplest archive type
  - **JavaArchive** – allows for addition of class/package, and manfiest entries
  - **EnterpriseArchive** - Java EAR type
  - **WebArchive** - Java WAR type
  - **ResourceAdapterArchive** – Java RAR type

```
WebArchive myArchive = ShrinkWrap.create(WebArchive.class,”app.jar");
```

ShrinkWrap – Adding Content

- **ArchiveAsset** – Nested archive content
- **ByteArrayAsset** – byte[] or InputStream content
- **ClassAsset** – Java class content
- **ClassloaderAsset** – Resource that can be loaded by optionally-specified Classloader
- **FileAsset** – File content
- **StringAsset** – String content
- **UrlAsset** – content located at a given URL
- **EmptyAsset** – empty content

ShrinkWrap Example

```
@Deployment
public static Archive<?> createDeployment() {
    return ShrinkWrap.create(WebArchive.class, archiveName: "test.war")
            .addPackage(TestMeetingBean.class.getPackage())
            .addClass(MeetingDao.class)
            .addClass(MemberDao.class)
            .addClass(MeetingBean.class)
            .addClass(TestingSupportBean.class)
            .addClass(EntityProducer.class)
            .addAsResource( resourceName: "beans.xml", target: "META-INF/beans.xml")
            .addAsResource( resourceName: "glassfish-resources.xml", target: "META-INF/glassfish-resources.xml")
            .addAsResource( resourceName: "test-persistence.xml", target: "META-INF/persistence.xml");
}
```

Debugging:

```
archive.as(ZipExporter.class).exportTo(new File( pathname: "test.war"));
```

ShrinkWrap Example 2

```
@Deployment
public static Archive<?> createDeployment()
{
    return ShrinkWrap.create(WebArchive.class, "test.war")
            .addClasses(Conference.class, ConferenceCalendar.class)
            .addResource("confcal/submit.xhtml", "submit.xhtml")
            .addResource("confcal/submission.xhtml", "submission.xhtml")
            .addWebResource("faces-config.xml")
            .addWebResource(EmptyAsset.INSTANCE, "beans.xml")
            .setWebXML("jsf-web.xml");
}
```

Configuration

Configuration

- Container selected based on classpath
  - Add container via Maven/Gradle
- *arquillian.xml*
  - Optional Arquillian configuration file
  - Located in root of the classpath
  - Configures how to locate and communicate with the container
    - Port container is running on
    - Hostname (if remote)

Containers

Container determined by classpath:

```
<dependency>
    <groupId>fish.payara.arquillian</groupId>
    <artifactId>arquillian-payara-server-4-embedded</artifactId>
    <version>1.0.Beta2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>fish.payara.extras</groupId>
    <artifactId>payara-embedded-all</artifactId>
    <version>4.1.2.173</version>
    <scope>test</scope>
</dependency>
```

`arquillian-payara-server-4-managed`
`arquillian-payara-server-4-remote`

Arquillian XML

arquillian.xml

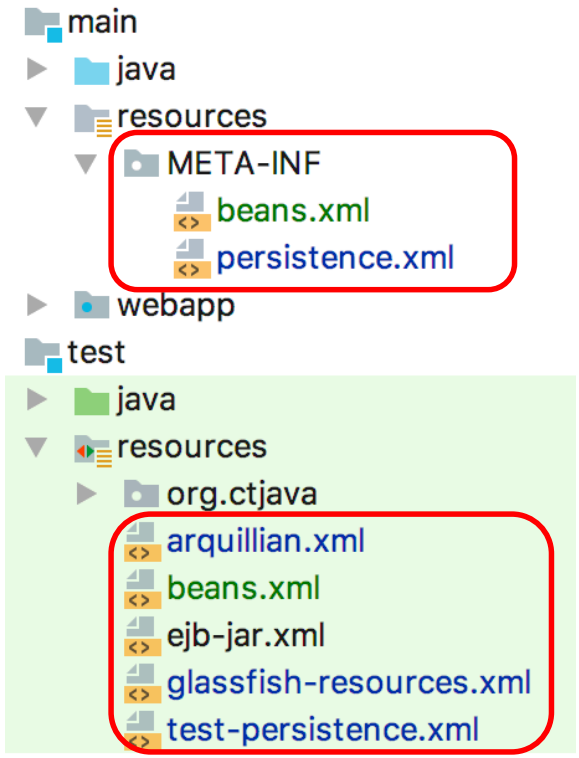
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <arquillian xmlns="http://jboss.org/schema/arquillian"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="
5          http://jboss.org/schema/arquillian
6          http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
7      <engine>
8          <property name="deploymentExportPath">target/deployments</property>
9      </engine>
10     <extension qualifier="webdriver">
11         <property name="browser">phantomjs</property>
12     </extension>
13     <container qualifier="glassfish-embedded" default="true">
14         <configuration>
15             <property name="bindHttpPort">4000</property>
16         </configuration>
17     </container>
18 </arquillian>
```

Container specific settings

Scenario

- Embedded testing using GlassFish
- Application uses JPA and a JTA data source
- Test database is developer specific

Test Configuration




