Demystifying Microservices for Java EE Developers

by David Heffelfinger

Payara Server - Derived from GlassFish

24/7 Production Support, Quarterly Releases, Bug Fixes & Patches

Contents

Lately there has been a lot of talk about microservices - a new way to develop and design enterprise applications. In this guide, we aim to cut through the hype, explaining microservices in terms that make sense to Java EE developers.

WHAT ARE MICROSERVICES?

Microservices are an architectural style in which code is deployed in small, granular modules. The microservice architecture reduces coupling and increases cohesion. Typically, microservices are implemented as RESTful web services by invoking HTTP methods (GET, POST, PUT, DELETE) on each other, commonly using JSON to pass data to one another. Since communication between microservices is done via HTTP methods, microservices written in different programming languages can interact with each other.

CAN JAVA EE DO MICROSERVICES?

Some may think that Java EE is "too heavyweight" for microservices development - this is simply not the case. Because of this misconception, some may think that Java EE may not be suitable for a microservice architecture, when in fact **Java EE fits microservices development very well**. Some time ago, Java EE applications were deployed to a "heavyweight" application server. Nowadays, most vendors offer lightweight application servers which use very little memory or disk space - as is the case with Payara Server.

Developing microservices with Java EE involves writing standard Java EE applications, while limiting yourself to a certain subset of Java EE APIs, typically JAX-RS and JSON-P; and perhaps some others, like CDI and - if interacting with a relational database - JPA. **Java EE developers can absolutely leverage their existing expertise**, the main requirement in this case is the development of RESTful web services using JAX-RS, then these web services would be packaged in a WAR file and deployed to a lightweight application server as usual.

When using modern, embeddable application servers such as Payara Micro (www.payara.fish/payara_micro), frequently only one application is deployed to each instance of the application server, and, in some cases, the "tables are turned" by having the application server be just a library that the application uses as a dependency. With these modern application servers, several instances of the application server are often deployed to a server, making modern Java EE very suitable for microservices development. Many modern, lightweight Java EE application servers are embeddable, allowing the creation of an "Uber JAR", which includes both the application code and the application server libraries. This "Uber JAR" is then transferred to the server and run as a standalone application. Payara Micro is an example of a lightweight application server that offers this functionality.

In addition to "Uber JARs", modern application servers can be added to a container image (such as Docker), allowing an application to be deployed as a thin WAR, typically only a few kilobytes in size. This approach has the advantage of very fast deployments, usually under 2 seconds.

By deploying to a contemporary Java EE Web Profile compliant application server (or, as explained in the previous paragraph, creating an "Uber JAR"), Java EE developers can certainly leverage their existing expertise to develop microservices compliant applications.

Can Java EE be used to develop microservices then? The answer is a resounding 'yes'!

Java EE developers can leverage their expertise and deploy their code to one of the lightweight application servers suitable for microservices development, such as Payara Micro.

Now that we've established that Java EE is very much suitable for a microservice architecture, let's explore some of the advantages and disadvantages of the microservices approach.

Microservices Advantages

Developing an application as a series of microservices offers several advantages over traditionally designed applications:

- **Smaller codebases** - since each microservice is a small, standalone unit, code bases for microservices tend to be smaller and easier to manage than traditionally designed applications.

- **Microservices encourage good coding practices** - a microservice architecture encourages loose coupling and high cohesion.
- **Greater resilience** - traditionally designed applications act as a single point of failure; if any component of the application is down or unavailable, the whole application is unavailable. Since microservices are independent modules, one component (i.e. one microservice) being down does not necessarily make the whole application unavailable.
- **Scalability** - since applications developed as a series of microservices are composed of a number of different modules, scalability becomes easier; we can focus only on those services that may need scaling without having to waste effort on parts of the application that do not need to be scaled.

Microservices Disadvantages

Developing and deploying applications adhering to a microservice architecture comes with its own set of challenges, regardless of what programming language or application framework is used to develop the application:

- **Additional operational and tooling overhead** - each microservice implementation would require its own (possibly automated) deployment, monitoring systems, etc.
- **Debugging microservices may be more involved than debugging traditional enterprise applications** - if an end-user reports a problem with their application, and internally that application utilizes multiple microservices, it is not always clear which of the microservices may be the culprit. This may be especially difficult if the microservices involved are developed by different teams with different priorities.
- **Distributed transactions may be a challenge** - rolling back a transaction involving several microservices may be hard. A common approach to work around this is to isolate microservices as much as possible, treat them as single units, then have local transaction management for each microservice. For example, if microservice A invokes microservice B, and microservice B runs into a problem, the transaction local to microservice B would roll back and return an HTTP status code of 500 (server error) to microservice A. Microservice A could then use this particular

status code as a trigger to initiate its own compensating transaction to bring the system back to its initial state.

