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
```

To verify Payara server is enabled you can again use the asadmin tool as follows:

```
asadmin list-hazelcast-members
```

Demo project (10 mins)

Setup demo project: Guestbook

1. Download the Guestbook project and extract it to a proper location
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 our 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 guestbook. 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` interface, implemented by two request scoped beans - one using JPA and the other - the cache. It has three operations: finding a user in the DB by name and by name and password and also adding a new user. The second operation corresponds to login and and the third to the register feature 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 simply 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.

What is particularly interesting about these controllers is the way they obtain the `UserManager`. 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 all controllers, as the other one is not ready yet.

```
@Inject
@JPA
private UserManager userManager;
```

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.

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
}
```

Tasks (50 mins)

Task 1: enable basic caching for the business components

Features to demonstrate (basic Cache operations)

- `CacheManager::createCache`

- `CacheManager::getCache`
- `Cache::put`
- `Cache::get`
- `Cache::remove`

First include the JCache API as part of your project's pom.xml as follows:

```
    <dependency>
         <groupId>javax.cache</groupId>
         <artifactId>cache-api</artifactId>
         <version>1.0.0</version>
         <scope>provided</scope>
   </dependency>
```

Our first task is to enable JCache for our comments. The key of the cache is the comment ID in our database (of type `Long`) and the value is the comment itself (of type `Comment`) We will initialize the cache instance in your JCache managers implementations. In order to do that, we will first inject the CacheManager provided by Payara:

```
@ApplicationScoped
public class JCacheCommentsManager implements CommentsManager {

    @Inject
    private CacheManager cacheManager;

}
```

As we want our comments cache to be automatically created (or obtained if it is already created) upon the comments manager initialization, we can put it in a `@PostConstruct` method. Remember that we should fallback to the JPA implementation if something is not present in the cache. So we will need to inject JPACommentsManager as well.

```
@ApplicationScoped
@JCache
public class JCacheCommentsManager implements CommentsManager {

    static final String COMMENTS_CACHE_NAME = "comments";

    @Inject
    @JPA
    private CommentsManager passThroughCommentsManager;

    @Inject
    private CacheManager cacheManager;

    private Cache<Long, Comment> cache;

    @PostConstruct
    public void getCommentsCache() {
        cache = cacheManager.getCache(COMMENTS_CACHE_NAME, Long.class, Comment.class);
        if (cache == null) {
            cache = cacheManager.createCache(
                    COMMENTS_CACHE_NAME,
                    new MutableConfiguration<Long, Comment>()
                            .setTypes(Long.class, Comment.class));
        }
    }
```

Now let's implement the methods of the `CommentsManager` interface using JCache. We'll start with getting all the comments. The JCache API is not really convenient for getting all the elements in the cache. It just gives you an iterator. We'll first check whether our cache is empty on not, by calling `hasNext()` on that iterator. If the cache is not empty, then we will return its content. Otherwise we'll go to the database, load all the comments present there into the cache and return them.

```
     @Override
     public List<Comment> getAllComments() {
        Iterator<Cache.Entry<Long, Comment>> commentsCacheIterator = cache.iterator();
        if (commentsCacheIterator.hasNext()) {
            // Converting iterator to Stream is a bit ugly, so doing it the Java 7 way
            List<Comment> foundComments = new ArrayList<>();
            while (commentsCacheIterator.hasNext()) {
                foundComments.add(commentsCacheIterator.next().getValue());
            }
            return foundComments;
        }

        List<Comment> dbComments = passThroughCommentsManager.getAllComments();
        dbComments.forEach(comment -> cache.put(comment.getId(), comment));
        return dbComments;
     }
```

Submitting a new comment means that it should be added to the cache as we as to the database:

```
    @Override
    public Comment submitComment(Comment newComment) {
       Comment submittedComment = passThroughCommentsManager.submitComment(
newComment);
       cache.put(submittedComment.getId(), submittedComment);
       return submittedComment;
    }
```

And finally if you want to delete a comment, you have to make sure that it is removed from the cache:

```
    @Override
    public void deleteCommentWithId(Long commentId) {
       passThroughCommentsManager.deleteCommentWithId(commentId);
       cache.remove(commentId);
    }
```

Now you are ready to change the `CommentsManager` implementation in both `CommentsController` and `NewCommentController`:

```
@Inject
@JCache
private CommentsManager commentsManager;
```

Redeploy your application and run it.