```
@Deployment
public static Archive<?> createDeployment() {
    return ShrinkWrap.create(WebArchive.class,
                archiveName: TEST_APP_NAME+".war")
                .addPackage(TestMeetingBean.class.getPackage())
                .addPackage(MemberBean.class.getPackage())
                .addPackage(MemberDao.class.getPackage())
                .addClass(TestingSupportBean.class)
                .addClass(EntityProducer.class)
                .addClass(ApplicationConfig.class)
                .addClass(RegistrationRS.class)
                .addClass(ValidationExceptionMapper.class)
                .addAsResource( resourceName: "beans.xml",
                        target: "META-INF/beans.xml")
                .addAsResource( resourceName: "glassfish-resources.xml",
                        target: "META-INF/glassfish-resources.xml")
                .addAsResource( resourceName: "test-persistence.xml",
                        target: "META-INF/persistence.xml");
}
```

Test configurations added via ShrinkWrap

GlassFish Server Configuration

glassfish-resources.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 Resource Definitions//EN" "
<resources>
  <jdbc-resource pool-name="testPool" jndi-name="jdbc/ctjava"/>
  <jdbc-connection-pool name="testPool" allow-non-component-callers="false" associate-with-thread="false"
                        connection-creation-retry-attempts="0"
                        connection-creation-retry-interval-in-seconds="10" connection-leak-reclaim="true"
                        connection-leak-timeout-in-seconds="0" connection-validation-method="auto-commit"
                        datasource-classname="org.postgresql.ds.PGPoolingDataSource"
                        fail-all-connections="false" idle-timeout-in-seconds="300"
                        is-connection-validation-required="true" is-isolation-level-guaranteed="true"
                        lazy-connection-association="false" lazy-connection-enlistment="false"
                        match-connections="false" max-connection-usage-count="0" max-pool-size="32"
                        max-wait-time-in-millis="60000" non-transactional-connections="false"
                        pool-resize-quantity="2" res-type="javax.sql.ConnectionPoolDataSource"
                        statement-timeout-in-seconds="-1" steady-pool-size="8"
                        validate-atmost-once-period-in-seconds="0" wrap-jdbc-objects="false">
    <!-- <property name="url" value="@env.db.url@"/> -->
    <property name="serverName" value="127.0.0.1"/>
    <property name="databaseName" value="ctjava"/>
    <property name="User" value="@env.db.user@"/>
    <property name="Password" value="@env.db.password@"/>
  </jdbc-connection-pool>
</resources>
```

Substituted by build – pulled from environment.

