Preface

You are a **developer** or an architect creating world-class applications and solutions. Well, all apps don't necessarily live in just one box, and for the recent past, Cloud is not only one more computer but a lot more. All applications use a network to communicate with each other, whether they connect to a database, a web service or storage, or other services.

Creating a distributed application means one has to take specials care of latency, performance, and, most importantly, the responsiveness to the end-users. End users could be systems or humans; what's important is that you have a timely and consistent response. There are a lot many things one can do to optimize the stack. Most importantly, this workshop focuses on one thing i.e., Red Hat Data Grid.

What is Red Hat Data Grid? Red Hat Data Grid 8.0 provides a distributed in-memory, NoSQL datastore solution. Your applications can access, process, and analyze data at in-memory speed to deliver a superior user experience. Whether you are using legacy applications or a new breed of microservices and functions, Red Hat Data Grid 8.0 will enable your applications to perform better with its in-memory solution.

This lab offers attendees an intro-level, hands-on session with Red Hat Data Grid. From the first line of code to building & consuming services and finally to assembling everything and deploying it on Openshift. It illustrates what a Cache is, how to build applications with a distributed cache, as well as best practices on designing applications with caching in mind.

Deploying and maintaining microservices can become challenging. There are many elements of software design one needs to ensure in constructing and maintaining distributed services. To help us deploy our microservices we are going to use Openshift, a Kubernetes distribution from Red Hat. Kubernetes (commonly referred to as "K8s") is an open-source system for automating deployment, scaling, and management of containerized applications that was originally designed by Google and donated to the Cloud Native Computing Foundation. It aims to provide a "platform for automating deployment, scaling, and operations of application containers across clusters of hosts". In this workshop, we are using a specific distribution of K8s named OpenShift that provide a few sets of features beneficial to maintain our microservices. We use OpenShift in this lab because of the complexity involved when dealing with multiple microservices, their updates, downtimes, and so on.

This is a BYOL (Bring Your Own Laptop) session. You will be provided with a CodeReady Workspace, an Openshift environment all pre-provisioned. If you prefer to run locally on your laptop, you can do development on your laptop. Still, you will need to use the provided OpenShift environment as Operators installed for you are already set up there. So bring your Windows, OSX, or Linux laptop. If you decide to run locally on your laptop you need JDK 8+ and Apache Maven (3.5+).

What you are going to learn:

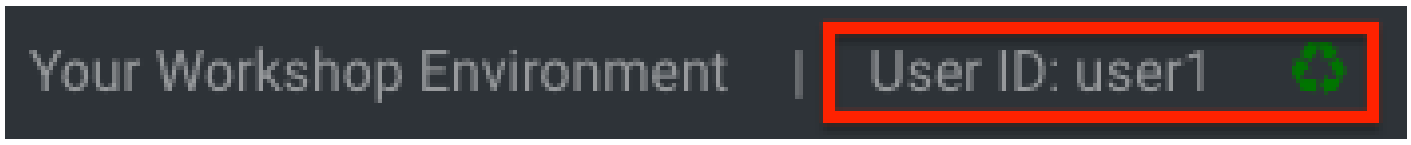
- What is Red Hat Data Grid 8.0
- What is a Cache, and how to start with the common use cases
- What is an Embedded Cache
- What is Clustering in a Cache scenario and how it works
- What is a Remote Cache and how to take benefit of it within Red Hat Data Grid
- How to build applications with known frameworks like Quarkus and Spring
- How to use the Red Hat Data Grid server REST API
- How to externalize sessions

And much more!

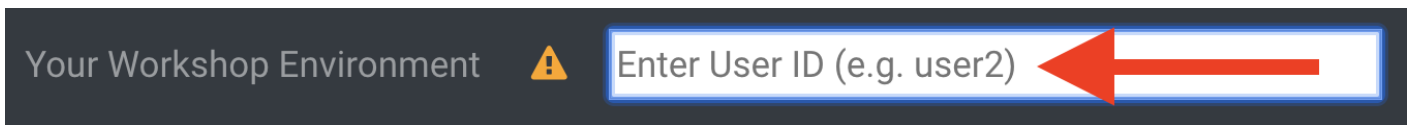
Setting up

First Step: Confirm Your Username!

Look in the box at the top of your screen. Is your username set already? If so it will look like this:



If your username is properly set, then you can move on. **If not, in the above box, enter the user ID you were assigned** like this:



This will customize the links and copy/paste code for this workshop. If you accidently type the wrong username, just click the green recycle icon to reset it.

Throughout this lab you'll discover how Quarkus can make your development of cloud native apps faster and more productive.

Click-to-Copy

You will see various code and command blocks throughout these exercises which can be copy/pasted directly by clicking anywhere on the block of text:

```java
/* A sample Java snippet that you can copy/paste by clicking */
public class CopyMeDirectly {
    public static void main(String[] args) {
        System.out.println("You can copy this whole class with a click!");
    }
}
```

Simply click once and the whole block is copied to your clipboard, ready to be pasted with `CTRL` + `V` (or `Command` + `V` on Mac OS).

There are also Linux shell commands that can also be copied and pasted into a Terminal in your Development Environment:

```sh
echo "This is a bash shell command that you can copy/paste by clicking"
```

The Workshop Environment You Are Using

Your workshop environment consists of several components which have been pre-installed and are ready to use. Depending on which parts of the workshop you're doing, you will use one or more of:

- Red Hat OpenShift (https://www.openshift.com/) - You'll use one or more **projects** (Kubernetes namespaces) that are your own and are isolated from other workshop students
- Red Hat CodeReady Workspaces (https://developers.redhat.com/products/codeready-workspaces/overview) - based on **Eclipse Che**, it's a cloud-based, in-browser IDE (similar to IntelliJ IDEA, VSCode, Eclipse IDE). You've been provisioned your own personal workspace for use with this workshop. You'll write, test, and deploy code from here.
- Red Hat Runtimes (https://www.redhat.com/en/products/runtimes) - a collection of cloud-native runtimes like Spring Boot, Node.js, and Quarkus (https://quarkus.io)

You'll be provided clickable URLs throughout the workshop to access the services that have been installed for you.

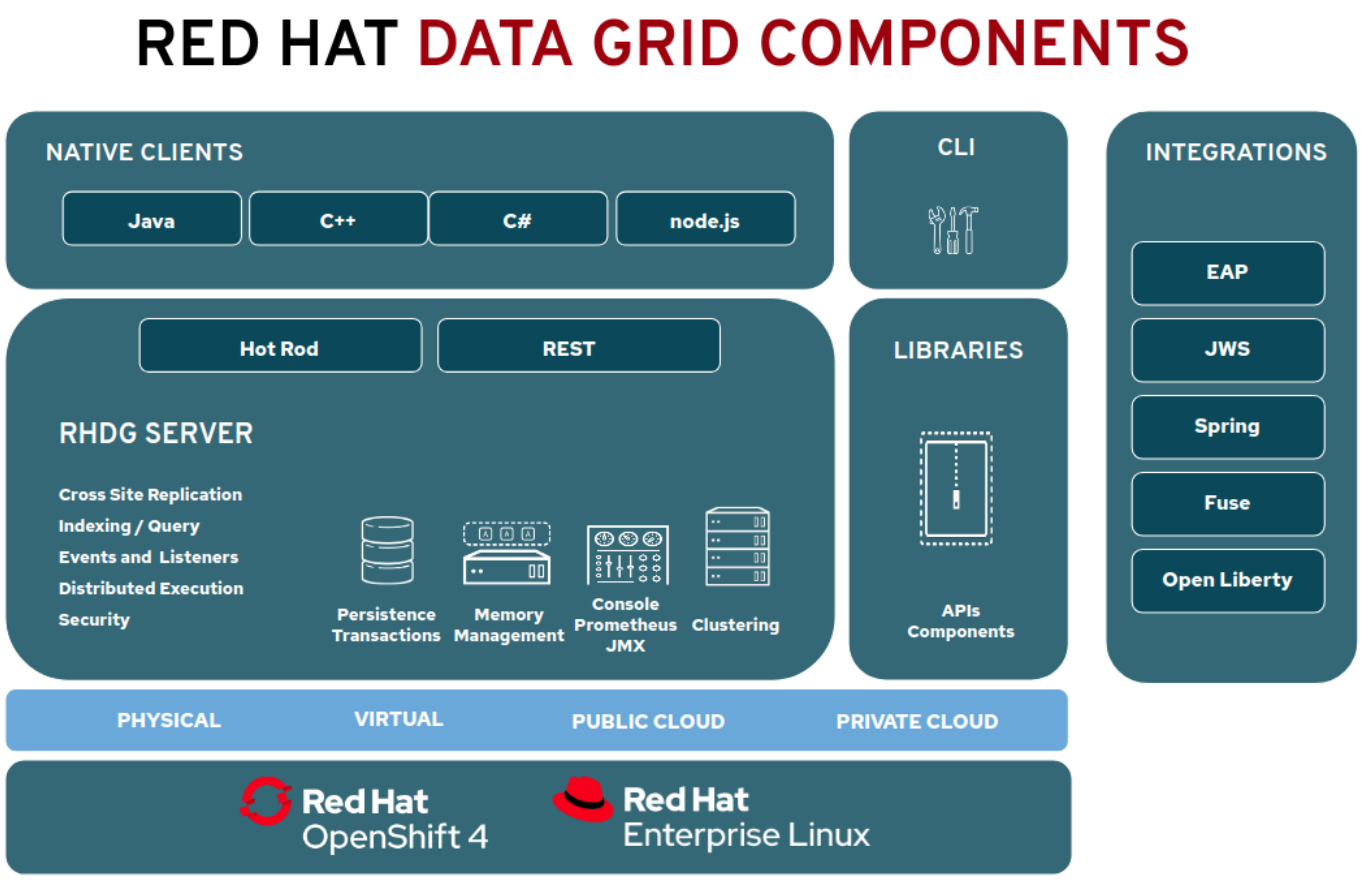
How to complete this workshop

Click the "Next >" button at the bottom to advance to the next topic. You can also use the menu on the left to move around the instructions at will.

Good luck, and let's get started!

DataGrid in 10 mins

The latest update to Red Hat Runtimes is with the release of Red Hat Data Grid 8.0, which provides a distributed in-memory, NoSQL datastore solution. Your applications can access, process, and analyze data at in-memory speed to deliver a superior user experience. Whether you are using legacy applications or a new breed of microservices and functions, Red Hat Data Grid 8.0 will enable the journey to Open Hybrid Cloud. Data Grid includes the Infinispan open-source software project. It is available to deploy as an embedded library, as a standalone server, or as a containerized application on Red Hat OpenShift Container Platform.



A Full Lifecycle Operator to reduce deployment and management overhead in OpenShift

An Operator enables the operations and lifecycle management for an application by using the underlying Kubernetes APIs. That means complex applications (e.g. consumed as services such as distributed caching, databases, etc) can easily get upgraded when newer versions arrive and more; also meaning no human intervention. The Operator SDK enables developers to write such Operators. Red Hat Data Grid 8.0 introduces a fully supported Data Grid Operator that provides operational intelligence.

A new server architecture

Cloud and Container Native Data Grid needs to have a reduced footprint, and that's what the latest version of Red Hat Data Grid brings. It reduces both the disk footprint and initial heap size upto 50%, leaving more memory for your data. You can now run the server without the Red Hat JBoss Enterprise Application Platform (EAP), ensuring a lower memory and disk footprint also simplifying configuration.

Moreover, Data Grid 8.0 servers provide several enhancements and improvements to security, including integration with Red Hat SSO and a smaller attack surface.

A more performant and rich REST API

Red Hat Data Grid 8.0 introduces REST API v2. The API has 50% faster response rates compared to v1. There are also new capabilities introduced such as

- Access data and manipulate objects (such as counters)
- Perform operations such as gracefully shutting down Data Grid clusters or transferring cache state to backup locations when using cross-site replication
- And lastly, monitor cluster and server health and retrieve statistics

Moreover, Red Hat Data Grid REST API v2 also automatically converts between storage formats such as JSON, XML, Protobuf, and plain text for increased interoperability. The Red Hat Data Grid engineering team develops and maintains comprehensive REST API Documentation.

A powerful CLI

In 8.0, Data Grid gives you a new CLI with intuitive commands for remotely accessing data and managing clusters. The CLI uses familiar Bash commands for navigating, such as `cd` and `ls`. It also provides command history and auto-completion for ease of use.

Additionally, the CLI provides help text and man pages for commands with clear examples.

Enhanced observability

Now you can use the `/metrics` endpoint for integration with Prometheus. Moreover, Red Hat Data Grid 8 is also compatible with Eclipse Microprofile Metrics API. More specific metrics and gauges are included. Data Grid 8.0 also offers improved statistics and management operations via JMX and updates to logging with coarse-grained logging categories and support for logs in JSON format.

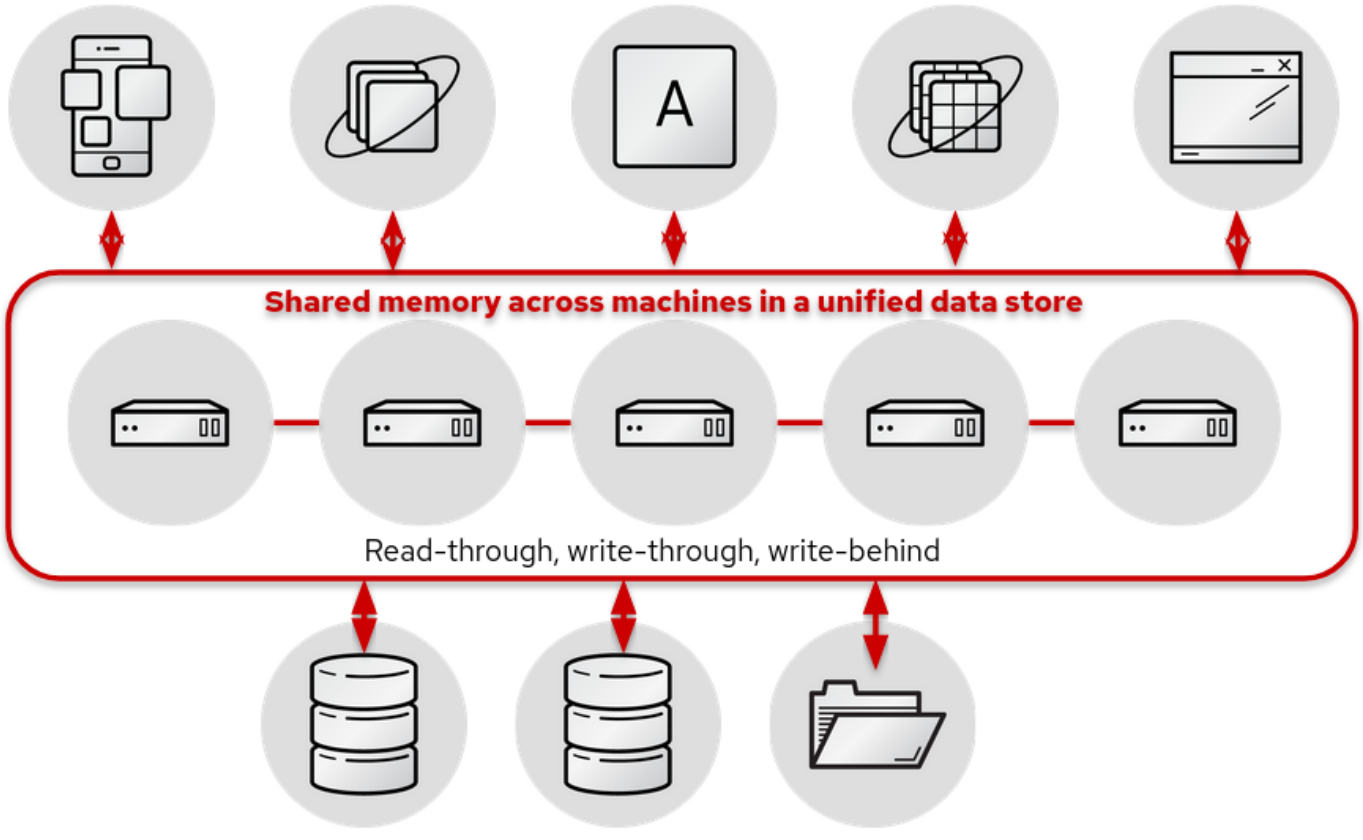
What is Caching and how to apply it with Red Hat Data Grid?

Modern applications use data and lets assume lots of data, whether it comes from databases, files, webservices, rpc call etc. When an application has to process this data, what is the most natural way of doing it? Mostly the application will process this data in memory. Let's assume I have a very slow database, this could be due to any reason, network latency, or even big queries that return alot of data. So the most straightforward way of handling this would be to store some of that data in the memory. By doing so, you would be able to process requests to your systems much faster. However there are challenges. How much data should you store in memory? And most importantly, what will happen in case of failure scenarios?

- Will you lose all the state of your data in memory?
- Will you need to re-read all your data and events inorder to get back to the same state where you failed. Or you might have to let go of that entirely.

Now the above two might seem very simple, but that tasks can be tedious and most importantly error prone. So at this point we could introduce a local cache (e.g. a `HashMap`) that a lot of us might have done this in the past. However as a developer you might know that this has not much effect in case of failures. So the need is entirely for a component that can not just cache data in the memory, but give

1. A consistent way to handle data and state in the memory
2. Resiliencey in case of failures
3. Processing efficency and performance
4. Events, streams, and distribution capabilites



By having such capabilites a cache is no longer just an in-memory data structure, but also as a developer now you have the possiblity to take this component out of your local in memory processing and distribute it out on the network. Thereby in case of application failures you will still be able to access this data from the last point where you left off.

Now getting back to our primary question, how much data should you store in memory? Partially we have already discussed this above. Whats important is that as a developer you should be able to specifiy TTL (Time To Live) for your cache and its entries. You should be able to define eviction and expiration. Note that eviction is to prevent from memory overuse and not to remove the entry from the cache, it will drop an entry from memory on this instance and does not affect other instances or the persistence. It must be used with a configured persistence to be consistent. Whereas expiration will retire the entry and remove it from the cache and its persistence completely. There by knowing when your cache is hot and what data resides in it. Most over you should be able to do this distributed, cluster wide, or remotely.

Once a cache is remote, we also want some of the distributed features, like monitoring for example. Lets take a look at some of the caching strategies.

Local cache

The primary use for Red Hat Data Grid is to provide a fast in-memory cache of frequently accessed data. Suppose you have a slow data source (database, web service, text file, etc) - you could load some or all of that data in memory so that it's just a memory access away from your code. Using Red Hat Data Grid is better than using a simple `ConcurrentHashMap` . By setting up an embedded cache, Red Hat Dat Grid also allows you to tap into more features e.g. expiration, eviction, events on the cache etc. All make out a much better way of handling your cache and component design. Moreover if you would want to cluster such a cache that is also easily possible.

As a clustered cache

Lets assume you started with a local embedded cache in your application and now you suddenly realize that one instance of your application is not enough to handle the load from your users or systems. What do you do? With Red Hat Data Grid you can now scale that cache into a cluster. You dont need to change how you use your cache, but adding a few additonal config params you can now have a clustered cache and by having multiple instances of your application listenting to the same coherent cache. Events will be fired accross the cluster, expiration will happen accross the cluster, etc. Eviction removes entries from the local instance memory if not used, but not from persistent cache stores or other cluster members to ensure that the local Data Grid does not exceed that maximum size. And most over, you now even have the possiblity to distribute your keys accross the cluster. Red Hat Data Grid can scale horizontally to hundreds of nodes.

As a remote cache

Lets just say you used the clusterd cache, and embedded it in your application, which means that everytime a new instance of your application started you would have a new instance of your embedded cache ready to become part of the cluster. Now this is all great, but what if you dont want that clustering in your application? Rather then you might want to use a component outside of your applications lifecycle. Or you would want to share this cache accross multiple applications. In that case the Red Hat Data Grid could be used as a remote data grid. Now you can access your cache via multiple programming runtimes (e.g. Vert.x, Quarkus, NodeJS, C#, C/C++ etc), and your cache lifecycle and memory consumption will be independant of the applications life cycle, which is a great advantage in many cases.

Congratulations! By now you understand the different patterns of caching, and the requirements. Lets go ahead and create our first application and learn how we can use Red Hat Data Grid to achieve caching. Press next!

Additional Resources:

- Traditional zip deployments are available on the Customer Portal (https://access.redhat.com) Red Hat Data Grid download page (https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=data.grid).
- The container distribution and operator are available in the Red Hat Container Catalog (https://catalog.redhat.com/software/containers/explore)
- Product documentation is available here (https://docs.redhat.com)
- Getting Started Guide that will get you running with RHDG 8 in 5 minutes.
- Migration Guide (https://access.redhat.com/documentation/en-us/red_hat_data_grid/8.0/html/data_grid_migration_guide/index)
- Starter Tutorials (https://github.com/redhat-developer/redhat-datagrid-tutorials)
- Supported Components (https://access.redhat.com/articles/4933371)
- Supported Configurations (https://access.redhat.com/articles/4933551)

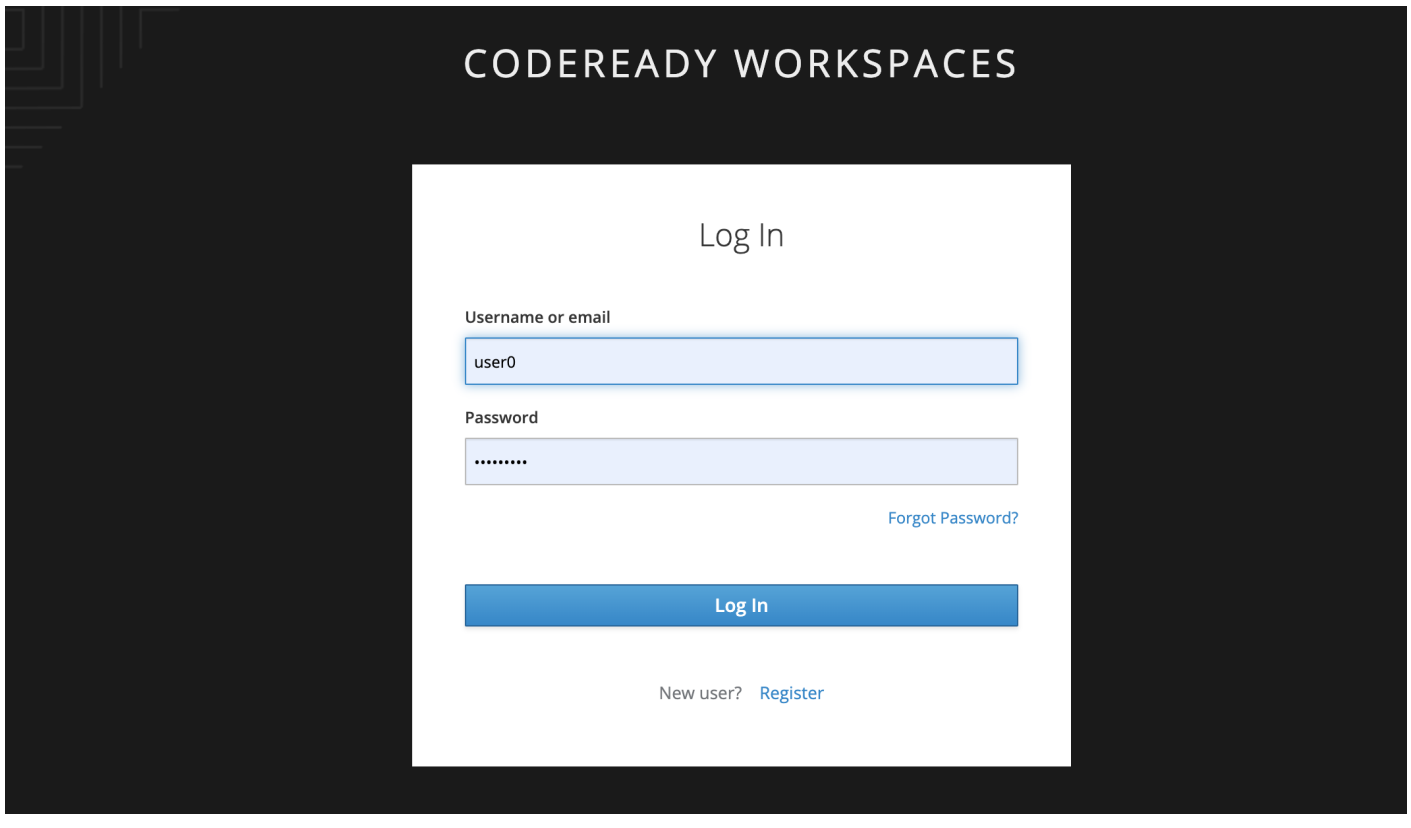
Getting started with Data Grid

Let's delve down into some of the basic features of a Red Hat Data Grid. Our workshop includes a lot more in-depth material going forward, but it's essential we set up a solid base with understanding more about Red Hat Data Grid and Infinispan. In this section, we have prepared six exercises for you. These exercises give a basic introduction to some of the features of the Red Hat Data Grid Java API.

