1 Introduction

1.1 Rationale

JEE7 has come a long way since the introduction of the Java Enterprise platform in the 1990s. In responses to development in the industry and open source projects, it moved from a mainly monolithic "one-all" solution, it has become a set of specifications that can be used individually, either as part of a larger JEE container (like Websphere, Jboss, Glassfish, Weblogic) or standalone in a Java Standard Edition solution.

The Java Enterprise adopted OSGi as a possible solution some years ago and OSGi technology is used or can be used with some of the JEE containers. However, OSGi is often seen as an enabling technology for the container and not as a starting point for development.
If you try to get various JEE technologies working together in an OSGi framework, your often end up in a clutter of dependencies between projects and versions that are hard to untangle. As a result you end up with a large amount of bundles in your target environment. The Apache Karaf project tries to handle/solve this for you, but its scope is much broader than the JEE standards.

1.2 Purpose

The purpose of the JEE extender is limited, but as a result relatively simple:

- It focuses on JEE7 standards only.
- It only uses OSGi services to couple various standards together.
- It does not introduce additional interfaces for the user.
- It discourages/disables JEE techniques that are possible harmful or may lead to unclear results in OSGi environments.

It brings the following JEE7 standards together on a standard OSGi framework:

- JPA 2.1. Via a bridge to a standard JEE persistence provider implementation exported as service. An example for eclipselink is provided.
- JTA 1.1. Using an external or own implementation of a JEE transaction manager.
- CDI 1.2. Using an OSGi bridge to the Weld CDI reference implementation that allows importing and exporting OSGi services via CDI-enabled bundles.
- JSF 2.2. Currently using the Mojarra JSF reference implementation and tested with Primefaces.

All code is maintained on https://github.com/arievanwi/osgi.ee

2 Installation

2.1 Pre-requirements

- Java 8.

- The latest eclipse version with the functionality for eclipse plugin development enabled. For example the eclipse version for JEE developers.

2.2 Bundles

Next to this bundles are needed for:

- The OSGi framework (obvious).
- Various JEE APIs (required).
- An OSGi service component runtime like Felix SCR (required).
- An OSGi configuration manager implementation (optional, but likely to be necessary).
- A JPA provider (in case JPA is used).
- Weld (in case CDI is used).
- Web extender and JSF bundles in case JSF is used.

A full excerpt of all these dependencies can be extracted from the repository project "Runtime":

- Check-out and import the project "Runtime".
- Open the target definition file "Target.target".
- Press "Set as target platform".

As a result the OSGi target platform is switched to this new configuration.

The extender functionality consists of the following bundles:

- datasource.factory. Bundle that is able to create javax.sql.DataSource services from configuration admin information. See 3.3 for more information.
- eclipselink.extender. Bundle that extends the eclipselink JPA provider to export a PersistenceUnitProvider to the OSGi service registry. See 3.2 for more information.
- osgi.ee.extender.jpa. Extender bundle that registers entity managers for bundles containing JPA persistence unit definitions. See 3.2 for more information.
- osgi.ee.extender.cdi. Extender bundle that processes CDI beans from bundles that need it. See 6 for more information.
- osgi.ee.extender.web. Web extender bundle. An implementation of an OSGi web extender as described in compendium chapter 128. See for more information 5.

3 JPA

3.1 JPA interfaces

JPA works around the following interfaces:

- javax.persistence.spi.PersistenceProvider, further called persistence provider.
- javax.persistences.EntityManagerFactory, further called the entity manager factory.
- javax.persistence.EntityManager, further calls the entity manager.

The *entity manager factory* is the central interface here and represents one persistence

unit, normally specified in the META-INF/persistence.xml file of a bundle. An entity manager factory is constructed by a *persistence provider*, which is a vendor provided implementation (eclipselink, hibernate) of the interface. The persistence provider interface has methods to construct an entity manager factory from the details from the persistence unit definition.
An entity manager is the unit of work used by the application: it provides methods to update/insert/query the objects stored via the persistence unit.

In normal EE environments, persistence providers are created via the Java service provider solution, entity manager factories are handled by the container and entity managers are constructed where needed. This all doesn't work in a plain OSGi environment.

3.2 Solution for OSGi

Needed bundles: osgi.ee.extender.jpa and eclipselink.extender.