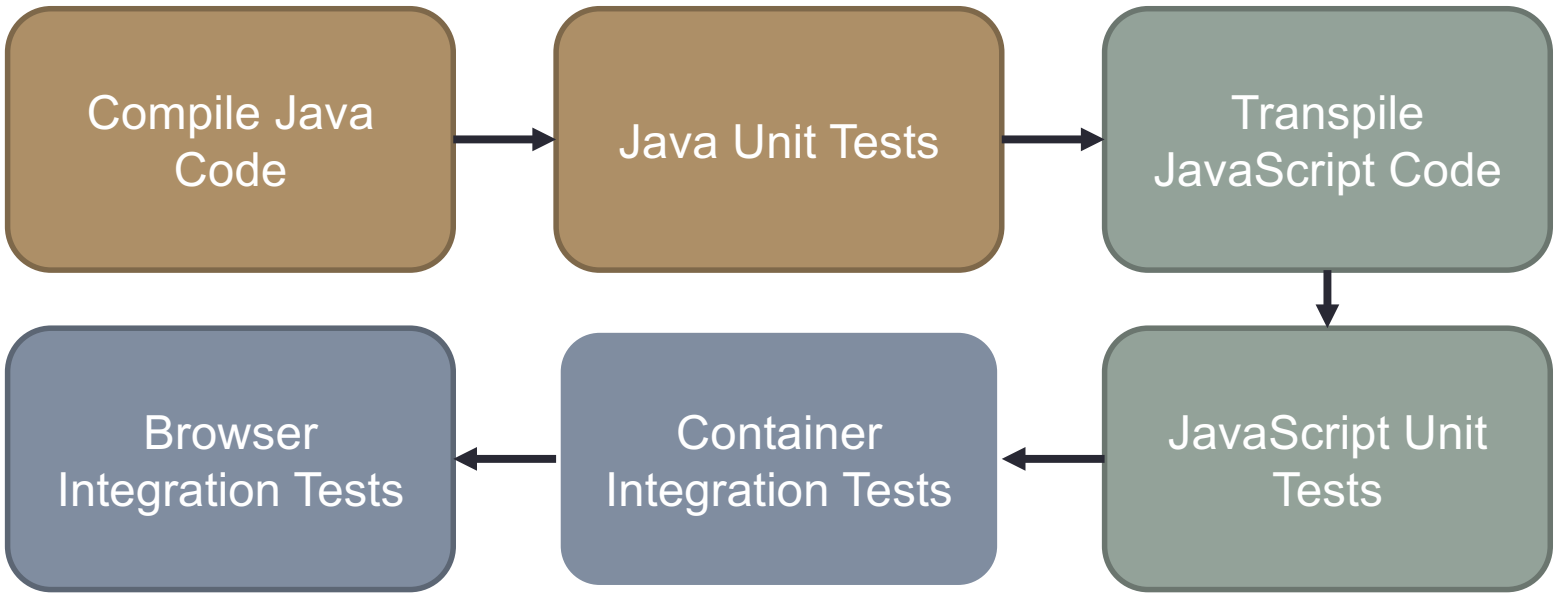
J2D

JavaScript + EE Testing

JavaScript Testing Challenges

- JavaScript is a separate ecosystem
  - Code usually “transpiled” (ECMAScript 6/7 -> 5)
  - Code implemented in TypeScript ”compiled” to ECMAScript 5
  - npm package management
  - JavaScript module loading
  - Complex frameworks – Angular 2+, React
  - State libraries: Redux
- JavaScript code depends on EE services

Build System



And the pipeline grows…

J2D

JavaScript + EE Example

- Overview
  - JavaScript dependencies managed via npm
  - Webpack module system
  - Angular 4 + TypeScript
  - npm invokes webpack via scripts
- Implementation
  - Gradle Plug-in to invoke npm -> webpack
  - Output is resources/webapp
  - Gradle WAR plug-in only sees final JS output

JavaScript + EE Example

```
plugins {
    id 'java'
    id 'war'
    id 'com.moowork.node' version '1.0.1'
}
task runWebPack(type: NpmTask) {
    workingDir = file( 'src/main/js' )
    args = ['run','build']
}

war.dependsOn(runWebPack)

task explodedWar(type: Copy) {
    into "$buildDir/exploded"
    with war
}
```

Raw ECMAScript / TypeScript files

JavaScript + EE Example

- Unit Tests: Gradle + npm
  - Transpiles JavaScript code and run webpack
  - Runs JavaScript unit tests (using Karma / Jasmine)
- Integration Tests: Arquillian +
  - Arquillian Drone – enables access to Selenium
  - Arquillian Graphene – enables AJAX testing

Selenium

- Automates web browsers
- Support multiple web browsers: Chrome, FireFox, etc.
- FireFox IDE for recording operations