About your workspace environment

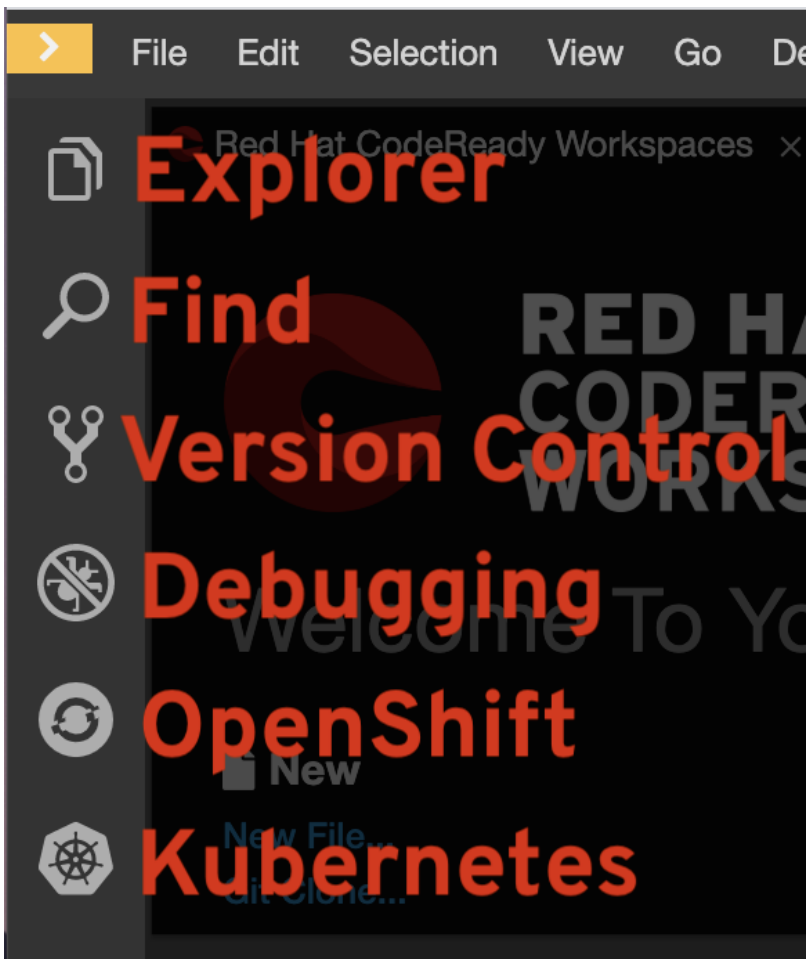
For developments and deployment, we use the using Red Hat CodeReady Workspaces, an online IDE based on Eclipe Che (https://www.eclipse.org/che/). **Changes to files are auto-saved every few seconds**, so you don't need to save changes explicitly.

To get started, access the CodeReady Workspaces instance (https://codeready-codeready.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com) and log in using the username and password you are assigned (e.g. `user10/openshift`):



By logging in to CodeReady, you get access to your development workspace. We have already created a workspace for you. Your development environment opens up by clicking the workspace on the left menu.

You can see icons on the left for navigating between project explorer, search, version control (e.g., Git), debugging, and other plugins. You'll use these during this workshop. Feel free to click on them and see what they do:

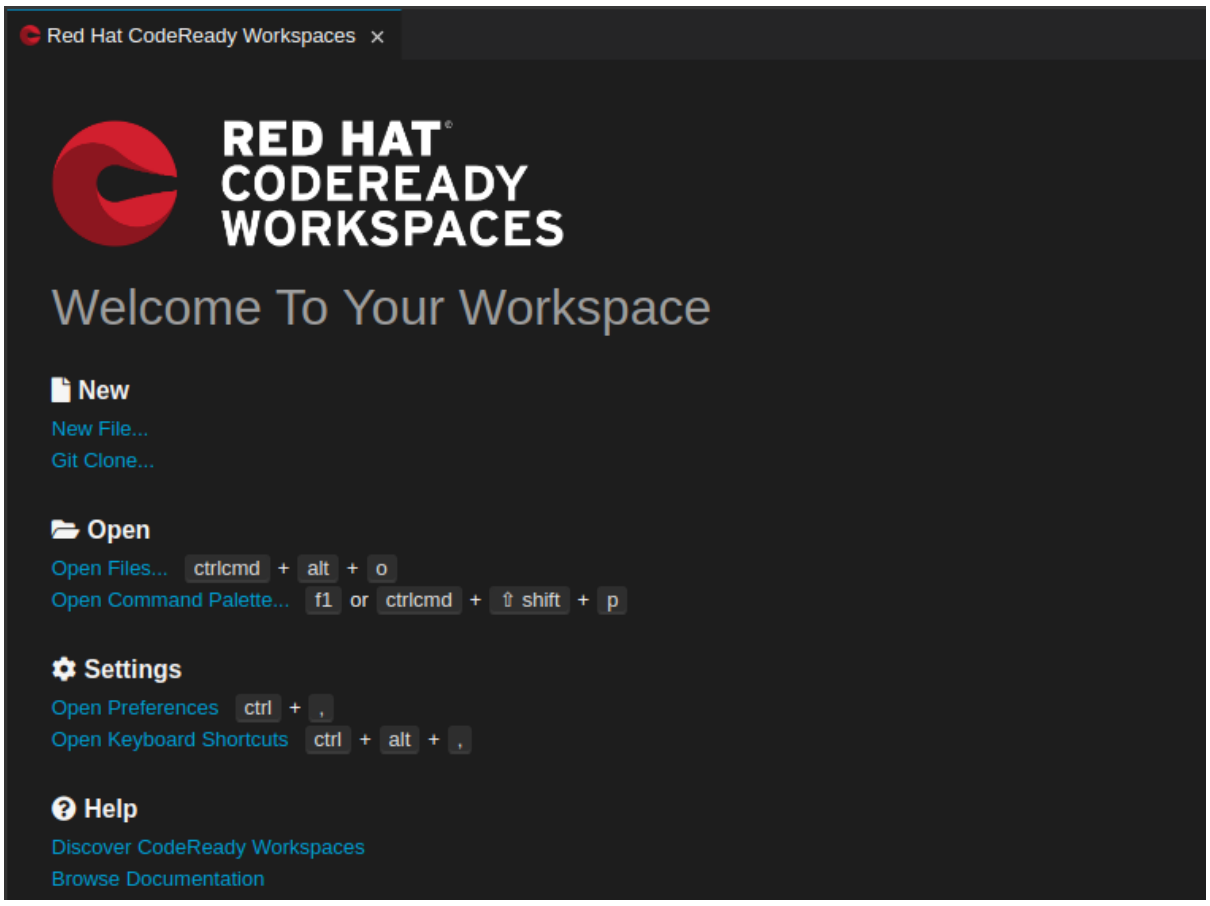


If things get weird or your browser appears, you can simply reload the browser tab to refresh the view.

Many features of CodeReady Workspaces are accessed via **Commands**. You can see a few of the commands listed with links on the home page (e.g., *New File..*, *Git Clone..*, and others).

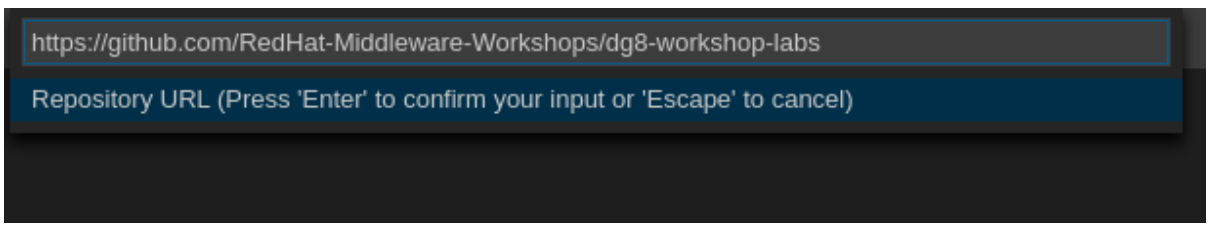
If you ever need to run commands that you don't see in a menu, you can press `F1` to open the command window, or the more traditional `Control` + `SHIFT` + `P` (or `Command` + `SHIFT` + `P` on Mac OS X).

Let's import our first project. Click on **Git Clone..** (or type `F1`, enter 'git' and click on the auto-completed *Git Clone..*)



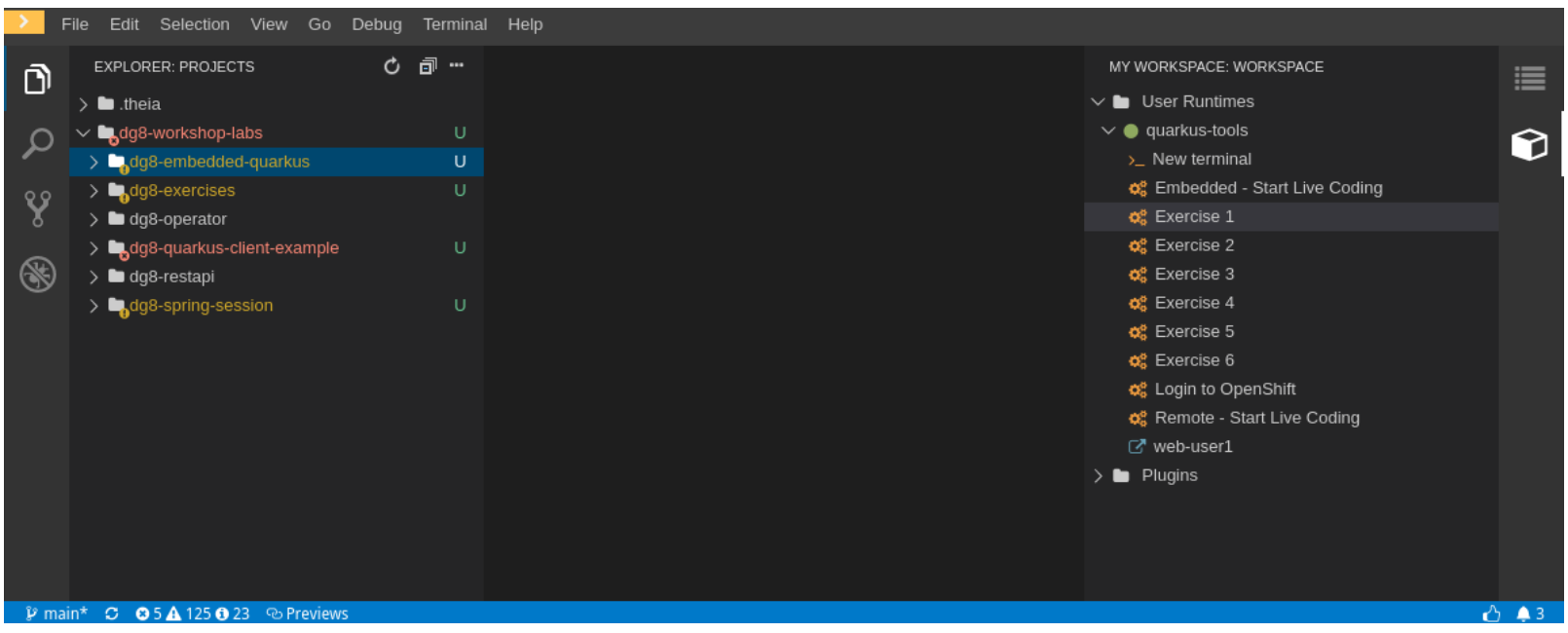
Step through the prompts, using the following value for **Repository URL**. If you use **FireFox**, it may end up pasting extra spaces at the end, so just press backspace after pasting:

```shell
https://github.com/RedHat-Middleware-Workshops/dg8-workshop-labs
```



The project is now imported into your workspace. Following screenshot shows the workspace after the lab projects have been imported.

1. On the left you can see the project explorer with the heading **EXPLORER:PROJECTS**. Project explorer can be used to navigate to source files. Once you click any source file. it will open up in the editor.
2. On the right is the `Workspace Command View` with the heading **MYWORKSPACE:WORKSPACE**. In this view we have created point and click commands. These commands will be used through out the workshop labs.



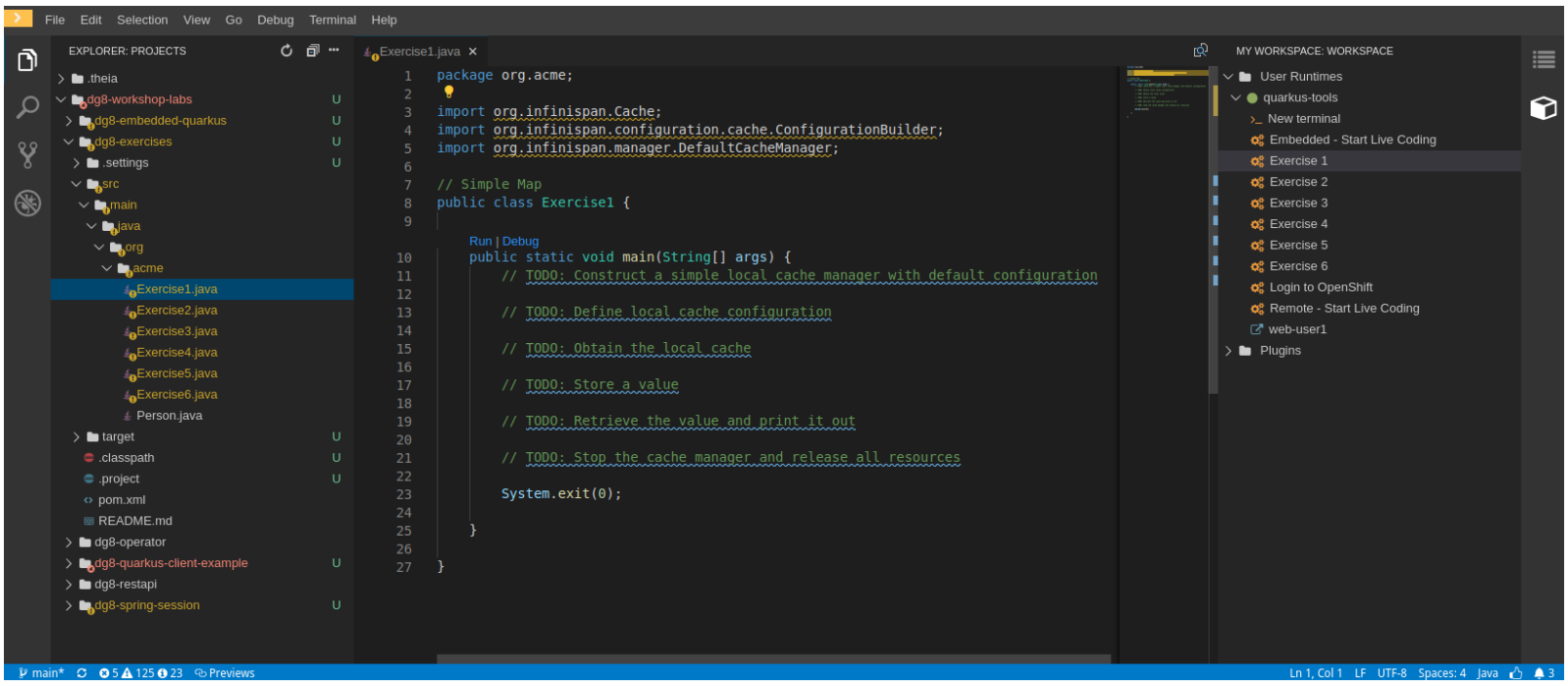
Exercises

This first lab comprises of 6 Exercises. The exercises will give you a glimpse into some of the features of Red Hat Data Grid, and is a good starting point to learn how to use Red Hat Data Grid with your applications.

Summary of the Exercises

1. **Exercise 1** - Creating a Cache
2. **Exercise 2** - JSR-107, JCache
3. **Exercise 3** - Functional API
4. **Exercise 4** - Streaming data from the Cache
5. **Exercise 5** - Using Transactions
6. **Exercise 6** - Queries to the Cache with Lucene

Each Exercise has a corresponding .java file e.g. `Exercise1.java`. The maven project required for this lab is `dg8-exercises`. Following screenshot shows where the Java files are placed. The package name we have used is `org.acme`



All the exercises are marked with a `//TODO:`. Where ever you see this `//TODO:` it signifies that you need to write some code below it. We have added some comments with it, so you know what is required. Incase if the instructions are not understandable, please ask the instructor.

Moreover you will find that preceding exercise sections will explain the `//TODO` in more details and what needs to be done.

Exercise 1: Creating a local Cache

First, a bit about Java *Maps*. Why are Maps good for a cache? Maps are fast, they use methods like `hashCode` and `equals` to determine how to add data to the map. This also means they can be fast enough (O(1)) time to read and write the data. That is exceptional performance, and that's what one would expect from a cache. Data storage is in key and value pairs. There is a lot more to `Map`s, but let's start with a basic cache how-to.

A **CacheManager** is the primary mechanism for retrieving a Cache instance and is often used as a starting point to using the cache. Essentially if you were using a `Map` object you would just create a `Map` and store all your key|value pairs in it. However, when you use a tool like Red Hat Data Grid/Inifinispan, you get more than just a simple map (e.g. Listeners, events, etc), all of which we will talk about in further sections.

CacheManagers are heavyweight objects, and it's not recommended to have more than one **CacheManager** being used per JVM (unless specific configuration requirements require more than one, but either way, this would be a minimal and finite number of instances).

Add the following to your main method in class Exercise1

```java
// TODO: Construct a simple local cache manager with default configuration
DefaultCacheManager cacheManager = new DefaultCacheManager();
```

Now that we have `cacheManager`, we can now define what a Cache should look like. We could choose many features from the system (e.g. if we were adding grouping, streams, listeners, strategies for eviction or clustering, etc) we would do that here. The following example just takes the default configuration.

```java
// TODO: Define local cache configuration
cacheManager.defineConfiguration("local", new ConfigurationBuilder().build());
```

Perfect, so now we have defined our cache, time for us to get that cache from our **CacheManager**. We have also defined that our cache should have both our Key and Value as `Strings`.

```java
// TODO: Obtain the local cache
Cache<String, String> cache = cacheManager.getCache("local");
```

Finally lets put an entry in the Cache. Change the "key" and "value" to anything you'd like.

```java
// TODO: Store a value
cache.put("key", "value");
```

Here we get the value by specifying the key. The key is the same as we used in our previous line's `cache.put`. By specifying a key to the cache, you can get the value stored in it; the same process is also used for an update.

```java
// TODO: Retrieve the value and print it out
System.out.printf("key = %s\n", cache.get("key"));
```

Finally, **CacheManager** is a heavy object; it does a lot, so no need to keep it going on. When done, we close that instance by calling the `stop()` method.

```java
// TODO: Stop the cache manager and release all resources
cacheManager.stop();
```

Great, now we have all we require to run this Exercise. Let's try to run it. You can choose to run it via the Workspace command view by clicking on `Exercise1`. Or you can just open a new terminal from the same view `>_ New Terminal` and run the it manually using maven commands. Both methods would work.

Remember incase of running maven directly via terminal the path to the exercises project is as follows. `/projects/dg8-workshop-labs/dg8-exercises`. Make sure you are in this directory before you run maven commands from the terminal.

```shell
mvn clean compile && \
mvn exec:java -Dexec.mainClass=org.acme.Exercise1
```

The first time a build runs, it may need to download and cache dependencies. Future builds will go much faster!

You should be able to see an output similar to the following.

```shell
Jun 22, 2020 6:40:02 PM org.infinispan.factories.GlobalComponentRegistry preStart
INFO: ISPN000128: Infinispan version: Red Hat Data Grid 'Turia' 10.1.5.Final-redhat-00001
Jun 22, 2020 6:40:03 PM org.infinispan.lock.impl.ClusteredLockModuleLifecycle cacheManagerStarted
INFO: ISPN029009: Configuration is not clustered, clustered locks are disabled
key = value
```

We can see the `key = value` printed from our code.

Exercise 2: JSR-107 JCache

The term cache is generally referred to as a component that stored data in memory so that its easy to read the value that might be hard to calculate or that need to be accessed rather quickly. As discussed earlier, simple `java.util` packages do now have all the capabilities required, and wiring them by oneself is complex if not hard enough. The Java Specification Request (JSR-107) has been created to define temporary caching API for Java. The specification defines some Standard APIs for storing and managing data both for local and distributed use cases.

Let's take a look at how you can use JSR-107 with Red Hat Data Grid/Infinispan. In `Exercise2.java`, add the following code at the designated comment markers:

```java
        // TODO: Construct a simple local cache manager with default configuration
        CachingProvider jcacheProvider = Caching.getCachingProvider();  1
        CacheManager cacheManager = jcacheProvider.getCacheManager();  2
        MutableConfiguration<String, String> configuration = new MutableConfiguration<>();  3
        configuration.setTypes(String.class, String.class);  4

        // TODO: create a cache using the supplied configuration
        Cache<String, String> cache = cacheManager.createCache("myCache", configuration);  5
```

Let's take a more in-depth look at the code above

1. We use a `CachingProvider`, which is part of the standards API
2. The Caching provider, in turn, gives us a `CacheManager`
3. We create a configuration object for our cache (in this case a `MutableConfiguration`)
4. Here we also set the type of keys & values in our Cache (If you remember this is different from our previous exercise since we are using the JSR-107 API now)
5. and finally we get our cache

Finally lets put an entry in the Cache. Change the "key" and "value" to anything you'd like.

```java
        // Store and retrieve value
        cache.put("key", "value");
        System.out.printf("key = %s\n", cache.get("key"));
```

And then lets close our `CacheManager`.

```java
        // TODO: Stop the cache manager and release all resources
        cacheManager.close();
```

Run the above exercise as follows in the CodeReady terminal, or you can also choose to execute the command `Exercise2` in your MyWorkspace Menu on the right.

```shell
mvn clean compile && \
mvn exec:java -Dexec.mainClass=org.acme.Exercise2
```

You should be able to see an output similar to the following. On the last line you can see your key, value printed.

```shell
Jun 22, 2020 6:54:25 PM org.infinispan.factories.GlobalComponentRegistry preStart
INFO: ISPN000128: Infinispan version: Red Hat Data Grid 'Turia' 10.1.5.Final-redhat-00001
Jun 22, 2020 6:54:25 PM org.infinispan.lock.impl.ClusteredLockModuleLifecycle cacheManagerStarted
INFO: ISPN029009: Configuration is not clustered, clustered locks are disabled
key = value
```

Exercise 3: Functional API

The approach taken by the Functional Map API when working with multiple keys is to provide a lazy, pull-style API. All multi-key operations take a collection parameter which indicates the keys to work with (and sometimes contains 'value' information too), and a function to execute for each key/value pair. Each function's ability depends on the entry view received as a function parameter, which changes depending on the underlying map: `ReadEntryView` for `ReadOnlyMap`, `WriteEntryView` for `WriteOnlyMap`, or `ReadWriteView` for `ReadWriteMap`. The return type for all multi-key operations, except the ones from `WriteOnlyMap`, return an instance of `Traversable`, which exposes methods for working with the returned data from each function execution. Let's see an example:

- This example demonstrates some of the key aspects of working with multiple entries using the Functional Map API
- All data-handling methods (including multi-key methods) for `WriteOnlyMap` return `CompletableFuture<Void>`, because there's nothing the function can provide that could not be computed in advance or outside the function.

There is a particular type of multi-key operations which work on all keys/entries stored in Infinispan. The behavior is very similar to the multi-key operations shown above, with the exception that they do not take a collection of keys (or values) as parameters.

There are a few interesting things to note about working with all entries using the Functional Map API:

- When working with all entries, the order of the `Traversable` is not guaranteed
- Read-only's `keys()` and `entries()` offer the possibility to traverse all keys and entries present in the cache — When traversing entries, both keys and values, including metadata, are available --- Contrary to Java's `ConcurrentMap`, there's no possibility to navigate only the values (and metadata) since there's little to be gained from such a method --- Once a key's entry has been retrieved, there's no extra cost to provide the key as well.

Let us start by initializing our cache with the `DefaultCacheManager` as we have done so in the previous labs. However, we use the functional API, and hence after getting the cache, our `Map` implementation is different.

How to use the Functional API?

Using an asynchronous API, all methods that return a single result return a `CompletableFuture` which wraps the result. To avoid blocking, it offers the possibility to receive callbacks when the `CompletableFuture` has completed, or it can be chained or composed with other `CompletableFuture` instances. You do not need to write the following snippet, it should already be there. Let's get started with Exercise3.java.

Please remove the following lines in the main method.

```
/* UNCOMMENT When starting this exercise
UNCOMMENT When starting this exercise */
```

JAVA
```
DefaultCacheManager cacheManager = new DefaultCacheManager();
cacheManager.defineConfiguration("local", new ConfigurationBuilder().build());
AdvancedCache<String, String> cache = cacheManager.<String, String>getCache("local").getAdvancedCache();
FunctionalMapImpl<String, String> functionalMap = FunctionalMapImpl.create(cache);
FunctionalMap.WriteOnlyMap<String, String> writeOnlyMap = WriteOnlyMapImpl.create(functionalMap); 1
FunctionalMap.ReadOnlyMap<String, String> readOnlyMap = ReadOnlyMapImpl.create(functionalMap);
```

Next, what you would want to do is asynchronously write to this cache. Copy and paste the following snippet to Exercise3.java

JAVA
```
// TODO Execute two parallel write-only operation to store key/value pairs
CompletableFuture<Void> writeFuture1 = writeOnlyMap.eval("key1", "value1",
        (v, writeView) -> writeView.set(v));  1
CompletableFuture<Void> writeFuture2 = writeOnlyMap.eval("key2", "value2",
        (v, writeView) -> writeView.set(v));
```

1 Write-only operations require locks to be acquired. They do not require reading previous value or metadata parameter information associated with the cached entry, which sometimes can be expensive since they involve talking to a remote node in the cluster or the persistence layer. Exposing write-only operations makes it easy to take advantage of this vital optimization.

And now lets do a read operation in similar

JAVA
```
//TODO When each write-only operation completes, execute a read-only operation to retrieve the value
CompletableFuture<String> readFuture1 =
        writeFuture1.thenCompose(r -> readOnlyMap.eval("key1", EntryView.ReadEntryView::get));  1
CompletableFuture<String> readFuture2 =
        writeFuture2.thenCompose(r -> readOnlyMap.eval("key2", EntryView.ReadEntryView::get));
```

1 Exposes read-only operations that can be executed against the functional map. The information that can be read per entry in the functional map. Read-only operations have the advantage that no locks are acquired for the duration of the operation.

