MicroProfile 1.2 Conference App Demo

1. Prerequisites

1. Install the Minishift binary[1]
2. Install Docker[2] if needed
3. Install VirtualBox[3] if needed
4. Clone the https://github.com/MicroProfileJWT/microprofile-conference.git project
5. cd microprofile-conference
6. Start minishift using the config-minishift.sh script in the microprofile-conference root directory
7. Configure your environment:

```
[starksm64-microprofile-conference 524]$ eval $(minishift oc-env)
[starksm64-microprofile-conference 525]$ type oc
oc is /Users/starksm/.minishift/cache/oc/v3.7.1/darwin/oc
[starksm64-microprofile-conference 526]$ eval $(minishift docker-env)
[starksm64-microprofile-conference 1568]$ oc login $(minishift ip):8443
 -u admin -p admin
Login successful.

You have access to the following projects and can switch between them
 with 'oc project <projectname>':

    default
    kube-public
    kube-system
  * myproject
    openshift
    openshift-infra
    openshift-node
```

[1] https://docs.openshift.org/latest/minishift/getting-started/installing.html
[2] https://docs.docker.com/install/
[3] https://www.virtualbox.org/wiki/Downloads

```
Using project "myproject".
```

8. open the minishift console uisng `minishift console`

2. Build and Deploy the Microservices

1. Build the services:

```
[starksm64-microprofile-conference 1559]$ mvn clean install -
DskipTests=true
[INFO] Scanning for projects...
[INFO]
 ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Conference
[INFO] Conference :: Bootstrap Data
[INFO] Conference :: Authorization
[INFO] Conference :: Session
[INFO] Conference :: Vote
[INFO] Conference :: Speaker
[INFO] Conference :: Schedule
[INFO] Conference :: Web
[INFO] Conference :: Start
...
[INFO]
 ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Conference ..................................... SUCCESS [
  0.693 s]
[INFO] Conference :: Bootstrap Data ..................... SUCCESS [
  2.593 s]
[INFO] Conference :: Authorization ...................... SUCCESS
 [ 12.907 s]
[INFO] Conference :: Session ............................ SUCCESS [
  8.802 s]
[INFO] Conference :: Vote ............................... SUCCESS
 [ 12.265 s]
[INFO] Conference :: Speaker ............................ SUCCESS [
  9.020 s]
[INFO] Conference :: Schedule ........................... SUCCESS
 [ 15.670 s]
```

```
[INFO] Conference :: Web ................................. SUCCESS
 [ 33.957 s]
[INFO] Conference :: Start ............................... SUCCESS [
  0.032 s]
[INFO]
 ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
 ------------------------------------------------------------------------
[INFO] Total time: 01:36 min
[INFO] Finished at: 2018-02-16T00:00:40-08:00
[INFO] Final Memory: 112M/1154M
[INFO]
 ------------------------------------------------------------------------
```

2. Install the services into the minishift environment using the cloud-deploy.sh script:

```
[starksm64-microprofile-conference 1571]$ ./cloud-deploy.sh
[INFO] Scanning for projects...
[INFO]
...
deployment "microservice-vote" created
service "microservice-vote" created
route "microservice-vote" exposed
[starksm64-microprofile-conference 1572]$ oc status
In project My Project (myproject) on server https://192.168.99.100:8443

http://microservice-authz-myproject.192.168.99.100.nip.io to pod port
 http (svc/microservice-authz)
  pod/microservice-authz-3124937629-wl8g7 runs example/microservice-
authz:latest

http://microservice-schedule-myproject.192.168.99.100.nip.io to pod port
 http (svc/microservice-schedule)
  pod/microservice-schedule-3040366544-n82zt runs example/microservice-
schedule:latest

http://microservice-session-myproject.192.168.99.100.nip.io to pod port
 http (svc/microservice-session)
  pod/microservice-session-1164112827-r8z9r runs example/microservice-
session:latest
```

```
http://microservice-speaker-myproject.192.168.99.100.nip.io to pod port
 http (svc/microservice-speaker)
  pod/microservice-speaker-2311407995-4mt9p runs example/microservice-
speaker:latest

http://microservice-vote-myproject.192.168.99.100.nip.io to pod port
 http (svc/microservice-vote)
  pod/microservice-vote-2774736211-wzzhz runs example/microservice-
vote:latest

View details with 'oc describe <resource>/<name>' or list everything
 with 'oc get all'.
[starksm64-microprofile-conference 1573]$
```

