JDG + EAP Lab 1 Guide

This explains the steps for lab 1, either follow them step-by-step or if you feel adventurous read the overview and try to accomplish goals without the help of the step-by-step

NOTE: If you are looking at the PDF version and have problems with for example copying text, the original Markdown is available here (http://bit.ly/Ybh0Hn) .

Background

Acme Inc has released a new cloud service application to manage tasks lists called Todo. The application is a new HTML5 interface using AngularJS (don't worry, you don't have to know AngularJS to complete the labs). For server side it is using CDI and REST on JBoss EAP to expose CRUD services. The application was initially only released using HTML5 clients, but following the successful start Acme have also implemented native applications for Android and iPhone that are using the same REST services.

The challenge for Acme right now is that the traffic to the backend is steadily increasing and within 6 month they predict that the database they are using (H2) will be overloaded and they will either have to buy more hardware or rewrite the application using a NoSQL store.

However during a sales meeting with Red Hat the JBoss Solution Architect suggested that implementing JDG as a side cache might be a easier solution, where minimal changes to the application would have to be implemented.

Goals

Increase read performance 10 times by implementing JDG as side cache to the H2 database without changing the UI, REST service or data model object.

Objectives

The main steps in lab1 is to:

1. Configure the environment for lab1
2. Run the JUnit/Arquillian tests (performance test should fail)
3. Install the mockup application and verify that is working
4. Add dependencies to the maven project and to the WAR file for JDG
5. Add dependencies to the JDG modules in EAP via jboss-deployment-structure.xml
6. Inject a local Cache into TaskService class and implement the logic to cache findAll.

Step-by-Step

The step-by-step guide is dived into 3 different sections matching the main steps in the overview.

The First step over is to setup the lab environment

Setup the lab environment

To assist with setting up the lab environment we have provided a shell script that does this.

1. Run the shell script by standing in the jdg lab root directory (~/jdg-workshops) execute a command like this

```
$ sh init-lab.sh --lab=1
```

Install and build the mock project

1. Start the JBoss EAP if not already started in a terminal.

```
$ target/jboss-eap-6.3/bin/standalone.sh
```

2. In another terminal (on the dev host) change directory to the project

```
$ cd projects/lab1
```

3. Run the JUnit test either in JBDS or by using command line. To run the test the `arquillian-jbossas-remote-7` profile will have to be activated.

```
$ mvn -P arquillian-jbossas-remote-7 test
```

4. Build and deploy the project

```
$ mvn package jboss-as:deploy
```

5. Verify in a browser that application deployed nice successfully by opening http://localhost:8080/todo (http://localhost:8080/todo) in a browser.

6. Click around and verify that you can add tasks and complete tasks etc.

   The Mock application is simple todo application that uses a database to store tasks. It uses angular.js on the client and the server side consists of REST services to list, create and update these tasks.

7. Go thourgh the code a bit to understand the application.

Add dependencies to the maven project

In this step-by-step section we will add dependecies to the maven project so that we can later on add the code to store tasks in JDG.

1. Open the maven pom.xml file in project/todo in an editor or IDE and add the following in the dependencyManagement section

   ```
   <dependencyManagement>
       ...
       ...
       ...
       <dependency>
           <groupId>org.infinispan</groupId>
           <artifactId>infinispan-bom</artifactId>
           <version>6.1.0.Final-redhat-4</version>
           <type>pom</type>
           <scope>import</scope>
       </dependency>
   </dependencyManagement>
   ```

   And add the following dependencies.

   ```
   <dependency>
       <groupId>org.infinispan</groupId>
       <artifactId>infinispan-core</artifactId>
       <scope>provided</scope>
   </dependency>
   <dependency>
       <groupId>org.infinispan</groupId>
       <artifactId>infinispan-cdi</artifactId>
       <scope>provided</scope>
   </dependency>
   ```

   Note: We use a bom file to manage the versions of the dependencies, if you choose not to use the bom file, just specify the version directly in each dependency instead.

2. Now we need fix the class loading so that we are using the correct JDG library in the container.

   JBoss EAP ships with infinispan libraries internally, but since are using JDG 6.3 we must make sure that we use the correct infinispan libraries/modules. One solution is to ship the JDG libraries in the WEB-INF/lib folder but that makes the WAR grow allot in since effecting not only deploymenttime, but we also have to create a new release to patch or update JDG. The other solution is to use the JDG modules new as of JDG 6.3.