Finally, let's print the operation as it completes.

```java
        //TODO When the read-only operation completes, print it out
        System.out.printf("Created entries: %n");
        CompletableFuture<Void> end = readFuture1.thenAcceptBoth(readFuture2, (v1, v2) -> System.out.printf("key1 = %s%nkey2 = %s%n", v1, v2));

        // Wait for this read/write combination to finish
        end.get();
```

So we have seen how a `WriteOnly` and `ReadOnly Map` works, let's also add the `ReadWriteMap`. Read-write operations offer the possibility of writing values or metadata parameters and returning previously stored information. Read-write operations are also crucial for implementing conditional, compare-and-swap (CAS) operations. Locks need to be acquired before executing the read-write lambda.

```java
        // Use read-write multi-key based operation to write new values
        // together with lifespan and return previous values
        Map<String, String> data = new HashMap<>();
        data.put("key1", "newValue1");
        data.put("key2", "newValue2");
        Traversable<String> previousValues = readWriteMap.evalMany(data, (v, readWriteView) -> {
            String prev = readWriteView.find().orElse(null);
            readWriteView.set(v, new MetaLifespan(Duration.ofHours(1).toMillis()));
            return prev;
        });
```

Now let's run our code and see how it works.

Run the above exercise as follows in the CodeReady terminal, or you can also choose to execute the command `Exercise3` in your MyWorkspace Menu on the right

```shell
mvn clean compile && \
mvn exec:java -Dexec.mainClass=org.acme.Exercise3
```

You should be able to see an output similar to the following. On the last line you can see your key, value printed.

```shell
Jun 22, 2020 6:59:09 PM org.infinispan.factories.GlobalComponentRegistry preStart
INFO: ISPN000128: Infinispan version: Red Hat Data Grid 'Turia' 10.1.5.Final-redhat-00001
Jun 22, 2020 6:59:09 PM org.infinispan.lock.impl.ClusteredLockModuleLifecycle cacheManagerStarted
INFO: ISPN029009: Configuration is not clustered, clustered locks are disabled
Created entries:
key1 = value1
key2 = value2
Updated entries:
ReadOnlySnapshotView{key=key1, value=newValue1, metadata=MetaParamsInternalMetadata{params=MetaParams{length=1, metas=[MetaLifespan=3600000]}}}
ReadOnlySnapshotView{key=key2, value=newValue2, metadata=MetaParamsInternalMetadata{params=MetaParams{length=1, metas=[MetaLifespan=3600000]}}}
Previous entry values:
value1
value2
```

Exercise 4: Streaming data from the cache

Infinispan Distributed Java Streams can be used to calculate analytics over existing data. Through the overloading of methods, Infinispan can offer a simple way of passing lambdas that are `Serializable` without the need for explicit casting. Being able to produce binary formats for the lambdas is an essential step for java streams executions to be distributed.

Please remove the following lines in the main method.

```
/* UNCOMMENT When starting this exercise
UNCOMMENT When starting this exercise */
```

With the following, we create a lambda to write data into our cache

```java
        // TODO: Store some values
        int range = 10;
        IntStream.range(0, range)
                .boxed()
                .forEach(i -> cache.put(i + "-key", i + "-value"));
```

And now we read that data summing up the values.

```java
        // TODO: Map and reduce the keys
        int result = cache.keySet().stream()
                .map(e -> Integer.valueOf(e.substring(0, e.indexOf("-"))))
                .collect(() -> Collectors.summingInt(Integer::intValue));
```

Now let's run our code and see how it works.

Run the above exercise as follows in the CodeReady terminal, or you can also choose to execute the command `Exercise4` in your MyWorkspace Menu on the right

```shell
mvn clean compile && \
mvn exec:java -Dexec.mainClass=org.acme.Exercise4
```

You should be able to see an output similar to the following. On the last line, you can see your key, the value printed.

```shell
Jun 22, 2020 7:00:04 PM org.infinispan.factories.GlobalComponentRegistry preStart
INFO: ISPN000128: Infinispan version: Red Hat Data Grid 'Turia' 10.1.5.Final-redhat-00001
Jun 22, 2020 7:00:05 PM org.infinispan.lock.impl.ClusteredLockModuleLifecycle cacheManagerStarted
INFO: ISPN029009: Configuration is not clustered, clustered locks are disabled
Result = 45
```

Exercise 5: Using Transactions

Transactions are essential in any business application. Usually, the transaction is used with the dataset, and quite often related to a database. Still, thats not exactly right, if you have a distributed dataset, one needs transactions for business logic to prevail. Infinspan provides transactions. There can be a scenario in which the cluster adds a node or entry has been written by another node. The Infinispan transaction manager is aware of such events and handles them. You can read more about the design of transactions here: https://github.com/infinispan/infinispan-designs

Please remove the following lines in the main method.

```
/* UNCOMMENT When starting this exercise
UNCOMMENT When starting this exercise */
```

Lets get the TransactionManager from the cache

```java
        //TODO Obtain the transaction manager
        TransactionManager transactionManager = cache.getAdvancedCache().getTransactionManager();
```

We begin our transaction, write two entries, and then close it.

```java
        // TODO Perform some operations within a transaction and commit it
        transactionManager.begin();
        cache.put("key1", "value1");
        cache.put("key2", "value2");
        transactionManager.commit();
```

Let’s also do a rollback scenario. So we write to entries and rollback.

```java
        //TODO Perform some operations within a transaction and roll it back
        transactionManager.begin();
        cache.put("key1", "value3");
        cache.put("key2", "value4");
        transactionManager.rollback();
```

Now let’s run our code and see how it works.

Run the above exercise as follows in the CodeReady terminal, or you can also choose to execute the command `Exercise5` in your MyWorkspace Menu on the right

```shell
mvn clean compile && \
mvn exec:java -Dexec.mainClass=org.acme.Exercise5
```

You should be able to see an output similar to the following. On the last line you can see your key, value printed.

```shell
Jun 22, 2020 7:01:31 PM org.infinispan.factories.GlobalComponentRegistry preStart
INFO: ISPN000128: Infinispan version: Red Hat Data Grid 'Turia' 10.1.5.Final-redhat-00001
Jun 22, 2020 7:01:31 PM org.infinispan.lock.impl.ClusteredLockModuleLifecycle cacheManagerStarted
INFO: ISPN029009: Configuration is not clustered, clustered locks are disabled
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Jun 22, 2020 7:01:31 PM org.infinispan.transaction.lookup.GenericTransactionManagerLookup useDummyTM
INFO: ISPN000104: Using EmbeddedTransactionManager
key1 = value1
key2 = value2
key1 = value1
key2 = value2
```

So as you can see, even though we wrote the new values, but by rolling back, they do not exist anymore.

Exercise 6: Queries to the Cache with Lucene

Infinispan includes a highly scalable distributed Apache Lucene Directory implementation.

This directory closely mimicks the same semantics of the traditional filesystem and RAM-based directories, being able to work as a drop-in replacement for existing applications using Lucene and providing reliable index sharing and other features of Infinispan like node auto-discovery, automatic failover, and rebalancing, optionally transactions, and can be backed by traditional storage solutions as filesystem, databases or cloud store engines.

The implementation extends Lucene's `org.apache.lucene.store.Directory` so it can be used to store the index in a cluster-wide shared memory, making it easy to distribute the index. Compared to rsync-based replication, this solution is suited for use cases in which your application makes frequent changes to the index, and you need them to be quickly distributed to all nodes. Consistency levels, synchronicity, and guarantees, total elasticity, and auto-discovery are all configurable; also, changes applied to the index can optionally participate in a JTA transaction, optionally supporting XA transactions with recovery.

Please remove the following lines in the main method.

```
/* UNCOMMENT When starting this exercise
UNCOMMENT When starting this exercise */
```

Since Lucene is part of Infinispan, We need to make sure that we have the right configuration for it.

JAVA
```
// Create cache config
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.indexing().index(Index.ALL)  1
        .addProperty("default.directory_provider", "ram")  2
        .addProperty("lucene_version", "LUCENE_CURRENT");  3

// Obtain the cache
Cache<String, Person> cache = cacheManager.administration()
        .withFlags(CacheContainerAdmin.AdminFlag.VOLATILE)
        .getOrCreateCache("cache", builder.build());
```

1 Here we are telling our Cache config that we want to index all entries

2 The storage for Lucene is in the memory

3 and we want to give it a version

Now, let's add a bit of more code to the above example. In the following code, we get the QueryFactory and create a query.

JAVA
```
// TODO: Obtain a query factory for the cache
QueryFactory queryFactory = Search.getQueryFactory(cache);
// Construct a query
Query query = queryFactory
    .from(Person.class)
    .having("name")
    .eq("William")
    .toBuilder()
    .build();
// Execute the query
List<Person> matches = query.list();
```

Now let's run our code and see how it works.

Run the above exercise as follows in the CodeReady terminal, or you can also choose to execute the command `Exercise6` in your MyWorkspace Menu on the right

SHELL
```
mvn clean compile && \
mvn exec:java -Dexec.mainClass=org.acme.Exercise6
```

You should be able to see an output similar to the following. On the last line you can see your key, value printed.

SHELL
```
Jun 22, 2020 7:03:18 PM org.hibernate.search.engine.Version <clinit>
INFO: HSEARCH000034: Hibernate Search 5.10.7.Final-redhat-00001
Jun 22, 2020 7:03:18 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.5.Final}
Jun 22, 2020 7:03:18 PM org.infinispan.query.impl.LifecycleManager createQueryInterceptorIfNeeded
INFO: ISPN014003: Registering Query interceptor for cache cache
Match: Person [name=William, surname=Wordsworth]Match: Person [name=William, surname=Shakespeare]
```

It's quite simple to add Lucene based search to your cache. Try to change the parameters a bit and experience this more.

Congratulations!

You have completed the first introductory exercises to Red Hat Data Grid 8.0.

1. **Exercise 1** - Creating a Cache
2. **Exercise 2** - JSR-107, JCache
3. **Exercise 3** - Functional API
4. **Exercise 4** - Streaming data from the Cache
5. **Exercise 5** - Using Transactions

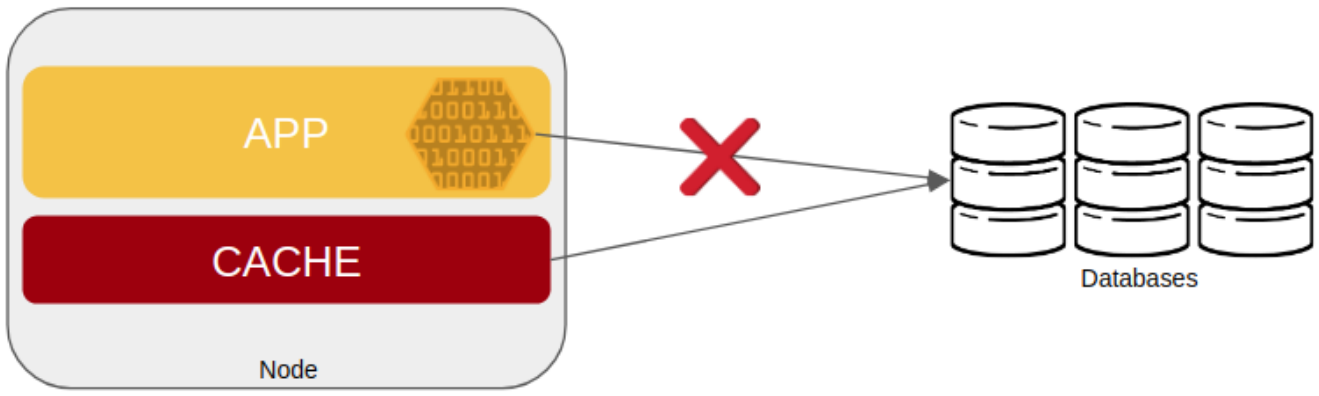
6. **Exercise 6** - Queries to the Cache with Lucene

You should now be able to create caches, stream data and so much more. Let's move on to the next section and experience more in-depth examples.

Caching 101, Lets get started with your first app

What could be cases where you want to use Cache. Let's take a moment and think about it. Where do you think you could use cache? Well there could be limitless answers to this question; Some common usecases are listed below

- Lookup Data; if you have an app, that needs some user profile data etc, thats unecessary to poll everytime, and it doesnt change much
- Latency or bulk; you might have a service or database that takes alot of time to load some of the data.
- Traffic; you might have loads of users and usage trends are spiking
- Session Storage; Storing your webapp sessions, this could be carts etc, that you can use to scale your application
- Global Counters; You might want to create distributed keys accross a distributed dataset. Use this to update and fetch data.



The above figure depicts a common scenario when you dont want to use a networked dataset that could be slow and with high latency, but rather use an inmemory cache. In this case the in-memory cache becomes the front for the data.

This lab is about an embedded cache in local or clustered mode, also a common usecase for applications. What is an Embedded Cache in Red Hat Data Grid?

Embedded Cache, Some basics

The CacheManager is Red Hat Data Grid's main entry point. You can use a CacheManager to

- configure and obtain caches
- manage and monitor your nodes
- execute code across a cluster

Depending on whether you are embedding Red Hat Data Grid in your application or you are using it remotely, you will be dealing with either an EmbeddedCacheManager or a RemoteCacheManager. While they share some methods and properties, be aware that there are semantic differences between them.

CacheManagers are heavyweight objects, and recommended use would be one CacheManager used per JVM (unless specific setups require more than one; but either way, this should be a minimal and finite number of instances).

The simplest way to create a CacheManager is:

```java
EmbeddedCacheManager manager = new DefaultCacheManager();
```

which starts the most basic, local mode, non-clustered cache manager with no caches. CacheManagers have a lifecycle and the default constructors also call start(). Overloaded versions of the constructors are available, that do not start the CacheManager, although keep in mind that CacheManagers need to be started before they can be used to create Cache instances.

Once constructed, CacheManagers should be made available to any component that require to interact with it via some form of application-wide scope such as JNDI, a ServletContext or via some other mechanism such as an IoC container.

When you are done with a CacheManager, you must stop it so that it can release its resources:

```java
manager.stop();
```

This will ensure all caches within its scope are properly stopped, thread pools are shutdown. If the CacheManager was clustered it will also leave the cluster gracefully.

Your first service with Caching

One of the common usecases for a Cache is to keep most used data in memory. Example having a Scoreboard in the cache. Let's assume theres a webpage that keeps the Score card for a round played by players on different tours. Now since this website expects people coming to check the top scores for example, or maybe based on a country etc. The best approach would be store this information in a cache rather then polling that information from different webservices or different databases as an example.

In our first service we will do exactly that. We will store this data in our Embedded Cache and understand not only how this works but the different ways of handling cache, getting events from it etc.

Project details

You can choose multiple runtimes to impelement this service, in our case today the example we have taken is with Quarkus. Quarkus is a Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards. Quarkus is also known for some of its cool features e.g Live Coding which we will also use in our labs. It makes it easier to code and see our changes as we do that.

Navigate to the project `dg8-embedded-quarkus` This is a template project, and you will be writing code into this project. As you can see there is already some files inplace. Let's take a look into what these files are and do.

The Maven dependencies

Open the pom.xml file in the project.

We will be using the following dependencies to create our service

```xml
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy</artifactId> 1
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-jsonb</artifactId> 2
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-infinispan-embedded</artifactId> 3
</dependency>
<dependency>
```

1. Quarkus-resteasy; for our REST endpoint
2. Quarkus-resteasy-jsonb; we will use this for Json serialization for our REST endpoint
3. Quarkus-infinispan-embedded; This extension will enable us to embed our cache in our service.

The Score Entity

We have also created a POJO called Score, which will serve as our datastructure for the ScoreCard. If you have played golf, you might wonder this is a very basic data structure and that's entirely true, we could have gone in more details but we have kept this short to cover all the features. And you are welcome to extending this datastructure after successfully finishing these labs.

If you open `Score.java` you will see the following first few lines

```java
// The number of holes played per round
public static final int HOLES = 18;

// The players is on this hole
private int currentHole = 0;

// Name of the player
private String playerName;

// players unique Id
private String playerId;

// The actual scoreCard
private int[] card = new int[HOLES];

// The course player is playing on.
private String course = "St.Andrews Links";

// the courseCard; the expected handicap
private int[] courseCard = {4,4,4,4,5,4,4,3,4,4,3,4,4,5,4,4,4,4};
```

The rest of the methods are accessors for these fields. Important to mention we do have three constructors

```java
// Used in Json serialization
public Score()

// Creating a new player with course and the courses score card
public Score(String playerName, String playerId, String course, int[] courseCard)

// Creating a new player with defaults
public Score(String playerName, String playerId)
```

Take a look at some of the other methods in the Score class and make yourself familiar with it. Do not change the class at this time.

Creating a service for caching

So now that you are familiar with the project template, Let's start by creating a service. Todo this open `ScoreService.java` .

Define the following three class level variables

```java
Cache<Object, Score> scoreCache; 1

Logger log = LoggerFactory.getLogger(ScoreService.class); 2

EmbeddedCacheManager cacheManager; 3
```

1 the `scoreCache` is an instance of `Cache`, which will be our point to store and retrieve values. Cache expects `<K,V>` types, in our case our key is an Object and our actual entry is a `Score`. Yes the same Score POJO we showed earlier. The Cache is also the central interface of Red Hat Data Grid. A Cache provides a highly concurrent, optionally distributed data structure with additional features such as: JTA transaction compatibility, Eviction support for evicting entries from memory to prevent `OutOfMemoryErrors`, Persisting entries to a `CacheLoader`, either when they are evicted as an overflow, or all the time, to maintain persistent copies that would withstand server failure or restarts. For convenience, `Cache` extends `ConcurrentMap` and implements all methods accordingly. Methods like `keySet()`, `values()` and `entrySet()` produce backing collections in that updates done to them also update the original Cache instance. Certain methods on these maps can be expensive however (prohibitively so when using a distributed cache). The `size()` and `Map.containsValue(Object)` methods upon invocation can also be expensive as well. The reason these methods are expensive are that they take into account entries stored in a configured CacheLoader and remote entries when using a distributed cache.

2 the `log`; straight forward logger incase we want to log something.

3 `cacheManager`; which is an instance of `EmbeddedCacheManager`, we inject this into our code using the dependency injection and this is possible due to the extension we added in our maven dependencies.

Next let's create some accessor methods for our service.

```java
public List<Score> getAll() { 1
    return new ArrayList<>(scoreCache.values());
}

public void save(Score entry) { 2
    scoreCache.put(getKey(entry), entry);
}

public void delete(Score entry) { 3
    scoreCache.remove(getKey(entry));
}

public void getEntry(Score entry){ 4
    scoreCache.get(getKey(entry));
}
```

1 We get all values from the cache and return them as a List of Scores

2 We are saving the entire entry, which we expect as a Score object.

3 We are deleting an entry from our cache

4 Finally we want to get 1 entry from our cache.

These are simple accessor methods, one thing you might have noticed is the use of the method `getKey`. This method described as follows has one simple task i.e. to get us the key, which in our case we use as a concatenated string of playerId+course. Since entry always has both of these values we concatenate them here.

Add the following methods to your class as well.

```java
public static String getKey(Score entry){ 1
    return entry.getPlayerId()+","+entry.getCourse();
}

public Score findById(String key) { 2
    return scoreCache.get(key);
}
```

1 to get the key, so we have the right combination when we get an entry request to our cache

2 find the entry in our cache incase we get a getOne request from the resource

Perfect! Almost to our final step for this service. What we are missing is initialization of our CacheManager and then we need to ask the CacheManager to give us a new cache.

The CacheManager has many purposes: - acts as a container for caches and controls their lifecycle - manages global configuration and common data structures and resources (e.g. thread pools) - manages clustering

A CacheManager is a fairly heavy-weight component, and you will probably want to initialize it early on in your application lifecycle. For that reason we use the onStart method in this Service to ensure that the CacheManager and Cache are both created at startup. This also benefits us when we change this to clustering mode, more on that in our next lab.

```java
void onStart(@Observes @Priority(value = 1) StartupEvent ev){
    cacheManager = new DefaultCacheManager();  1
    ConfigurationBuilder config = new ConfigurationBuilder();  2

    cacheManager.defineConfiguration("scoreboard", config.build());  3
    scoreCache = cacheManager.getCache("scoreboard");  4

    log.info("Cache initialized");

}
```

1 Constructing a CacheManager is done via one of its constructors, which optionally take in a Configuration or a path or URL to a configuration XML file. In our current config we do not need to add much, but use the defaults

2 We use defaults for the Configuration builder. its a very handy Object that enables us to define different cache configurations which we will notice further on in this lab.

3 We are passing our configuration to the CacheManager.

4 You obtain Cache instances from the CacheManager by using one of the overloaded getCache(), methods. Note that with getCache(), there is no guarantee that the instance you get is brand-new and empty, since caches are named and shared. Because of this, the CacheManager also acts as a repository of Caches, and is an effective mechanism of looking up or creating Caches on demand. In our case we expect this to be the first Cache and local embedded one. This is also not clustered.

You might have noticed, that a CacheManager can have multiple Caches; which is great, since in any application you could store multiple unrelated data in different caches, not just that you might even want to have different behaviour with different Caches, e.g. Eviction or Expiration could differ etc. This gives us a lot more to work with then we would in a ConcurrentHashMap as an example.

Creating a REST Resource for our app

Let's create our REST resource. This should be simple. Open the `ScoreResource.java` file. Since we already implemented most of our code in the service, we need to make sure we can respond on the correct REST calls.

First Let's inject our ScoreService so we can use all the caching functions we need. Copy the following codes in `// Inject ScoreService`:

```java
@Inject
ScoreService scoreService;
```

Let's implement the create end point, here we are simply calling the save function on the scoreService. Copy the following codes in `// Implement the create end point`:

```java
@POST
public Response create(@Valid Score item) {
    scoreService.save(item);
    return Response.status(Status.CREATED).entity(item).build();
}
```

And we also want to be able to get one entry from our cache. following method will do that by calling the scoreService.findById. Copy the following codes in `// To be able to get one entry`:

```java
@GET
@Path("/{id}")
public Object getOne(@PathParam("id") String id) {
    Object entity = scoreService.findById(id);
    if (entity == null) {
        throw new WebApplicationException("ScoreCard with id of " + id + " does not exist.", Status.NOT_FOUND);
    }
    return entity;
}

@GET
public List<Score> getAll() {
    return scoreService.getAll();
}
```

And incase we wanted to update an entry. that would normally the case when we the player is playing the round. so the score will be updated. Copy the following codes in `// To update an entry`:

```java
@PATCH
@Path("/{id}")
public Response update(@Valid Score card, @PathParam("id") Long id) {
    scoreService.save(card);
    return Response.status(Status.CREATED).entity(card).build();

}
```

Take a look into some of the other methods in the ScoreResource to make your self familiar with the code there.

If you might have noticed at the class declaration we are using the following annotations

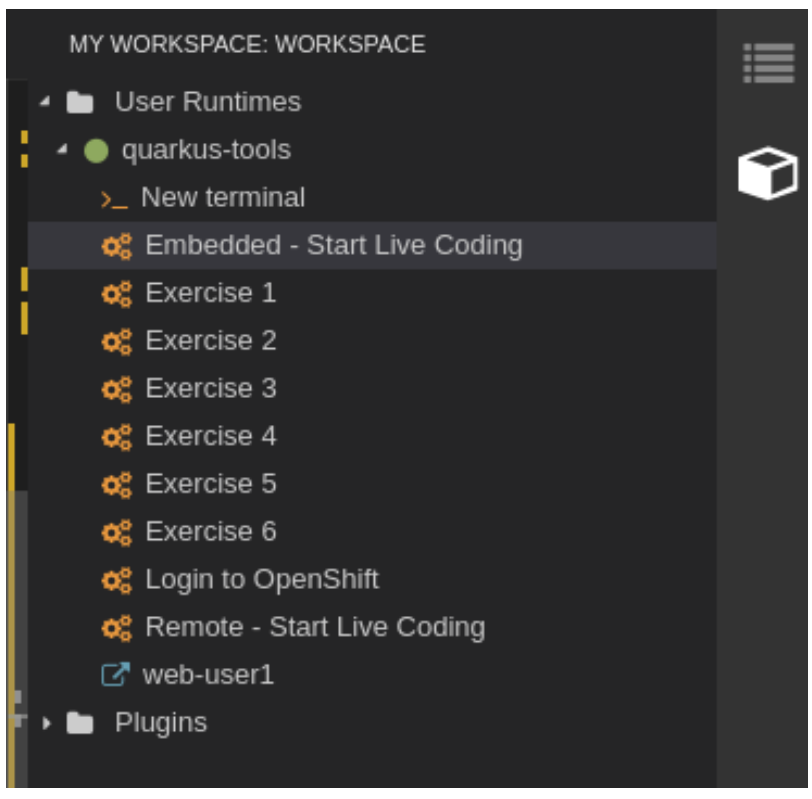
```java
@Produces(MediaType.APPLICATION_JSON) 1
@Consumes(MediaType.APPLICATION_JSON) 2
@Path("/api") 3
```

1. This means we are producing JSON from our responses
2. This means we only listen to JSON, this helps us to consume the JSON directly and serialize it into our Score POJO as an example.
3. And `api` is the path to our resource. e.g. localhost:8080/api

It is suggested that at this moment you close all terminal windows that you might have opened in the previous labs. to keep a clear view of our lab

Run the Service

A quick look at our side bar menu on the right called `MyWorkspace`



We will use this menu through out the labs. There is a bunch of commands created specifically for this workshop.

