Adopt-a-JSR workshop

JCache

Introduction (10 mins)

JCache

JCache, apart from being one of the longest running JSRs (about 13 years from 2001 to 2014), is a Java API that provides a unified mechanism for interacting with various caching implementations. The operations provided by the API allow for a uniform way to access, update, create and remove entries from a cache.

Payara

Payara is a Java application server derived from the Glassfish code base. Support for JCache is provided to Payara by means of the Hazelcast JCache provider.

Prerequisites (20 mins)

Setup your IDE for Java EE development

Download Eclipse, IntelliJ or NetBeans with JavaEE support:

- Eclipse for Java EE developers package can be downloaded from http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/mars2;
- IntelliJ community edition can be downloaded from https://www.jetbrains.com/idea/download (unless you have license for the Ultimate edition that provides better support for Java EE development);
- NetBeans with Java EE support can be downloaded from https://netbeans.org/downloads/

Setup Payara server

1) Download Payara server full (latest version) from http://www.payara.fish/downloads

2) Extract the Payara server zip to a proper directory (e.g. D:\Software on Windows or /opt on Linux). We'll call that directory `$(PAYARA_HOME)`

3) Run the Java DB RDBMS that comes with Payara by executing the `$(PAYARA_HOME)/javadb/bin/startNetworkServer` script.

Java DB (also known as Apache Derby) is also shipped with Glassfish distributions. By default the Java DB server accepts connections on port 1527.

4) Start Payara server using the `asadmin` script from the `$(PAYARA_HOME)/bin` directory as follows:

```
asadmin start-domain
```

5) Verify that server is started by logging in the admin interface at http://localhost:4848/ (Payara uses a modified version of the Glassfish admin interface)

If the admin interface does not load the check the Payara logs at `$(PAYARA_HOME)/glassfish/domains/domain1/logs` errors during initialization.

6) Enable Hazelcast by running the following command in the `$(PAYARA_HOME)/bin` directory:

```
asadmin set-hazelcast-configuration --enabled=true --dynamic=true
```

Guide (60 mins)

Setup demo project: Guestbook

1) Clone the Guestbook project to a proper location:

```
git clone https://github.com/bgjug/jcache-workshop.git guestbook
```

2) Import the project as a Maven project in your IDE of choice (e.g. for Eclipse: File → Import → Existing Maven Projects)

3) Add Payara (Glassfish 4) deployment support for in your IDE.

For Eclipse add a new server from the Servers view in Eclipse (Window → Show View → Servers) by right clicking in the view and selecting **Glassfish 4** from the **Glassfish** category. As a name specify **Payara**, as a location specify **D:/software/payara-4.1.1.161.1/payara41/glassfish** and as a Java Development Kit specify JDK 8 (must be installed on your system and added as a Runtime environment from Window → Preferences in Eclipse). After you finish creating the server right click on it from the server view and select 'Add or Remove', select the 'guestbook' project for deployment and click **Finish**.

4) Verify that when the project is synchronized with the server you are able to launch it by navigating to the **Applications** tab in the Payara admin panel and clicking the 'Launch' action next to the deployed **guestbook** application.

Short description of the project

Sample app

The project that we are going to use for showcasing JCache is a very simple guestbook. All its features are available for logged in users only. So the first screen invites you to either log in or register. The guest book comes with four predefined users (nayden, misho, marto and mitya) with passwords matching their respective username. You can also add your own by clicking the Register button and filling out the registration form. Once logged in the user can see the list of all the

comments entered in the guest book. A comment has title, content and author. The user can then add a comment of their own, by clicking the *Add comment* button If the user has admin privileges (at the moment only nayden has), they can also delete comments by clicking on the *delete* link to the right of each message.

Implementation notes

We've used a combination of Java EE 7 and some Java EE 8 technologies to implement the guestbook so far.

Model layer

The modeling and persistence layer is implemented with JPA. There are just two entities - User and Comment. They are connected with OneToMany be-directional relationship - one user can publish multiple comments and the comment keeps track which is the user that posted it. There are just a few named queries for each JPA entity:

- We need to find a user by name and password upon login, that is why there is a special query for that
- We need to get all the comments in our guestbook when we login

The entity manager is initialized in a dedicated CDI producer - `EntityManagerProducer`.

```
@ApplicationScoped
public class EntityManagerProducer {

    @PersistenceUnit
    private EntityManagerFactory emf;

    @RequestScoped
    @Produces
    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }
}
```

Each time when we need an entity manager to be injected in our app, CDI will call the `getEntityManager()` method. Now we are safe to inject directly `EntityManager` in our beans, rather then going through getting the `EntityManagerFactory` first.