   The setup script that we run to setup the environment installs JDG as JBoss EAP modules, which means that we don't have to ship them as part of the WAR file. For example if we need to patch JDG we don't have to patch the application. We do however need to tell the cointainer (JBoss EAP) that our application depends on these modules. This can be done via adding dependencies to the `MANIFEST.MF` file (can be created as part of the maven built) or by using `jboss-deployment-structure.xml`. We are going to use the later since it works better with Arquillian testing.

Create a file called `jboss-deployment-structure.xml` under `src/main/webapp/WEB-INF` that looks like this:

```
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module name="org.infinispan" slot="jdg-6.3" services="import"/
>
            <module name="org.infinispan.cdi" slot="jdg-6.3" meta-inf="impo
rt"/>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

3. Run the build and deploy command again

```
$ mvn package jboss-as:deploy
```

4. Make sure that the above command are succesfull and you are done with this section.

Inject a local Cache into TaskService class and implement the logic to findAll, create, update.

1. Open TaskSevice.java in an editor or IDE and add the following as a field to the class

```
@Inject
Cache<Long, Task> cache;
```

you also need to add the follwing import statement if you IDE doesn't fix that

```
import org.infinispan.Cache;
import org.jboss.infinispan.demo.model.Task;
```

2. Change the implementation of the findAll method to look like this:

```
public Collection<Task> findAll() {
    return cache.values();
}
```

3. Change the create method to look like this:

```
public void insert(Task task) {
    if(task.getCreatedOn()==null) {
        task.setCreatedOn(new Date());
    }
    em.persist(task);
    cache.put(task.getId(),task);
}
```

4. Add the implementation of the update method as shown below:

   public void update(Task task) {

```
em.merge(task);
cache.replace(task.getId(),task);
```

   }

5. Add the implementation of the delete method as shown below:

   public void delete(Task task) {

```
em.remove(em.getReference(task.getClass(),task.getId()));
cache.remove(task.getId());
```

   }

6. We also need fill the cache with the existing values in the database using by adding the following method:

```
 @PostConstruct
 public void startup() {

     log.info("### Querying the database for tasks!!!!");
     final CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
     final CriteriaQuery<Task> criteriaQuery = criteriaBuilder.createQuery(T
ask.class);

     Root<Task> root = criteriaQuery.from(Task.class);
     criteriaQuery.select(root);
     Collection<Task> resultList = em.createQuery(criteriaQuery).getResultLis
t();

     for (Task task : resultList) {
         this.insert(task);
     }

 }
```

7. Next make sure that the TaskServiceTest class adds the jboss-deployment-structure.xml, which should look like this:

```
 .addAsWebInfResource(new File("src/main/webapp/WEB-INF/jboss-deployment-stru
cture.xml"))
```

8. Run the JUnit test to see that everything works as expected

9. Your TaskService.java implementation should look something like this:

```
package org.jboss.infinispan.demo;

import java.util.Collection;
import java.util.Date;
import java.util.logging.Logger;

import javax.annotation.PostConstruct;
import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

import org.infinispan.Cache;
import org.jboss.infinispan.demo.model.Task;

@Stateless
public class TaskService {

    @PersistenceContext
    EntityManager em;

    @Inject
    Cache<Long,Task> cache;

    Logger log = Logger.getLogger(this.getClass().getName());

    /**
     * This methods should return all cache entries, currently contains mock
up code.
     * @return
     */
    public Collection<Task> findAll() {
        return cache.values();
    }

    public void insert(Task task) {
        if(task.getCreatedOn()==null) {
            task.setCreatedOn(new Date());
        }
        em.persist(task);
        cache.put(task.getId(),task);
    }
```

```
    public void update(Task task) {
        em.merge(task);
        cache.replace(task.getId(),task);
    }
```

```
        @PostConstruct
        public void startup() {

            log.info("### Querying the database for tasks!!!!");
            final CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
            final CriteriaQuery<Task> criteriaQuery = criteriaBuilder.createQuery(
Task.class);

            Root<Task> root = criteriaQuery.from(Task.class);
            criteriaQuery.select(root);
            Collection<Task> resultList = em.createQuery(criteriaQuery).getResultL
ist();

            for (Task task : resultList) {
                this.insert(task);
            }

        }

    }
```

1. Hold on with deploy to the application server. There are one issue with the current setup that we will solve in the next

Configure the cache programatically

What just happend is that we have implemented a local cache solution where we can offload the database based on the default configuraiton. We haven't yet configured any setting with the cache. There are allot of different possibilities to tweak the JDG library mode settings, but at the moment we will only do some basic configuration settings. Settings can be done in XML or in code. In this example we will use the code API, but later we will use the XML to configure JDG in standalone mode.

Below is a code snipped that shows how to create configuration objects for the cache.

