JBoss Data Grid lab guides

Lab 1

Thomas Qvarnström

v1.1 2015-12-06

Table of Contents

1. Background . . . 1
2. Goals . . . 1
3. Objectives . . . 1
4. Step-by-Step . . . 2
    4.1. Setup the lab environment . . . 2
    4.2. Install and build the mock project . . . 2
    4.3. Add dependencies to the maven project . . . 3
    4.4. Inject a local Cache into TaskService class and implement the logic to findAll, create, update. . . . 5
    4.5. Configure the cache programmatically . . . 8

This guide explains the steps running for lab 1, either follow the steps in the step-by-step section or if you feel adventurous try to accomplish goals without the step-by-step guide.

NOTE	If you are looking at the PDF version and have problems with for example copying text, the original Asciidoc version is available here.

1. Background

Acme Inc has released a new cloud service application to manage tasks lists called `todo`. The application is implemented using HTML5 and AngularJS (don't worry, you don't have to know AngularJS to complete the labs) for the client side. The server side uses CDI and REST on JBoss EAP to expose CRUD services on top of a database. There are also Android and iPhone apps for `todo` that are using the same REST services.

The main challenge for Acme right now is that the round trip to a database is to expensive for the smart apps and the responsive UI interface.

During a meeting with the local Red Hat Sales team, the JBoss SA suggested that Acme should use JDG to avoid the expensive round trip. Initially the JBoss SA recommends that Acme implements JDG as a side cache with minimal changes to the application.

2. Goals

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

3. 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.

4. 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

4.1. 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-labs) execute a command like this

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

4.2. Install and build the mock project

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

```
$ target/jboss-eap-6.4/bin/standalone.sh
```

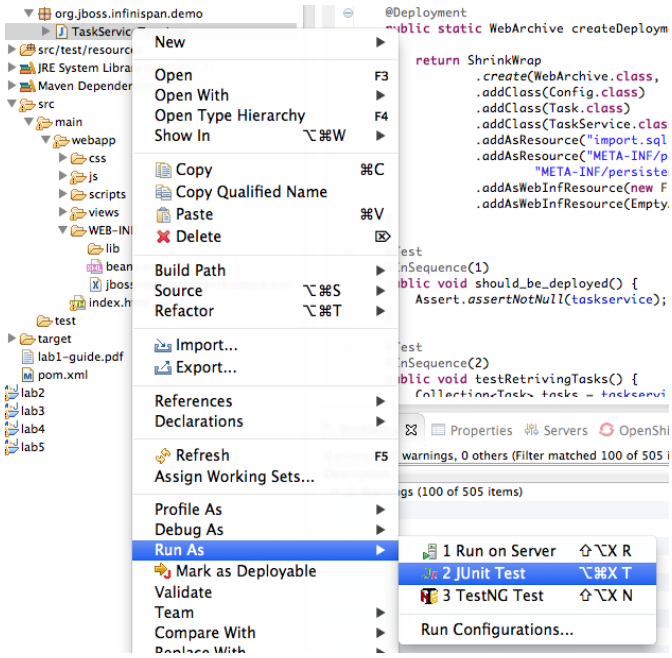
2. In another terminal (on the dev host) change directory to the project

```
$ cd projects/lab1
```

3. Run the JUnit test either in JBDS (see an example in the next step) or by using command line (below). To run the test the `arquillian-jbossas-remote-7` profile will have to be activated.

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

4. Run the JUnit test by right clicking TaskServiceTest.java and select Run As ... → JUnit Test



5. Build and deploy the project

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

6. Verify in a browser that application deployed nice successfully by opening http://localhost:8080/mytodo in a browser.
7. 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.

8. Go through the code a bit to understand the application.

4.3. Add dependencies to the maven project

In this step-by-step section we will add dependencies to the maven project so that we can later on add the code to store tasks in JDG. Adding JDG to a project is made simple the the use of uber-jars, so that only a single dependency is required.

1. Open the maven pom.xml file in an editor or IDE and add the following dependency:

```
<dependencies>
    ...
    ...
    <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-embedded</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

NOTE	We use a bom reference in the `parent/pom.xml` 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 6.5 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 significantly. This not only effects deployment time, but also requires that we create a new release to patch or update JDG. A better 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. Because we are using JBoss modules, if we need to patch JDG we don't have to patch the application. We do however need to tell the container (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.

   Update the file called `jboss-deployment-structure.xml` under `src/main/webapp/WEB-INF` to look like this:

```
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module name="org.infinispan" slot="jdg-6.5"/>
            <module name="org.infinispan.cdi" slot="jdg-6.5" meta-inf="import"/>
            <module name="org.jgroups" slot="jdg-6.5"/>
            <module name="org.infinispan.persistence.jpa" slot="jdg-6.5" services=
"export"/>
            <module name="org.hibernate"/>
        </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 successful and you are done with this section.

4.4. 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 following import statement if you IDE does not fix that (please ignore the message: "No bean is eligible for injection to the injection point [JSR-299 §5.2.1]")

```
import javax.inject.Inject;
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 insert 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. Change the implementation of the update method to look like this:

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

5. Change the implementation of the delete method to look like this:

```
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 updating the startup method to look like this:

```
@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).getResultList();

    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-structure.xml
"))
```

8. 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 mockup 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).getResultList();

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

    }

}
```

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

4.5. Configure the cache programmatically

What just happened is that we have implemented a local cache solution where we can offload the database based on the default configuration. 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 objects:

- `GlobalConfiguration` is used for the global configuration attributes that are applied to all caches created from this cache container. For example, the network transport and security are both configured in GlobalConfiguration.
- `Configuration` to hold the local configuration. In this example we allow multiple domains since otherwise we get a nasty exception saying that the cache already exists. In the local configuration we enable JMX statistics (needed 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 the TaskService class. This can be done using something called Producer in CDI. So all we have to do is create 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. Update the Config class in package org.jboss.infinispan.demo to look like this:

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

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 {

    @Produces
    @ApplicationScoped
    @Default
    public EmbeddedCacheManager defaultEmbeddedCacheConfiguration() {
        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();
        return new DefaultCacheManager(glob, loc, true);
    }

}
```

2. We are nearly 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, like this:

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

3. Execute the test and verify that the performance test that was failing is now passing.
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/mytodo
6. Congratulations you are done with lab1.