Do the same for the `UserManager`. There the Cache should be of type `Cache<String, User>` with the user name being the key and the corresponding user object the value. We'll implement the UserManager interface with JCache in mind:

```
@JCache
public class JCacheUserManager implements UserManager {

  static final String USERS_CACHE_NAME = "users";

  @Inject
  @JPA
  private UserManager passThroughUserManager;

  @Inject
  private CacheManager cacheManager;

  private Cache<String, User> cache;

  @PostConstruct
```

```
  public void createUserCache() {
      cache = cacheManager.getCache(USERS_CACHE_NAME);
      if (cache == null) {
          cache = cacheManager.createCache(
                  USERS_CACHE_NAME,
                  new MutableConfiguration<String, User>()
                          .setTypes(String.class, User.class));
      }
  }

    @Override
    public User getUser(String userName, String password) {
        User user = cache.get(userName);
        if (user != null) {
            if (user.getPassword().equals(password)) {
                return user;
            } else {
                return null;
            }
        }

        User userInDb = passThroughUserManager.getUser(userName, password);
        if (userInDb != null) {
            cache.put(userName, userInDb);
        }
        return userInDb;
    }

    @Override
    public void addUser(User newUser) {
        passThroughUserManager.addUser(newUser);
        cache.put(newUser.getUserName(), newUser);
    }

    @Override
    public User findUserByName(String userName) {
        User user = cache.get(userName);
        if (user != null) {
            return user;
        }

        User userInDb = passThroughUserManager.findUserByName(userName);
        if (userInDb != null) {
            cache.put(userName, userInDb);
        }
        return userInDb;
    }
}
```

And of course change the `UserManager` implementation to `@JCache` in the three controllers:

`LoginController`, `RegisterController` and `UserController`:

```
@Inject
@JCache
private UserManager userManager;
```

Task 2: Use the JCache annotations instead of manipulating the cache manually

Features to demonstrate (Cache interceptor annotations)

- `@CacheDefaults`
- `@CachePut`
- `@CacheResult`
- `@CacheKey`
- `@CacheValue`
- `@CacheResult`

Sometimes all you want to do is simply get a value from the cache, which key is passed as a parameter to your method. Or insert a key value pair, which happen to be your method parameters. Let's look for example at the `JCacheUserManager`. The `findUserByName()` method just takes the parameter and looks it up in the cache. If it is not there, then it goes to the pass through implementation. It doesn't do anything else apart from that.

For such situations the JCache specification has defined a group of annotations, which you can put on your methods to instruct the implementation to do the cache operations for you. First of all, let's declare the cache that will be used by our annotation based caching:

```
@CacheDefaults(cacheName = JCacheUserManager.USERS_CACHE_NAME)
public class JCacheUserManager implements UserManager {
```

Now it is time to tell JCache that our `findUserByName` method will look up a value in the cache and that it will look it up by key matching the only parameter. For that we use the `@CacheResult` and `@CacheKey` annotations:

```
@CacheResult
public User findUserByName(@CacheKey String userName) {
    return passThroughUserManager.findUserByName(userName);
}
```

As you might have noticed we have got rid of the direct call to the cache object - it is done by Payara for us. We only left the pass through call.