- **Network latency** - since microservices rely on HTTP method calls for communication, performance can suffer due to network latency.
- **Potential for complex interdependencies** - while independent microservices tend to be simple, they are dependent on each other. A microservice architecture can potentially create a complex dependency graph. This situation can be worrisome if some of our services depend on microservices developed by other teams that may have conflicting priorities (i.e. we find a bug in their microservice, however fixing the bug may not be a priority for the other team).
- **Susceptible to the fallacies of distributed computing** - applications developed following a microservice architecture may make some incorrect assumptions, such as network reliability, zero latency, infinite bandwidth, etc.

When developing a brand-new application from scratch, **carefully evaluate the application requirements and weigh them against the various advantages or disadvantages of a microservices architecture,** then decide if implementing the new application by following a microservice architecture would make sense. Migrating existing applications to microservices requires some consideration as well.

MIGRATING TO MICROSERVICES

If we have an existing, traditionally designed application, migrating to microservices may or may not make sense. In this case, not only do we need to consider the benefits vs. the disadvantages of a microservice architecture, we also need to consider the fact that a migration to microservices may not provide much value to our end users. As much as we software developers like to modernize existing applications, the fact of the matter is that redesigning existing applications does not bring direct value to end users. If there are pressing new business requirements or defects to fix, then users may be better served by keeping the existing architecture.

If we decide that migrating to a microservice architecture makes sense, existing legacy systems typically cannot be changed overnight. There are a few approaches we can follow to migrate existing applications to a microservice architecture.

Iterative refactoring

One approach we can use to refactor an existing application to a microservice architecture is to identify the existing components and refactor them as microservices individually. This approach allows us to iteratively refactor existing traditional applications into microservices, whilst mitigating the risk of not implementing new functionality during the redesign. By following this approach, we would eventually end up with our application completely refactored to a microservice architecture.

Partial refactoring

In some cases, it may not be necessary or practical to migrate an existing application completely into a microservice architecture. If portions of the existing application provide functionality that may be useful to other applications, this functionality may be refactored into microservices. Once we do this, our new microservice(s) can be used by other applications, regardless of what programming language was used to implement them. By partially refactoring our application we would end up with a hybrid approach, using microservices only where it makes sense.

Implementing new application requirements as microservices

Existing applications deployed to production rarely remain unchanged, new requirements and enhancement requests come from users all the time. Instead of completely migrating an existing application to microservice architecture, new requirements may be developed as microservices, and our traditionally designed application can invoke the newly developed modules implementing these new requirements. Just like with partial refactoring, by implementing new requirements as microservices we would end up with a hybrid approach, with existing application functionality developed in a more traditional way, and new application functionality developed as microservices.

EXAMPLES USING PAYARA MICRO

Now that we have given a brief introduction to microservices, we are ready to show an example microservice application written using Java EE. Our example application should be very familiar to most Java EE developers, it is a simple

CRUD (Create, Read, Update, Delete) application developed as a series of microservices. The application will follow the familiar MVC design pattern, with the “View” and “Controller” developed as microservices. The application will also utilize the very common DAO (Data, Access, Object) pattern, with our DAO developed as a microservice as well.

The example code is not a full CRUD application, for simplicity, we decided to only implement the “Create” part our CRUD application.

We will be using Payara Micro to deploy our example code. Our application will be developed as three modules, first a microservices client, followed by a microservice implementation of a controller in the MVC design pattern, then an implementation of the DAO design pattern implemented as a microservice.

Developing microservices client code

Before delving into developing our services, we will first develop a microservices client in the form of an HTML5 page using the popular Twitter Bootstrap CSS library, as well as the ubiquitous jQuery JavaScript library. The JavaScript code in the front end service will invoke the controller microservice, passing a JSON representation of user entered data. The controller service will then invoke the persistence service and save data to a database. Each microservice will return an HTTP code indicating success or error condition.

The most relevant parts of our client code are the HTML form and the jQuery code to submit the form to our Controller microservice.