Business logic

The business logic is implemented in a few request scoped CDI beans. It is structured in a package per component way: there is a package for the comments and a package for the users. The `CommentsManager` is an interface holding our business logic. We have defined two implementations of this interface: one that uses directly JPA and the other one - using JCache. The application classes distinguish between the two using the qualifiers that are defined in the same package: `@JPA` and `@JCache` and are put on each one of them:

```
@RequestScoped
@JCache
public class JCacheCommentsManager implements CommentsManager {
  // implementation
}

@RequestScoped
@JPA
public class JCacheCommentsManager implements CommentsManager {
  // implementation
}
```

The interface has defined three business methods: `getAllComments()`, `submitComment()` and `deleteCommentById()`. As the latter two change the database, someone needs to start a transaction. Instead of bothering to do that by our own, we've used the `@Transactional` annotation coming from the JTA spec in Java EE 7

```
@Transactional
public Comment submitComment(Comment newComment) {
    em.persist(newComment);
    return newComment;
}

@Transactional
public void deleteCommentWithId(Long commentId) {
    final Comment comment = em.find(Comment.class, commentId);
    if (comment != null) {
        em.remove(comment);
    }
}
```

The users package contains the business classes dealing with users. Again, there is a `UserManager` request scoped bean, that tries to find a user in the DB and also adds a new user. This corresponds to login and register features of our guestbook. One very special class is the `UserContext`. It is session scoped, which means that an instance of it will be created in the beginning of the browser session and will be destroyed once that session is invalidated. So it is a perfect means to use that for keeping session information, such as whether the user is logged in and if yes, which is that user. For that we use the `currentUser` field. The class that handles logging in (we'll come to it in a minute) has to make sure that it initializes it once a user is successfully logged in. Then the other classes, which require information about the currently logged in user, can siJSmply look that up from the user context. Which, remember, is one and the same instance throughout the whole user session.

So, how does that logged in user lookup work? The naive way is to just inject the `UserContext` bean and call its `getCurrentUser()` method. Of course it will work, but there is even neater way - inject directly the user that is currently logged in, rather than calling the getter each time. It will again work with a CDI producer - make the `getCurrentUser()` produced that user:

```
@Produces
@LoggedIn
public User getCurrentUser() {
    return currentUser;
}
```

You maybe noticed the special `@LoggedIn` qualifier. We've added that so that we can distinguish between all the different types of users that we might want to produce and inject in our application. So, for example, if we want to later inject the admin user for some new feature, then we can add a new qualifier (e.g. @Admin) and use that at the injection point.

But let's get back to our current state of the guestbook. Now, if we need somewhere the current user, its injection is as simple as that:

```
@Inject
@LoggedIn
private User currentUser;
```

The frontend

We've chosen MVC 1.0 (JSR 371) to manage the connection between frontend and backend of our application. There's another workshop going through the new features of that, which you can check here.

There are a couple of controllers for each of our components. Let's start with the users. One of the controllers there manages login. When a GET request arrives at the *login* URI, the showLoginForm is called and it returns the string `"login.jsp"`. This tells MVC to look for that file in the WEB-INF/views folder of our application.

There are plenty of other combinations of return values (and types), view locations and view technologies that you may use in your application. It's a good practice when you pick one, to stick to it in your whole app

There's also a method that handles POST requests (`login()`). It receives the userName and password entries from the login form, as parsed by the MVC application. Then it tries to look for a user via the `UserManager`. If it finds one, it stores it in the `UserContext` and redirects to the comments page. Otherwise, it simply redirects to the login page, which will finally end in a GET request to the same controller.

There's nothing completely different in the other controller in the user package - `RegisterController`. Its GET method returns the register.jsp, which is then parsed on the server and rendered in the browser. The POST method is a bit different than the one in the `LoginController`. Its job is to get the data from the registration form, convert it to a user object and store that in the database. Also make sure that the entered data is valid and after that put the user in the UserContext. All the plumbing is done by the MVC framework. We only make sure to define the mapping in our `UserModel` class. There is also the validation check whether the entries in the *password* and *reenterPassword* fields match.

The comments component contains two controllers as well. The first one is responsible for returning the comments view and populating its backing model with the comments that are currently available in the database and with the currently logged in user:

```
@GET
public String showAllComments() {
    models.put("comments", commentsManager.getAllComments());
    models.put("user", currentUser);
    return "comments.jsp";
}
```

This data is then available via the expression language in the JSP itself:

```
<div class="logged-user">
  Hello, <c:out value="${user.firstName}"/>
</div>

<c:forEach items="${comments}" var="comment">
    <tr>
        <td><c:out value="${comment.title}"/></td>
        <td><c:out value="${comment.content}"/></td>
        <td><c:out value="${comment.byUser.firstName}"/>
            <c:out value="${comment.byUser.lastName}"/></td>
        <c:if test="${user.admin}"><td><a href="
comment/delete?commentId=${comment.id}">Delete</a></td></c:if>
    </tr>
</c:forEach>
```

The other method here is the one that is used to delete comment with a certain ID. It first makes sure that the user that performed the request has admin role.