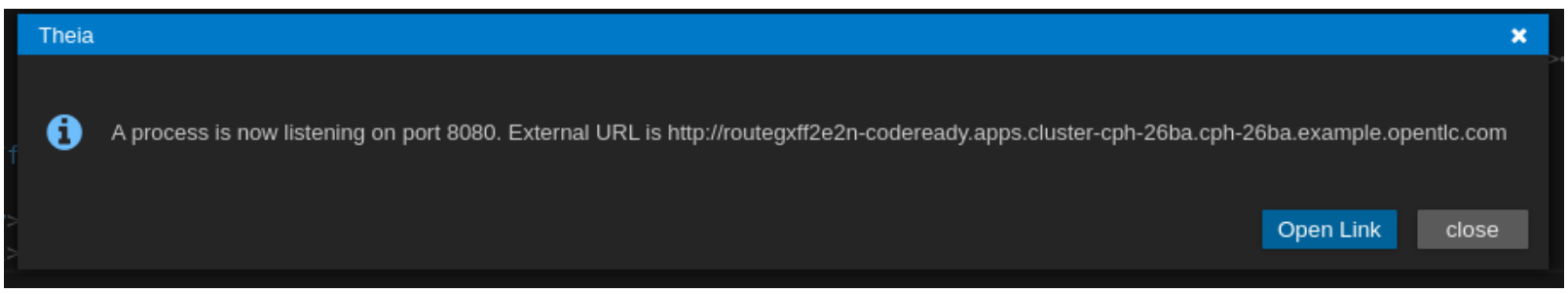
First Let's login to Openshift. You will find the button in the right corner in MyWorkspace menu. Click `Login to Openshift` and login with:

- **Username**: `user10`
- **Password**: `openshift`

After you log in using **Login to OpenShift**, the terminal is no longer usable as a regular terminal. You can close the terminal window. You will still be logged in when you open more terminals later!

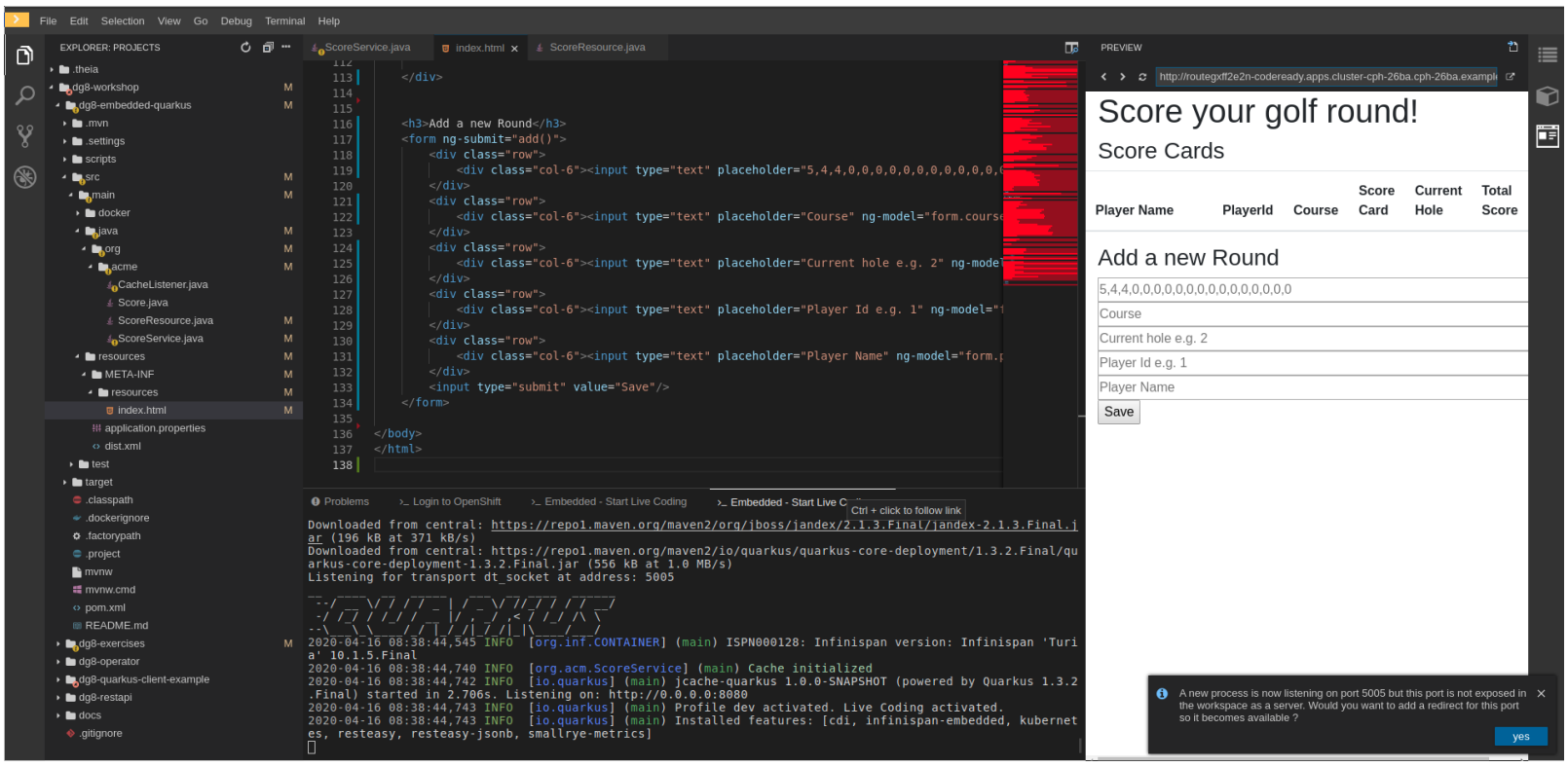
Let's run our project click on the Command `Emebdded - Start Live Coding` This will enable Live coding, it will open up a small terminal to build your artifact and then open up a browser view

Make sure you click on the Open link for port `8080` (dismiss the one for `5005` - that's for the debugger):



It takes a few moments to establish the ingress route, so you may need to click the `Reload` button if you see `Application not responding`.

You can also click on the link icon in the browser view, which will open a browser tab.



Now run the following bash script in a **new** terminal(assuming live coding is still on):

```sh
sh $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-embedded-quarkus/scripts/load.sh
```

Reload **PREVIEW** again in CodeReady workspaces and you will see some scores updated. Now these scores were posted directly:

1. Via our ScoreResource
2. Into our ScoreService
3. And passed into the cache

We just created a bunch of POST requests, to create a bunch of scores. The way the algorithm is working is that, we assume the score card is updated after every hole. or at the end. so if you place the data:

- card: `5,4,4,4,3,5,0,0,0,0,0,0,0,0,0,0,0,0`
- course: `Bethpage`
- currentHole: `6`
- playerId: `2`
- playerName: `James`

In this case, we are still doing a POST request like before, however the cache is checking whats coming in the put request, it will create the combination key like we have in our Score service i.e. PlayerId+Course and put the new data in it. This means that again its just one call to make this update, no lookups were needed specifically to perform before updating.

Try this data entry again, and this time change the course to `Firestone`, and you will notice that there will be a new entry for James. So now James will have two rounds on the scoreboard.

> Its important to know what our key is and its important to find the right combination of what kind of key our data should posses when it comes to a Cache.

Let's do that, enter this data in the form in your browser view and press save, it will updated James's round score.

If you goto your endpoint/api which should be route of your app/api in the browser you will also see the same JSON data there as well.

So what we have successfully done so far. Read, Write and update our Cache.

Let's move on to the next step and do some more interesing additions to our project.

> Since we are using the Live Coding mode here, at any time if you terminate or restart the session it will clear the cache.

Expiration of Entries

Let's assume you are pulling this data off from a database. You might want that it should be removed from the cache after a certain time period. You can do this by defining this either on the a single entry or the entire cache. By default entries created are immortal and do not have a lifespan or maximum idle time. Using the cache API, mortal entries can be created with lifespans and/or maximum idle times

Expiration is a top-level construct, represented in the configuration as well as in the cache API. - While eviction is local to each cache instance , expiration is cluster-wide . Expiration lifespan and maxIdle values are replicated along with the cache entry. - Maximum idle times for cache entries require additional network messages in clustered environments. For this reason, setting maxIdle in clustered caches can result in slower operation times. - Expiration lifespan and maxIdle are also persisted in CacheStores, so this information survives eviction/passivation.

Let's start with doing this for one entry.

In Infinispan entry expiration can happen in two ways:

- a certain time after the data was inserted into the cache (i.e. lifespan)
- a certain time after the data was last accessed (i.e. maximum idle time)

The Cache interface offers overloaded versions of the put() method that allow specifying either or both expiration properties. The following example shows how to insert an entry which will expire after 5 seconds

Open the **ScoreService.java** and change the `save` method to the following.

```java
    public void save(Score entry) {
        scoreCache.put(getKey(entry), entry, 5, TimeUnit.SECONDS);
    }
```

In the above code, we have used TimeUnit and we specify 5 as the unit which is seconds. Following are the units you can use in the TimeUnit:

- NANOSECONDS
- MICROSECONDS
- MILLISECONDS
- SECONDS
- MINUTES
- HOURS
- DAYS

Okay now its time to test this change. Go back into the terminal and run `load.sh`.

```shell
sh $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-embedded-quarkus/scripts/load.sh
```

Reload **PREVIEW** right away. and you will see the entries again. Now wait for 5 seconds and reload again. You will see the entries have expired. This is because we set the lifespan to 5 seconds.

In the previous step we used the overloaded put() method to store mortal entries. But since we want all of our entries to expire with the same lifespan, we can configure the cache to have default expiration values. To do this we will construct the DefaultCacheManager by passing in a org.infinispan.configuration.cache.Configuration object. A configuration in Infinispan is mostly immutable, aside from some runtime-tunable parameters, and is constructed by means of a ConfigurationBuilder. Using the above use-case, let's create a cache configuration where we want to set the default expiration of entries to 5 seconds.

Add the following line to `onStart` method in **ScoreService.java**. It should be on right under the `ConfigurationBuilder` instantiation:

```java
config.expiration().lifespan(5, TimeUnit.SECONDS);
```

Also change the `save` method implementation back to the following:

```java
    public void save(Score entry) {
        scoreCache.put(getKey(entry), entry);
    }
```

and re-run the load script:

```shell
sh $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-embedded-quarkus/scripts/load.sh
```

Reload **PREVIEW** right away. and you will see the entries again. Now wait for 5 seconds and refresh again. You will see the entries have expired. This is becuase we set the lifespan to 5 seconds for the CacheManager.

Now this is a configuration change for the cache and this will expire all entries after 5 seconds.

Challenge yourself: Next task for you is to change the lifespan to 5 minutes and see if that works for you.

When an entry expires it resides in the data container or cache store until it is accessed again by a user request. An expiration reaper is also available to check for expired entries and remove them at a configurable interval of milliseconds. More information can be found in the Product documentation

Eviction

Red Hat Data Grid supports eviction of entries, such that you do not run out of memory. Eviction is typically used in conjunction with a cache store, so that entries are not permanently lost when evicted, since eviction only removes entries from memory and not from cache stores or the rest of the cluster. Red Hat Data Grid supports storing data in a few different formats. Data can be stored as the object iself, binary as a byte[], and off-heap which stores the byte[] in native memory.

Eviction occurs on a local basis, and is not cluster-wide. Each instance will analyze on adding a new entry whether the threshold for evction is reached and decide what to evict. Eviction does not take into account the amount of free memory in the JVM as threshold to starts evicting entries. You have to set size attribute of the eviction element to be greater than zero in order for eviction to be turned on. If size is too large you can run out of memory. The size attribute will probably take some tuning in each use case.

Eviction is not recommended to use without a persistence as it will cause inconsistencies between the clustered instances!

Difference between Eviction and Expiration

Both Eviction and Expiration are means of cleaning the cache of unused entries and thus guarding the heap against OutOfMemory exceptions, but eviction is primary to control the data in memory and expirationis to control the lifecycle of entries. So now a brief explanation of the difference.

- With eviction you set maximal number of entries you want to keep in the memory and if this limit is exceeded if entries are added, a candidate is found to be dropped from memory according to the eviction strategy.
- eviction stategy depends on the memory configuration, see Configuring Data Grid - Eviction for more details
- Eviction can be set up with passivation in that case the entry is only persisted if evicted from memory
- With expiration you set time criteria for entries to specify how long you want to keep them in the cache.
- **lifespan** Specifies how long entries can remain in the cache before they expire. The default value is -1, which is unlimited time.

A new `put()` will reset the lifespan.

- **maximum idle time** Specifies how long entries can remain idle before they expire. An entry in the cache is idle when no operation is performed with the key. The default value is -1, which is unlimited time.

Perfect now we know what eviction and expiration API we have at our disposal and how we can use them in our app.

Add the following line to `onStart` method in **ScoreService.java**. It should be on right under the `ConfigurationBuilder` instantiation. In our example below we are going to limit our Cache to only 2 entries, anything above that will not be added to the Cache.

```java
config.memory().size(2).build();
```

and re-run the load script:

```shell
sh $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-embedded-quarkus/scripts/load.sh
```

Reload **PREVIEW** right away and you will see the entries again. But this time note that there are only two entries. And thats what we had specified in our Cache configuration.

This is only to demonstrate how eviction works, it is not deterministic which entry is evicted, and without a persistent cache store it effectively means removed. Consider eviction is local and will evict an entry based on the local access history, so different instances will evict different entries. In that case, without persistence, it depends which node is used to retrieve an entry and the result will be different for the same key.

Listeners

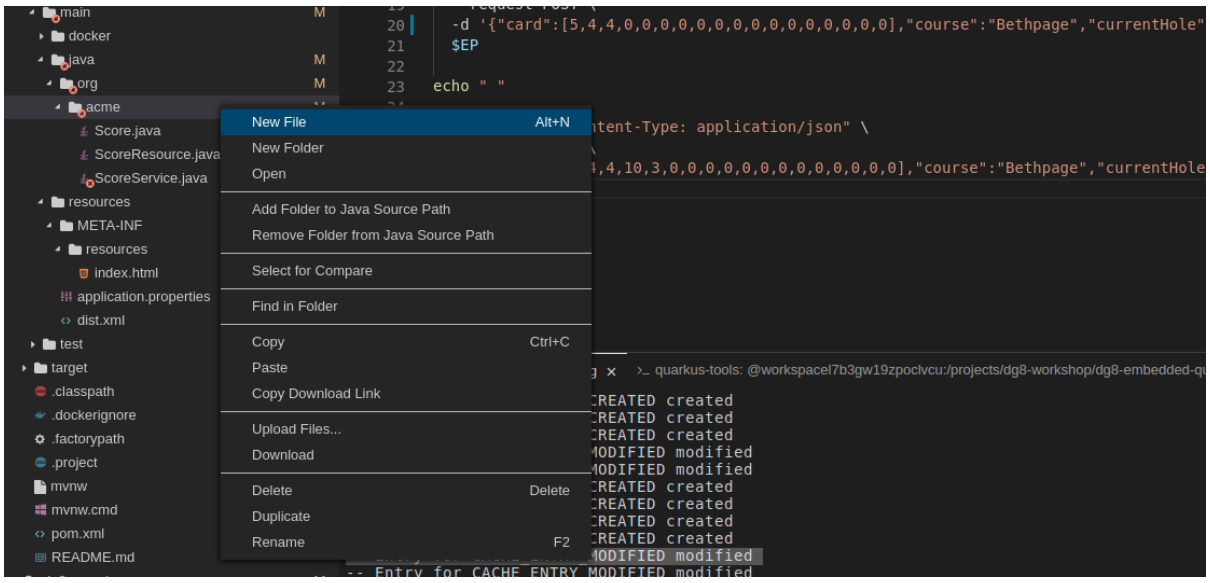
Red Hat Data Grid offers a listener API, where clients can register for and get notified when events take place. This annotation-driven API applies to 2 different levels: cache level events and cache manager level events.

Events trigger a notification which is dispatched to listeners. Listeners are simple POJO s annotated with `@Listener` and registered using the methods defined in the Listenable interface.

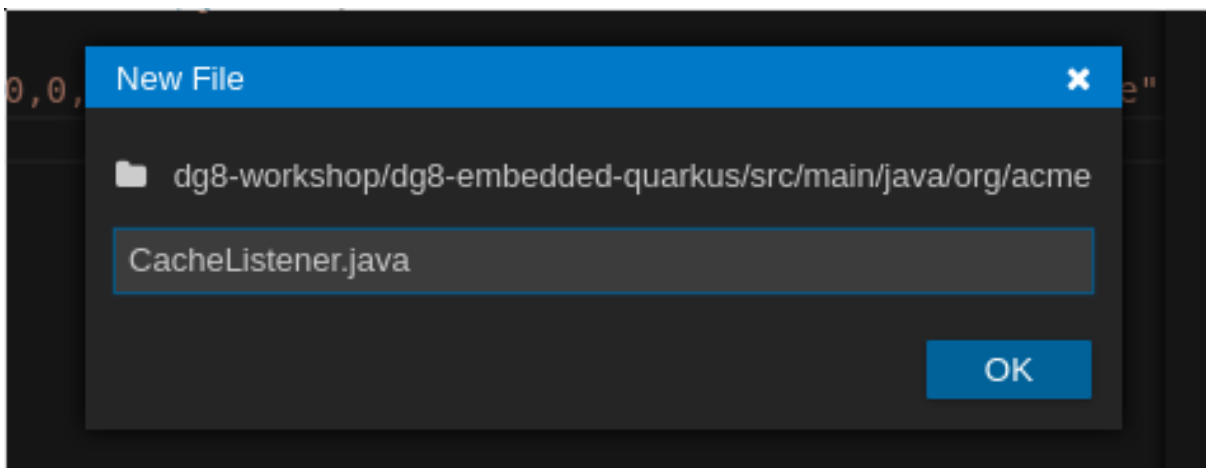
Both Cache and CacheManager implement `Listenable`, which means you can attach listeners to either a cache or a cache manager, to receive either cache-level or cache manager-level notifications.

Implement a new class `CacheListener`

1 Create a **New File** by right clicking on your project's package name i.e. `acme`, also shown in the screenshot below



2 Next specifiy the name of the file `CacheListener.java` , also in the screenshot below



JAVA
```java
package org.acme;

import org.infinispan.notifications.Listener;
import org.infinispan.notifications.cachelistener.annotation.CacheEntryCreated;
import org.infinispan.notifications.cachelistener.annotation.CacheEntryModified;
import org.infinispan.notifications.cachelistener.event.CacheEntryCreatedEvent;
import org.infinispan.notifications.cachelistener.event.CacheEntryModifiedEvent;

@Listener
public class CacheListener {

    @CacheEntryCreated
    public void entryCreated(CacheEntryCreatedEvent<String, Score> event) {
        System.out.printf("-- Entry for %s created \n", event.getType());
    }

    @CacheEntryModified
    public void entryUpdated(CacheEntryModifiedEvent<String, Score> event){
        System.out.printf("-- Entry for %s modified\n", event.getType());
    }
}
```

Also, important thing is to add this listener to our Cache configuration. Add the following line to the config. It should be pasted after **scoreCache = cacheManager.getCache("scoreboard");** line:

JAVA
```java
scoreCache.addListener(new CacheListener());
```

Now if we update the entries in our cache or create new ones; we will see a notification on the our console. Let’s test this out.

and re-run the load script:

SHELL
```shell
sh $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-embedded-quarkus/scripts/load.sh
```

Now check the terminals tab where it says `Embedded - Live Coding` you should messages like follows

```
2020-04-16 09:29:38,664 INFO  [org.acm.ScoreService] (vert.x-worker-thread-3) Cache initialized
2020-04-16 09:29:38,665 INFO  [io.quarkus] (vert.x-worker-thread-3) jcache-quarkus 1.0.0-SNAPSHOT (powered by Quarkus 1.3.2.Final) started in 0.074s. Listening
on: http://0.0.0.0:8080
2020-04-16 09:29:38,665 INFO  [io.quarkus] (vert.x-worker-thread-3) Profile dev activated. Live Coding activated.
2020-04-16 09:29:38,665 INFO  [io.quarkus] (vert.x-worker-thread-3) Installed features: [cdi, infinispan-embedded, kubernetes, resteasy, resteasy-jsonb,
smallrye-metrics]
2020-04-16 09:29:38,666 INFO  [io.qua.dev] (vert.x-worker-thread-3) Hot replace total time: 0.371s
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_CREATED created
-- Entry for CACHE_ENTRY_MODIFIED modified
```

SHELL

if you start to re run the load.sh a couple of times, you will start to see the modified messages more frequently. Assuming that the lifespan of the cache is more then 5 seconds.

Congratulations we are at the end of this lab!

Recap

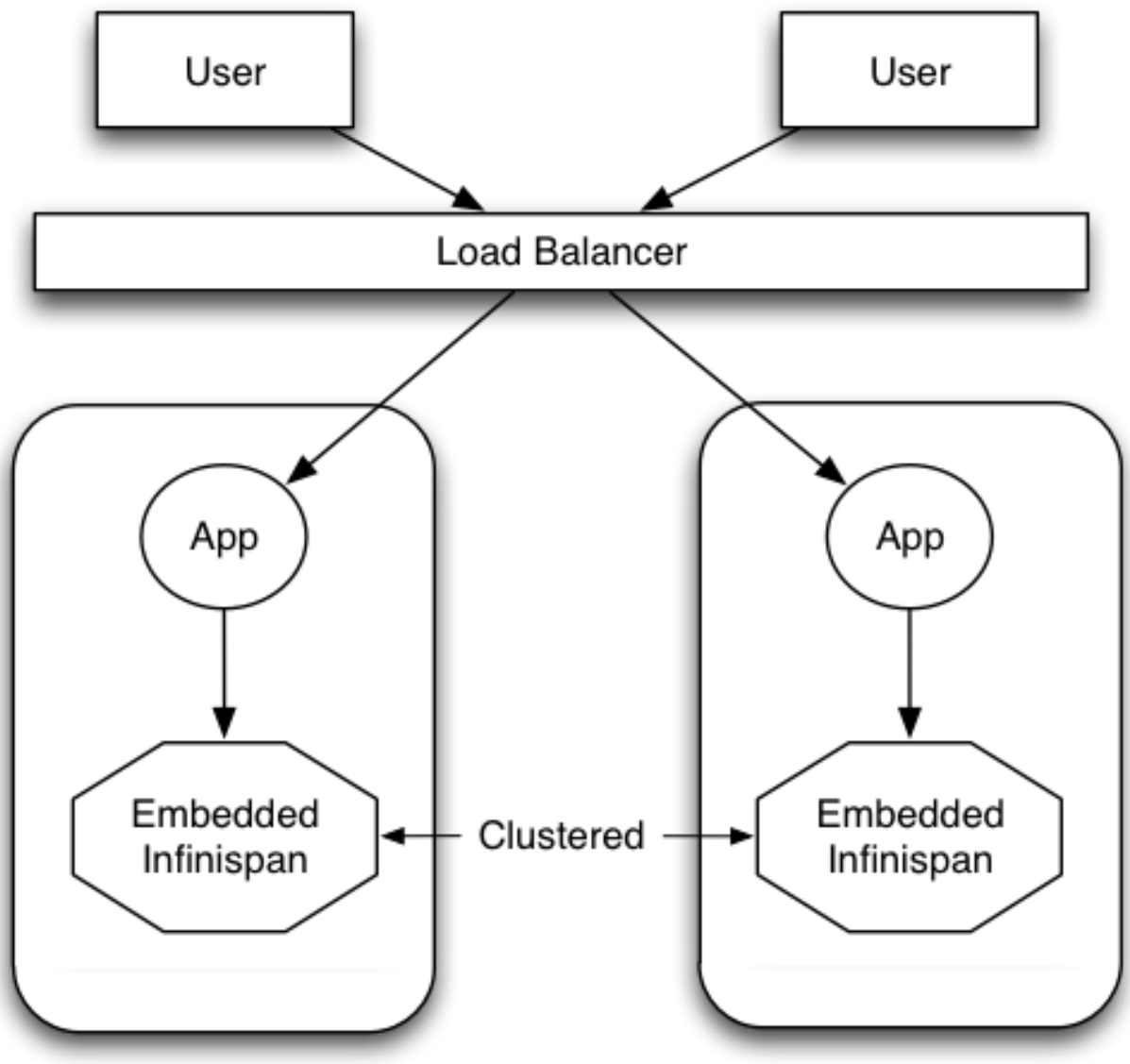
1. You created our own Cache and learnt how to us EmbeddedCacheManager
2. You learnt how to use ConfigurationBuilder and Configuration objects to define our Configurations for the Cache and CacheManager
3. You learnt about Expiration and Eviction
4. And lastly you implemented your own Listener.

Congratulations!! you have completed the first lab of this workshop. Let's move to the next lab and learn how we can cluster this Cache and also deploy this on a cloud environment like Openshift.

Embedded Clustered Cache

Embeded Cache with a cluster

In this lab we are going to cluster the embedded cache. Inifinispan does this very nicely. We do not need to change a lot of configuration to achieve this. After configuration, we will deploy our application to Openshift and see how the cluster will work in a cloud environment. Following diagram illustrates the topology we want to achieve. Although clustering cache nodes can be in 100s, we will start with a couple of them and see how we can get the feature set we need for a cache.



Clustering

Open the ScoreService.java again in our project `dg8-embedded-quarkus`

We are going to add the following lines of code to our onStart method

```java
        GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();
        global.transport().clusterName("ScoreCard");
        cacheManager = new DefaultCacheManager(global.build());
```

Replace the onStart method in our ScoreService.java

```java
    void onStart(@Observes @Priority(value = 1) StartupEvent ev){
        GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();  1
        global.transport().clusterName("ScoreCard").addProperty("configurationFile", "default-configs/jgroups-kubernetes.xml")
            .serialization().addContextInitializer(new org.acme.ScoreInitializerImpl());  2
        cacheManager = new DefaultCacheManager(global.build());

        ConfigurationBuilder config = new ConfigurationBuilder();

        config.expiration().lifespan(5, TimeUnit.MINUTES)
                .clustering().cacheMode(CacheMode.REPL_SYNC);  3

        cacheManager.defineConfiguration("scoreboard", config.build());
        scoreCache = cacheManager.getCache("scoreboard");
        scoreCache.addListener(new CacheListener());


        log.info("Cache initialized");

    }
```

1. We define a global configuration, since we are going to be in a clustered mode, hence everytime a new instance of our app will be created it will have the global configuration parameters to talk to the other nodes if they are present.
2. Infinispan nodes rely on a transport layer to join and leave clusters as well as to replicate data across the network. Infinispan uses JGroups technology to handle cluster transport. You configure cluster transport with JGroups stacks using DNS_PING on OpenShift, which define in *default-configs/jgroups-kubernetes.xml*. Here we have defined a unique clusterName for our app's embedded cache manager.
3. We setup a distributed, replicated cache, which means that all our nodes will have all the keys in its memory. But each node will return the correct values for all the keys.