We will only show small snippets of code here, the complete code for the sample application can be found at http://info.payara.fish/hubfs/payara_microprofile_example.zip

Markup for the form in our HTML5 page looks as follows:

```
<form id="customerForm">

    <div class="form-group">

        <label for="salutation">Salutation</label><br/>

        <select id="salutation" name="salutation"
           class="form-control" style="width: 100px !important;">

            <option value=""> </option>

            <option value="Mr">Mr</option>

            <option value="Mrs">Mrs</option>

            <option value="Miss">Miss</option>

            <option value="Ms">Ms</option>

            <option value="Dr">Dr</option>

        </select>

    </div>

    <div class="form-group">

        <label for="firstName">First Name</label>

        <input type="text" maxlength="10" class="form-control"
        id="firstName" name="firstName"  placeholder="First Name">

    </div>

    <div class="form-group">

        <label for="middleName">Middle Name</label>

        <input type="text" maxlength="10" class="form-control"
        id="middleName" name="middleName" placeholder="Middle Name">

    </div>

    <div class="form-group">

        <label for="lastName">Last Name</label>

        <input type="text" maxlength="20" class="form-control"
        id="lastName" name="lastName" placeholder="Last Name">

    </div>

    <div class="form-group">

        <button type="button" id="submitBtn"
        class="btn btn-primary">Submit</button>

    </div>

</form>
```

As we can see, this is a standard HTML form using Twitter Bootstrap CSS classes. Our page also has a script to send form data to the controller microservice.

```
<script>
  $(document).ready(function () {
      $("#submitBtn").on('click', function () {
          var customerData = $("#customerForm").serializeArray();
          $.ajax({
              headers: {
                  'Content-Type': 'application/json'
              },
              crossDomain: true,
              dataType: "json",
              type: "POST",
              url: "http://localhost:8180/CrudController/webresources/customercontroller/",
              data: JSON.stringify(customerData)
          }).done(function (data, textStatus, jqXHR) {
              if (jqXHR.status === 200) {
                  $("#msg").removeClass();
                  $("#msg").toggleClass("alert alert-success");
                  $("#msg").html("Customer saved successfully.");
              } else {
                  $("#msg").removeClass();
                  $("#msg").toggleClass("alert alert-danger");
                  $("#msg").html("There was an error saving customer data.");
              }
          }).fail(function (data, textStatus, jqXHR) {
              console.log("ajax call failed");
              console.log("data = " + JSON.stringify(data));
              console.log("textStatus = " + textStatus);
              console.log("jqXHR = " + jqXHR);
              console.log("jqXHR.status = " + jqXHR.status);
          });
      });
  });
</script>
```

The script is invoked when the Submit button on the page is clicked. It uses jQuery's `serializeArray()` function to collect user-entered form data and create a JSON formatted array with it. The `serializeArray()` function creates an array of JSON objects, each element on the array has a `name` property matching the name attribute on the HTML markup, and a `value` property matching the user-entered value.

For example, if a user selected “Mr” in the salutation drop down, entered “John” in the first name field, left the middle name blank, and entered “Doe” as the last name, the generated JSON array would look like this:

```
[{"name":"salutation","value":"Mr"},{"name":"firstName","value":"John
"},{"name":"middleName","value":""},{"name":"lastName","value":"Doe"}
]
```

Notice that the value of each “name” property in the JSON array above matches the “name” attributes in the HTML form and the corresponding “value” attributes match the user entered values.

Since the generated HTTP request will be sent to a different Payara Micro instance, we need to set the `crossDomain` property of the Ajax settings object to `true`, even though we are deploying all of our microservices to the same server (or, in our case, to our local workstation).

Notice that the `url` property value of the Ajax setting objects has a port of `8180`, as we need to make sure our Controller microservice is listening to this port when we deploy it.

We can deploy our View microservice from the command line as follows:

```
java -jar java -jar payara-micro-4.1.1.164.jar --noCluster --deploy
/path/to/CrudView.war
```

Payara Micro is distributed as an executable JAR file; therefore, we can start it via the `java -jar` command. The exact name of the jar file will depend on the version of Payara Micro you are using.