3. Update the web-application/src/main/local/webapp/WEB-INF/conference.properties service URLs to use the value for `minishift ip` in your environment. In my environment 192.168.99.100 is the IP address. Globally replace 192.168.99.100 with whatever is returned in your minishift setup.
4. Run the web application front end

```
mvn package tomee:run -pl :web-application -DskipTests
[INFO] Scanning for projects...
[INFO]
[INFO]
 ------------------------------------------------------------------------
[INFO] Building Conference :: Web 1.0.0-SNAPSHOT
[INFO]
 ------------------------------------------------------------------------
[INFO]
...
miTargets] to [true] as the property does not exist.
INFO - Starting ProtocolHandler [http-nio-8080]
INFO - Starting ProtocolHandler [ajp-nio-8009]
INFO - Server startup in 3177 ms
```

5. Open the web application http://localhost:8080/

3. Code Walkthrough

In this section we take a look at the code behind the Microprofile features in use in the conference application.

3.1. MP-JWT

The JWT RBAC for MicroProfile[4](MP-JWT) feature defines how JSON web tokens(JWT) may be used for authentication and role based authorization. The MP-JWT feature also defines an API for accessing the claims associated with JWTs. In the conference application demo, the microservice-session uses the JWT groups claim and a custom application claim. The following code snippet demonstrates the MP-JWT API.

Code from: microservice-session/src/main/java/io/microprofile/showcase/session/SessionResource.java

```
import org.eclipse.microprofile.jwt.JsonWebToken;


@ApplicationScoped
public class SessionResource {


    /**
     * The current MP-JWT for the authenticated user
     */
    @Inject
    JsonWebToken jwt; ❶

...

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Timed
    public Collection<Session> allSessions(@Context SecurityContext
 securityContext) throws Exception {
        requestCount.inc();
        if (jwt == null) { ❷
            // User was not authenticated
            System.out.printf("allSessions, no token\n");
            return Collections.emptyList();
        }
        String userName = jwt.getName();
```

[4] https://github.com/eclipse/microprofile-jwt-auth

```
        // Use the isUserInRole of container to check for VIP role in
the JWT groups claim
        boolean isVIP = securityContext.isUserInRole("VIP"); ❸
        System.out.printf("allSessions(%s), isVIP=%s, User token: %s\n",
userName, isVIP, jwt);
        // Check if the user has a session_time_preference custom claim
in the token
        Optional<String> sessionTimePref =
jwt.claim("session_time_preference"); ❹
        if(sessionTimePref.isPresent()) {
            // Create a session filter for the time preference...
        }

        // If the user does NOT have a VIP role, filter out the VIP
sessions
        Collection<Session> sessions;
        if (!isVIP) { ❺
            sessions = sessionStore.getSessions()
                .stream()
                .filter(session -> !session.isVIPOnly())
                .collect(Collectors.toList());
        } else {
            sessions = sessionStore.getSessions();
        }
        return sessions;
    }
```

❶ Injection of the MP-JWT token as a JsonWebToken interface.

❷ If there is no token, return an empty collection of sessions

❸ Check for a VIP role in the token using the container's isUserInRole(String) method. This internally maps to the token's MP-JWT defined group claims.

❹ Illustrates programatic lookup of a custom claim not defined by the MP-JWT spec. In this example the session results would be filtered to only return those matching the session time of day preference.

