Constructor Injection and other new features in Declarative Services 1.4

BJ Hargrave, IBM
Declarative Services

- A declarative model for publishing and consuming OSGi services
- Introduced in Release 4 in 2005, it greatly simplified programming OSGi services
  - DS 1.1 added minor enhancements
  - DS 1.2 added build-time annotations
  - DS 1.3 added field injection, component property types and introspection
- Service Component Runtime, SCR, is the runtime which implements the spec and manages the service components

Open Liberty

>900 projects
>590 DS Components

https://openliberty.io/
Service Component

- A Java class which can optionally be registered as a service and can optionally use services

- Described by XML in the bundle which is processed at runtime by SCR

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="example.provider.ExampleImpl" activate="activate">
<implementation class="example.provider.ExampleImpl"/>
<service>
  <provide interface="example.api.Example"/>
</service>
<reference name="Log" interface="org.osgi.service.log.LogService" bind="setLog"/>
</scr:component>
```
Service Component using annotations

• But writing XML is for computers

• So DS supports for programming with annotations

• Tools, like \texttt{bnd}, process the annotations at \texttt{build-time} into the XML used by SCR at \texttt{runtime}
Service Component using annotations

- **@Component** declares a service component and any service it may provide.

- **@Activate** declares the activate method which is called to complete activation of the component.

- **@Reference** declares a service references by the component and can be applied to a bind method (method injection) or a field (field injection).

```java
@Component
public class ExampleImpl implements Example {
    private Map<String, Object> properties;
    private LogService log;

    @Activate
    void activate(Map<String, Object> map) {
        properties = map;
    }

    @Override
    public boolean say(String message) {
        log.log((int) properties.get("loglevel"), message);
        return false;
    }

    @Reference
    void setLog(LogService log) {
        this.log = log;
    }
}
```
So what is new in Declarative Services 1.4?

- Constructor Injection
- Activation fields
- Component Property Types enhancements
  - Use to annotate components
  - Name mapping enhancements: single-element annotations and PREFIX_
  - Predefined component property types
- Component factory properties
- OSGi Logger support
Constructor Injection
Injection

- **Method injection**

  ```java
  @Reference
  void setExample(Example example) {
      this.example = example;
  }
  ```

- **Field injection**

  ```java
  @Reference
  private Example example;

  private final Example example;
  ```

- **Constructor injection**

  ```java
  @Activate
  public ExampleImpl(@Reference Example example) {
      this.example = example;
  }
  ```
Constructor Injection

- Static policy only!

- Scalar cardinality
  ```java
  private final Example example;
  @Activate
  public ExampleImpl(@Reference Example example) {
      this.example = example;
  }
  ```

- Multiple cardinality
  ```java
  private final List<Example> examples;
  @Activate
  public ExampleImpl(@Reference List<Example> examples) {
      this.examples = examples;
  }
  ```

- Service type from generic signature
Constructor Injection

- Can also inject parameters of types related to services

  ```java
  @Activate
  public ExampleImpl(
      @Reference ServiceReference<Example> sr,
      @Reference ComponentServiceObjects<Example> so,
      @Reference(service=Example.class) Map<String,Object> serviceProps,
      @Reference Map.Entry<Map<String,Object>,Example> tuple)
  {
      ... 
  }
  ```

- Service type from generic signature
Constructor Injection

• Can also inject activation objects

• Parameters not annotated @Reference

```java
public @interface Props {
    int my_port() default 8080;
}

@Activate
public ExampleImpl(
    ComponentContext context,
    BundleContext bc,
    Map<String, Object> componentProps,
    Props props) {
    ... }
```
Activation Fields
Activation Fields

- Activation objects can now also be injected in fields in addition to being parameters to the activate method or the constructor

```java
@Activate
private ComponentContext context;
@Activate
private BundleContext bc;
@Activate
private Map<String,Object> componentProps;
@Activate
private Props props;
```
Component Property Type enhancements
Component Property Types

- Define and use your component properties in a type safe manner using annotations

```java
public @interface Props {
    int my_port() default 8080; // my.port property
}

@Component
public class ExampleImpl implements Example {
    private final int my_port;
    @Activate
    public ExampleImpl(Props props) {
        my_port = props.my_port();
    }
}
```
Component Property Types

• But if you want to specify a different value for your component, you need to use a String

• Make sure to get property name and type right

```java
public @interface Props {
    int my_port() default 8080; // my.port property
}
@Component(property = {"my.port:Integer=9080"})
public class ExampleImpl implements Example {
    ...
}
```
Component Property Types

- Now you can use the component property type to annotate the component and specify property values for your component in a type safe manner.

```java
@ComponentPropertyType
public @interface Props {
    int my_port() default 8080; // my.port property
}

@Component
@Props(my_port=9080)
public class ExampleImpl implements Example {
    ...
}
```
**Single-element Annotation**

- A single-element annotation is an annotation which has one element named `value` and all other elements, if any, have default values.

- Property name for `value` element is derived from type name rather than method name.

```java
@Component
@MyPort(9080)
public class ExampleImpl implements Example {
    ...
}
```

```java
@Component
@MyPort
public @interface MyPort {
    int value() default 8080; // my.port property
}
```
Common prefix for property names

- Value of `PREFIX_` field, whose value is a compile-time constant String, is prefixed to property name

- Useful when property names have a common prefix to keep annotation element names short

```java
@Component
@PropertyType
public @interface ServerPort {
    String PREFIX_ = "com.acme.";
    int value() default 8080; // com.acme.server.port property
}
```
Predefined Component Property Types

- OSGi specifications will define Component Property Types for specified properties
- Common service properties: @ServiceRanking, @ServiceDescription
- HttpWhiteboard service properties: @HttpWhiteboardServletName, @HttpWhiteboardServletPattern
- JAX-RS service properties: @JaxrsName, @JaxrsResource

```java
@ComponentPropertyType
public @interface JaxrsName {
    String PREFIX_ = "osgi.";
    String value(); // osgi.jaxrs.name property
}
```
Component Factory Properties
Component Factory Properties

- Component Factory services only had two properties: component.name and component.factory

```java
@Component(factory="widget.factory")
public class WidgetImpl implements Widget {
    ...
}
```

```java
@Component
public class WidgetUser {
    @Reference(target="(component.factory=widget.factory)")
    private ComponentFactory<Widget> factory;
    @Activate
    void activate() {
        ComponentInstance<Widget> widget =
            factory.newInstance(null);
    }
}
```
Component Factory Properties

- Component Factory Properties are added to allow more decoration of the factory

```java
@Component(factory="widget.factory",
           factoryProperty={"widget.quality=AAA"})
public class WidgetAAAImpl implements Widget { ... }

@Component(factory="widget.factory",
           factoryProperty={"widget.quality=B"})
public class WidgetBImpl implements Widget { ... }

@Component
public class WidgetUser {
    @Reference(target="(&(component.factory=widget.factory)(widget.quality=AAA))")
    private ComponentFactory<Widget> factory;
}
```
OSGi Logger Support
OSGi Logger Support

- R7 includes a big update to the OSGi Log Service
  - LoggerFactory which makes Loggers
  - Like slf4j
  - DS will support creating and injecting a Logger for the component implementation class

```java
@Component
public class ExampleImpl implements Example {
    @Reference(service=LoggerFactory.class)
    // LoggerFactory.getLogger(ExampleImpl.class)
    private Logger logger;
    @Activate
    void activate() { logger.info("initialized"); }
}
```
And a number of other small improvements
Evaluate the Sessions
Sign in and vote at eclipsecon.org

-1  0  +1