Payara Micro Revealed

Cloud-Native Application Development with Java

—

David R. Heffelfinger

Apress®

CHAPTER 2

Developing Microservices Using Payara Micro

Microservices are nothing but RESTful web services; they typically perform a single task; as such, their code bases tend to be small. In a microservices architecture, applications are composed of several different microservices communicating with one another.

Payara Micro is a MicroProfile-compliant runtime; as such, applications deployed with Payara Micro must conform to the MicroProfile specification. MicroProfile uses the Jakarta RESTful Web Services API for RESTful web services deployment. In this chapter, we will see how to develop RESTful web services using said API and execute them against Payara Micro.

Setting Up Your Environment

Payara provides a Maven plug-in that makes working with Payara Micro projects easier. In this section, we will discuss how to set up Maven projects for Payara Micro applications.

NetBeans IDE can create a Payara Micro project out of the box with all necessary configuration in place.

Payara Micro Maven Plug-in

To add the Payara Micro Maven plug-in to your project, add it to the <plugins> section of your *pom.xml* as illustrated in the following example.

© David R. Heffelfinger 2022
D. R. Heffelfinger, *Payara Micro Revealed*, https://doi.org/10.1007/978-1-4842-8161-1_2

```
<build>
  <plugins>
    <plugin>
      <groupId>fish.payara.maven.plugins</groupId>
      <artifactId>payara-micro-maven-plugin</artifactId>
      <version>1.4.0</version>
      <configuration>
        <payaraVersion>5.2021.5</payaraVersion>
        <deployWar>false</deployWar>
        <commandLineOptions>
          <option>
            <key>--autoBindHttp</key>
          </option>
          <option>
            <key>--deploy</key>
            <value>
             ${project.build.directory}/${project.build.finalName}
            </value>
          </option>
        </commandLineOptions>
        <contextRoot>/</contextRoot>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Under the <configuration> section, we can set several configuration options used when running our application in Payara Micro.

Maven automatically downloads the Payara Micro version we specify in the <payaraVersion> tag.

We can specify if we want to automatically deploy the project WAR file in the <deployWar> tag. Usually when developing an application, it makes sense to deploy the application as an exploded WAR file (more on that later); therefore, it is a good idea to set this value to false.

We can specify command-line options to pass to Payara Micro in the <commandLineOptions> tag. Table 2-1 lists the options that are available.

Table 2-1. *Payara Micro Command-Line Options*

Command-Line Option	**Description**
autoBindHttp	If set to true, Payara Micro will automatically find an available port and bind it as the httP port
deploy	specifies either an exploded or standard War file to Payara Micro. as illustrated in the example, it is a good idea to take advantage of Maven environment variables when deploying our project from Maven
port	If autoBindHttp is set to false, used to specify the httP port to use

the complete list of command-line options for Payara Micro can be found at *https://docs.payara.fish/community/docs/documentation/payara-micro/appendices/cmd- line- opts.html*.

Additionally, we can specify the context root of our application via the <contextRoot> tag. In a microservices application, there is typically only one RESTful web service per application; therefore, typically, we would use the root context ("/") here.

Payara BOM

Maven Bill of Materials (BOM) are a special kind of POM used to keep dependencies in sync. Payara provides a BOM that provides all necessary dependencies for Payara Micro. In order to successfully build and deploy our code to Payara Micro, we need to include the Payara BOM in the <dependencyManagement> section of our *pom.xml* file. For example:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>fish.payara.api</groupId>
      <artifactId>payara-bom</artifactId>
      <version>${version.payara}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Payara Maven Repository

In order to find Payara-specific dependencies, we need to add the Payara Maven Repository to the <repositories> section of our *pom.xml*.

```
 <repositories>
  <repository>
    <id>payara-nexus-artifacts</id>
    <name>Payara Nexus Artifacts</name>
    <url>https://nexus.payara.fish/repository/payara-artifacts</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

Nexus is a popular open source repository management tool. the Payara Maven repository uses Nexus to manage its Maven artifacts.