The OSGi enterprise specification chapter 127 gives a solution how JPA should be used in OSGi. However, it doesn't really use the application developer as main starting point, since its endpoint is an entity manager factory while an application programmer basically uses an entity manager. Therefore, the JPA extender bundle has some additional logic to bridge the additional gap between the application developer and the specification. Globally, it works as follows:

- The extender bundle tracks persistence provider implementations that are registered in the OSGi services registry.
- The extender bundle tracks bundles that indicate the presence of one or more persistence units via the Meta-Persistence bundle header (according to the OSGi enterprise specification).
- It creates entity manager factory instances for the persistence units that can be constructed with the available persistence providers and registers them as service for the bundle specifying the persistence unit.
- It creates thread-local entity manager instances on demand to service the application. The entity manager can be accessed via an entity manager service that is registered for a persistence unit.

This all sounds a little complex (and maybe it is), but from an application point of view it means that you can just reference an entity manager service from you application and use it. Example:
Assume that you defined a persistence unit named “Orders” that allows access to Order instances and you want to access those orders from any other bundle. This can be done using service component annotations as follows:

```
@Component
public class OrderDao {
  private EntityManager entityManager;

  @Reference(target = "(osgi.unit.name=Orders)")
```

```
  void setEntityManager(EntityManager m) {
    this.entityManager = m;
  }

  public List<Order> getOrders() {
    TypedQuery<Order> query = entityManager.createQuery(
      "select o from Order", Order.class);
    return query.getResultList();
  }
}
```

Note that the property "osgi.unit.name" must be used/filtered to reference the correct persistence unit. Otherwise, you may just end up with a different persistence unit entity manager.

3.3 Persistence unit data sources

Persistence units need to be defined as specified in the OSGi enterprise specification. This means that a "Meta-Persistence" header must be added to a bundle to declare it persistence units definition files.

Persistence units use data sources to connect to a database. This can be done either via the non-jta-datasource/jta-datasource elements or by providing the database properties directly in the persistence description file. The second solution should not be used. In stead a data source OSGi service should be created and used.

3.3.1 Creating a datasource service

javax.sql.DataSource is an interface and therefore can easily be exported as OSGi service. Normally, this is done using a JDBC connection, optionally added with connection pooling, etc. Base of this all is a JDBC driver that is provided by your database supplier.

The OSGi enterprise specification chapter 125 specifies how to get a data source using this specification. However, in practice no-one implements this chapter and we are just left with a JDBC database driver for our vendor.

The bundle datasource.factory provides a solution for creating a (pooled) data source via a configuration manager and publishing it in the service registry. This is done via the following configuration:

- A factory pid of "datasource" or "XAdatasource" for respectively a normal datasource or a datasource that is able to interact with a transaction manager. The use depends on your persistence unit definition:
  - If your transaction type is JTA and you therefore use the jta-data-source definition in your persistence file, a XAdatasource is needed.
  - Otherwise, you can just use "datasource".
- jdbc.driver, jdbc.user, jdbc.password, jdbc.url. Indicate the JDBC parameters for the data source. Standard convention.

- pool.idle.min, pool.idle.max, pool.active.max, pool.wait are the connection pool parameters for minimum idle connection, maximum idle connections, maximum active at the same time and the wait time for a connection to become available.
- validation.query and validation.timeout specify the validation query and the validation query timeout.

When using the org.avineas.cm.persister bundle as back-end for persistence storage, a configuration could look as follows:

```
# Type of service, either "XAdatasource" or "datasource"
ds.1..service.factoryPid = XAdatasource
# JDBC configuration.
ds.1..jdbc.url=jdbc\:oracle\:thin\:@localhost\:1521\:XE
ds.1..jdbc.driver = oracle.jdbc.driver.OracleDriver
ds.1..jdbc.user=oratest
ds.1..jdbc.password=orapassword
# Property set on the service to find it back.
ds.1..name = Orders
```

Note that properties not mentioned above are just copied to the service registration and therefore can be used to filter the service.

As indicated above, the driver must be specified in the JDBC parameters. However, drivers are not included with the bundle and must be separately attached via a fragment bundle. The actions to take are as follows:

- Create a fragment project for your JDBC driver(s):
  - New project → Plugin development → new fragment project.
  - MANIFEST → Overview → Host plugin: datasource.factory.
- Put your driver jar somewhere in the project.
- Add the jar to your bundle class path:
  - MANIFEST → Runtime → Classpath.