```
GlobalConfiguration glob = new GlobalConfigurationBuilder()
        .globalJmxStatistics().allowDuplicateDomains(true).enable() // This
        // method enables the jmx statistics of the global
        // configuration and allows for duplicate JMX domains
        .build();
Configuration loc = new ConfigurationBuilder().jmxStatistics()
        .enable() // Enable JMX statistics
        .eviction().strategy(EvictionStrategy.NONE) // Do not evic objects
        .build();
DefaultCacheManager manager = new DefaultCacheManager(glob, loc, true);
```

There are two main configuration object: `GlobalConfiguration` for the Global configuration if we use for example multiple clustred configurations and `Configuration` to hold the local configuration. In this example we allow muliple domains since otherwise we get a nasty exception saying that the cache allready exists. In the local configuration we enable JMX statistics (need for JON for example) and we set the eviction.strategy to NONE, meaning that no objects are evicted.

We can then create a cache manager object using these configuration and pass it true to also start it.

Since we are using CDI in our example we can actually override the cache manager that is used when someone injects a cache with `@Inject Cache<?,?> cache;` like we do in TaskService class. This can be done using something called Producer in CDI. So all we have to do is crate a method that looks like this:

```
@Produces
@ApplicationScoped
@Default
public EmbeddedCacheManager defaultEmbeddedCacheConfiguration() { ... }
```

Then we put this class somewhere in our classpath (or even better in our source) and add the configuration code from above in it.

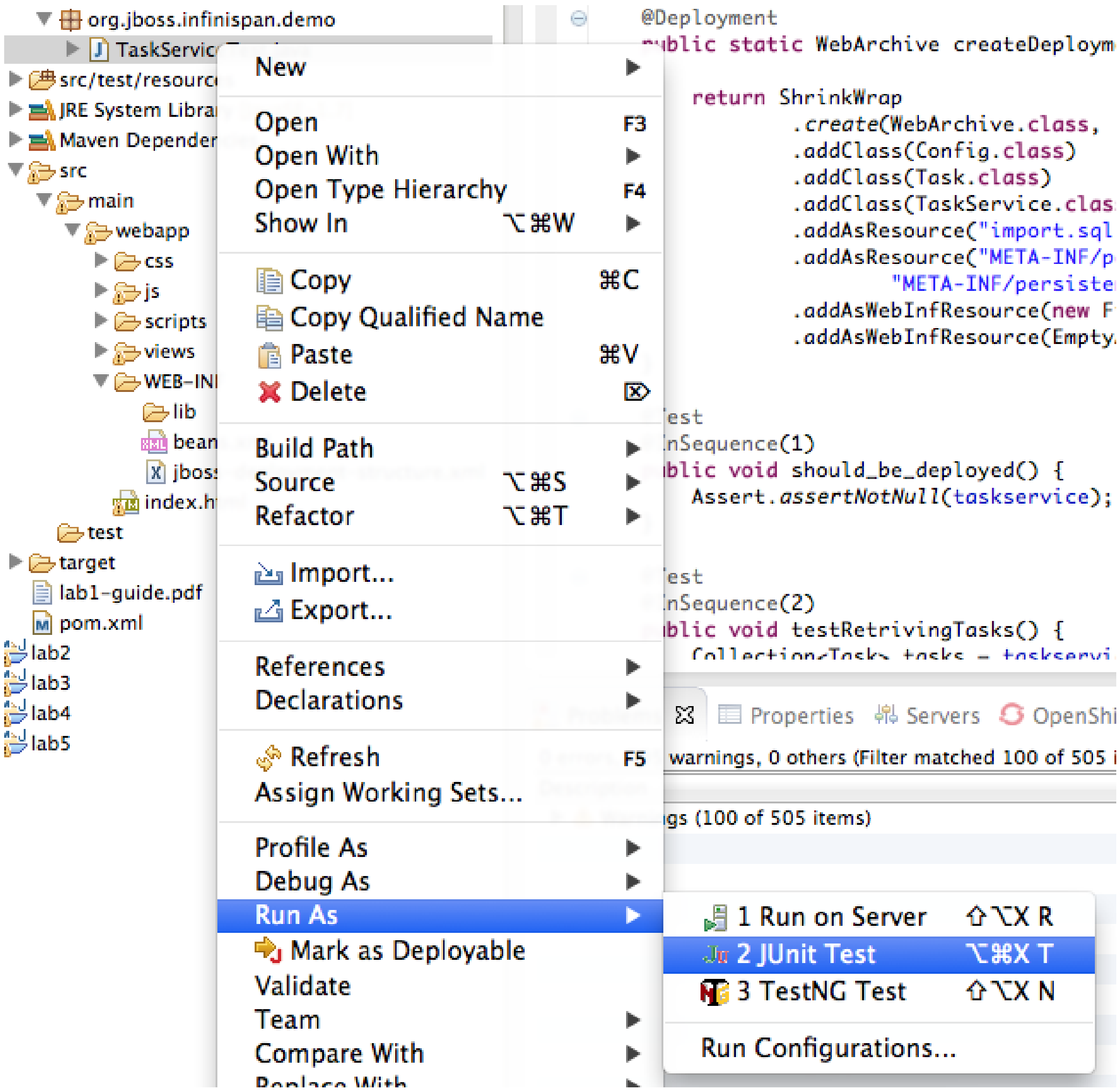
1. Add a Config class in package org.jboss.infinispan.demo that looks loke this:

```
package org.jboss.infinispan.demo;

import javax.annotation.PreDestroy;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Default;
import javax.enterprise.inject.Produces;

import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.configuration.global.GlobalConfiguration;
import org.infinispan.configuration.global.GlobalConfigurationBuilder;
import org.infinispan.eviction.EvictionStrategy;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.manager.EmbeddedCacheManager;

public class Config {

    private EmbeddedCacheManager manager;

    @Produces
    @ApplicationScoped
    @Default
    public EmbeddedCacheManager defaultEmbeddedCacheConfiguration() {
        if (manager == null) {
            GlobalConfiguration glob = new GlobalConfigurationBuilder()
                    .globalJmxStatistics().allowDuplicateDomains(true).enab
le() // This
                    // method enables the jmx statistics of the global
                    // configuration and allows for duplicate JMX domains
                    .build();
            Configuration loc = new ConfigurationBuilder().jmxStatistics()
                    .enable() // Enable JMX statistics
                    .eviction().strategy(EvictionStrategy.NONE) // Do not ev
ic objects
                    .build();
                manager = new DefaultCacheManager(glob, loc, true);
            }
            return manager;
        }

        @PreDestroy
        public void cleanUp() {
            manager.stop();
            manager = null;
        }
    }
```

2. Soon we are ready to deploy the application, but first we need to make sure that test passes. Before we run the test, lets check that TaskServiceTest.java add the Config class to the test, liek this:

```
.addClass(Config.class)
```

3. Run the JUnit test by right clicking TaskServiceTest.java and select Run As ... -> JUnit Test



```
@Deployment
ublic static WebArchive createDeploym

   return ShrinkWrap
           .create(WebArchive.class,
           .addClass(Config.class)
           .addClass(Task.class)
           .addClass(TaskService.clas
           .addAsResource("import.sql
           .addAsResource("META-INF/p
                   "META-INF/persiste
           .addAsWebInfResource(new F
           .addAsWebInfResource(Empty.

est
nSequence(1)
blic void should_be_deployed() {
   Assert.assertNotNull(taskservice);

est
nSequence(2)
blic void testRetrivingTasks() {
```

```
warnings, 0 others (Filter matched 100 of 505 i
gs (100 of 505 items)
```

4. If everything is green we are ready to deploy the application with the following command in a terminal

```
$ mvn package jboss-as:deploy
```

5. Test the application by opening a browser window to http://localhost:8080/todo ()

6. Congratulations you are done with lab1.