At the moment of writing of this hands on lab, it was impossible to combine `@CacheResult` methods with methods that work directly with the JCache API for one and the same cache. It resulted in `IllegalArgumentException`. Everything along with a workaround is explained in this gist

Next up, let's pretend that we badly need a new method in our `UserManager` - `adduser(String userName, User user)`. Let's add it to the interface and to both implementations. As long as `JCacheUserManger` is concerned, we'll implement it using `@CachePut`, `@CacheKey` and `@CacheValue` annotation and by just calling the pass through implementation:

```
interface UserManager {

    // ....
    void addUser(String userName, User newUser);
}

@JPA
@ApplicationScoped
public class JPAUserManager implements UserManager {

    @Inject
    private EntityManager em;

    // ....

    @Transactional
    @Override
    public void addUser(User newUser) {
        em.persist(newUser);
    }

    @Override
    @Transactional
    public void addUser(String userName, User newUser) {
        addUser(newUser);
    }
}

@ApplicationScoped
@JCache
@CacheDefaults(cacheName = JCacheUserManager.USERS_CACHE_NAME)
public class JCacheUserManager implements UserManager {

    static final String USERS_CACHE_NAME = "users";

    @Inject
    @JPA
    private UserManager passThroughUserManager;

    // ....

    @Override
    @CachePut
    public void addUser(@CacheKey String userName, @CacheValue User newUser) {
        passThroughUserManager.addUser(userName, newUser);
    }
}
```

Finally, let's look at the comments manager. There we have the `deleteCommentWithId` method, which is a perfect fit for the `@CacheRemove` annotation:

```
@CacheRemove
public void deleteCommentWithId(@CacheKey Long commentId) {
    passThroughCommentsManager.deleteCommentWithId(commentId);
}
```

Task 3: Expiry policy

Features to demonstrate

- `CacheConfiguration.setExpiryPolicyFactory()`

To supply an expiry policy for the comments cache refactor the `getCommentsCache()` in `JCacheCommentsManager` class as follows:

```
@PostConstruct
public void getCommentsCache() {
    cache = cacheManager.getCache(COMMENTS_CACHE_NAME, Long.class, Comment.class);
    if (cache == null) {
        cache = cacheManager.createCache(
                COMMENTS_CACHE_NAME,
                new MutableConfiguration<Long, Comment>()
                        .setTypes(Long.class, Comment.class)
                        .setExpiryPolicyFactory(
                                FactoryBuilder.factoryOf(
                                        new AccessedExpiryPolicy(
                                                new Duration(TimeUnit.MINUTES, 3)))));
    }
}
```

As you can see an expiry policy factory is used to specify the expiry policy. In that case we specify an expiry of three minutes for the cache entries upon creation.

Note that at this point you may add additional code to your comments JSP page in order to observe the currently set expiry policy for a cache. The JCache API itself provides limited capabilities to inspect the current expiry policy - we can have to store configuration for further statistics. We can, however, check that an access policy is in effect by retrieving a Cache configuration instance.

In order to check that an expiry policy is set we can add a `getStatistics()` default method to the `CommentsManager` interface:

```
    default Object getStatistics() {
        return new Object();
    }
```

Then we can provide implementation of the above method for the `JCacheCommentsManager` class:

```
public Object getStatistics() {
    return cache.getConfiguration(CompleteConfiguration.class);
}
```

After that, add a statistics instance to the model in the `showAllComments` method of the `CommentsController` class:

```
    models.put("statistics", commentsManager.getStatistics());
```

Finally, add a display in comments.jsp of whether an expiry policy is set (or not) by checking whether there is an expiry policy factory in place:

```
    <br /><br /><br /><span>Expiry policy is : ${statistics.getExpiryPolicyFactory()
!= null ? 'set' : 'not set'}</span>
```