❺ This code block makes a check of the incoming MP-JWT token to see if it has a group claim that contains the VIP value. If the VIP claim does not exist, the sessions are filtered to remove those that the isVIPOnly property. Otherwise, all sessions are returned if the token has the VIP group.

3.2. MP-Configuration

The Microprofile config[5] (MP-Config) supports injection and programtic lookup of external configuration information via a common API. The MP-Config spec defines 3 common configuration sources: * System environment variables * System properties * A META-INF/microprofile-config.properties

SPIs are defined for adding configuration sources as well as for converting from string to arbitrary types.

The microprofile conference app makes use of injection of META-INF/microprofile-config.properties, environment variables, and the conversion SPI. The first code snippet we will look at injects a value from the bundled META-INF/microprofile-config.properties as a `java.security.PrivateKey`. The `AuthzResource` from the microservice-authz project shows the injection:

Code from: microservice-authz/src/main/java/io/microprofile/showcase/tokens/AuthzResource.java,PrivateKeyConverter.java

```
import java.security.PrivateKey;

@ApplicationScoped
public class AuthzResource {

/**
 * An example of injecting a custom property type
 */
@ConfigProperty(name="authz.signingKey") ❶
@Inject
private PrivateKey signingKey; ❷

...
```

```
# The META-INF/microprofile-config.properties entries
authz.signingKey=/privateKey.pem ❶
```

[5] https://github.com/eclipse/microprofile-config

```
import java.security.PrivateKey;

import org.eclipse.microprofile.config.spi.Converter;

import static io.microprofile.showcase.tokens.TokenUtils.readPrivateKey;

/**
 * A custom configuration converter for {@linkplain PrivateKey}
 injection using
 * {@linkplain org.eclipse.microprofile.config.inject.ConfigProperty}
 */
public class PrivateKeyConverter implements Converter<PrivateKey> { ❶
    /**
     * Converts a string to a PrivateKey by loading it as a classpath
 resource
     * @param s - the string value to convert
     * @return the PrivateKey loaded as a resource
     * @throws IllegalArgumentException - on failure to load the key
     */
    @Override
    public PrivateKey convert(String s) throws IllegalArgumentException
 {

        PrivateKey pk = null;
        try {
            pk = readPrivateKey(s);
        } catch (Exception e) {
            IllegalArgumentException ex = new
 IllegalArgumentException("Failed to parse ");
            ex.initCause(e);
            throw ex;
        }
        return pk;
    }
}
```

❶ The config property name reference to match against a config source.

❷ The custom value PrivateKey value injection site.

❶ The mapping from the referenced "authz.signingKey" name to a string value in the standard META-INF/microprofile-config.properties.

❶ The custom converter implementation that takes the input string value and transforms it into a PrivateKey by loading it as a resource from the classpath.

A further example usage of the MP-Config will be seen in the next section on the health check feature.

3.3. MP-Health

The Microprofile health check[6] (MP-Health) feature allows on to define application health check endpoints as commonly used in cloud environment to validate avaiability and liveness. The MP-Health feature supports this along with an ability to define a JSON payload that can be used to convey additional information.

The following microservice-session MP-Health code snippet shows an example health implementation that makes use of the MP-Config API to inject configuration that is used during construction the health response.

Code from: microservice-session/src/main/java/io/microprofile/showcase/session/SessionCheck.java

```
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.eclipse.microprofile.health.Health;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;

@Health ❶
@ApplicationScoped
public class SessionCheck implements HealthCheck { ❷
    @Inject
    private SessionStore sessionStore;
    @Inject
    @ConfigProperty(name = "sessionCountName", defaultValue =
 "sessionCount") ❸
    private String sessionCountName;
    @ConfigProperty(name = "JAR_SHA256") ❹
    @Inject
    private String jarSha256;

    @Override
    public HealthCheckResponse call() { ❺
        return HealthCheckResponse.named("sessions-check")
```

[6] https://github.com/eclipse/microprofile-health

```
            .withData(sessionCountName,
 sessionStore.getSessions().size()) ❻
            .withData("lastCheckDate", new Date().toString())
            .withData("jarSHA256", jarSha256)
            .up()
            .build();
    }
}
```

❶ The annotation marking the bean as a health check endpoint.