The final controller (`NewCommentController`) is responsible for handling new comments in the guestbook. Its GET method returns the newComment.jsp form, while its POST method handles the submission itself.

What is particularly interesting about these controllers is the way they obtain the `CommentsManager`. As we've mentioned already - there are two implementations of this interface. In order to avoid ambiguities upon deployment, we need to specify at injection point which of them we want to use. At the moment we are using the JPA implementation in both controllers, as the other one is not ready yet.

```
@Inject
@JPA
private CommentsManager commentsManager;
```

Miscellaneous

There are some classes which functionality is not directly connected with any of the business components that we looked so far.

The security package contains a servlet filter class. Its responsibility is to intercept incoming requests to the `comment` URI and check whether there is a user logged in. If not, the request is redirected to the login page. Otherwise the request is passed through.

```
@Override
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
    if (userContext.getCurrentUser() != null) {
        chain.doFilter(request, response);
    } else {
        ((HttpServletResponse)response).sendRedirect("login");
    }
}
```

The test package contains a class that inserts test data in the database when the application is started by the server. This is where the initial users and comments are created, so that you are able to login and see them right after the initial deployment. It is implemented with a singleton Enterprise Java Bean, that is created upon startup, rather than upon first use:

```
@Singleton
@Startup
public class TestDataInserter {
}
```

When the EJB container instantiates and initialized the above class, it will call the method annotated with `@PostConstruct`. That is why we put there the initialization of our test data:

```
@PostConstruct
public void insertTestData() {
    // Test data initialization goes here
}
```

Enable JCache

- Cache::put
- Cache::get
- Cache::remove

Utilize additional JCache APIs

- EntryProcessor
- CacheEntryListeners
- ExpiryPolicy
- CacheWriter / CacheReader?/CacheLoader

Refactor project to use CDI

Summary

How many times faster is the application with JCache ?

References

JCache overview

- JSR 107: JCache - Java Temporary Caching API: https://jcp.org/en/jsr/detail?id=107
- Introduction to JCache JSR 107: https://dzone.com/articles/introduction-jcache-jsr-107
- Sneak peek into the JCache API: https://www.javacodegeeks.com/2015/02/sneak-peek-jcache-api-jsr-107.html
- JCache, why and how ?: https://vaadin.com/blog/-/blogs/jcache-why-and-how-
- JCache is Final! I Repeat: JCache is Final!
- Java Caching: Strategies and the JCache API
- How to speed up your application using JCache: https://www.jfokus.se/jfokus16/preso/How-to-Speed-Up-Your-Application-using-JCache.pdf
- After 13 years, JCache specification is finally complete: http://sdtimes.com/13-years-jcache-specification-finally-complete/
  .JCache support
- Hazelcast blogs (JCache category): http://blog.hazelcast.com/category/jcache/
- Hazelcast JCache implementation: http://docs.hazelcast.org/docs/3.3/manual/html-single/hazelcast-documentation.html#hazelcast-jcache-implementation
- Hazelcast 3.5 Manual: Introduction to the JCache API: http://docs.hazelcast.org/docs/3.5/manual/html/jcache-api.html
- Infinispan JCache support: http://infinispan.org/docs/7.0.x/user_guide/user_guide.html#_using_infinispan_as_a_jsr107_jcache_provider
- Infinispan JCache example: http://infinispan.org/tutorials/simple/jcache/
- Oracle Coherence JCache support: https://docs.oracle.com/middleware/1213/coherence/develop-applications/jcache_intro.htm#COHDG5778

- Ehcache JCache support: https://github.com/ehcache/ehcache-jcache
- Apache Ignite JCache provider: https://ignite.apache.org/use-cases/caching/jcache-provider.html
- Google App Engine support for JCache: https://cloud.google.com/appengine/docs/java/memcache/usingjcache
- Couchbase JCache Implementation Developer Preview 2: http://blog.couchbase.com/jcache-dp2
- Couchbase JCache implementation: https://github.com/couchbaselabs/couchbase-java-cache
- JCache (Payara 4.1.153): https://github.com/payara/Payara/wiki/JCache-(Payara-4.1.153)
- Spring JCache annotations support: https://spring.io/blog/2014/04/14/cache-abstraction-jcache-jsr-107-annotations-support .JCache & CDI
- Using JCache with CDI: http://www.tomitribe.com/blog/2015/06/using-jcache-with-cdi/
- High Performace Java EE with JCache and CDI: http://www.slideshare.net/Payara1/high-performance-java-ee-with-jcache-and-cdi
- Using the JCache API with CDI on Payara server: http://blog.payara.fish/using-the-jcache-api-with-cdi-on-payara-server