3.3.2 Using a datasource service for a persistence unit

To reference a data source service from a persistence unit descriptor file, you can just reference a normal or XA datasource in the respective elements in the persistence.xml file. This is done by using the JNDI OSGi reference URL format as described in the OSGI enterprise specification chapter 126. Although the use of JNDI itself is strongly discouraged, the extender functionality accepts the osgi:service JNDI lookup format for datasources in persistence description files. This format is as follows:

```
osgi:service/javax.sql.DataSource/<filter>
```

where:

- osgi:service indicates a service reference.
- javax.sql.DataSource indicates the interface to reference.
- <filter> is a standard OSGi filter.

To reference a datasource with name “OrdersDS”, the reference would become:

```
osgi:service/javax.sql.DataSource/(name=OrdersDS)
```

A full persistence.xml for our orders persistence unit could become:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="Orders" transaction-type="JTA">
    <jta-data-source>
      osgi:service/javax.sql.DataSource/(name=OrdersDS)
    </jta-data-source>
    <class>orders.objects.Order</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
  </persistence-unit>
</persistence>
```

3.3.3 Pre-processing persistence unit definitions

During development you want to add additional logging, etc. to persistence units definitions, or add some other options. In practice this means that you need to edit the persistence.xml files and change them back later. Using the JPA extender, it is possible to transform a persistence.xml file before it is used. This is done via a system property "osgi.jpa.transformer". For example: if you want to use file "persistence-transformer.xsl" in /opt as a transformation, you need to start the framework with option ": `-D/opt/persistence-transformer.xsl"`.

4 JTA

The Java Transaction API specifies how to handle transactions that possibly span multiple different resources. In practice however this is seldom used because most applications just use one single database.

However, to use a single solution independent of the number of databases/connections used, the extender bundle provides a TransactionManager implementation that is exported as service and default enabled.

In normal situations, its use is as follows:

- At the start of an action (either via the web or otherwise), a transaction must be started using TransactionManager.begin().
- If something goes wrong, the transaction must be rolled back using TransactionManager.rollback().
- Otherwise, the transaction must be committed using TransactionManager.commit().

For web based applications, the extender provides a web servlet filter (osgi.jta.servlet.filter.TransactionFilter). In other situations, something alike must be provided using a different solution.

Alternatively, the transaction manager can be completely disabled by setting the "service.ranking" property of CM pid "osgi.extender.jta.tm" lower than -1000. A full example:

```
# Transaction timeout timer. In seconds.
osgi.extender.jta.tm..timeout = 30
# The ranking of the service. -1001 will disable it completely
osgi.extender.jta.tm..service.ranking = 0
```

5 Web extender

Web applications do not automatically run in an OSGi container. The reason for this is that web applications have a specific format that need to be processed by a JEE aware container. That functionality is not provided by OSGi.

The OSGi enterprise specification chapter 128 defines how to deal with web applications in an OSGi environment. The OSGi web extender provided by this project conforms to that specification with some restrictions:

- It does not convert standard war files to bundles. As such, the war must be a WAB (web application bundle) and as such a normal OSGI bundle.
- It does not provide the webbundle URL handling.
- It does not publish events as a result of the extending of the WABs.
- It is not tested for JSPs (since JSPs are assumed to be outdated).

5.1 Environment set-up

To be able to use the web extender, the following requirements need to be met:

- The OSGi environment must provide a standard HTTP service, as specified by chapter 102 of the compendium/enterprise specification. Bundles that provide this functionality are provide by both the Felix (see note) and Eclipse equinox projects.
- A service component runtime (like Felix SCR) to start the components in the bundle.

Note on using the Felix HTTP service:
The Felix HTTP service is a one-stop solution that provides various chapters of the compendium/enterprise specification related to HTTP handling. It therefore conflicts in some way with the implementation here (because it also implements chapters 128 and chapter 140 of the specifications). Therefore, when using Felix, replace "Web-ContextPath" with "X-Web-ContextPath" below and be aware that event listeners will be called from the Felix implementation as well.

5.2 Declaring a web bundle

To set-up a web-aware bundle, a bundle must contain the "Web-ContextPath" header, like:

```
Web-ContextPath: /MyContext
```

This means that the bundle is servicing requests to the path /MyContext on the HTTP port where the OSGI HTTP service is configured for.

When the bundle is started, it is picked up, configured and an OSGi service is registered for the created ServletContext as specified.

To add servlets to the context, there are two options:

- Creating a web.xml file (the normal JEE way).
- Registering servlets in the OSGi service registry.

5.2.1 Web.xml

Bundles that declare a web context in the bundle headers as indicated above, may have a /WEB-INF/web.xml file declaring the web application details. This file is automatically picked up by the extender bundle to configure the servlet context. The following information is used from the file:

- Servlet context parameters (context-param elements).
- Servlet definitions and mapping (servlet and servlet-mapping elements).
- Listener definitions (listener elements).
- Filter definitions and mapping (filter and filter-mapping elements).
- Welcome files and error files (welcome-file and error-page elements).
- Session timing parameters (session-config elements).

All other elements are silently ignored and as a result not all sub-elements from elements that are parsed are used (like servlet role definitions which are security related).

5.2.2 Web context definition service

Declaring a web.xml file means that only one context can be defined by a bundle. In normal situations this is sufficient. However, it is possible to add additional context definitions by registering an OSGi service of type osgi.extender.web.WebContextDefinition. This basically adds a web context to the bundle as is normally done with the combination Web-ContextPath bundle header and web.xml file. More specific: this is exactly how the extender handlers the bundle header/web.xml combination.

5.2.3 Servlet and filter service registrations

The disadvantage of declaring servlets, filters and listeners in a file is that the classes must be self-supporting: they are just instantiated via the default constructor and should be able to handle that. In some cases this is insufficient, for example if you want to use OSGi services in your servlet/filter which is much more simple using dependency injection.

For these cases, next to the declarations in the context file, servlets and filter that are registered as OSGi services are also pick-up by the web extender if they:

1. Have a web-context service property set.
2. Have the standard annotations from the servlet specification (WebServlet or WebFilter) so their mapping can be determined.

Ad 1.
The web-context service property is used by the extender to determine for which contexts

the servlets/filters must apply. The value must be regex expression that match the context.

Ad 2.
The servlet specification indicates that servlet and filter classes can have annotations and are automatically picked up from the class path. In OSGi environments classpath scanning is bad practice and the way to do this is by using OSGi services. As a result, the extender processes these annotations on OSGi services.

Example:
```
@Component(service = Servlet.class, property =
WebContextDefinition.WEBCONTEXTPATH + "=/Test")
@WebServlet(name = "Service", urlPatterns = "/xxx/*")
public class SimpleServlet extends HttpServlet {
```

Declares a servlet that services on URLS "/Test/xxx/*" (context path /Test, URL pattern withing the context /xxx/*).

The extender will make sure that always the init and destroy methods of the servlets/filters are called independent of the starting order of the services.

Note: care should be taken if a fitler or servlet context path service property matches more that one context. In those cases the init and destroy methods are called for all matching contexts and requests are dispatched for multiple contexts, possibly at the same time.

5.2.4 Listener services

Listeners can also be defined as OSGi services. Only listeners with the context-path service property set will be processed and called.
The extender will only call services that are registered at the time an event occurs. This is something to regard while writing listeners as services: it is for example very well possible that a session created event is not received by a SessionListener because the listener was not up at the time the event occurred.

5.3 JSF and classpath scanning

JSF uses two types of configuration:

- faces-config.xml files. Contain declarations of the configuration needed by components, etc.
- Facelets tag libraries. Used to extend the standard JSF components.

Unfortunately, to find these type of configurations, the JSF implementations make heavily use of classpath scanning. As already indicated earlier: this is a pain with OSGi and should be avoided when possible.
To still allow definitions from components to be automatically added for Web bundles, the extender treats the resource path /WEB-INF/classes/META-INF in a special way: it remaps this path to the META-INF directories of the bundles the web bundle depends on. As a

result, these files can be found by the JSF implementation.

For example:
Suppose a web bundle uses Primefaces JSF components. The primefaces bundle contains in the /META-INF directory the following files:

- /META-INF/faces-config.xml (a faces configuration file)
- /META-INF/primefaces-p.taglib.xml and /META-INF/primefaces-pm.taglib.xml (tag library definitions).

These files can be reached from the servlet context via the paths: /WEB-INF/classes/META-INF/faces-config.xml, /WEB-INF/classes/META-INF/primefaces-p.taglib.xml and /WEB-INF/classes/META-INF/primefaces-pm.taglib.xml.

Of course this may cause problems when bundles contain the same file names in their META-INF directories. Fortunately, it appears that implementations prefer to use the class loader for loading resources if the name of the resource is known in advance. As a result, the class loader is able to differentiate by returning different URLs for the files that would map to the same name in the WEB-INF/classes/META-INF namespace.

Note that the extender only remaps resources from bundles that are directly referenced from the bundle declaring the web context (for example via Import-Package or Require-Bundle manifest headers).

6 CDI

The Contexts and Dependency Injection specification provides the JEE standard for dependency injection. Its reference implementation is done by the Jboss Weld project. CDI is the suggested JEE7 approach for declaring beans that are used by JSF, but it can also be used without a web part.

The standard Java interface for CDI is the BeanManager interface, further referred to as bean manager. Luckily however, as an application programmer you don't use that interface much (or at all). However, it is mentioned here because bean manager services are published for every bundle that is CDI-extended by the extender bundle.

The following chapters describe how to enable your bundle for CDI extension and how to use and publish OSGi services in a CDI environment.

6.1 Enable a bundle for CDI extension

To enable a bundle for processing its contents for CDI bean creation, a specific requirement must be set in the bundle header:

```
Require-Capability: osgi.extender; filter:="(osgi.extender=osgi.cdi)"
```

This requires an extender of type “osgi.cdi” which is provided by the extender bundle.

A bundle with this requirement does not need to provide a beans.xml file.

6.2 Using OSGi services in beans

A CDI bundle can import services from the OSGi service registry by using the @ServiceReference qualifier (from package osgi.cdi.annotation) at an injection point. Example:

```
@Named
@ApplicationScoped
public class OrderView {
	@Inject @ServiceReference(filter="(source=local)")
	private OrderDao dao;
```

This indicates that an OrderDao service with property "source" set to "local" must be injected. The "filter" property of the annotation is optional, but allows you to narrow down the selected service via a standard OSGi service property filter.

Next to normal singleton services, it is also possible to inject a collection of services using this annotation. For example:

```
public class OrderView {
	@Inject @ServiceReference
	private Collection<OrderDao> daos;
```

Both solutions are backed by a proxy, so even for long lasting scopes like ApplicationScope, the references always result in the actual (set of) available services.

As an additional annotation parameter, a timeout value can be specified that indicates the time to wait for services to become available in case they are not available when a method on the reference is called. In normal situations, the default behaviour is fine (causing a object reference to wait for a small time and collections to perform the operation on an empty list immediately).

Note: service references **must** be interfaces.

6.3 Exporting beans as OSGi services

To mark a bean for export as service in the OSGi service registry, a class must be annotated with the @Service annotation (from package osgi.cdi.annotation). Example:

```
@ApplicationScoped
@Service(properties = "source=local")
public class OrderDaoStub implements OrderDao {
```

This indicates that a service must be exported for this bean with service property "source" set to "local". The properties are optional and may be a list of "key=value" strings that follow the same conventions as the properties that can be declared on Declarative Service components (see compendium specification). This means that it is possible to specify the type of the properties, like: "service.ranking:Integer=1".

Services can only be exported from global scopes, meaning application scope or

component scope (@ComponentScoped annotation, defined in osgi.cdi.annotations, see 6.5). The bean will be registered In the OSGi service registry for all the Java interfaces that the bean implements.

6.4 RequestScoped and SessionScoped beans

CDI defines scopes for requests and session, meaning that beans defined at this scope remain available during either the complete request or the user session. To make the CDI container aware about when to start and end requests and sessions, an (OSGi service) interface is defined by the extender that can be used to start and end scopes and set the right scope for a specific thread. This interface is defined by the osgi.extender.cdi.scopes.ExtenderContext interface. It is normally not directly used by applications, but used via the ScopeListener class that should be declared as servlet listener in case the session and scopes are used in web applications. Like:

```
<listener>
  <listener-class>osgi.extender.cdi.scopes.ScopeListener</listener-class>
</listener>
```

in WEB-INF/web.xml.

6.5 Additional scopes

The CDI extender defines, next to the standard scopes, the following additional scopes:

- *ComponentScoped*. Indicates an additional normal scope. In practice the usage is hardly different from the standard ApplicationScoped scope, but beans in this scope are eagerly instantiated (in contrast to normal application scope beans which are loaded on demand). This scope can be used in the case a component needs to be initialized at startup of the CDI container. Note that beans that are annotated as OSGi services are automatically eagerly instantiated, even if they are in application scope.
- *ViewScoped*. A CDI extension to mimic the ViewScoped scope as known from JSF. IView scoped means that beans defined in this scope will remain active for the specific page until either the session times out or the user explicitly destroys the current page (only works in JSF environments). In normal situations this means that posts/gets to the same page will have a consistent bean view, for example to fetch/update data via Ajax calls.

The scopes are defined in the package osgi.cdi.annotation and are handled via the servlet listener as defined in 6.4.

7 JSF

JSF uses Java expression language to reference beans for displaying information and executing actions on pages. With JEE7, these beans can be any bean managed in CDI, which practically removes the need of "JSF beans" from earlier specification versions. Therefore, the assumption is that beans referenced in JSF refer to beans in the CDI container.

In JEE containers, the linking between JSF and CDI is done by the (JEE) container. In OSGi

this functionality is performed by the CDI extender bundle.

7.1 Set-up

In a normal JSF application, the Faces implementation automatically looks for tag libraries and faces-config.xml files in the META-INF directory in the jars on the classpath of a web application. In an OSGi environment the META-INF directory is not exported and therefore this set-up does not work.

Chapter 128 of the OSGi enterprise specification specifies however how web applications can run in an OSGi environment. An implementation of this chapter is a pre-requirement for working with JSF. Possible solutions are:

- Pax-Web. A combination of bundles that automatically extends a web bundle with the tag libraries and faces-config files needed for JSF application: it locates these files in bundles that are referenced by the web application and adds them to the servlet parameters.
- The Web Extender that is part of this project, see 5.

Both implementations have been tested.

Since normally the scope listeners and CDI are also required for a JSF application, the OSGi container creates a dependency from the JSF based web application bundle to the CDI extender bundle. As a result, automatically the faces-config.xml from the extender bundle is detected and processed by the JSF implementation. This completes all the necessary work for integrating JSF with CDI. However, it also requires additional package imports by the web bundle: osgi.cdi.faces and osgi.jta.faces.

7.2 Dynamic extension

Writing one Web Application Bundle (WAB) for JSF is simple. However, the interesting part of using OSGi comes from the fact that bundles can come and go and that these bundles can dynamically extend functionality provided to the set-up.

As a result it is possible to extend the functionality of a JSF web application dynamically by simply adding bundles that follow that extension pattern. This is done as follows:

1. Create the pages, resources, etc. needed for the extension as you would normally do with a JSF application and store these resources somewhere in your bundle.
2. Define the necessary beans for the pages, etc. using CDI annotations.
3. Define the bundle as a CDI bundle, see before.
4. Define the resources (pages, library contents, etc.) to be exported for usage by the JSF application. This is done by adding a "Bundle-Resources" header to the bundle specifying the resources to export.

Example:
Suppose you created a new page "bla.xhtml" and want to navigate to this page from the main application which is a different bundle. In that case you would:

- Place "bla.xhtml", say, in resources/pages below your bundle.

- Write the bean classes for the page as you would normally do.
- Put the following headers in the manifest file:
  - `Require-Capability: osgi.extender; filter:="(osgi.extender=osgi.cdi)"`
    (for enabling CDI).
  - `Bundle-Resources: resources`
    (to indicate that the directory "resources" should be exported as root of the resource directory).

  It is possible to specify multiple locations for different kind of files, for example:
  `Bundle-Resources: resources, css=library/css`
  to indicate that the "css" library can be found in the library/css directory.

Note that the default behaviour is to allow the (main) web application access to all bean managers of all CDI extended bundles and to allow access to all exported resources from all bundles. This can be limited by filtering on:

- The bundle symbolic name, or
- The bundle category.

These headers are copied as service attributes to the exported interfaces and can be filtered on using the "osgi.extender.cdi.faces.filter" context init parameter:

```
<context-param>
  <description>Filter of resources/CDI bean containers</description>
  <param-name>osgi.extender.cdi.faces.filter</param-name>
  <param-value>(Bundle-Category=*web-group*)</param-value>
</context-param>
```

This allows for multiple web applications to run in one OSGi container without interfering each other.

8 Thanks

This project would not have been possible without:

- Partial sponsoring from (Imtech) Traffic and Infra and testing in two projects there.
- Testing parts of the extender at Fujifilm Manufacturing Europe.