In theory the above should be working once you deploy your application. However - we are not living in a perfect world and due to a bug in the Payara-Hazelcast bridge: https://github.com/payara/Payara/issues/318 we need to provide a workaround for directly placing custom `Comment` instances as values in the cache with causes `ClassNotFoundException`s to be thrown when working with the cache. For that reason the Payara team has provided a `PayaraValueHolder` class as part of the Payara utilities that must be included in the pom.xml of the project as follows:

```
<dependency>
    <groupId>fish.payara.extras</groupId>
    <artifactId>payara-embedded-all</artifactId>
    <version>4.1.152.1</version>
    <type>jar</type>
    <scope>provided</scope>
</dependency>
```

Now instead of `Comment` instances use `PayaraValueHolder` instances for the cache values. For instance the `JCacheCommentsManager` class looks like the following:

```
@RequestScoped
@JCache
public class JCacheCommentsManager implements CommentsManager {

    @Inject
    @JPA
    private CommentsManager passThroughCommentsManager;

    @Inject
    private Cache<Long, PayaraValueHolder> cache;

    @Override
```

```
    public List<Comment> getAllComments() throws ClassNotFoundException,
            IOException {

        Iterator<Cache.Entry<Long, PayaraValueHolder>> commentsCacheIterator = cache
                .iterator();
        if (commentsCacheIterator.hasNext()) {
            // Converting iterator to Stream is a bit ugly, so doing it the
            // Java 7 way
            List<Comment> foundComments = new ArrayList<>();
            while (commentsCacheIterator.hasNext()) {
                Comment comment = (Comment) commentsCacheIterator.next()
                        .getValue().getValue();
                foundComments.add(comment);
            }
            return foundComments;
        }

        List<Comment> dbComments = passThroughCommentsManager.getAllComments();

        dbComments.forEach(comment -> {
            try {
                cache.put(comment.getId(), new PayaraValueHolder(comment));
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        return dbComments;

    }

    @Override
    public Comment submitComment(Comment newComment) throws IOException {
        Comment submittedComment = passThroughCommentsManager
                .submitComment(newComment);
        cache.put(submittedComment.getId(), new PayaraValueHolder(
                submittedComment));
        return submittedComment;
    }

    @Override
    public void deleteCommentWithId(Long commentId) {
        passThroughCommentsManager.deleteCommentWithId(commentId);
        cache.remove(commentId);
    }

    @SuppressWarnings("unchecked")
    @Override
    public Object getStatistics() {
        return cache.getConfiguration(CompleteConfiguration.class);
    }
```

```
}
```

Utilize additional JCache APIs

Features to demonstrate (supporting JCache APIs):

- setting `Cache` configuration (such as `ExpiryPolicy` for cache entry expiry timeout)
- using `CacheEntryListeners` to intercept `Cache` lifecycle events (creation, removal, update or retrieval of entries)
- using `Entry` to provide composite atomic operations over mutable representations of `Cache` entries
- using `CacheWriter` / `CacheLoader` instances in order to establish cache integrations (for read-through/write-through) operations with other systems

CacheEntryListener

We have several concrete types of a CacheEntryListener:

- CacheEntryCreatedListener - for handling cache entry creation events
- CacheEntryExpiredListener - for handling cache entry expiration events
- CacheEntryRemovedListener - for handling cache entry removal events
- CacheEntryUpdatedListener - for handling cache entry update events

We'll create a cache entry event listener that logs information about entries added to the comments cache. In the `bg.jug.guestbook.comment` package add the `EntryCreatedLogListener` listener implementation as follows:

```
public class EntryCreatedLogListener implements
        CacheEntryCreatedListener<Long, PayaraValueHolder> {

            private final Logger LOGGER = Logger.getLogger(this.getClass().getName());

            int hits = 0;

            @Override
            public void onCreated(
                    Iterable<CacheEntryEvent<? extends Long, ? extends
PayaraValueHolder>> events)
                    throws CacheEntryListenerException {
                for (CacheEntryEvent<? extends Long, ? extends PayaraValueHolder>
event : events) {
                    hits++;
                    LOGGER.info("New entry value added in comments cache. Current
additions: " + hits);
                }
            }
        }
```