We will take a look into replicaiton and distribution furhter in this lab.

Also open the CacheListener and make sure that the `@Listener` class annotation is changed to:

```java
@Listener(primaryOnly = false)
```

This will ensure all nodes get notified on changes.

It is possible in a non transactional cache to receive duplicate events. This is possible when the primary owner of a key goes down while trying to perform a write operation such as a put. Infinispan internally will rectify the put operation by sending it to the new primary owner for the given key automatically, however there are no guarantees in regards to if the write was first replicated to backups. Thus more than 1 of the following write events (CacheEntryCreatedEvent, CacheEntryModifiedEvent & CacheEntryRemovedEvent) may be sent on a single operation.

It is recommended that at this moment you press KBD:[CTRL+C] in the `Start Live Coding` terminal to quit our app, then close all terminal windows that you might have opened in the previous labs. to keep a clear view of our lab

First lets login to Openshift. You will find the button in the right corner in MyWorkspace menu. Click `Login to Openshift` and login with:

- **Username**: `user10`
- **Password**: `openshift`

After you log in using **Login to OpenShift**, the terminal is no longer usable as a regular terminal. You can close the terminal window. You will still be logged in when you open more terminals later!

Next, run the following command to add the Openshift extension for Quarkus (Be sure you're in the right directory: `cd /projects/dg8-workshop-labs/dg8-embedded-quarkus`).

The Openshift extension makes it easy to deploy your application to openshift, rather then taking all the different steps from an oc command line, you can do that through your maven build.

run the following in your terminal, you should see a BUILD SUCCESSFUL message when done.

```shell
mvn quarkus:add-extension -Dextensions="openshift"
```

Now open the application.properties file in `src/main/resources/application.properties`

Add the following properties to it

```shell
quarkus.http.cors=true
quarkus.openshift.expose=true  1

# if you dont set this and dont have a valid cert the deployment wont happen
quarkus.kubernetes-client.trust-certs=true  2
```

1 The first property makes sure that once our application is deployed it will expose a route

2 The second property makes sure that incase you dont have valid certificates the build wont stop. in our case that can likely be the case since its not a production environment rather a demo one.

Perfect everything is in order. Make sure you are logged into openshift. You can run the following command in your terminal to confirm:

```shell
oc whoami
```

The command should return your user name: user10, if you are logged in.

Let's first generate a container image for our application

```shell
mvn clean package -Dquarkus.container-image.build=true
```

You should see a build successful message at the end. That mean everything worked out.

Now lets deploy our application to Openshift

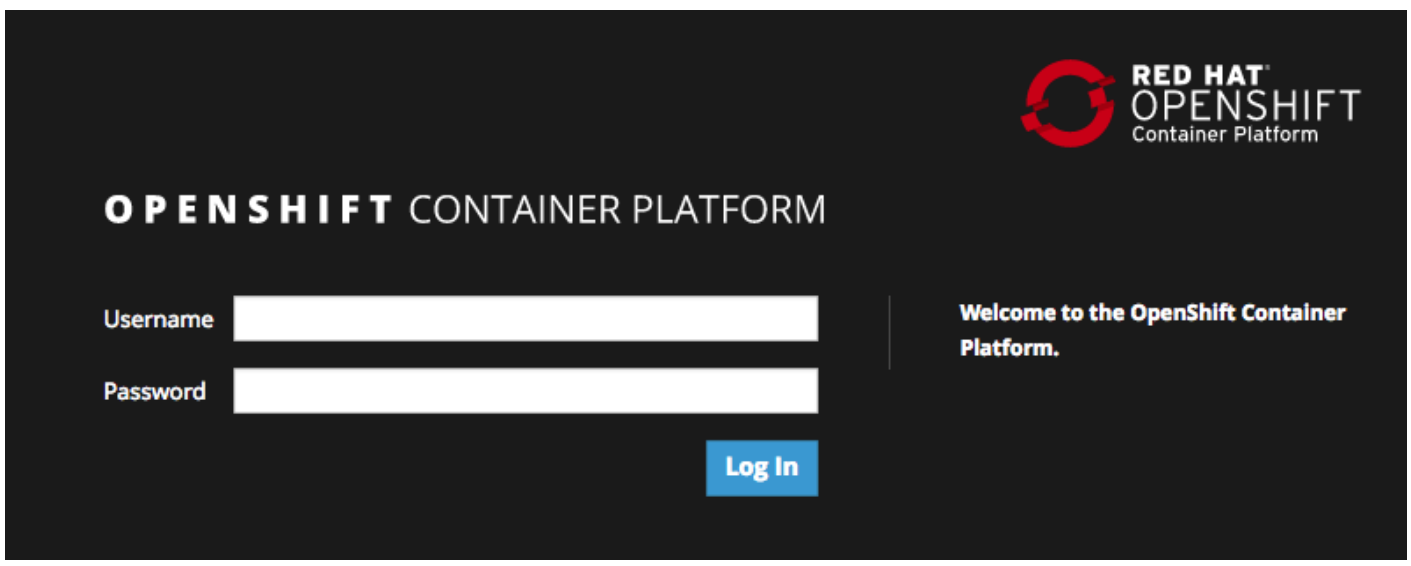
```shell
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Also remmember next time we need to deploy we just need to run the above deploy command again. thats all!

Lets wait for this build to be successfull!

Openshift Console

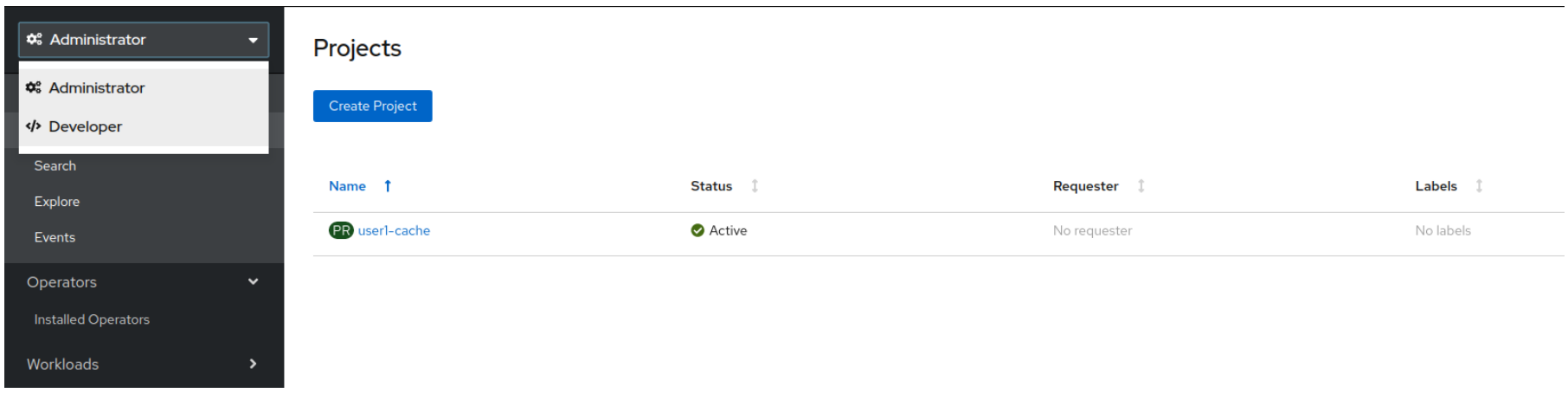
First, open a new browser with the OpenShift web console (https://console-openshift-console.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com)



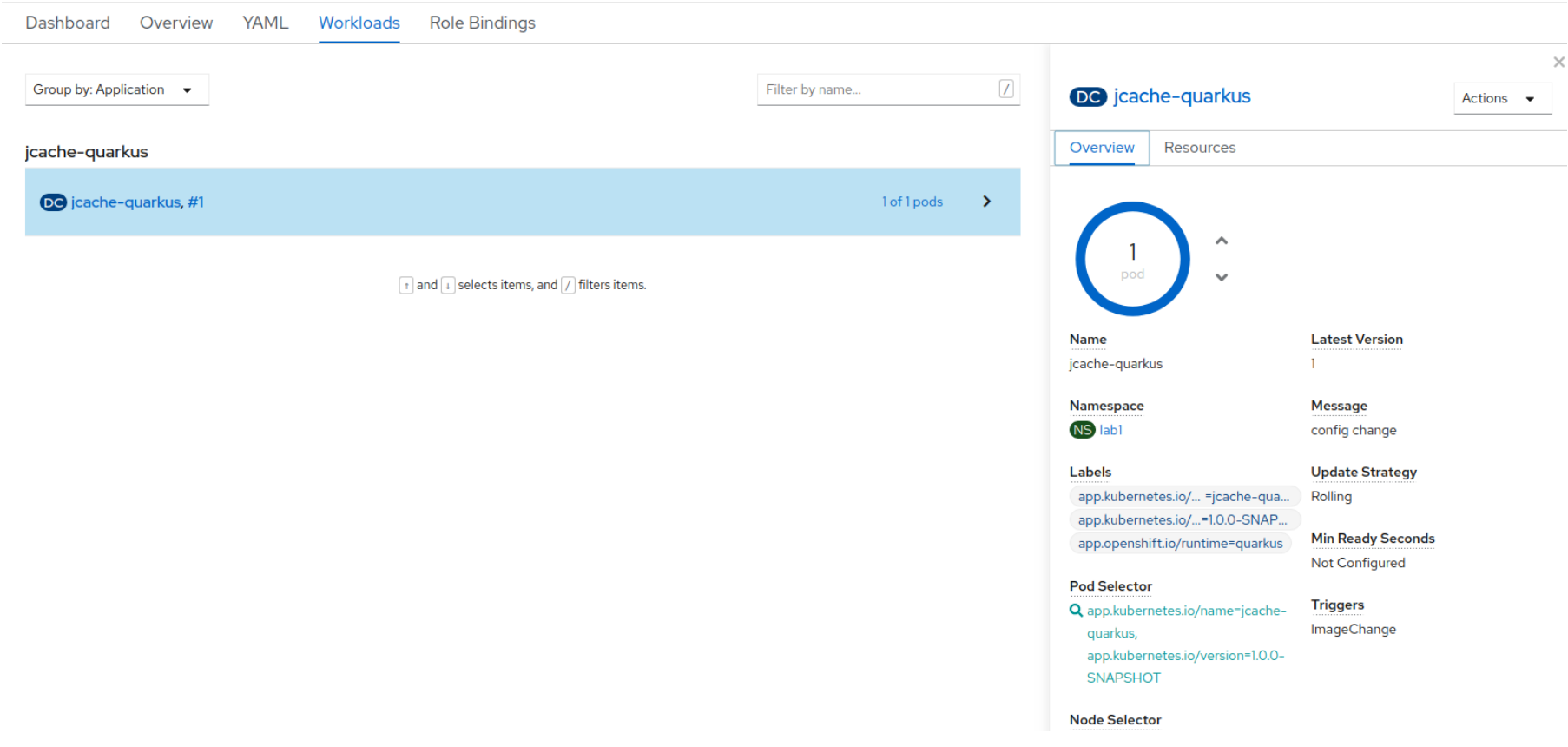
Login using:

- Username: `user10`
- Password: `openshift`

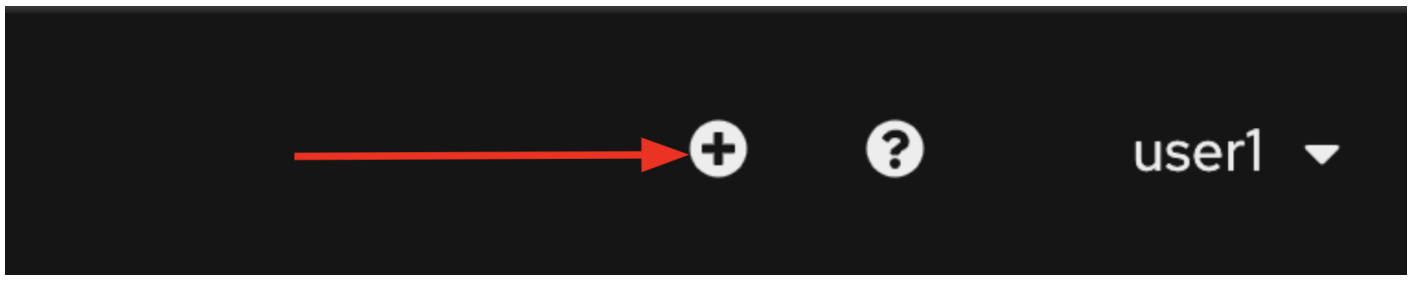
You should see something as follows:



Click on the project name and you should see something similar:



Create a new *JGroup* service to find members among Infinispan servers. Press the plus sign on the right top corner as shown in the picture:



Paste the below Service YAML code into the editor and click **Create**:

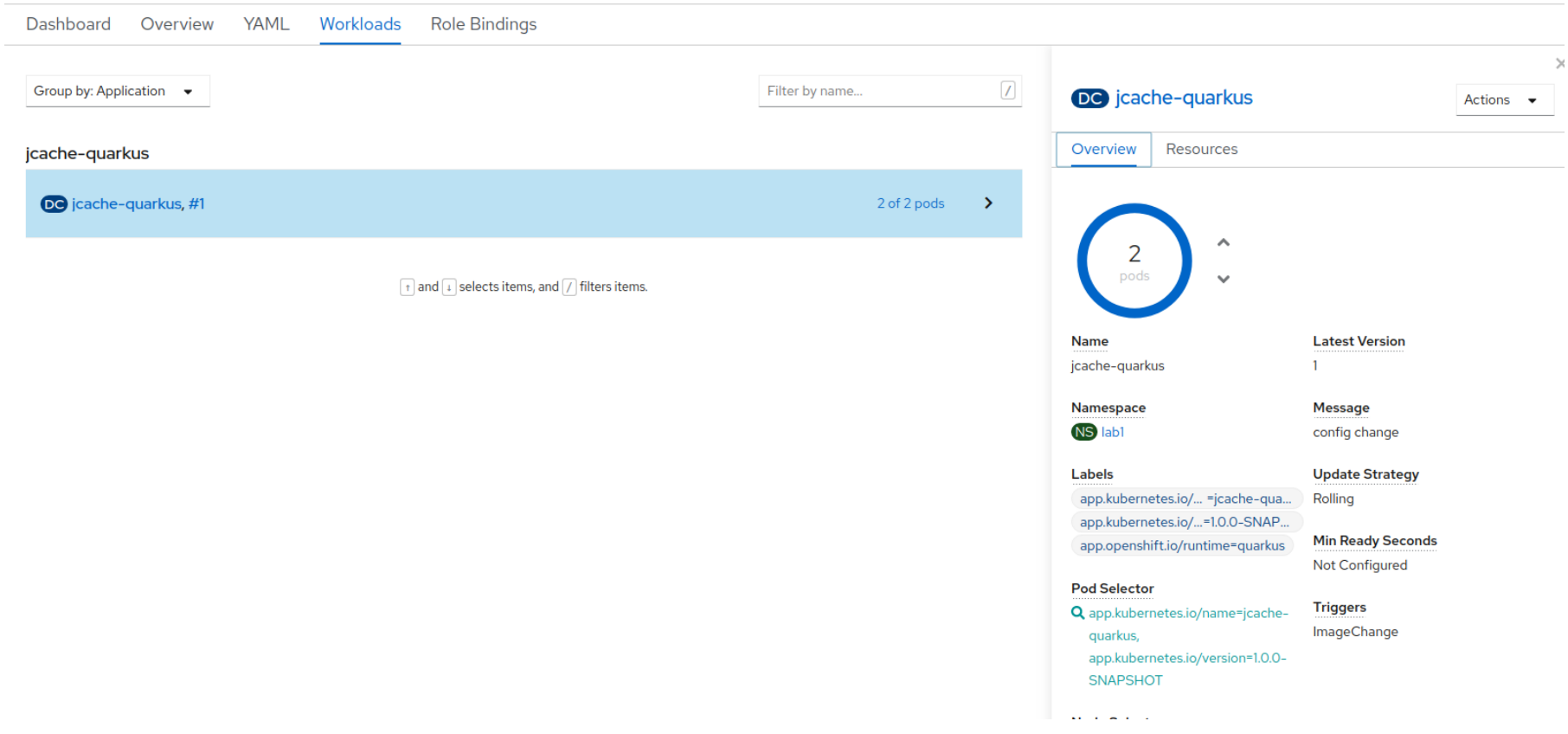
```yaml
kind: Service
apiVersion: v1
metadata:
  name: jcache-quarkus-ping
  namespace: user10-cache
spec:
  ports:
    - name: ping
      protocol: TCP
      port: 7800
      targetPort: 7800
  selector:
    app.kubernetes.io/name: jcache-quarkus
  clusterIP: None
  type: ClusterIP
```

Let's patch *DeploymentConfig* to add the above target port to the embedded cache application. Go back to CRW terminal window and execute the following commands:

```sh
oc patch dc/jcache-quarkus -p '{"spec": {"template": {"spec": {"containers": [{"name": "jcache-quarkus","ports":  [{"name": "http","containerPort": 8080,"protocol": "TCP"},{"name": "ping","containerPort": 7800,"protocol": "TCP"}]}]}}}}' && oc rollout latest dc/jcache-quarkus && oc rollout status -w dc/jcache-quarkus
```

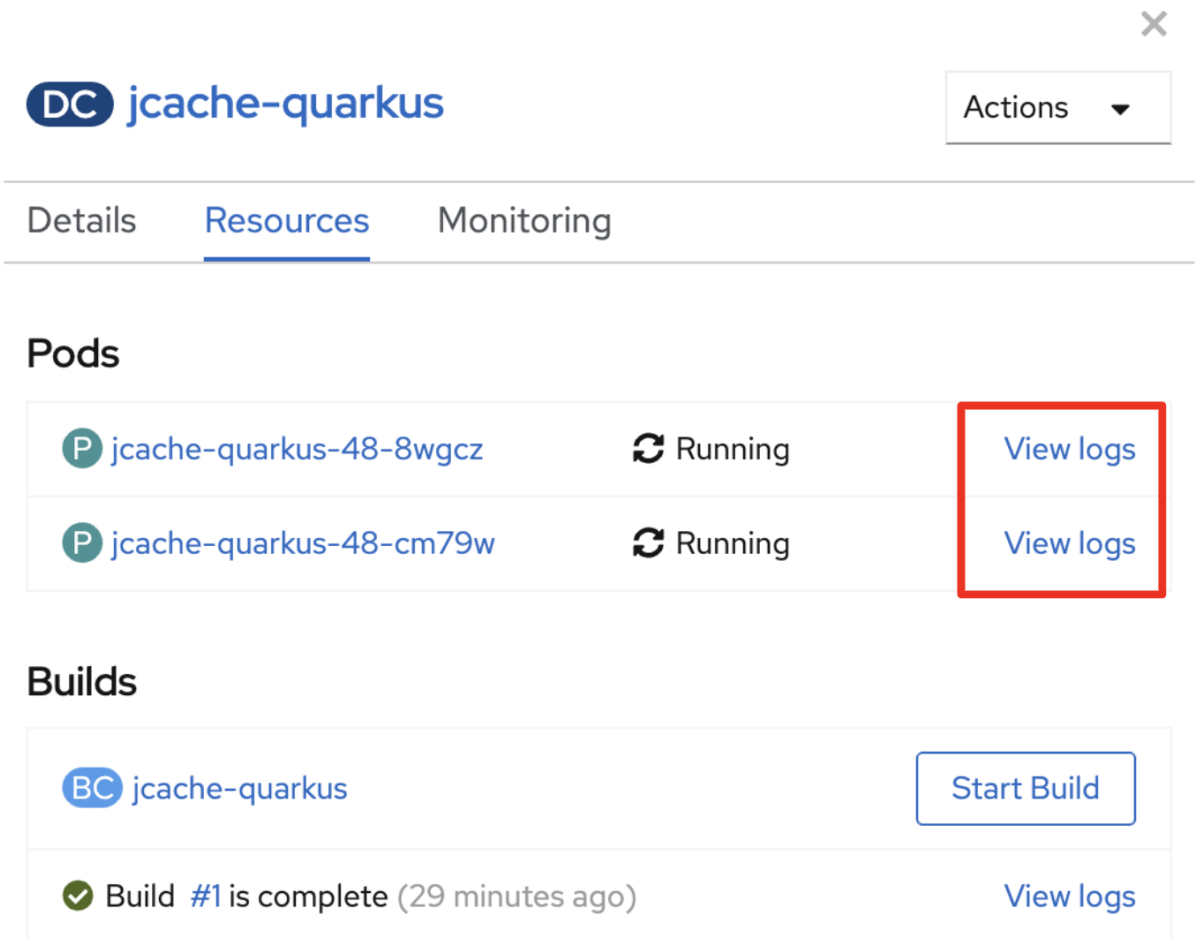
Navigate to *Workloads* tab, then click on the `jcache-quarkus` workload. Next, click on the *Resources* tab on the right, and at the bottom you will see the route to your application. You can also click at the route and it will take you to the application page, same as we have done in the previous lab. if append /api to the url you will be on the api endpoint.

Now go back to the `Details` tab for the applicaiton and Click on the pod scaler and scale to 2 pods.



This will spin up another instance of the app, and cluster them together automatically.

Let's find out if 2 Inifinspan cache servers joined the cluster. Click on `View Logs` in *Resources* tab:



You will see the following ISPN logs. It sometimes takes a min to join the cluster:

Now open another terminal in CodeReady workspaces and change to the scripts directory