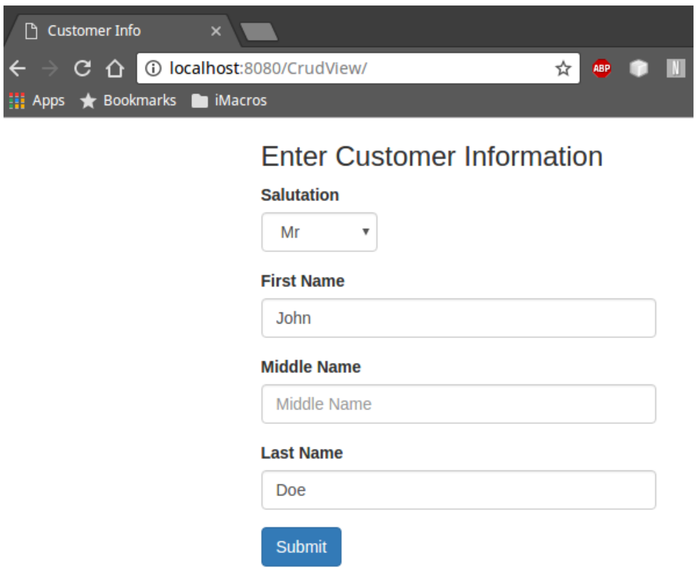
By default, Payara Micro instances running on the same server form a cluster automatically, for our simple example we don't need this functionality, therefore we used the `--noCluster` command line argument.

The `--deploy` command line argument is used to specify the artifact we want to deploy, in our case it is a war file containing the HTML5 page serving as the user interface of our example application.

We can examine Payara Micro output to make sure our application was deployed successfully.

```
[2017-01-18T20:35:54.211-0500] [Payara Micro 4.1] [INFO] [AS-WEB-GLUE-00172] [javax.enterprise.web] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1484789754211] [levelValue: 800] Loading application [CrudView] at [/CrudView]

[2017-01-18T20:35:54.282-0500] [Payara Micro 4.1] [INFO] [] [javax.enterprise.system.core] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1484789754282] [levelValue: 800] CrudView was successfully deployed in 1,477 milliseconds.

[2017-01-18T20:35:54.285-0500] [Payara Micro 4.1] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1484789754285] [levelValue: 800] No META-INF/deploy directory

[2017-01-18T20:35:54.285-0500] [Payara Micro 4.1] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1484789754285] [levelValue: 800] Deployed 1 archives
```

We can now point our browser to our CrudView application URL (http://localhost:8080/CrudView in our example). After entering some data the page will look as shown in the following screenshot.



Enter Customer Information

Salutation: Mr

First Name: John

Middle Name: Middle Name

Last Name: Doe

When the user clicks on the Submit button, the client passes a JSON representation of user-entered data to the controller service.

The controller service

The controller service is a standard RESTful web service implementation of a controller in the MVC design pattern.

```
package fish.payara.crudcontroller.service;

//imports omitted for brevity

@Path("/customercontroller")

public class CustomerControllerService {

    public CustomerControllerService() {

    }

    @OPTIONS

    public Response options() {

        return Response.ok("")

                .header("Access-Control-Allow-Origin",
                        "http://localhost:8080")

                .header("Access-Control-Allow-Headers", "origin,” +
                        “content-type, accept, authorization")

                .header("Access-Control-Allow-Credentials", "true")

                .header("Access-Control-Allow-Methods",
                        "GET, POST, PUT, DELETE, OPTIONS, HEAD")

                .header("Access-Control-Max-Age", "1209600")

                .build();

    }

    @POST

    @Consumes(MediaType.APPLICATION_JSON)

    public Response addCustomer(String customerJson) {

        Response response;

        Response persistenceServiceResponse;

        CustomerPersistenceClient client =
          new CustomerPersistenceClient();

        Customer customer = jsonToCustomer(customerJson);
        persistenceServiceResponse = client.create(customer);
        client.close();
        if (persistenceServiceResponse.getStatus() == 201) {
            response = Response.ok("{}").
                    header("Access-Control-Allow-Origin",
                           "http://localhost:8080").build();
        } else {
            response = Response.serverError().
                    header("Access-Control-Allow-Origin",
"http://localhost:8080").build();
        }
        return response;
    }
    private Customer jsonToCustomer(String customerJson) {
        Customer customer = new Customer();
        JsonArray jsonArray;
        try (JsonReader jsonReader = Json.createReader(
                new StringReader(customerJson))) {
            jsonArray = jsonReader.readArray();
        }
        for (JsonValue jsonValue : jsonArray) {
            JsonObject jsonObject = (JsonObject) jsonValue;
            String propertyName = jsonObject.getString("name");
            String propertyValue = jsonObject.getString("value");
            switch (propertyName) {
                case "salutation":
                    customer.setSalutation(propertyValue);
                    break;
                case "firstName":
                    customer.setFirstName(propertyValue);
                    break;
                case "middleName":
                    customer.setMiddleName(propertyValue);
                    break;
                case "lastName":
                    customer.setLastName(propertyValue);
                    break;
                default:
                    LOG.log(Level.WARNING, String.format(
                            "Unknown property name found: %s",
                             propertyName));
                    break;
            }
        }
        return customer;
    }
}
```