Jakarta EE and MicroProfile Dependencies

Last but not least, we need to add dependencies for any Jakarta EE or MicroProfile APIs we will use in our project. Our simple example uses only the Jakarta RESTful Web Services API; therefore, we only need one dependency.

```
<dependencies>
  <dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-web-api</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Specifying the Base URI for Our Web Services

Before we can start developing RESTful web services, we need to specify the base URI that all web service endpoints in our module will share. This can be done via the @ApplicationConfig annotation, as illustrated in the following example.

```
package com.ensode.microservice;

import javax.ws.rs.ApplicationPath; import
javax.ws.rs.core.Application;

@ApplicationPath("webresources") public class
ApplicationConfig extends Application { }
```

The annotated class must extend the Application class; the value of the ApplicationPath annotation corresponds to the root URI of our RESTful web services (webresources, in our example). The root URI we specify here will precede the URIs of the RESTful web services we develop; for example, if we write a RESTful web service with a "sample" URI, the complete URI for the web service would be /webresources/sample.

Running Our Application

Maven can create an exploded WAR file from any Java web project via the war:exploded goal. Similarly, we can start Payara Micro straight from Maven via the payara- micro:start Maven goal provided by the Payara Micro Maven Project.

By invoking Maven using the aforementioned two Maven goals, we can start Payara Micro and automatically deploy our application code.

```
mvn war:exploded payara-micro:start
```

We should see some output in the command line similar to the following:

```
[2021-08-07T13:21:06.418-0400] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1
_ThreadName=main] [timeMillis: 1628356866418] [levelValue: 800] [[

Payara Micro URLs: http://192.168.1.165:8080/
'microservice-1.0-SNAPSHOT' REST Endpoints:
GET    /openapi/
GET    /openapi/application.wadl
GET    /webresources/application.wadl
DELETE /webresources/sample
GET    /webresources/sample
PATCH  /webresources/sample
POST   /webresources/sample
PUT    /webresources/sample

]]

[2021-08-07T13:21:06.418-0400] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1
_ThreadName=main] [timeMillis: 1628356866418] [levelValue: 800] Payara
Micro  5.2021.4 #badassmicrofish (build 740) ready in 7,751 (ms)
Handling HTTP GET requests
```

Notice that the output specifies the IP address and HTTP port where Payara Micro is listening, as well as all RESTful web service endpoints along with the HTTP request types they listen for.

---

some of these endpoints are automatically generated by Payara Micro.

Starting our application this way, combined with the compile-on-save feature most modern Java IDEs possess, allows us to hot deploy our application, meaning that code changes are automatically picked up by Payara Micro as we save our code, eliminating the need to manually build and run our code to test our code changes.

Developing RESTful Web Services

In order to turn any plain old Java object into a RESTful web service, all we need to do is annotate it with the @Path annotation. This annotation allows us to specify the URI for our web service; it also lets the runtime (Payara Micro, in our case) know that this class is a web service.

```
package com.ensode.microservice;

@Path("sample") public class
SampleRestFulService { }
```

The @Path annotation, combined with the @ApplicationPath annotation we previously discussed, determines the full URI of our RESTful web service. In our sample project, the complete URI for the RESTful web service would be webresources/sample. A RESTful web service needs to respond to one or more HTTP request types (GET, POST, PUT, DELETE, or PATCH); this is accomplished via method-level annotations.

Handling HTTP GET Requests

We can handle HTTP GET requests in our RESTful web services by annotating one of our methods with the @GET annotation. If our method returns a value to the client, we need to specify the mime type of the returned value via the @Produces annotation.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public String processGetRequest() {

  String json = "{"
    + "\"msg\":\"Service processed HTTP GET request!\""
    + "}";

  return json;
}
```

JSON is by far the most common mime type used for data transmission in RESTful web services and microservices, but it is by no means the only one. All supported mime types are defined as constants in the MediaType class; others supported include XML (MediaType.TEXT_XML), HTML (MediaType.TEXT_HTML), and plain text (MediaType.TEXT_PLAIN), among others.