```shell
cd dg8-embedded-quarkus/scripts
```

in this directory we have a load.sh file. Open this file in CodeReady Workspaces and change the variable `EP` to the application route from the browser (including the `/api` suffix):

load.sh

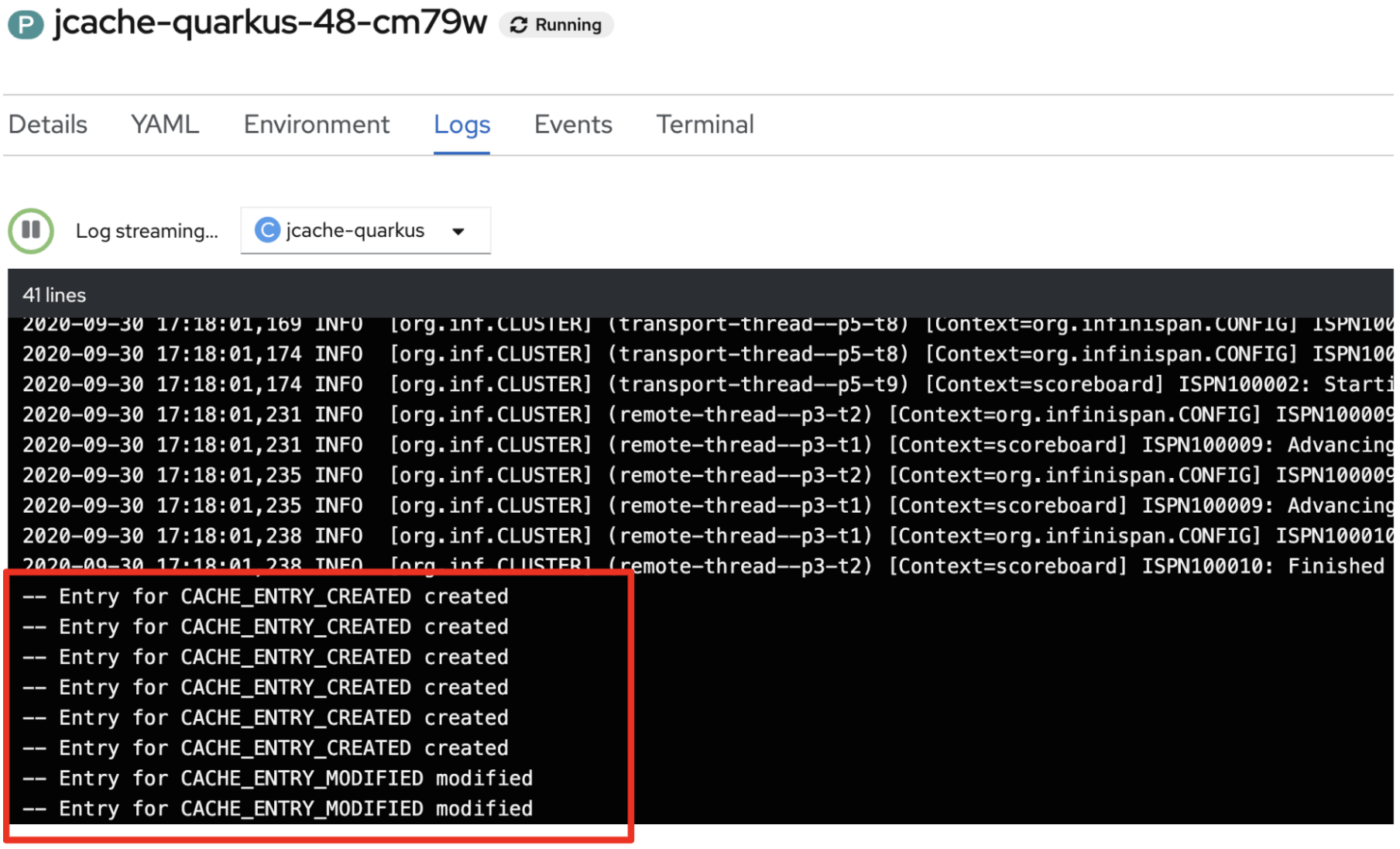
```
#!/usr/bin/env bash
#export EP=http://localhost:8080/api
export EP=http://jcache-quarkus-user1-cache.apps.cluster-mco-0c21.mco-0c21.sandbox1805.opentlc.com/api

curl --header "Content-Type: application/json" \
  --request POST \
  -d '{"card":[5,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],"course":"Bethpage","currentHole":3,"playerId":"4","player
  $EP
```

and then run load.sh

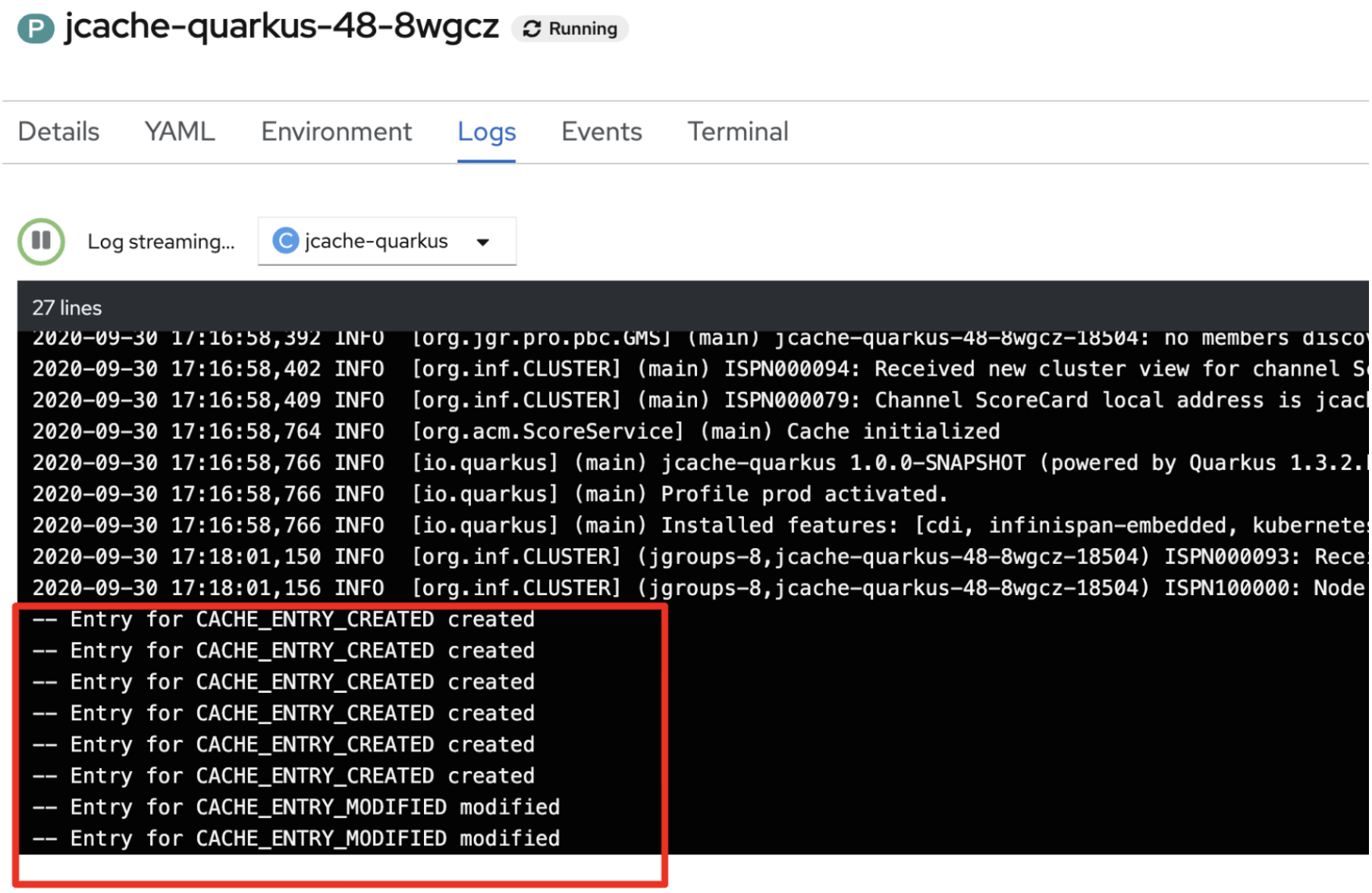
```
./load.sh
```
SHELL

Go back to the resrouce view of your application and then click view logs, you should see same Entry logs as follows in both nodes.

Node1:



Node2:



Design Considerations

Firstly, p2p deployments are simpler than client-server ones because in p2p, all peers are equals to each other and this simplifies deployment. If this is the first time you are using Infinispan, p2p is likely to be easier for you to get going compared to client-server.

Client-server Infinispan requests are likely to take longer compared to p2p requests, due to the serialization and network cost in remote calls. So, this is an important factor to take in account when designing your application. For example, with replicated Infinispan caches, it might be more performant to have lightweight HTTP clients connecting to a server side application that accesses Infinispan in p2p mode, rather than having more heavyweight client side apps talking to Infinispan in client-server mode, particularly if data size handled is rather large. With distributed caches, the difference might not be so big because even in p2p deployments, you're not guaranteed to have all data available locally.

Environments where application tier elasticity is not important, or where server side applications access state-transfer-disabled, replicated Infinispan cache instances are amongst scenarios where Infinispan p2p deployments can be more suited than client-server ones.

Congratulations we are at the end of this lab!

Recap

1. You created our own Cache and learnt how to us EmbeddedCacheManager
2. You learnt how to use ConfigurationBuilder and Configuration objects to define our Configurations for the Cache and CacheManager
3. You learnt about how to create and Embedded Cluster
4. You learnt how to deploy a Quarkus application with emebedded cache and scale it.
5. You learnt the difference between Replicated and Distributed Cache and how clustering and listeners works.

Congratulations!! you have completed the second lab of this workshop. Lets move to the next lab and learn how we can create a remote cache and how it can benefit our applications.

Operators and RHDG 8

Deployment: What's an Operator and how does it help us?

An Operator is a method of packaging, deploying, and managing a Kubernetes-native application. A Kubernetes-native application is an application that is both deployed on Kubernetes and managed using the Kubernetes APIs and tooling. An Operator is essentially a custom controller and encapsulates operational knowledge.

A controller is a core concept in Kubernetes and is implemented as a software loop that runs continuously on Kubernetes comparing, and if necessary, reconciling the expressed desired state and the current state of an object. Objects are well known resources like `Pods`, `Services`, `ConfigMaps`, or `PersistentVolumes`. Operators apply this model at the level of entire applications and are, in effect, application-specific controllers.

The Operator is a piece of software running in a `Pod` on the cluster, interacting with the Kubernetes API server. It introduces new object types through Custom Resource Definitions, an extension mechanism in Kubernetes. These custom objects are the primary interface for a user; consistent with the resource-based interaction model on the Kubernetes cluster.

An Operator watches for these custom resource types and is notified about their presence or modification. When the Operator receives this notification it will start running a loop to ensure that all the required connections for the application service represented by these objects are actually available and configured in the way the user expressed in the object's specification.

The Operator Lifecycle Manager (OLM) is the backplane that facilitates management of operators on a Kubernetes cluster. Operators that provide popular applications as a service are going to be long-lived workloads with, potentially, lots of permissions on the cluster.

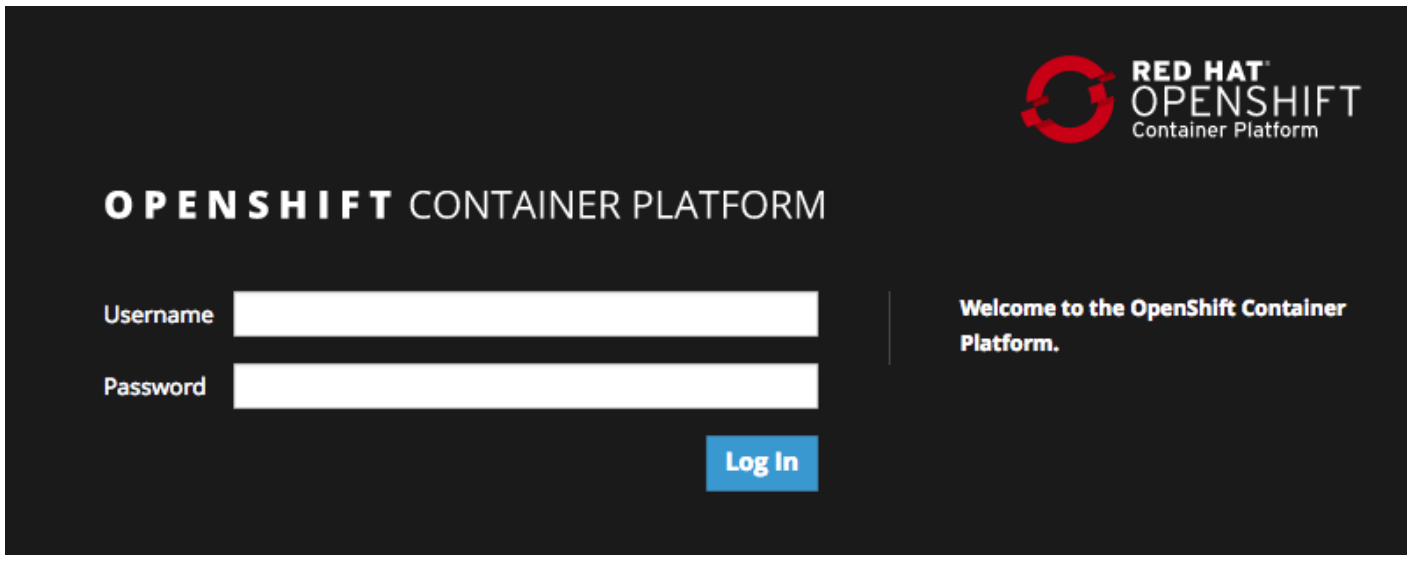
With OLM, administrators can control which Operators are available in what namespaces and who can interact with running Operators. The permissions of an Operator are accurately configured automatically to follow a least-privilege approach. OLM manages the overall lifecycle of Operators and their resources, by doing things like resolving dependencies on other Operators, triggering updates to both an Operator and the application it manages, or granting a team access to an Operator for their slice of the cluster.

Red Hat Data Grid 8.0 comes with an Operator. The administrators of the cluster you are working with today have already installed the Data Grid Operator. What we need to do as a user is define a Custom Resource as to how and what configuration we want for our Red Hat Data Grid instances.

Installing

If you have not already logged into OpenShift from the CodeReady Workspaces terminal, please do that now.Click on the `Login to Openshift` menu in the right menu called 'My Workspace'. When prompted, your **Username** is `user10` and your **Password** is `openshift`.

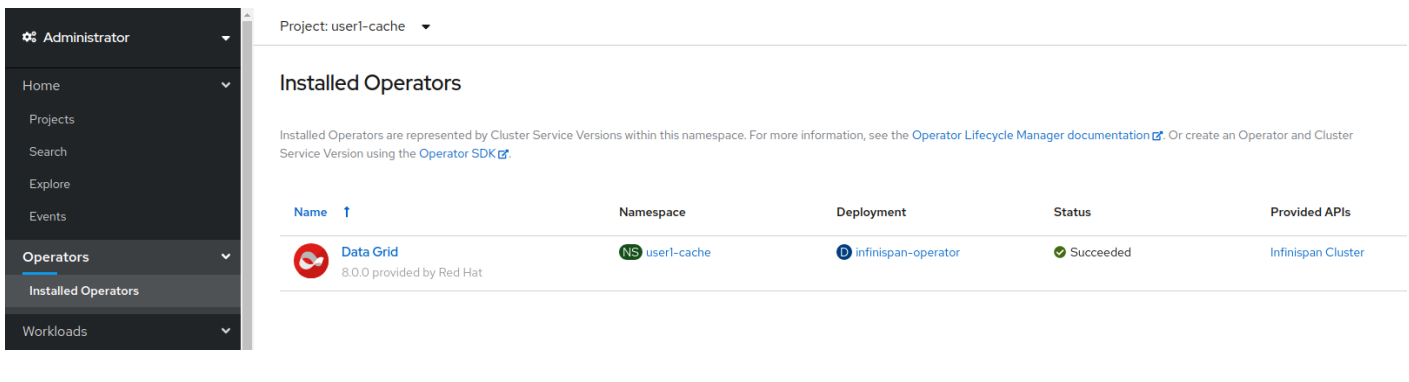
Open a new browser to the OpenShift web console (https://console-openshift-console.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com)



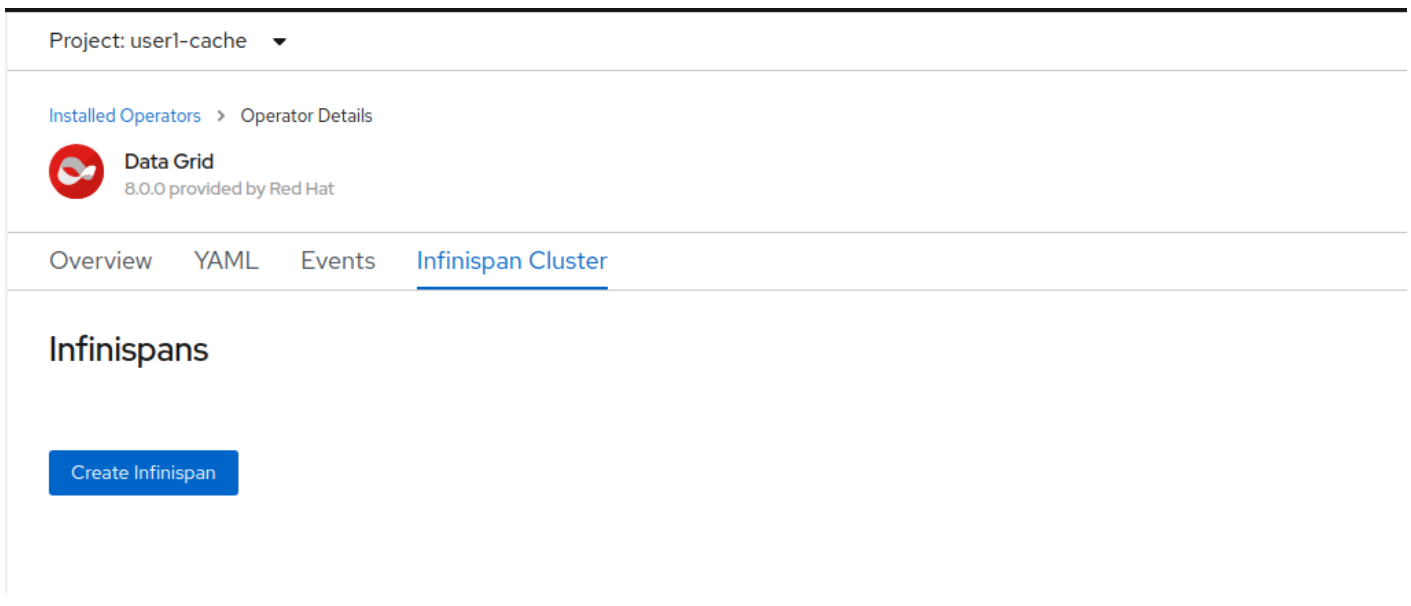
Login using:

- Username: `user10`
- Password: `openshift`

Once Logged in, goto your project `user10-cache` Click the link Installed Operator on the left, as shown in the picture below.



Notice that the DataGrid operator is already installed in your namespace.



You can see there are no clusters installed in our namespace. Let's go ahead and do that.

Click on `Create Infinispan` and replace the follwing YAML with the default sample:

```yaml
apiVersion: infinispan.org/v1
kind: Infinispan  1
metadata:
  name: datagrid-service  2
  namespace: user10-cache
spec:
  replicas: 2  3
  expose:
    type: LoadBalancer  4
```

1 Tell Kubernetes/Openshift that the Custom resource type is Infinispan

2 Specify the name of our cluster as datagrid-service

3 Specify the replicas we want for our service

4 Finally we want this instance to be accessible from outside (i.e. OpenShift Web Console)

Also notice that we are calling our service `datagrid-service`, we will use this name in the following labs to access our cluster.

Click **Create** at the bottom.

You can watch the Red Hat Data Grid Operator creating the instances by running the following command:

```shell
oc get pods
```

Above command should render a similar output as below:

```shell
[jboss@workspacel7b3gw19zpoclvcu dg8-operator]$ oc get pods
NAME                                READY   STATUS    RESTARTS   AGE
datagrid-service-0                  1/1     Running   0          2m59s
datagrid-service-1                  1/1     Running   0          2m14s
infinispan-operator-544ff55c59-4s7wl   1/1     Running   1          2d10h
```

```shell
oc get services
```

The above command should render a similar output as shown in the example below. Showing all the services:

```shell
NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP                                                              PORT(S)
AGE
datagrid-service           ClusterIP      172.30.115.185   <none>                                                                   11222/TCP
5m55s
datagrid-service-external  LoadBalancer   172.30.90.75     abd9b45a50a174648af684c05cba0bd9-1926931502.ap-southeast-1.elb.amazonaws.com   11222:32206/TCP
5m55s
datagrid-service-ping      ClusterIP      None             <none>                                                                   8888/TCP
5m55s
```

You can see that there are three datagrid-services,

- 1 for use within the cluster,
- 1 for ping service which ensures that the clusters are healthy and operational
- and lastly the external service, which we will use to goto the Admin console.

Run the following command to get the `LoadBalancer` address

```shell
oc get services | grep datagrid-service-external | awk '{ print $4 }'
```

As you can see we have a service with our `LoadBalancer`. Let's get that url and paste it in the browser as follows

The following is an example, your `LoadBalancer` url will most likely differ:

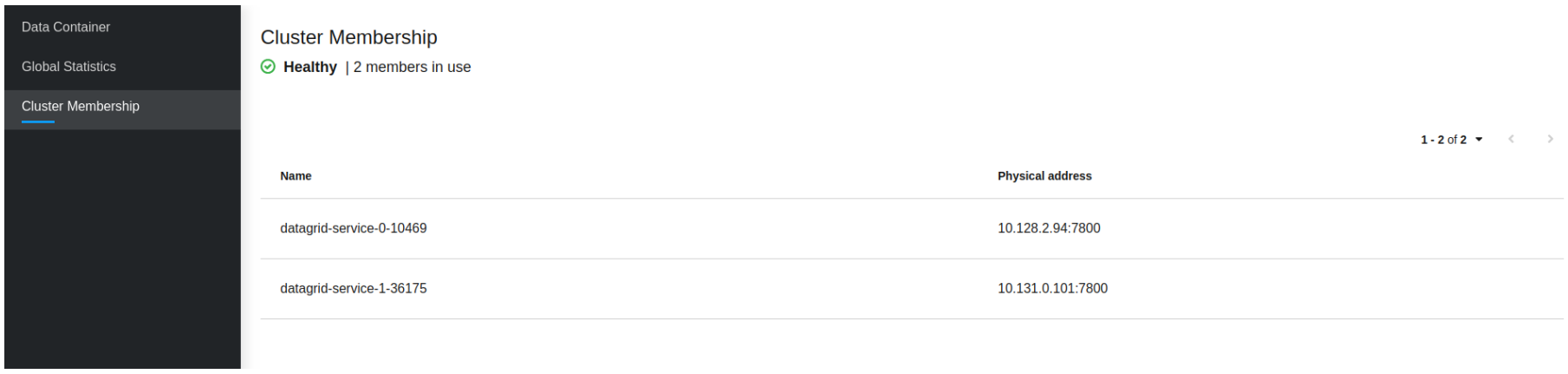
- `http://ad6cd35d6e6aa46fcb96558204c35f08-872149037.us-east-1.elb.amazonaws.com:11222`

If you try to access the url; you would need to provide credentials.

The datagrid operator creates the credentials during installation time and they should be stored in your namespace secrets. Let's get the secret with the following command.

```shell
oc get secret datagrid-service-generated-secret -o jsonpath="{.data.identities\.yaml}" | base64 --decode
```

And now the final test to check we have a running cluster; login with the username developer and the password from the above secret.



Recap

1. You created your own CR
2. Deployed the CR to Openshift using the DataGrid operator
3. You installed your DataGrid instance

Congratulations!! you have completed the first Datagrid installation of this workshop. Let's move to the next lab and learn how we can use this instance as a RemoteCache with a Quarkus Application.

Remote Cache

HotRod Client

Hot Rod is a binary TCP protocol that Infinispan offers for high-performance client-server interactions with the following capabilities:

- Load balancing — Hot Rod clients can send requests across Infinispan clusters using different strategies to increase reliability
- Failover — Hot Rod clients can monitor Infinispan cluster topology changes and automatically switch to available nodes to ensure requests never go to offline nodes
- Efficient data location — Hot Rod clients can find key owners and make requests directly to those nodes, which can help reduce latency

RemoteCache

A `RemoteCache`, as the name suggests, is a cache that can be accessed remotely. The Data Grid server will host this remote cache, and clients will connect to this remote cache. From a design perspectivem it gives more flexibility and also allows to have central deployments with multiple caches residing in it; there by making managemetn and operations a bit more simpler.

There are some symantic differences how Inifinspan/Red Hat Data Grid expose `RemoteCache` vs `EmbeddedCache`. The collection methods `keySet`, `entrySet`, and `values` are backed by the remote cache. Every method called is sent back into the `RemoteCache`. This is useful as it allows for the various keys, entries or values to be retrieved lazily, and not requiring them all be stored in the client memory at once if the user does not want. These collections adhere to the `Map` specification, noting that `add` and `addAll` are not supported but all other methods are supported. One thing to note is the `Iterator.remove`, `Set.remove`, or `Collection.remove` methods require more than 1 round trip to the server to operate. You can check out the `RemoteCache` Javadoc to see more details about these and the other methods.

Project details

For this example we are going to create a simple web application. it will take some input from a web form and then add the entries into the Cache. However in this case we will be using the `RemoteCacheManager` and we will be using ProtoStream API. All of this with a Quarkus based application.

So let's get cracking. But first let's take a look at the project.

Back in CodeReady Workspaces (https://codeready-codeready.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com), navigate to the project `dg8-quarkus-client-example`. This is a template project, and you will be writing code into this project. As you can see there is already some files inplace. Let's take a look into what these files are and do.

The Maven dependencies

Open the `pom.xml` file in the project.

We will be using the following dependencies to create our service

```xml
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy</artifactId> 1
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-jsonb</artifactId> 2
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-infinispan-client</artifactId> 3
</dependency>
<dependency>
```

1 `quarkus-resteasy` for our REST endpoint

2 `quarkus-resteasy-jsonb` for Json serialization for our REST endpoint

3 `quarkus-infinispan-client` for enabling us to use Data Grid's `RemoteCache`

Protobuf

Protobuf or Protocol Buffers are a method of serializing structured data. Protocol buffers are a flexible, efficient, and automated mechanism for serializing structured data. You can easily write and read your structured data to and from a variety of data streams and using a variety of languages. Protobuf is all about structured data, so the first thing to do is to define the structure of your data. This is accomplished by declaring Protobuf message types in .proto files.

In our example the `game.proto` file looks like this:

PROTOBUF
```
package quickstart;  1

message Game {  2
    required string name = 2;  3
    required string description = 3;  4
}
```

1 We define a package for our message

2 We define a name for our message. A message is similar to an entity.

3 We specify that our message has a string attribute called `name` and that it is required

4 We also specify that our message has a string attribute called `description` and that it is required.

Save the above content in the following file: `src/main/resources/META-INF/game.proto`

Marshallers

As described in the previous section, a fundamental concept of the Protobuf format is the definition of messages in the .proto schema to determine how an entity is represented. However, in order for our Java applications to utilize the Protobuf format to transmit/store data, it's necessary for our Java objects to be encoded. This is handled by the ProtoStream library and its configured Marshaller implementations, which convert plain old Java objects into the Protobuf format.

Although generating resources is the easiest and most performant way to utilize ProtoStream, this method might not always be viable. For example, if you are not able to modify the Java object classes to add the required annotations. For such use cases, it's possible to manually define the .proto schema and create a manual marshaller implementation. Let's define our Marshaller.

Open the `GameMarshaller` class in the `dg8-quarkus-client-example/src/main/java/org/acme/rest/json` folder.

Add the following method to our `GameMarshaller` class. In the following code we specify how we are going to read from our ProtoStream. We could add any additional processing on the stream if we wanted to. For now we take a simplified read and return a `Game` object. Hence everytime a stream is read from the Cache, this method will be called.

JAVA
```
    @Override
    public Game readFrom(MessageMarshaller.ProtoStreamReader reader) throws IOException {
        String name = reader.readString("name");
        String description = reader.readString("description");
        return new Game(name, description);
    }
```

Next we can also define a writer method. It takes a Game object and translates that into a stream.

JAVA
```
    @Override
    public void writeTo(MessageMarshaller.ProtoStreamWriter writer, Game game) throws IOException {
        writer.writeString("name", game.getName());
        writer.writeString("description", game.getDescription());
    }
```

Let's specify which class handles our Stream data.

JAVA
```
    @Override
    public Class<? extends Game> getJavaClass() {
        return Game.class;
    }
```

And finally here we let the Serialization process know what type we are doing this for. i.e. packagename.Class

JAVA
```
    @Override
    public String getTypeName() {
        return "quickstart.Game";
    }
```

Perfect we have our Marshaller configured.

Configuring our RemoteCache

Let’s move on and create our RemoteCache configuration

For this open the `Init.java` and add the following member variables to it.

