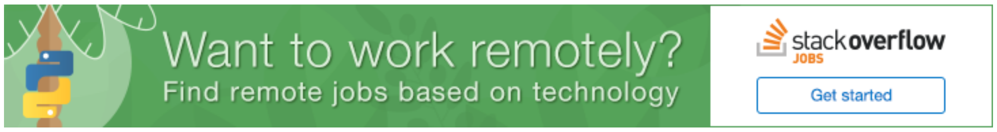
What exactly is the ResourceConfig class in Jersey 2?



I have seen a lot of Jersey tutorials that starts with something like

```
@ApplicationPath("services")
public class JerseyApplication extends ResourceConfig {
    public JerseyApplication() {
        packages("com.abc.jersey.services");
    }
}
```

without explaining what exactly the `ResourceConfig` class is. So where can I find its documentation, usage, etc.? Googling for "jersey resourceconfig" does not yield any official doc.

Some of my questions about this class and its usage are:

- What things can I do inside the subclass of `ResourceConfig` ?
- Do I need to register the subclass of `ResourceConfig` somewhere so that it can be found or is it automatically detected by Jersey?
- If the subclass is automatically detected what happens if I have multiple subclasses of `ResourceConfig` ?
- Is the purpose of `ResourceConfig` the same as the `web.xml` file? If so what happens if I have both in my project? Does one of them take precedence over the other?

jersey jax-rs jersey-2.0

edited Aug 11 '17 at 2:30

asked Aug 11 '17 at 2:14
Chin
5,705 19 63 112

1 Answer

Standard JAX-RS uses an `Application` as its configuration class. `ResourceConfig` *extends* `Application` .

There a three different ways to configure Jersey (JAX-RS):

1. With only web.xml
2. With both web.xml *and* an `Application/ResourceConfig` class
3. With only an `Application/ResourceConfig` class annotated with `@ApplicationPath` .

With only web.xml

It is possible to configure the application in a standard JAX-RS way, but the following is specific to Jersey

```
<web-app>
    <servlet>
        <servlet-name>jersey-servlet</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>com.mypackage.to.scan</param-value>
        </init-param>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Since Jersey runs in a servlet container, it is only right that the Jersey application runs as a servlet. Jersey's servlet that handles incoming requests is the `ServletContainer` . So here we configure its servlet. We also configure an init-param for Jersey to know which package(s) to scan for our `@Path`

Under the hood, Jersey will actually create a `ResourceConfig`, as that's what it uses to configure the application. Then it will register all the classes that it discovers through the package scan.

With both web.xml and `Application/ResourceConfig`

We can use the above web.xml, but instead of setting any init-params we can configure our `ResourceConfig/Application`

```
<servlet>
    <servlet-name>jersey-servlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.example.JerseyApplication</param-value>
    </init-param>
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
</servlet>
```

Here, Jersey will not need to create a `ResourceConfig`, as we are already passing one to it. It will just create *our* `ResourceConfig` or `Application`.

With only `Application/ResourceConfig`

Without a web.xml Jersey needs a way for us to provide the servlet-mapping. We do this with the `@ApplicationPath` annotation.

```
@ApplicationPath("services")
public class JerseyApplication extends ResourceConfig {
    public JerseyApplication() {
        packages("com.abc.jersey.services");
    }
}
```

Here with the `@ApplicationPath`, it's just like if we configured the servlet mapping in the web.xml

```
<servlet-mapping>
    <servlet-name>JerseyApplication</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

Your call to `packages` is the same as if we were to use only web.xml and provided the init-param for the package scan above.

When using only Java code for configuration, there needs to be some way for Jersey to discover our configuration class. This is done with the use of a `ServletContanerInitializer`. This is something that was introduced in the Servlet 3.0 Specification, so we cannot use "Java only" configuration in earlier servlet containers. Basically what happens is that we can provide the initializer what classes to look for, and the servlet container will pass those classes to Jersey's initializer. In this case, Jersey configures it to look for `Application` class and classes annotated with `@ApplicationPath`. See this post for further explanation.

What things can I do inside the subclass of ResourceConfig

Just look at the javadoc I linked to above. Its mostly just registration of classes. Not much else you need to do with it. The main methods you will be using is just the `register`, `packages`, and `property` method. The former lets you register classes and instances of resources and providers manually, while the latter will allow you give packages for Jersey to scan to pick up classes and register them.

The `ResourceConfig` is just a convenience class. Remember, it extends `Application`, so we could even use the standard `Application` class

```
@ApplicationPath("/services")
public class JerseyApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        classes.add(MyResource.class);
        return classes;
    }
    @Override
    public Set<Object> getSingletons() {
        final Set<Object> singletons = new HashSet<>();
        singletons.add(new MyProvider());
        return singletons;
    }

    @Override
    public Map<String, Object> getProperties() {
        final Map<String, Object> properties = new HashMap<>();
        properties.put("jersey.config.server.provider.packages",
                       "com.mypackage.to.scan");
        return properties;
    }
}
```

With a `ResourceConfig` , we would just do

```
JerseyApplication() {
    register(MyResource.class);
    register(new MyProvider());
    packages("com.mypackages.to.scan");
}
```

Aside from being more convenient, there are also a few thing under the hood that help Jersey configure the application.

edited Jan 16 at 7:50

answered Aug 11 '17 at 4:49

Paul Samsotha
129k 16 219 399

Thank you for the awesome answer! This clears up tons up things for me. – Chin Aug 11 '17 at 13:07