Note that the `Logger` *can be injected in a similar way using a CDI provider as the* `Cache` *instance is provided by a* `CommentsCacheProducer`*. For the purpose of simplicity we have provided the static approach.*

Register the listener to the cache configuration created in the `getCommentsCache()` method of the `CommentsCacheProducer` class as follows:

```
    cacheConfig.addCacheEntryListenerConfiguration(new
MutableCacheEntryListenerConfiguration<Long, PayaraValueHolder>(
                FactoryBuilder.factoryOf(EntryCreatedLogListener.class),null, true,
true));
```

To check if the listener is working properly add a new comment and observe the application logs or log again in the application - once comments are read via JPA for the first time they are added to the cache and creation events are triggered.

Entry

Next we will create an entry that appends the author of the comment in the comment contents before placing it into the cache. As you remember - due to a bug in the way value classes are deserialized we cannot access directly the `Comment` instance from the `PayaraValueHolder` instance holding it. For that reason we would change our `CommentsCacheProvider` implementation a bit by removing the `@Inject` annotation from the cache manager instance and initializing it using the following init block:

```
    {
        ClassLoader appClassLoader = getClass().getClassLoader();
        Config config = new Config();
        config.setClassLoader(appClassLoader);
        HazelcastInstance instance = Hazelcast.newHazelcastInstance(config);
        CachingProvider cp = HazelcastServerCachingProvider.createCachingProvider
(instance);
        cacheManager = cp.getCacheManager(cp.getDefaultURI(), appClassLoader);
    }
```

Create the `CommentAuthorEntry` entry in the `bg.jug.guestbook.comment` package as follows:

```
public class CommentAuthorEntry implements
        Entry<Long, PayaraValueHolder, Comment>, Serializable {

    private static final long serialVersionUID = 5257349479693469233L;

    @Override
    public Comment process(MutableEntry<Long, PayaraValueHolder> entry,
            Object... arguments) throws EntryException {

        try {
            Comment comment = (Comment) entry.getValue().getValue();
            comment.setContent(comment.getContent() + " ["
                    + comment.getByUser().getUserName() + "]");
            entry.setValue(new PayaraValueHolder(comment));
            return comment;
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

At the following to the `JCacheCommentsManager` methods for retriving all comments and adding a new comment right after a cache entry is created and placed in the cache:

```
    cache.invoke(comment.getId(), new CommentAuthorEntry());
```

Redeploy the application and check that once you log-in or add a comment the author is appended at the end of the comment.

CacheWriter/CacheLoader

Refactor project to use CDI

Features to demonstrate (JCache API annotations):

- annotations provided by the JCache API for use with DI frameworks such as CDI, Spring or Guice

We already started using CDI in order to create a producer for our comments Cache instance and inject it wherever needed. However the JCache API provides further a set of annotations that can be utilized by DI frameworks in order to provide direct caching support. These annotations include:

- @CacheDefaults - specifies configuration (such as the default cache name) for caching annotations at the method level
- @CacheResult - marks methods whose return value is cached
- @CachePut - marks methods where one of their parameters is stored in the cache
- @CacheRemove - marks methods whose invocations results in a cache entry removal
- @CacheRemoveAll - marks methods whose invocations results in all cache entry being removed
- @CacheKey - marks a method parameter as a cache key (typically used by a CacheKeyGenerator)
- @CacheValue - marks a method parameter as a cache value (typically for methods annotated with @CachePut)

Refactor the `submitComment` and `deleteCommentWithId` methods from the `JCacheCommentsManager` class to use the `@CachePut` and `@CacheRemove` annotations instead for interacting with the cache.

Summary

How many times faster is the application with JCache ? How can you manage your JCache caches and gather statistics on the various cache operations ?

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!: https://blogs.oracle.com/theaquarium/entry/jcache_is_final_i_repeat
- Java Caching: Strategies and the JCache API: https://www.linkedin.com/pulse/java-caching-strategies-jcache-api-somenath-panda
- 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