❷ The `HealthCheck` interface the endpoint implements to provide the health callback.

❸ An example of externalizing a data label used in health check response whose value is defined in the application META-INF/microprofile-config.properties.

❹ An example of injection of a config value whose source is an environment variable that is defined in the microservice-session openshift deployment descriptor.

❺ The `HealthCheck` call endpoint that returns the `HealthCheckResponse`.

❻ The various withData calls add labelled values, including the injected config values, to the JSON payload.

3.4. MP-Metrics

The Microprofile metrics[7] (MP-Metrics) feature aims to provide a unified way for Microprofile services to export Monitoring data via common API.

Code from: microservice-session/src/main/java/io/microprofile/showcase/session/SessionResource.java

```
import org.eclipse.microprofile.metrics.Counter;
import org.eclipse.microprofile.metrics.Histogram;
import org.eclipse.microprofile.metrics.Metadata;
import org.eclipse.microprofile.metrics.MetricRegistry;
import org.eclipse.microprofile.metrics.MetricType;
import org.eclipse.microprofile.metrics.annotation.Metric;
import org.eclipse.microprofile.metrics.annotation.Timed;
```

[7] https://github.com/eclipse/microprofile-metrics

```
@ApplicationScoped
public class SessionResource {

    @Inject
    @Metric(name = "requestCount", description = "All JAX-RS request
 made to the SessionResource",
            displayName = "SessionResource#requestCount") ❶
    private Counter requestCount;

    /**
     * The application metrics registry that allows access to any metric
 to be accessed/created
     */
    @Inject
    private MetricRegistry metrics; ❷

    @PostConstruct
    void init() {
        Collection<Session> sessions = sessionStore.getSessions();
        System.out.printf("SessionResource.init, session count=%d\n",
 sessions.size());
        // Create a histogram of the session abstract word counts
        Metadata metadata = new
 Metadata(SessionResource.class.getName()+".abstractWordCount",
 MetricType.HISTOGRAM);❸
        metadata.setDescription("Word count histogram for the session
 abstracts");
        Histogram abstractWordCount = metrics.histogram(metadata);❹
        for(Session session : sessions) {
            String[] words = session.getAbstract().split("\\s+");
            abstractWordCount.update(words.length);❺
        }
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Timed❻
    public Collection<Session> allSessions(@Context SecurityContext
 securityContext) throws Exception {
        requestCount.inc();❼
...
    }
```

```
    @GET
    @Path("/{sessionId}")
    @Produces(MediaType.APPLICATION_JSON)
    @Timed❽
    public Response retrieveSession(@PathParam("sessionId") final String
 sessionId) throws Exception {
        requestCount.inc();❾
        ...
    }


    @GET
    @Path("/{sessionId}/speakers")
    @Produces(MediaType.APPLICATION_JSON)
    @Timed❿
    public Response sessionSpeakers(@PathParam("sessionId") final String
 sessionId) throws Exception {
...
    }
```

❶ Define a `Counter` type metric named requestCount.

❷ Injection of the `MetricRegistry` interface allows for programmatic creation and lookup of metrics as will be done in init().

❸ Sets up the metadata for an abstractWordCount metric of type Histogram.

❹ The actual creation of the Histogram metric via the injected MetricRegistry instance.

❺ Population of the abstractWordCount from the various session abstracts.

❿ The allSessions, retrieveSession and sessionSpeakers endpoint methods are annotated with @Timed to indicate that the MP-Metrics layer should intercept the method invocations and create statistics for them.

❾ Programmatic updates of the injected requestCount metric are seen in the allSessions and retrieveSession endpoint methods.