  Generates Java code

Selenium Supported Browsers

- Google Chrome
- Internet Explorer 7, 8, 9, 10, and 11 on appropriate combinations of Vista, Windows 7, Windows 8, and Windows 8.1, Windows 10
- Firefox: latest ESR, previous ESR, current release, one previous release
- Safari
- Opera
- phantomjs
- Android (with Selendroid or appium)
- iOS (with ios-driver or appium)

Testing with Arquillian & Selenium

```
dependencies {
    testCompile 'org.jboss.arquillian:arquillian-bom:1.1.11.Final',
                'org.jboss.shrinkwrap.resolver:shrinkwrap-resolver-depchain:2.2.2',
                'org.jboss.arquillian.container:arquillian-glassfish-embedded-3.1:1.0.0.CR4',
                'org.jboss.arquillian.junit:arquillian-junit-container:1.1.11.Final',
                'junit:junit:4.11',
                'org.postgresql:postgresql:9.3-1102-jdbc41',
                'org.glassfish.main.extras:glassfish-embedded-all:4.1',
                'org.seleniumhq.selenium:selenium-firefox-driver:2.45.0',
                'org.seleniumhq.selenium:selenium-support:2.45.0',
                'com.github.detro:phantomjsdriver:1.2.0'
```

J2D

Construct WAR

```
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.xhtml$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.js$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.html$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.css$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_TEST_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.xhtml$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_TEST_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.js$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_TEST_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.html$"))
    .merge(ShrinkWrap.create(GenericArchive.class).as(ExplodedImporter.class)
        .importDirectory(WEBAPP_TEST_SRC).as(GenericArchive.class), path: "/", Filters.include( regexp: ".*\\.css$"))
    .addAsResource( resourceName: "glassfish-resources.xml", target: "META-INF/glassfish-resources.xml")
    .addAsResource( resourceName: "beans.xml", target: "META-INF/beans.xml")
    .addAsResource( resourceName: "test-persistence.xml", target: "META-INF/persistence.xml");
war.setWebXML(new File( pathname: "src/main/webapp/WEB-INF/web.xml"));
```

JavaScript Unit Test

```
describe("Test Registration", function () {
    var success = false;
    beforeEach(function (done) {
        console.log('executing the before...');
        var test = {
         'test': 'hi',
         'two': 'three'
        };
        $.ajax({
            type: 'POST',
            data: JSON.stringify(test),
            dataType: 'application/json',
            contentType: 'application/json',
            url: 'rest/register',
            success: function (jsonObj, textStatus, xhr) {
                success = true;
                done();
            },
            error: function (xhr, status, errorThrown) {
                success = false;
                console.log('failure: ' + errorThrown);
                done();
            }
        });
    });
    it("should have performed registration.", function () {
        expect(success).toEqual(true);
    });
});
```

Invokes server (Arquillian GlassFish) container.

Simple Selenium Test

```
@Test
@RunAsClient
public void runRegistrationTest() {
    final DesiredCapabilities capabilities = new DesiredCapabilities();
    capabilities.setJavascriptEnabled(true);
    capabilities.setCapability( capabilityName: "takesScreenshot",  value: true);
    capabilities.setCapability(PhantomJSDriverService.PHANTOMJS_EXECUTABLE_PATH_PROPERTY,phantomPath);
    WebDriver driver = new PhantomJSDriver(capabilities);
    driver.get("http://localhost:8181/test/SpecRunner.html");
    ExpectedCondition e = (ExpectedCondition<Boolean>) d -> {
        try {
            Thread.sleep( millis: 10);
        } catch (InterruptedException ex) {
            Logger.getLogger(RegistrationJSTest.class.getName()).log(Level.SEVERE,  msg: null, ex);
        }
        if(d != null) {
            List<WebElement> we = d.findElements(By.className("bar.filed"));
            return we.stream().anyMatch((element) -> (element.getText().contains("failure")));
        }
        return false;
    };
    Wait w = new WebDriverWait(driver,  timeOutInSeconds: 5);
    w.until(e);
    driver.close();
}
```

J2D

Arquillian Drone

- Manages Life-cycle of the browser
- Easy to test multiple browsers in single test
- Integration with mobile browsers
- Integration with QUnit
- Compatible with WebDriver (Selenium 2) and Selenium Grids