Use your IDE Code Completion on the MediaType class to see all supported mime types.

In our simple example, we are simply hard-coding a JSON string and returning that. Typically, we would process some data and generate a JSON string with one of Jakarta EE’s JSON processing libraries (JSON-B or JSON-P). We will cover JSON processing later in the book.

Handling HTTP POST Requests

HTTP POST requests are handled via the @POST annotation; usage is identical to the previously discussed @GET annotation.

```
@POST
@Produces(MediaType.APPLICATION_JSON)
public String processPostRequest() {

  String json = "{"
        + "\"msg\":\"Service processed HTTP POST request!\""
        + "}";

  return json; }
```

This example is nearly identical to the one in the previous section, the main difference being we annotated our method with the @POST annotation, which causes the method to be invoked when our service responds to an HTTP POST request.

Handling HTTP PUT Requests

HTTP PUT requests are handled via the @PUT annotation.

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
public String processPutRequest() {

  String json = "{"
        + "\"msg\":\"Service processed HTTP PUT request!\""
        + "}";

  return json; }
```

The @PUT annotation on this method causes the method to be invoked when our service responds to an HTTP PUT request.

Handling HTTP DELETE Requests

HTTP DELETE requests are handled via the @DELETE annotation.

```
@DELETE
@Produces(MediaType.APPLICATION_JSON)
public String processDeleteRequest() {

  String json = "{"
        + "\"msg\":\"Service processed HTTP DELETE request!\""
        + "}";

  return json; }
```

The @DELETE annotation on this method causes the method to be invoked when our service responds to an HTTP DELETE request.

Handling HTTP PATCH Requests

We can also handle HTTP PATCH requests via the @PATCH annotation.

```
@PATCH
@Produces(MediaType.APPLICATION_JSON)
public String processPatchRequest() {

  String json = "{"
        + "\"msg\":\"Service processed HTTP PATCH request!\""
        + "}";

  return json; }
```

As expected, the @PATCH annotation on this method causes the method to be invoked when our service responds to an HTTP PATCH request.

Path and Query Parameters

You can pass parameters to a RESTful web service; there are two ways to do that: using path parameters and query parameters; both ways are supported by the Jakarta RESTful Web Services API.

Path Parameters

Path parameters are added to a RESTful web services path when invoked; for example, the following RESTful web service URI contains two path parameters; the first parameter value is “Mr”; the second one is “Heffelfinger”.

http://localhost:8080/webresources/pathparams/Mr/Heffelfinger

In order for our RESTful web services to accept path parameters, we need to add the @Path annotation at the method level, as well as the @PathParam annotation for each method argument corresponding to a path parameter.

```
package com.ensode.pathparams; //imports

omitted
```

```
@Path("pathparams")
public class PathParamsSampleService {

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  @Path("/{title}/{lastName}")   public String
sayFormalHello(@PathParam("title") String title,
@PathParam("lastName") String lastName) {     return String.format("{\n"
          + "  \"msg\":\"Hello, %s %s\"\n"
          + "}", title, lastName);
  }

}
```

The method-level @Path annotation provides a URI template, specifying the location of each path parameter. In our example, the first parameter is called title; the second parameter is called lastName; we specify the order in which we expect path parameters by using the @Path annotation in this manner.

To bind path parameters to method arguments, we used the @PathParameter annotation as illustrated in the example; notice that the value of the @PathParameter annotation must match the name of the URI template variables specified in the method- level @Path annotation (title and lastName, in our example).

When a client invokes our service, it is expected for the client to pass the expected parameters in the right order; for example, we could invoke our service using curl as follows:

```
curl http://localhost:8080/webresources/pathparams/Mr/Heffelfinger
```

In this case, *Mr* corresponds to the value of our first parameter, as defined in the URI template of the method-level *@Path* annotation; *Heffelfinger* corresponds to the second path parameter, again defined in the URI template; these values are automatically bound to the corresponding method arguments by annotating these arguments with the *@PathParam* annotation whose values match the URI variables defined in the URI template (*title* and *lastName*, in our example).

Once we invoke our service, it will return a JSON string with the corresponding values.

