## Customizing the REST API

Add custom endpoints to the new REST API by using the existing WorldServer SDK. However, you should use this only as an entry point into the REST API and delegate the customization logic to other classes.

### About this task

You are inside the Spring context, so you can use its full functionality. For example, you can add the customization logic in a bean that you autowire in RestController. There is no management UI, so you will have to manage API customizations through REST calls.
Build your customization as a Maven project with Shade Plugin. This facilitates the creation of the uber **.jar** packages that will contain the customization and its libraries. The customization must be in a valid **.jar** package; this package must contain the customization classes and the libraries that are not already provided by WorldServer. Also, make sure that the RestController beans are in the `com.sdl.lt.worldserver.customizations` subpackage.
To create a new endpoint, you need to create a new RestController. In Spring, you can do this by annotating the class with `@RestController`. This instructs the customization process that it needs to add this class to the Spring context as a RestController bean.
To make sure that your **.jar** package is as minimalist as possible, set the dependencies that already exist in WorldServer as `provided`. For example, wssdk and spring-webmvc are already in WorldServer, so you should set them as `provided`. This makes shade-plugin exclude them from the **.jar** file. Also, you can use the shade-plugin configuration to include or exclude artifacts.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.1.7.RELEASE</version>
    <scope>provided</scope>
</dependency>
```

If you do not have wssdk or other libraries as Maven dependencies, you can install them in your local repository. You can do this using `mvn install:install-file`, and then use it as a regular Maven dependency.

```
mvn install:install-file -Dfile=wssdk-server.jar -DgroupId=com.sdl.lt.worldserver
-DartifactId=wssdk -Dversion=1.0 -Dpackaging=jar
```

Also, you can set its scope to `provided`, to avoid including it in the **.jar** package unnecessarily. For example, wssdk is already in WorldServer, so there is no need to provide it in every customization.
Another option is to use the `system` scope and reference the library from the disk.

```
<dependency>
    <groupId>com.sdl.lt.worldserver</groupId>
    <artifactId>wssdk</artifactId>
    <version>1.0</version>
    <scope>system</scope>
    <systemPath>c:\wssdk\wssdk_10.4.4.144\lib\server\wssdk-server.jar</systemPath>
</dependency>
```

**Important:** Dependencies with the `system` scope are not included in the uber **.jar** package.

### Procedure

1. Create a Maven project.
    a. Add the wssdk dependency.
    b. Add the `provided` dependencies.
    c. Add the dependencies that are not provided.
    d. Add the Shade Plugin. You can use the following definition as a reference:

```
e. <plugin>
f.     <groupId>org.apache.maven.plugins</groupId>
g.     <artifactId>maven-shade-plugin</artifactId>
h.     <version>2.4.2</version>
i.     <executions>
j.         <execution>
k.             <phase>package</phase>
l.             <goals>
m.                 <goal>shade</goal>
n.             </goals>
o.             <configuration>
p.                 <artifactSet>
q.                     <excludes>
r.                         <exclude>org.springframework:spring-
   webmvc:*</exclude>
s.                         <exclude>log4j:log4j:jar:</exclude>
t.                     </excludes>
u.                 </artifactSet>
v.                 <filters>
w.                     <filter>
x.                         <artifact>*:*</artifact>
y.                         <excludes>
z.                             <exclude>META-INF/**</exclude>
aa.                        </excludes>
bb.                    </filter>
cc.                </filters>
dd.            </configuration>
ee.        </execution>
ff.    </executions>
   </plugin>
```

2. Create a RestController bean.

a. Create a class in a package under `com.sdl.lt.worldserver.customizations`.

b. Annotate it with `@RestController`.

c. Annotate it with `@RequestMapping` ( "`/extensions/`*`<new-endpoint>`*").

**Tip:** Every customization mapping has to start with `/extensions/`.

d. Autowire context: Inject the `com.idiominc.wssdk.WSContext` in the controller.

e. Create the request handler method. To do so, you need to create a method that calls the customization logic and add the method request mapping by annotating the method with `@RequestMapping` ( `value = "clients", method = GET, produces = APPLICATION_JSON_VALUE`).

3. Build the project ( `mvn clean install`). This generates the uber **.jar**: **target/*<project-name>*-*<version>*.jar**

4. Upload the **.jar** file into WorldServer. Use any REST client you want. To upload the **.jar** file, make a POST request to **http://*<ws-host>*:*<ws-port>*/ws-api/v1/customizations/api?token=*<ws-token>***. The body should be form-data.

5. Test the customization.

6. Optional: Do any of the following:

   - See the existing customizations. To do so, make a request to **http://*<ws-host>*:*<ws-port>*/ws-api/v1/customizations?token=*<ws-token>***
   - Enable or disable customizations. To do so, make a PATCH request to **http://*<ws-host>*:*<ws-port>*/ws-api/v1/customizations?token=*<ws-token>*** changing the value of isActive to `true` or `false`.
   - Delete customizations. To do so, make a DELETE request to **http://*<ws-host>*:*<ws-port>*/ws-api/v1/customizations/api?token=*<ws-token>*&name=*<customization-name>***. Make sure your Content-Type is application/json.

**Parent topic:** The WorldServer REST API