```java
public static final String GAME_CACHE = "games"; 1

@Inject
RemoteCacheManager cacheManager; 2

private static final String CACHE_CONFIG = 3
        "<infinispan><cache-container>" +
                "<distributed-cache name=\"%s\"></distributed-cache>" +
                "</cache-container></infinispan>";
```

1 First we specify a class level variable which is the name of our Cache

2 We inject the `cacheManager` to our file. We only want to load the `CacheManager` once, and since its a heavy object, we want to do it at startup.

3 In addition to defining cache configuration within code, we can also configure a cache with xml. We are doing that here just to show that it is possible. We could have also loaded this from a file in the `META-INF` directory, but for a short demo this works okay as well.

```java
void onStart(@Observes @Priority(value = 1) StartupEvent ev) {
    String xml = String.format(CACHE_CONFIG, "games"); 1
    cacheManager.administration().getOrCreateCache(GAME_CACHE, new XMLStringConfiguration(xml)); 2
}
```

You might remember the `onStart` method from our previous labs. We are doing the same thing here. <1> We use the xml defined in a `String` and pass it on to the Red Hat Data Grid server to parse it and create a new cache called `games` <2> We then ask the `cacheManager` to get the Cache for us or create a new one if it doesnt exist

Now we should have a `RemoteCacheManager` configured, all we need to do now is to inject it in our REST resource.

REST endpoint

Open up the `GameResource.java` class. This class uses JAX-RS to define REST resources for our application.

In the following code we inject our `RemoteCache` , and we specify which remote cache we want by passing the variable `GAME_CACHE` to it, which we have initialized previously in our `Init` class.

Add this code to the `GameResource.java`

```java
@Inject
@Remote(GAME_CACHE)
RemoteCache<String, Game> gameStore;
```

The following are two simple GET and POST method implementation.

```java
@GET
public Set<Game> list() {
    return new HashSet<>(gameStore.values());
}

@POST
public Set<String> add(Game game) {
    gameStore.putAsync(game.getName(), game);
    return gameStore.keySet();
}
```

1 The `list` method is simply returning the games back to the front-end

2 The `add` method is using the Async api of Infinispan/Red Hat Data Grid to add the entry into the cache

Perfect. We are all set to deploy our application to Openshift and see how the `RemoteCache` will work.

Deploying to Openshift and scaling

Let’s prepare to deploy the application to Openshift

For this open up the `application.properties file` located in `src/main/resources/application.properties`

```properties
quarkus.infinispan-client.server-list=datagrid-service:11222 1
quarkus.infinispan-client.client-intelligence=BASIC 2
quarkus.infinispan-client.auth-username=developer 3
quarkus.infinispan-client.auth-password= 4


quarkus.http.cors=true

# Openshift extension settings.
quarkus.openshift.expose=true  5

# if you dont set this and dont have a valid cert the deployment wont happen

quarkus.kubernetes-client.trust-certs=true 6
quarkus.container-image.build=true
quarkus.kubernetes.deploy=true
```

1. Sets the Infinispan hostname/port to connect to. Each one is separated by a semicolon (eg. host1:11222;host2:11222)
2. Sets client intelligence used by authentication , in our case its basic, since we deployed a minimal server config
3. Sets username used by authentication, in our case its developer, thats the default from the operator.
4. Sets password used by authentication, we do not have this yet. we will find it out from the secrets.
5. We make sure that our applications route will be exposed once its deployed.
6. Finally we also put this property to true, in case our server does not have trusted certificates, which in our case will be true, since we are in a demo denvironment.

Let's go fill that password field in the above properties file.

Run the following command on the terminal and the passwords will be shown. Copy the password belonging to the `developer` user and add it to the password field `quarkus.infinispan-client.auth-password=` .

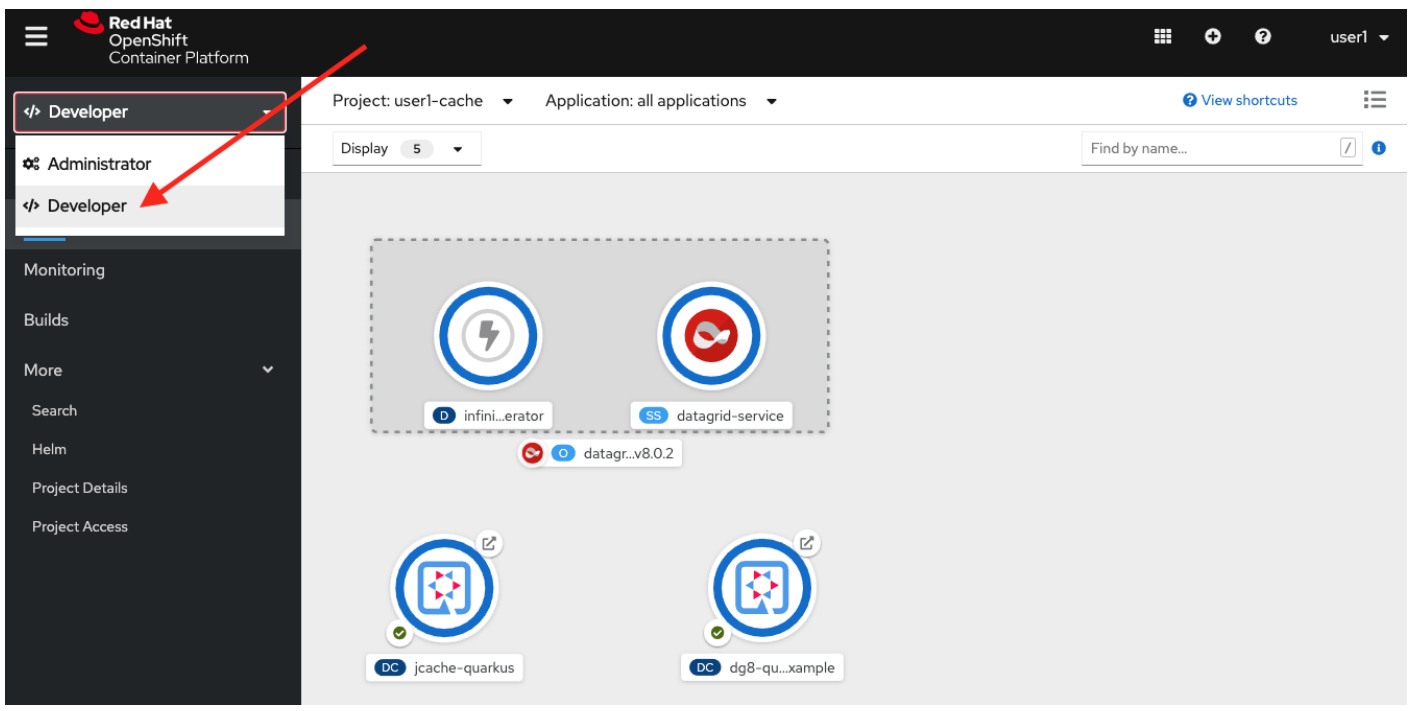
```shell
oc get secret datagrid-service-generated-secret -o jsonpath="{.data.identities\.yaml}" | base64 --decode
```

Let's go ahead and deploy the application to OpenShift.

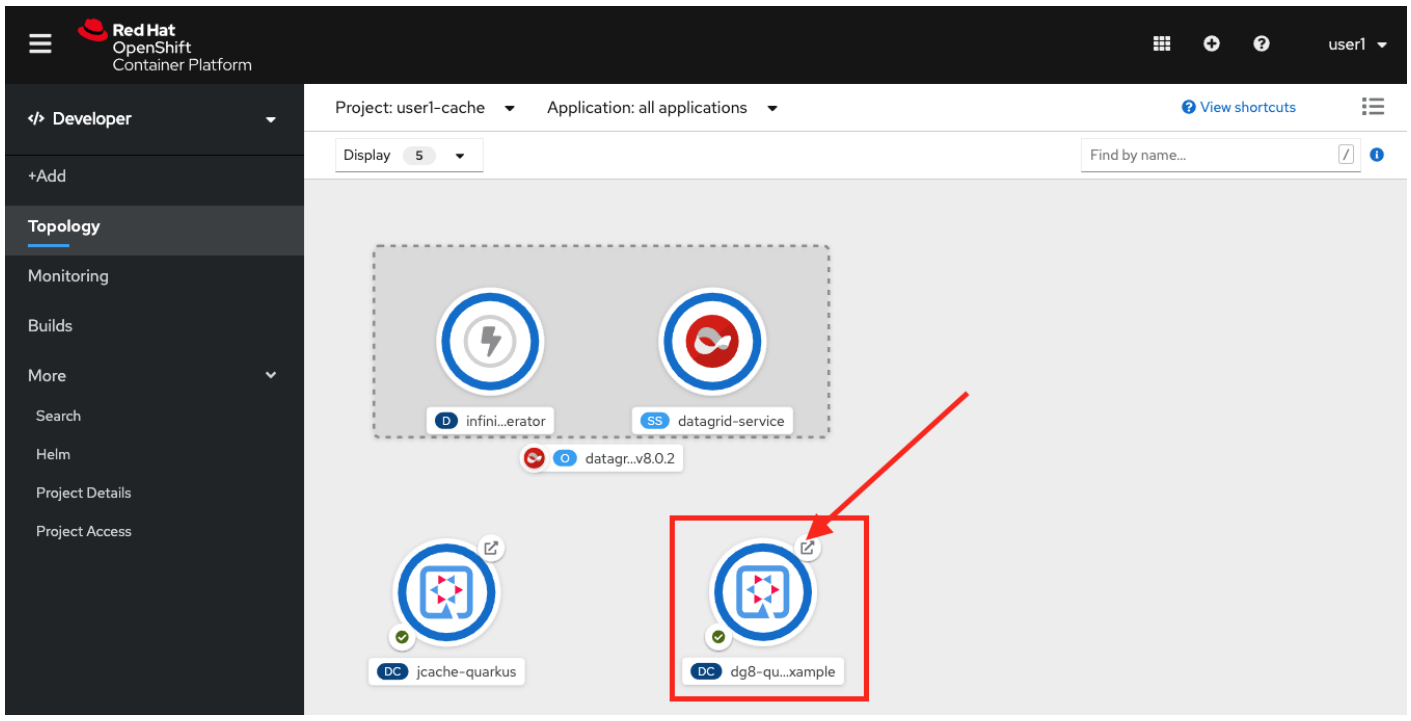
```shell
mvn clean package -DskipTests -f $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-quarkus-client-example
```

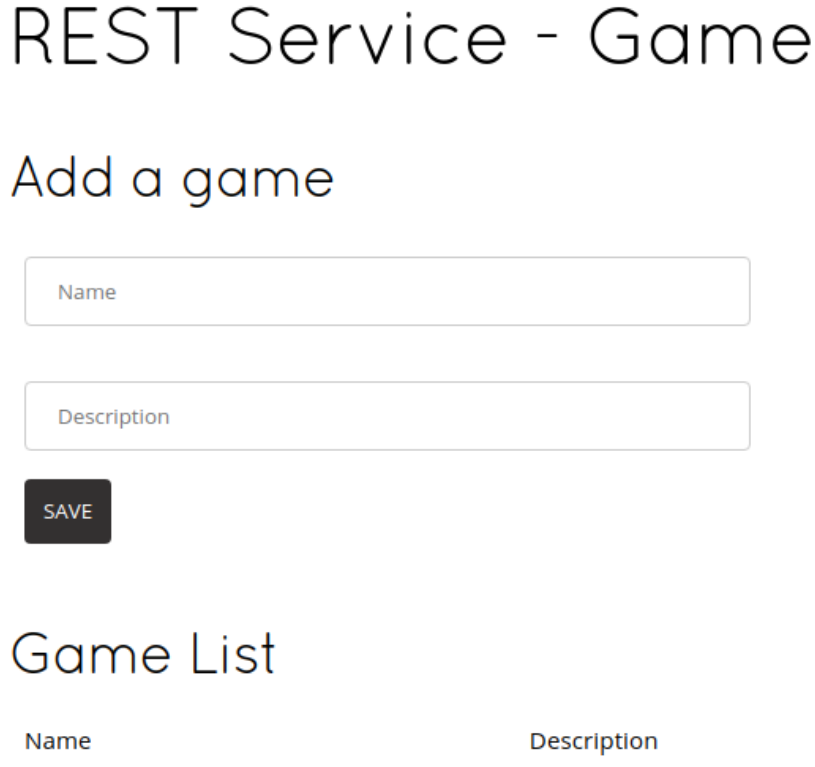
Let's wait for this build to be successful!

Now navigate to the OpenShift web console (https://console-openshift-console.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com) and switch to the topology view



Find the `dg8-quarkus-client-example` application and click on the route to navigate to the application





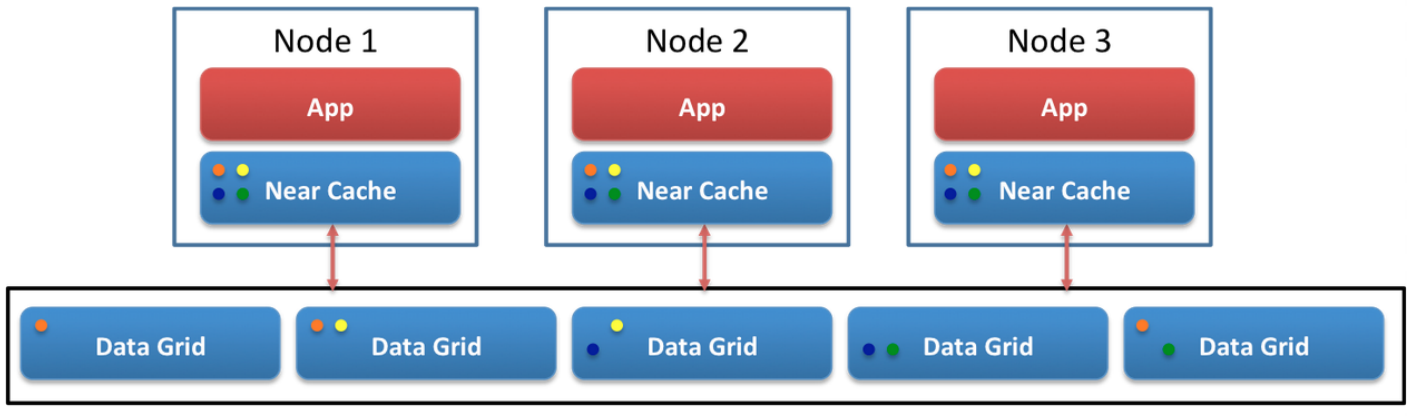
Try playing around with the application and adding some games.

Enabling Near Cache

Near caches are optional caches for Hot Rod Java client implementations that keep recently accessed data close to the user, providing faster access to data that is accessed frequently. This cache acts as a local Hot Rod client cache that is updated whenever a remote entry is retrieved via `get` or `getVersioned` operations.

In Red Hat Data Grid, near cache consistency is achieved by using remote events, which send notifications to clients when entries are modified or removed (refer to Remote Event Listeners). With near caching, local cache remains consistent with remote cache. Local entry is updated or invalidated whenever remote entry on the server is updated or removed. At the client level, near caching is configurable as either of the following:

- **DISABLED** - the default mode, indicating that near caching is not enabled
- **INVALIDATED** - enables near caching, keeping it in sync with the remote cache via invalidation messages.



When should I use it?

Near caching can improve the performance of an application when most of the accesses to a given cache are read-only and the accessed dataset is relatively small. When an application is doing lots of writes to a cache, invalidations, evictions, and updates to the near cache need to happen. In such a scenario the benefits a near cache provides won't necessarily be beneficial.

For Quarkus, near caching is disabled by default You can enable it by setting the profile config property `quarkus.infinispan-client.near-cache-max-entries` to a value greater than `0`. You can also configure a regular expression so that only a subset of caches have near caching applied through the `quarkus.infinispan-client.near-cache-name-pattern` property.

Add the following properties to our `application.properties` to enable near caching.

```properties
quarkus.infinispan-client.near-cache-max-entries=40
quarkus.infinispan-client.near-cache-name-pattern=*i8n-.
```

Let's go ahead and re-deploy the application to OpenShift.

```shell
mvn clean package -DskipTests -f $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-quarkus-client-example
```

You should see a Build Successful message from this run as well.

Notice that any entries that you might have added to the cache prior to this deployment are still there. That wasnt the case in the embedded cache, since we were not using any stores and everytime the application started the cache was empty. But in this case since the cache is remote, you will still see the entries from last time. Its important to note that there are different ways you can configure and setup the cache. For more details visit the Documenatation pages for Red Hat Data Grid.

Caching with Hibernate and JPA and Quarkus

When using Hibernate ORM in Quarkus, you don't need to have a `persistence.xml` file for configuration. Using such a classic configuration file is an option, but unnecessary unless you have specific advanced needs. Let's see first how Hibernate ORM can be configured without a persistence.xml resource.

In Quarkus, you just need to

- add your configuration settings in `application.properties`
- annotate your entities with `@Entity` and any other mapping annotation as usual

Other configuration needs have been automated: Quarkus will make some opinionated choices and educated guesses.

```java
package org.acme;

@Entity
@Cacheable
public class Country {
    // ...

    @OneToMany
    @Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
    List<City> cities;

    // ...
}
```

In the above code, just using the `@Cacheable` annotation will make sure that inifinspan is used as the second level cache for the entities. You also don't need to pick an implementation. A suitable implementation based on technologies Infinispan is included as a transitive dependency of the Hibernate ORM extension, and automatically integrated during the build.

Recap

1. You learnt about `RemoteCache` and HotRod client
2. You learnt about Protostream and marshallers in Infinispan
3. You deployed your Quarkus app using `RemoteCache` to OpenShift
4. You learnt about near caching and its use case
5. And finally we sum it up with JPA and Second Level Cache

Congratulations!! you have completed the this lab on RemoteCache. Let's move to the next lab and learn how we can use the new REST API in DataGrid to our advantage.

REST API

The Data Grid REST API allows you to monitor, maintain, and manage Data Grid deployments and provides access to your data. Although the REST API has a vast amount of features that one can use, in our lab we will demonstrate only some of them. For more details on the documentation on the Red Hat Data Grid REST API please visit the documentation page. Infinispan servers provide RESTful HTTP access to data through a REST endpoint built on Netty.

Supported Formats

You can write and read data in different formats and Data Grid can convert between those formats when required.

The following "standard" formats are interchangeable:

- `application/x-java-object`

- `application/octet-stream`
- `application/x-www-form-urlencoded`
- `text/plain`

We are using this example from the very good lists of demo collection on github: https://github.com/infinispan-demos/links We will do the following

1. Create a Cache in infinispan with Lucene support
2. Define our protostream
3. Load bulk data into the cache
4. Use search queries directly into the cache

Let's get started!

The Pokeman Example

What is a Pokemon?

Reminiscences from the GameBoy the creatures that inhabit the world of Pokémon are also called Pokémon. Many species of Pokémon are capable of evolving into a larger and more powerful creatures. The change is accompanied by stat changes, generally a modest increase, and access to a wider variety of attacks. There are multiple ways to trigger an evolution, including reaching a particular level, using a special stone, or learning a specific attack. For example, at level 16 Bulbasaur is capable of evolving into Ivysaur. Most notably, the Normal-type Eevee is capable of evolving into eight different Pokémon: Jolteon (Electric), Flareon (Fire), Vaporeon (Water), Umbreon (Dark), Espeon (Psychic), Leafeon (Grass), Glaceon (Ice), and Sylveon (Fairy). In Generation VI, a new mechanic called Mega Evolution—as well as a subset of Mega Evolution called Primal Reversion—was introduced into the game. Unlike normal evolution, Mega Evolution and Primal Reversion last only for the duration of a battle, with the Pokémon reverting to its normal form at the end. Forty-eight Pokémon are capable of undergoing Mega Evolution or Primal Reversion as of the release of Sun and Moon. In contrast, some species such as Castform, Rotom, Unown, and Lycanroc undergo form changes that may provide stat buffs or changes and type alterations but are not considered new species. Some Pokémon have differences in appearance due to gender. Pokémon can be male or female, male-only, female-only, or genderless.

Setup

All the project sources are in the project `dg8-restapi`. We will be using the `curl` utility as well as our browser.

Let's open up a new terminal in the CodeReady workspace. Make sure you are logged into OpenShift.

Run the following command on the terminal which will get the password for user developer and store it in and environment variable.

```shell
export PASSWORD=$(oc get secret datagrid-service-generated-secret -o jsonpath="{.data.identities\.yaml}" | base64 --decode | awk 'NR==3' | awk '{print $2}')
```

We are using the same service we created from the operator in previous labs.

We will also need our `LoadBalancer` address to reach the datagrid server. You can do that by running the following command in the terminal, which will set the `LoadBalancer` address to $LB in the terminal.

```shell
export LB="http://$(oc get services | grep datagrid-service-external | awk '{ print $4 }'):11222"
```

At anytime if you close the terminal, you will need to run these two commands again!

First let's register our protobuf

Let's see whats our data structure looks like.

PROPERTIES

```
/**
  * @Indexed
  */
message Pokemon  {
  repeated string abilities = 1;
  optional float against_bug = 2;
  optional float against_dark = 3;
  optional float against_dragon = 4;
  optional float against_electric = 5;
  optional float against_fairy = 6;
  optional float against_fight = 7;
  optional float against_fire = 8;
  optional float against_flying = 9;
  optional float against_ghost = 10;
  optional float against_grass = 11;
  optional float against_ground = 12;
  optional float against_ice = 13;
  optional float against_normal = 14;
  optional float against_poison = 15;
  optional float against_psychic = 16;
  optional float against_rock = 17;
  optional float against_steel = 18;
  optional float against_water = 19;
  optional int32 attack = 20;
  optional int32 base_egg_steps = 21;
  optional int32 base_happiness = 22;
  optional int32 base_total = 23;
  optional string capture_rate = 24;

  /* @Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO) */
  optional string classfication = 25;
  optional int32 defense = 26;
  optional int32 experience_growth = 27;
  optional float height_m = 28;
  optional int32 hp = 29;

  /* @Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO) */
  optional string japanese_name = 30;

  /* @Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO) */
  optional string name = 31;
  optional float percentage_male = 32;
  optional int32 pokedex_number = 33;
  optional int32 sp_attack = 34;
  optional int32 sp_defense = 35;
  optional int32 speed = 36;

  /* @Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO) */
  optional string type1 = 37;
  optional string type2 = 38;
  optional float weight_kg = 39;
  optional int32 generation = 40;
  optional int32 is_legendary = 41;
}
```

In our project `dg8-restapi` open the file `pokemon.proto` and paste the above data structure. The structure defines the capabilites of a Pokemon.

When caches are indexed, or specifically configured to store `application/x-protostream`, you can send and receive JSON documents that are automatically converted to and from Protostream. In order for this conversion to work you must register a protobuf schema.

First let's cd into the current project

SHELL

```
cd $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-restapi
```

To register protobuf schemas via REST, invoke a POST or PUT in the `___protobuf_metadata` cache as in the following command

SHELL

```
curl -u developer:$PASSWORD -X POST --data-binary @./pokemon.proto $LB/rest/v2/caches/___protobuf_metadata/pokemon.proto
```

Create a Cache

Now let's create an indexed cache since we want to retrieve data at speed from our Cache and Lucene store.

SHELL

```
curl -u developer:$PASSWORD -H "Content-Type: application/json" -d '{"distributed-cache":{"mode":"SYNC","indexing":{"auto-config":true,"index":"ALL"}}}'
$LB/rest/v2/caches/pokemon
```

Bulk loading the REST endpoint

Now we will load all the pokemon data we have in a json format. You can view the json files in `dg8-restapi/data`

for Example Abra's capabilities listed as follows

```JSON
{
    "_type": "Pokemon",
    "abilities": "['Synchronize', 'Inner Focus', 'Magic Guard']",
    "against_bug": 2.0,
    "against_dark": 2.0,
    "against_dragon": 1.0,
    "against_electric": 1.0,
    "against_fairy": 1.0,
    "against_fight": 0.5,
    "against_fire": 1.0,
    "against_flying": 1.0,
    "against_ghost": 2.0,
    "against_grass": 1.0,
    "against_ground": 1.0,
    "against_ice": 1.0,
    "against_normal": 1.0,
    "against_poison": 1.0,
    "against_psychic": 0.5,
    "against_rock": 1.0,
    "against_steel": 1.0,
    "against_water": 1.0,
    "attack": 20.0,
    "base_egg_steps": 5120.0,
    "base_happiness": 70.0,
    "base_total": 310.0,
    "capture_rate": 200.0,
    "classfication": "Psi Pok\u00e9mon",
    "defense": 15.0,
    "experience_growth": 1059860.0,
    "height_m": 0.9,
    "hp": 25.0,
    "japanese_name": "Casey\u30b1\u30fc\u30b7\u30a3",
    "name": "Abra",
    "percentage_male": 75.4,
    "pokedex_number": 63.0,
    "sp_attack": 105.0,
    "sp_defense": 55.0,
    "speed": 90.0,
    "type1": "psychic",
    "type2": 0,
    "weight_kg": 19.5,
    "generation": 1.0,
    "is_legendary": 0.0
}
```

Let’s run our loading script which is placed in our project `dg8-restapi`. Run the shell script in the terminal

```SHELL
./ingest-data.sh
```

The script run can take some time, wait for it to finish.

So by now we should have loaded about 801 Pokemon’s from the Pokemon universe.

So what does the script look like? Below you can se that we are loading each of the json files one by one to the cache rest end point

```SHELL
status=0
for f in data/*.json
do
  curl -u developer:$PASSWORD -XPOST --data-binary @${f}  -H "Content-Type: application/json; charset=UTF-8"  $LB/rest/v2/caches/pokemon/$(basename $f .json)
  let status=status+1
  echo  "Imported $f (total $status pokemons)"
done
```

Query the data

Get all Pokemons:

```URL
curl -u developer:$PASSWORD $LB/rest/v2/caches/pokemon'?action=search&query=from%20Pokemon'
```

Count Pokemons by generation:

```URL
# select count(p.name) from Pokemon group by generation
curl -u developer:$PASSWORD $LB/rest/v2/caches/pokemon'?action=search&query=select%20count(p.name)%20from%20Pokemon%20p%20group%20by%20generation'
```

Do a full text search on the name