```
{
  "msg":"Hello, Mr Heffelfinger"
}
```

Query Parameters

The second way we can pass parameters to our RESTful web services is via query parameters. In this case, parameters are sent as a query containing name value pairs in the URI for our web service.

For example, using curl, we could pass query parameters to a RESTful web service endpoint as follows:

```
curl "http://localhost:8080/webresources/queryparams?title=Mr&lastName= Heffelfinger"
```

In this example, the URI for the RESTful web service endpoint is http:// localhost:8080/webresources/queryparams; this example has two parameters, named title and lastName. A question mark is used to delimit the boundary between the URI and the query.

```
package com.ensode.queryparams; //imports

omitted

@Path("queryparams")
public class QueryParamsSampleService {

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public String sayFormalHello(@QueryParam("title") String title,
@QueryParam("lastName") String lastName) {     return String.format("{\n"
          + "  \"msg\":\"Hello, %s %s\"\n"
          + "}", title, lastName);
  }

}
```

Parameter names are defined by the @QueryParam annotation; values sent by the client are automatically bound to the corresponding method arguments. Output of this example would be identical to the path parameters examples we discussed in the previous section.

Parsing JSON Data

RESTful web services can consume and produce data in various formats, with JSON being by far the most common one.

Jakarta RESTful Web Services can automatically convert JSON data to and from Java with almost no effort on our part. Having a Java class with fields matching a JSON string's properties will result in those fields being automatically populated with the corresponding values. All we need to do is annotate our method with the @Consumes and @Produces annotations using the appropriate media type.

```
package com.ensode.jsonprocessing; //imports

omitted

@Path("messageprocessor") public
class MessageProcessor {

  @PUT

@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
  public Message processMessage(Message message) {     Message
returnedMessage = new Message();

    returnedMessage.setMsgText(message.getMsgText().       toUpperCase());

    return returnedMessage;
  }
}
```

Our Message class is a simple POJO (plain old Java object).

```
package com.ensode.jsonprocessing;

public class Message {   private String

msgText;

  public String getMsgText() {
return msgText;   }

  public void setMsgText(String msgText) {
this.msgText = msgText;   }

}
```

In our example, the client is expected to send a JSON string that can be parsed into a Message object. By simply annotating our method with the @Consumes annotation and specifying *application/JSON* as the media type, data from the client is used to automatically populate an instance of our Message class. We could invoke our service using curl as follows:

```
curl -X PUT http://localhost:8080/webresources/messageprocessor -H
"Content-Type: application/json" -d '{"msgText":"Yay for seamless JSON parsing!"}'
```

Notice that the JSON data we are sending to the service has a msgText property; this property is used to populate the corresponding msgText instance variable in our Message class.

Our example service method returns another instance of the Message class, since we are using the @Produces annotation with a media type of *application/json*; this return value is automatically converted to a JSON string, which we can see in the output of the preceding curl command.

```
{"msgText":"YAY FOR SEAMLESS JSON PARSING!"}
```

Summary

In this chapter, we saw how we can develop RESTful web services with the Jakarta RESTful Web Services API. We saw how using a few simple annotations can convert a Java class to a RESTful web service and handle HTTP requests, including GET, PUT, POST, DELETE, and PATCH requests.

We also covered parameters to our RESTful web services. We saw how to process path parameters via the @PathParam annotation. Additionally, we saw how to process query parameters via the @QueryParam annotation.

Finally, we saw how we can seamlessly parse JSON strings and populate our Java objects and how to easily generate JSON strings from plain old Java objects.