Arquillian Graphene

- Simple API for developing reusable tests
- Forces development of AJAX enabled tests
- Improved waiting API
- Abstracts: Page Objects and Page Fragments
- JQuery selectors as a location strategy

Arquillian Drone

```
@RunWith(Arquillian.class)
public class JPAWebDriverTest {

    @Drone
    WebDriver driver;

    @ArquillianResource
    URL deploymentUrl;

    @FindBy(id = "registerForm:username")
    WebElement registerUserNameField;

    @FindBy(id = "registerForm:password")
    WebElement registerPasswordField;

    @FindBy(id = "registerForm:register")
    WebElement submitRegistration;
```

Arquillian Drone

```
@Test
public void register() {
    // Register
    driver.get(deploymentUrl + "register.jsf");
    registerUserNameField.sendKeys(USERNAME);
    registerPasswordField.sendKeys(PASSWORD);
    // ensure that HTTP request is fired and wait for the response to be delivered
    Graphene.guardHttp(submitRegistration).click();
    Assert.assertTrue(loginHeader.isDisplayed());
    // And try to log in
    Assert.assertTrue("User should be registered and redirected to login page!"
            , loginUserNameField.isDisplayed()
                    && loginPasswordField.isDisplayed());

    loginUserNameField.clear();
    loginUserNameField.sendKeys(USERNAME);
    loginPasswordField.clear();
    loginPasswordField.sendKeys(PASSWORD);
    // ensure that HTTP request is fired and wait for the response to be delivered
    Graphene.guardHttp(submitLogin).click();
    Assert.assertTrue("User should be at welcome page!", welcomeMessage.isDisplayed());
}
```

Docker

Docker Overview

Docker Gradle Plug-ins

- Creating Docker images
- Starting/stopping Docker containers during testing

Arquillian Cube

- Enables management of containers hosted in Docker
- Reads Docker compile file and starts containers in correct order
- Executes tests in the running environment

Docker Gradle Plugins

- Build Docker images from project output:
  - Transmode/gradle-docker - http://tinyurl.com/k7o7nab
    - Build/publish docker files from build script – not Dockerfile
  - bmuschko/gradle-docker-plugin - http://tinyurl.com/hg4q6jr
    - docker-remote-api – interacts with Docker via remote API
    - docker-java-application – creates/pushes docker images for java applications
- Run Docker containers during build
  - palantir/gradle-docker - http://tinyurl.com/hpw853h
    - docker – building and pushing docker images
    - docker-compose - populating placeholders in a docker-compose template
    - docker-run – starting/stopping/status on named images

Gradle: Building Docker Images

```
apply plugin: 'java'
apply plugin: 'com.bmuschko.docker-remote-api'
import com.bmuschko.gradle.docker.tasks.image.DockerBuildImage

repositories {
    jcenter()
}

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.bmuschko:gradle-docker-plugin:3.0.3'
    }
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.21'
    testCompile 'junit:junit:4.12'
}

docker { url = 'https://127.0.0.1:4243' }

task buildImage(type: DockerBuildImage) {
    inputDir = file('.')
    tag = 'rcuprak/payara'
}
```

Gradle: Running Docker Images

```
apply plugin: 'java'
apply plugin: 'com.palantir.docker-run'

buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "gradle.plugin.com.palantir.gradle.docker:gradle-docker:0.9.0"
  }
}

dockerRun {
    name 'postgresql-test'
    image 'postgres'
    daemonize true
    env 'POSTGRES_PASSWORD': 'password'
    command 'sleep', '100'
}
```

Available Tasks:

- dockerRun
- dockerStop
- dockerRunStatus
- dockerRemoveContainer

Gradle: Docker & Testing Example

- Challenge:
  - Testing persistence code reliably is hard
  - Need a database in a “known state”
  - Minimize environment setup/configuration
- Solution:
  - Use Gradle Docker plugin to start/stop