```URL
curl -u developer:$PASSWORD $LB/rest/v2/caches/pokemon'?action=search&query=from%20Pokemon%20where%20name:%27pikachu%27'
```

Select top 5 Pokemons that can better withstand fire:

```
curl -u developer:$PASSWORD $LB/rest/v2/caches/pokemon'?action=search&query=from%20Pokemon%20order%20by%20against_fire%20asc&max_results=5'
```
URL

Get Pokemon by key (name)

```
curl -u developer:$PASSWORD $LB/rest/v2/caches/pokemon/Whismur
```
URL

Recap

1. You how the REST API works
2. You created a cache and protobuf via REST API
3. You loaded bulk data into the cache
4. And finally you queried through that data.

Congratulations!! you have completed the this lab on REST API!!

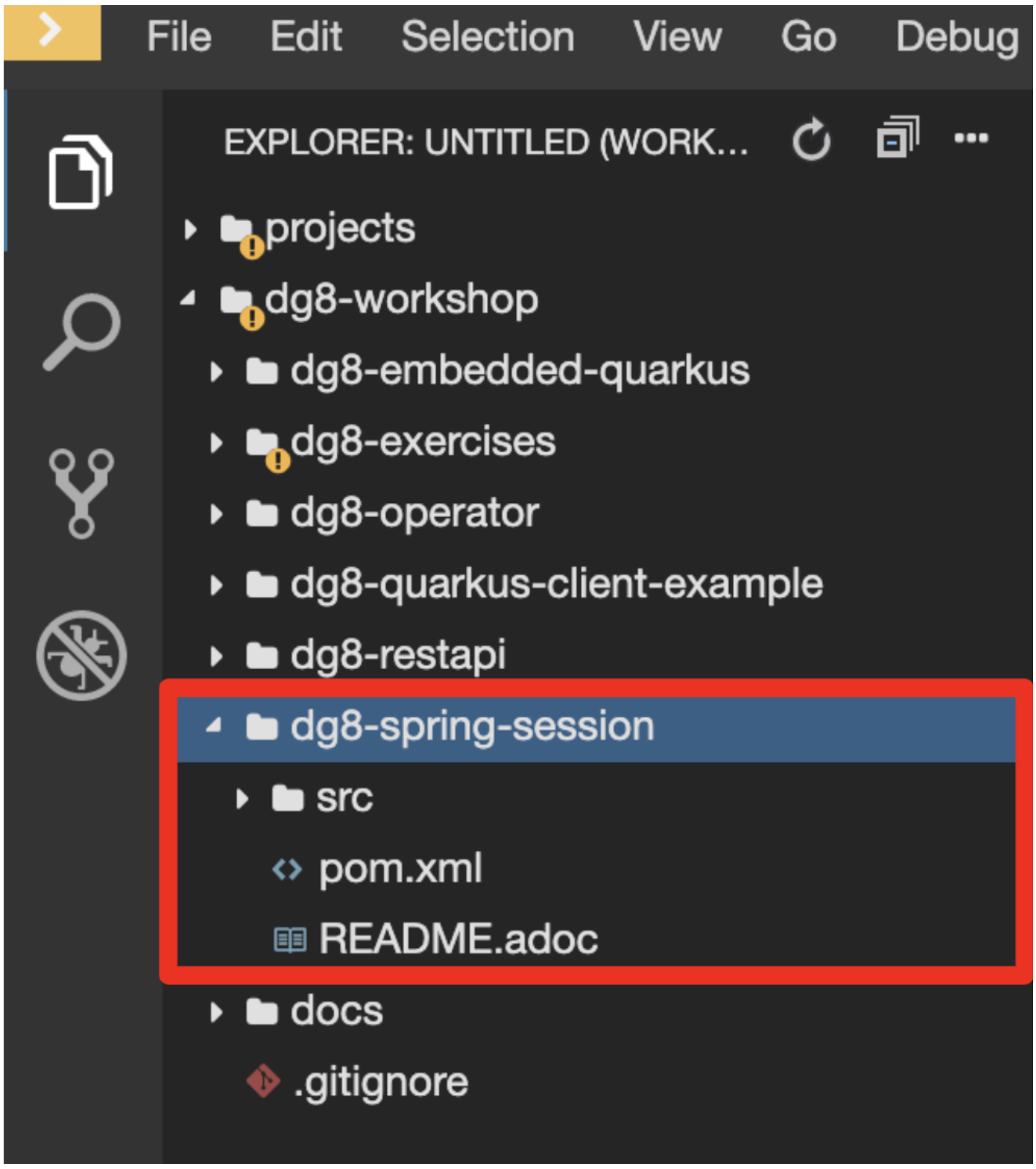
Externalizing Sessions

In this lab, we'll show how to externalize **HTTP Sessions** from **Spring Boot** to **Data Grid**. Data Grid Spring Session support is built on *SpringRemoteCacheManager* and *SpringEmbeddedCacheManager*, which means developers don't need to store HTTP session data in Data Grid manually for clustering the session data across multiple Spring Boot applications. Behind the scenes, Data Grid will autowire Spring Boot Session to distributed caches in Data Grid.

1. Developing EmbeddedCache Service

An embedded cache service allows Spring Boot applications to emded HTTP session data in an in-memory data storage when users invoke RESTful endpoints in a frontend web page. Lets's go through quickly how this works. We'll be using **Spring Session** combined with an **In-Memory Cache** provided by Data Grid.

Go to *Explorer: /projects* in *CodeReady Workspaces* Web IDE and expand **dg8-spring-session** directory.



There're a few interesting things what we need to take a look at in this Spring Boot application before we will develop it in CodeReady Workspaces.

This embeddedCache service is not using the default BOM (Bill of material) that Spring Boot projects typically use. Instead, we are using an Infinispan BOM provided by Red Hat that provides a high-level API to ensure compatibility between major versions of Data Grid. You can also enforce a specific version of Data Grid with the infinispan-bom module. Let's take a look at **infinispan-bom** to your `pom.xml` file as follows:

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.infinispan</groupId>
            <artifactId>infinispan-bom</artifactId>
            <version>${version.infinispan}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>javax.transaction</groupId>
            <artifactId>transaction-api</artifactId>
            <version>1.1</version>
        </dependency>
        <dependency>
            <groupId>javax.cache</groupId>
            <artifactId>cache-api</artifactId>
            <version>1.1.0.redhat-1</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

In order to use **Embedded Mode** in Spring Boot, `infinispan-spring-boot-starter-embedded` dependency is already pulled in your `pom.xml` file. This starter produces a `SpringEmbeddedCacheManager` bean by default:

```xml
<dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-spring-boot-starter-embedded</artifactId>
    <version>2.2.3.Final-redhat-00001</version>
</dependency>
```

Create an **InfinispanCacheConfigurer** bean to customize the cache manager. Open the Java class called `EmbeddedCacheConfig.java` in the `com.redhat.com.rhdg.config` package and copy below the `// TODO: Add cacheConfigurer method here` marker:

```java
@Bean
public InfinispanCacheConfigurer cacheConfigurer() {
   return manager -> {
      final org.infinispan.configuration.cache.Configuration ispnConfig = new ConfigurationBuilder()
            .clustering()
            .cacheMode(CacheMode.REPL_SYNC)
            .build();

      manager.defineConfiguration("sessions", ispnConfig);
      manager.getCache("sessions").addListener(new CacheListener());

   };
}
```

Copy below the `// TODO: Add globalCustomizer method here` marker to customize InfinispanGlobalConfigurer bean:

```java
@Bean
public InfinispanGlobalConfigurer globalCustomizer() {
   return () -> {
      GlobalConfigurationBuilder builder = GlobalConfigurationBuilder.defaultClusteredBuilder();
      builder.serialization().marshaller(new JavaSerializationMarshaller());
      builder.transport().clusterName("rhdg");
      builder.serialization().whiteList().addClass("org.springframework.session.MapSession");
      builder.serialization().whiteList().addRegexp("java.util.*");
      return builder.build();
   };
}
```

Finally, add the `@EnableInfinispanEmbeddedHttpSession` annotation to the `EmbeddedCacheConfig` class to enable Spring Cache support. When this starter detects the EmbeddedCacheManager bean, it instantiates a new `SpringEmbeddedCacheManager`, which provides an implementation of Spring Cache (https://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html).

Copy below the `// TODO: Add an Infinispan annotation here` marker:

```java
@EnableInfinispanEmbeddedHttpSession
```

Perfect! Now we have all the building blocks ready to use the cache. Let's start using our cache.

2. Deploying EmbeddedCache Service

Now we will build and deploy the project using the following command, which will use the maven plugin to deploy via CodeReady Workspaces Terminal:

```sh
mvn clean package spring-boot:repackage -f $CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-spring-session
```

Create a build configuration for your application using OpenJDK base container image in OpenShift:

```sh
oc new-build registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:1.5 --binary --name=cacheapp -l app=cacheapp
```

Start and watch the build, which will take about minutes to complete:

```sh
oc start-build cacheapp --from-file=$CHE_PROJECTS_ROOT/dg8-workshop-labs/dg8-spring-session/target/rhdg-0.0.1-SNAPSHOT.jar --follow
```

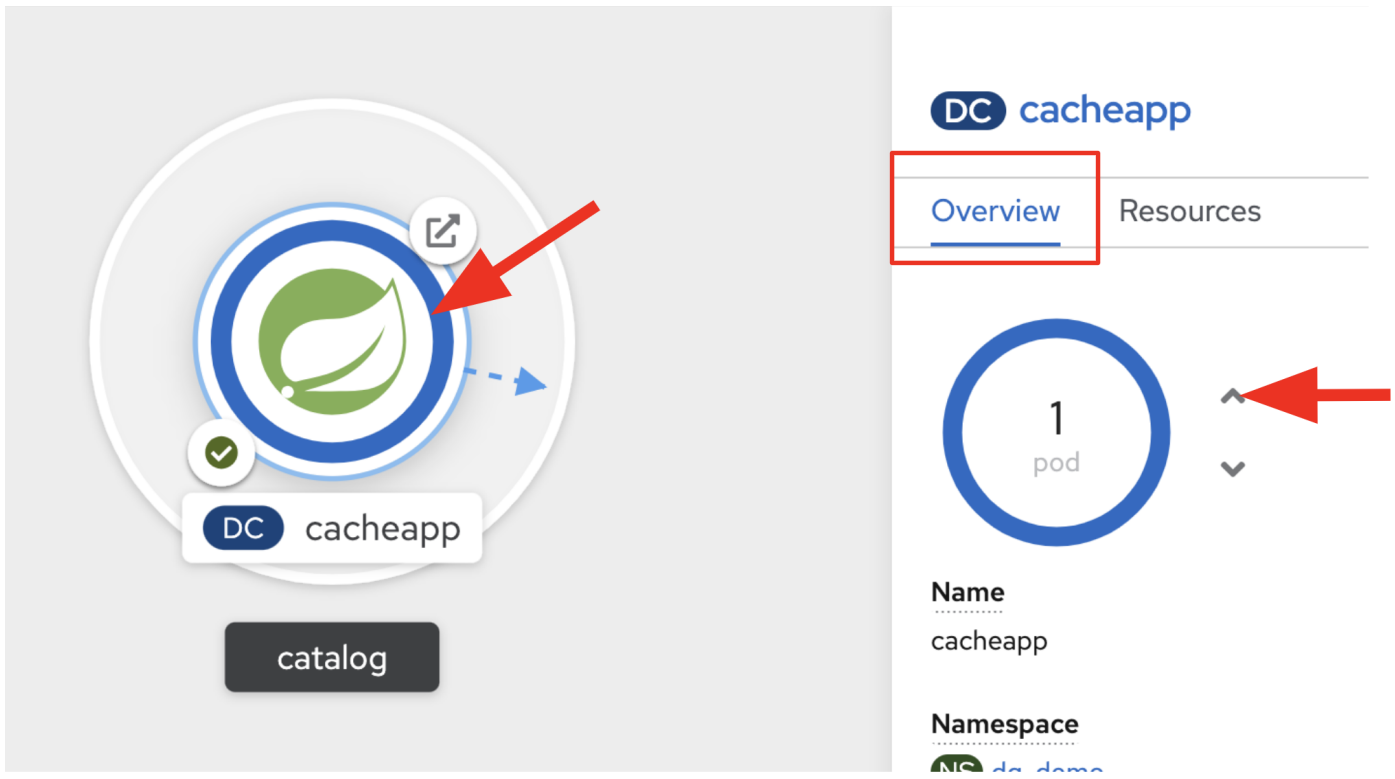
Deploy it as an OpenShift application after the build is done:

```sh
oc new-app cacheapp && oc expose svc/cacheapp && \
oc label dc/cacheapp app.kubernetes.io/part-of=catalog app.openshift.io/runtime=spring --overwrite
```

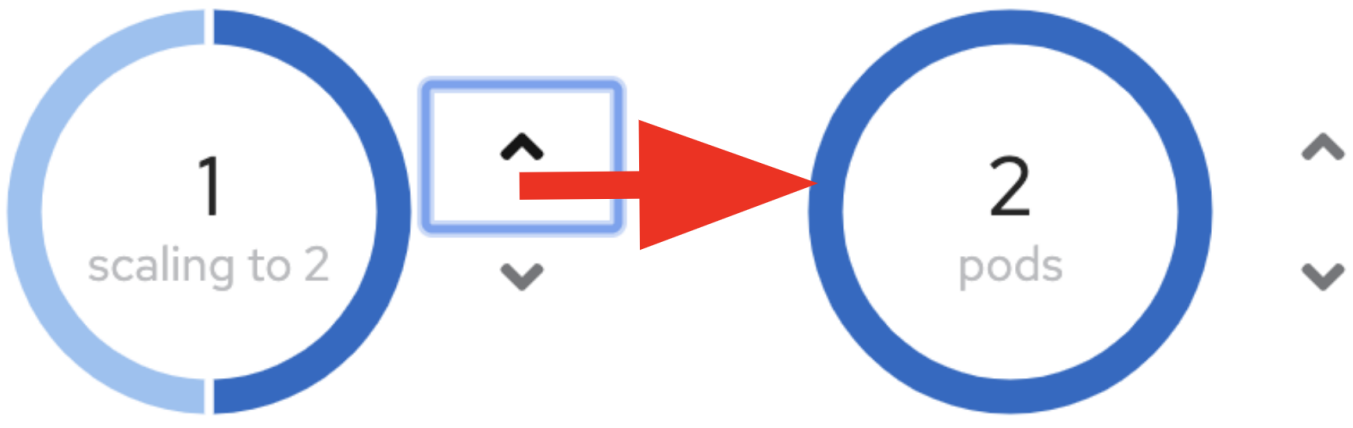
Finally, make sure it's actually done rolling out. Visit the Topology View (https://console-openshift-console.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com/topology/ns/user10-cache) for the cache service, and ensure you get the blue circles!

3. Testing EmbeddedCache Service

Let's scale up the cache service to make sure the clustered Spring applications refer to *Spring Session* in Data Grid. Click on `Up Arrow` once in *Overview* page:



Then you will see how the pod is scailing up:



Let's go externalizing Spring Session to Data Grid! Access the Cache Service UI (http://cacheapp-user10-cache.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com)!

Externalizing Spring Session to Data Grid

The Caching example demonstrates how to externalize Spring Session to Data Grid using SpringRemoteCacheManager or SpringEmbeddedCacheManager. It contains two components: the `session` service and `DataGrid`.

- The `session` service provides REST APIs to retrieve HTTP session if multiple Spring Boot applications can share the session using cache data in DataGrid.
- When the `session` service is invoked in Spring Boot app(A), it shows a created session data and an active cache value.
- Then clustered Spring Boot app(B) is invoked by the `session` service, it shows the same session data and active cache value in the app(A).
- When you click on `Clean the cache`, existing session data will be invalidated and the cache value will be removed too. New session and cache will be created in 5 sec.

Use the section below to invoke the `cache` service. Each call to the `cache` service records the time to complete the call. This shows the difference between retrieving a cached value and a value directly from `datagrid`.

Greeting service

Invoke the service | Clear the cache

Result:

```
Invoke the service to see the result.
```

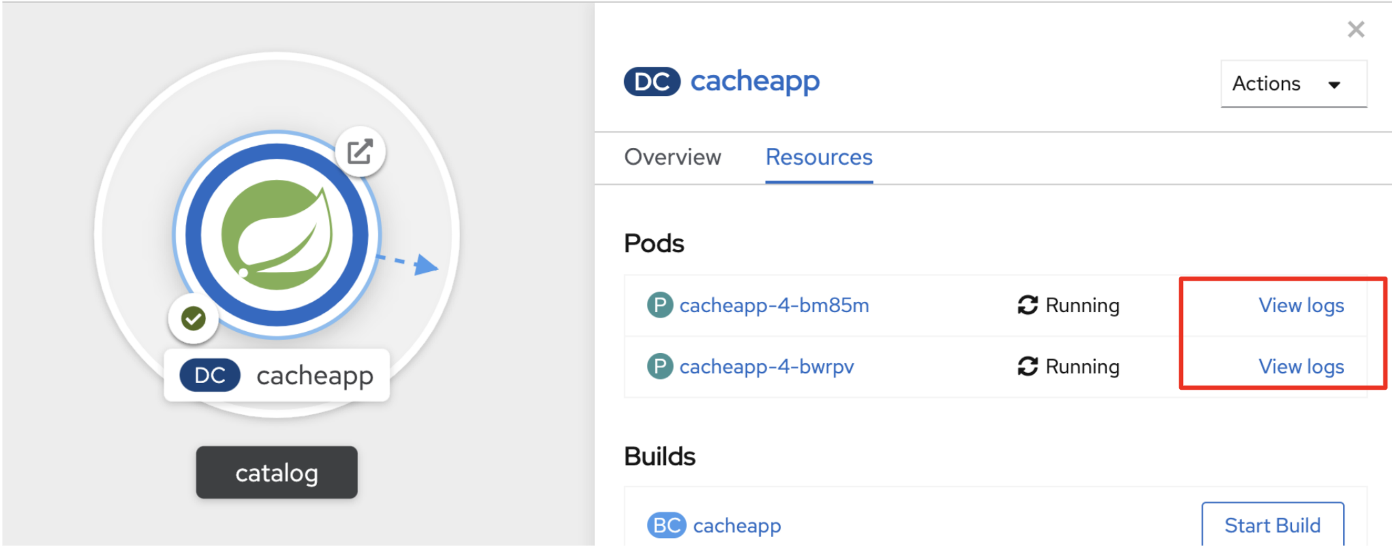
Click on `Invoke the service` then the **created** *Spring Session ID* is already stored at in-memory datagrid as **active** data in the *Result* box:

Open a new web browser window then access the the Cache Service UI (http://cacheapp-user10-cache.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com).

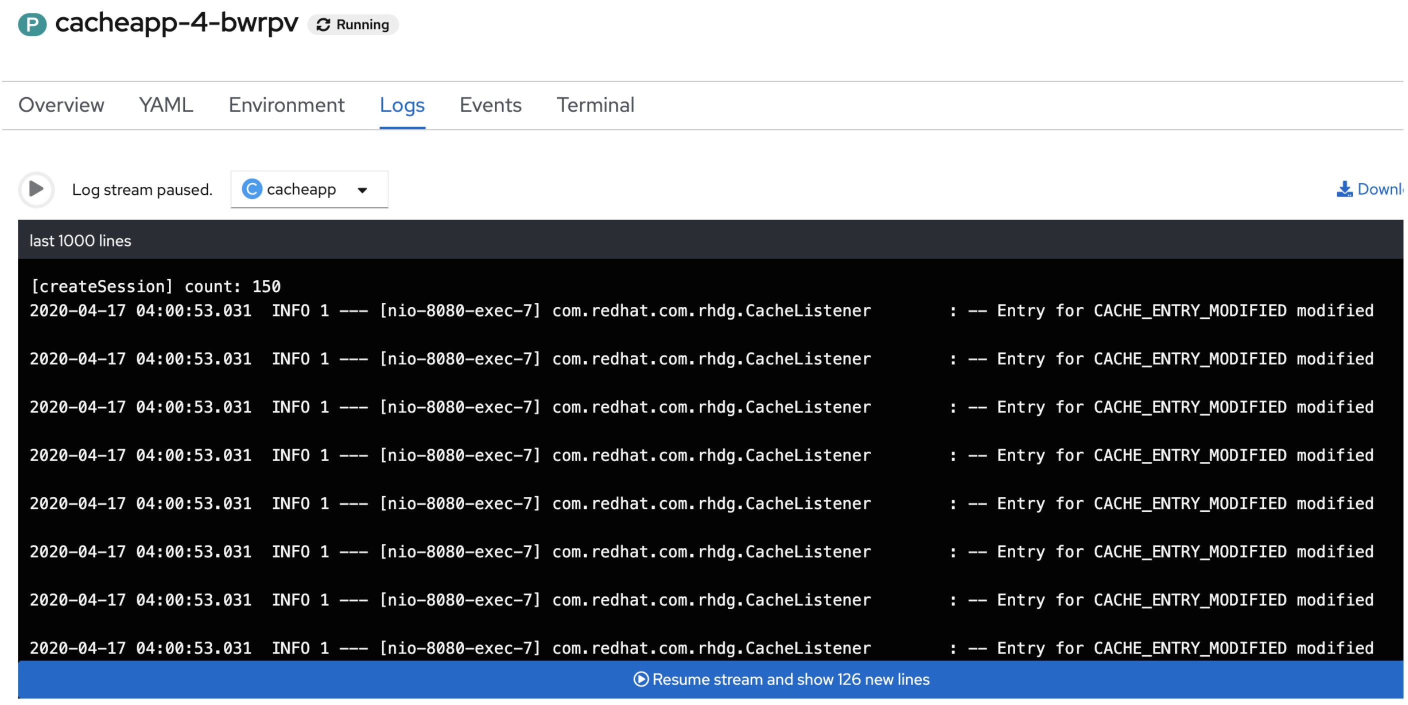
Click on `Invoke the service` once again then you will see the exact same *Spring Session ID* and *active data* but the **count** is increased to `2`. So two applications are clustered and refer to the *embedded Infinispan cache:

Go back to the **first** web browser then click on `Clear the cache`. Move to the **second** web browser then click on `Invoke the service`. You will see new **Session ID**, **active data** and the count is reset to **1** again:

Let's double-check if the Spring Session is clustered in the all running pods. Go back to the Topology View (https://console-openshift-console.apps.cluster-caba-d558.caba-d558.sandbox93.opentlc.com/topology/ns/user10-cache) and click on 'View logs' in the pods:



Now that we know how to react on changes in the cluster topology, we can also react to changes to the data within the cluster. The **CacheListener** separates the roles of our two pods such as putting data in the cache(-- *Entry for CACHE_ENTRY_MODIFIED created*) and showing the cache modifications(-- *Entry for CACHE_ENTRY_MODIFIED modified*):



We now have implemented Spring Session with embedded in-memory datagrid for clustering HTTP sessions across Spring Boot microservices. **Congratulations!**

You made it!

Conclusion

You made it ! Or you jumped to this section. Anyway, congratulations. We hope you enjoy this lab and learn some *stuff*. There is many other things about Red Hat Data Grid and Inifinispan that you can do and that was not illustrated here.

Don't forget Red Hat Data Grid can be used in many usecases:

- Red Hat Data Grid is a Cloud native caching system that you can embed in your application or use it as a remote server
- You can use it with the some of the well known frameworks and runtimes like Java, Node, C, C# etc.
- And it intergrates very well with Kubernetes/Openshift e.g. Operators, Observability etc.

As soon as you jump into the microservice, functions or cloud native applications, you will need a better application environment. Kubernetes/OpenShift are perfect weapon to build, deploy, bind, and manage your microservices.

If you want, and we hope so, to go further here are some references:

- Traditional zip deployments are available on the Customer Portal (https://access.redhat.com).
- The container distribution and operator are available in the Red Hat Container Catalog (https://catalog.redhat.com/software/containers/explore)
- Product documentation is available here (https://docs.redhat.com)
- Getting Started Guide that will get you running with RHDG 8 in 5 minutes.
- Migration Guide (https://access.redhat.com/documentation/en-us/red_hat_data_grid/8.0/html/data_grid_migration_guide/index)
- Starter Tutorials (https://github.com/redhat-developer/redhat-datagrid-tutorials)
- Supported Components (https://access.redhat.com/articles/4933371)
- Supported Configurations (https://access.redhat.com/articles/4933551)

Setting up your own environment

APPENDIX: Setting up my own machine for the lab

The labs are designed to run entirely on openshift, there is no mandatory requirement to install the following components. The following instructions are means of guidance for anyone who wants to try them out on their own machine.

*Recommended path is to use the Openshift RHMI/Integreatly environment provided in this workshop.

Java Development Kit

We need a JDK 8+ installed on our machine. Latest JDK can downloaded from:

- Oracle JDK 8 (http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)
- OpenJDK (http://openjdk.java.net/install/)

You can use either Oracle JDK or OpenJDK.

Apache Maven

You need Apache Maven 3.5+. If you don't have it already:

- Download Apache Maven from https://maven.apache.org/download.cgi.
- Unzip to a directory of your choice and add it to the `PATH`.

IDE

We recommend you use an IDE. You can use Eclipse, IntelliJ, VS Code or Netbeans.

No IDE ?

If you don't have an IDE, here are the steps to get started with Eclipse.

1. First download Eclipse from the download page (http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen1).
2. In the *Download Links* section, be sure to select the right version for your operating system. Once selected it brings you to a download page with a `Download` button.
3. Once downloaded, unzip it.
4. In the destination directory, you should find an `Eclipse` binary that you can execute.
5. Eclipse asks you to create a workspace.
6. Once launched, click on the *Workbench* arrow (top right corner).

Getting the code

```
git clone https://github.com/RedHat-Middleware-Workshops/dg8-workshop-labs.git
```

References

References