J2D

Gradle: Docker & Testing Example

```
public class TestMonitor implements TestListener {

    private Project project;

    public TestMonitor(Project project) {
        this.project = project;
    }

    @Override
    public void beforeSuite(TestDescriptor suite) {}

    @Override
    public void afterSuite(TestDescriptor suite, TestResult result) {}

    @Override
    public void beforeTest(TestDescriptor test) {}

    @Override
    public void afterTest(TestDescriptor test, TestResult result) {
        if(result.getFailedTestCount() > 0) {
            Task task = project.getTasks().getByName("dockerStop");
            if(task != null) {
                task.setEnabled(false);
            }
        }
    }
}
```

Gradle: Docker & Testing Example

```
import org.ctjava.build.TestMonitor
apply plugin: 'java'
apply plugin: 'com.palantir.docker-run'

repositories {
    jcenter()
}

buildscript {
    repositories {
        maven { url "https://plugins.gradle.org/m2/" }
    }
    dependencies {
        classpath "gradle.plugin.com.palantir.gradle.docker:gradle-docker:0.9.0"
    }
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.21'
    testCompile 'junit:junit:4.12'
}

dockerRun {
    name 'postgresql-test'
    image 'postgres'
    daemonize true
    env 'POSTGRES_PASSWORD': 'password'
    command 'sleep', '100'
}
gradle.addListener(new TestMonitor(project))
test.dependsOn 'dockerRun'
test.finalizedBy 'dockerStop'
```

Starts Docker with initial username/password

Arquillian Cube

- Arquillian Cube better integrates Docker for testing
- Docker settings configured in arquillian.xml
- Configuration settings for Cube setup:

serverUri	Docker-registry	username
Docker-containers	Docker-containersFile	Docker-containersFiles
certPath	Machine-name	password
tlsVerify		

Arquillian Cube Dependency

```
<dependency>
   <groupId>org.arquillian.cube</groupId>
   <artifactId>arquillian-cube-docker</artifactId>
   <version>${project.version}</version>
   <scope>test</scope>
</dependency>
```

Arquillian Cube

```
<dependency>
   <groupId>org.jboss.arquillian.container</groupId>
   <artifactId>arquillian-glassfish-remote-3.1</artifactId>
   <version>1.0.0.CR4</version>
   <scope>test</scope>
</dependency>
```

Remote Container

Arquillian Cube

arquillian.xml

```
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://jboss.org/schema/arquillian"
            xsi:schemaLocation="http://jboss.org/schema/arquillian
  http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
    <extension qualifier="docker">
        <property name="dockerContainersFile">docker-compose.yml</pr
    </extension>
</arquillian>
```

J2D

Arquillian Cube

docker-compose.yaml

```
version: '2'
services:
  myservice:
    env_file: envs
    image: superbiz/myservice:${version:-latest}
    ports:
      - "8081:8080"
  db:
    image: zhilvis/h2-db
    ports:
      - "1521:1521"
      - "8181:81”
```

Arquillian Cube

```
@RunWith(Arquillian.class)
public class HelloWorldTest {

  @Drone
  WebDriver webDriver;

  @CubeIp(containerName = "helloworld")
  String ip;

  @Test
  public void shouldShowHelloWorld() throws Exception {
    URL url = new URL("http", ip, 80, "/");
    webDriver.get(url.toString());
    final String message = webDriver.findElement(By.tagName("h1")).getText();
    Assert.assertThat(message).isEqualTo("Hello world!");
  }

}
```

Challenges

- Logging – where's the failure
  - Error messages often VERY misleading
- Code structure and complexity
- Cannot mix different containers in same runtime
  - Can't test GlassFish and WildFly embedded

Summary

Java EE can be Tested!

Resources

- Books:
  - https://www.manning.com/books/testing-java-microservices
  - https://www.amazon.com/Arquillian-Testing-Guide-John-Ament/dp/1782160701
  - https://www.manning.com/books/ejb-3-in-action-second-edition
- Guides:
  - http://arquillian.org/guides/
  - https://github.com/arquillian/arquillian-examples

J2D

Q&A

Twitter: @ctjava

Email: rcuprak@gmail.com / r5k@3ds.com

Blog: cuprak.info

Linkedin: www.linkedin.com/in/rcuprak

Slides: www.slideshare.net/rcuprak/presentations

J2D