The `options()` method, annotated with the `javax.ws.rs.OPTIONS` annotation, is necessary since the browser automatically calls it before invoking the actual post request containing the main logic of our server. In this method we set some header values to allow CORS (Cross-Origin Resource Sharing), which in simple terms means we allow our service to be invoked from a different server than the one where our service is running. In our case, the client is deployed to a different instance of Payara Micro, therefore it is considered a different origin, these headers are necessary to allow our client code and controller service to communicate with each other. Notice that we explicitly allow requests from http://localhost:8080, which is the host and port where our client code is deployed.

The main logic of our controller service is in the `addCustomer()` method. This method receives the JSON string sent by the client as a parameter. In this method, we create an instance of `CustomerPersistenceClient`(), which is a client for the persistence service implemented using the JAX-RS client API.

We then create an instance of a `Customer` class by invoking the `jsonToCustomer()` method, this method takes the JSON string sent by the client, and, using the standard Java EE JSON-P API, populates an instance of the Customer class with the corresponding values in the JSON string.

Our `addCustomer()` method then invokes the persistence service by invoking the `create()` method on `CustomerPersistenceClient`, checks the HTTP status code returned by the persistence service, then returns a corresponding status code to the client.

The Customer class is a simple Data Transfer Object (DTO), containing a few properties matching the input fields in the form in the client, plus corresponding getters and setters. The class is so simple we decided not to show it.

Let's now take a look at the implementation of our JAX-RS client code.

```
public class CustomerPersistenceClient {


    private final WebTarget webTarget;

    private final Client client;

    private static final String BASE_URI =
      "http://localhost:8280/CrudPersistence/webresources";


    public CustomerPersistenceClient() {

        client = javax.ws.rs.client.ClientBuilder.newClient();

        webTarget =
          client.target(BASE_URI).path("customerpersistence");

    }


    public Response create(Customer customer) throws
      ClientErrorException {

        return webTarget.request(
          javax.ws.rs.core.MediaType.APPLICATION_JSON).
          post(javax.ws.rs.client.Entity.entity(customer,
          javax.ws.rs.core.MediaType.APPLICATION_JSON),
          Response.class);

    }


    public void close() {

        client.close();

    }

}
```

As we can see, our client code is a fairly simple class making use of the JAX-RS client API: We declare a constant containing the base URI of the service we are invoking (our persistence service), and then create a new instance of `javax.ws.rs.client.ClientBuilder` in the class constructor and set its base URI and path to match the appropriate values for our persistence service. Our client class has a single method, which submits an HTTP POST request to the persistence service, then returns the response sent back from it.

Our controller service only uses two Java EE APIs, JAX-RS and JSON-P. Both APIs are supported by the MicroProfile initiative, therefore we can deploy it to the scaled-down version of Payara Micro,

```
java -jar payara-microprofile-1.0-4.1.1.164.1.jar --noCluster --port
8180 --deploy /path/to/CrudController.war
```

By examining Payara Micro's output we can see that our code deployed successfully.

```
[2017-01-19T19:34:05.758-0500] [Payara Microprofile 4.1] [INFO] [AS-WEB-GLUE-00172]
[javax.enterprise.web] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1484872445758]
[levelValue: 800] Loading application [CrudController] at [/CrudController]

[2017-01-19T19:34:05.828-0500] [Payara Microprofile 4.1] [INFO] [] [javax.enterprise.system.core]
[tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1484872445828] [levelValue: 800]
CrudController was successfully deployed in 7,005 milliseconds.

[2017-01-19T19:34:05.830-0500] [Payara Microprofile 4.1] [INFO] [] [PayaraMicro] [tid:
_ThreadID=1 _ThreadName=main] [timeMillis: 1484872445830] [levelValue: 800] No META-
INF/deploy directory

[2017-01-19T19:34:05.830-0500] [Payara Microprofile 4.1] [INFO] [] [PayaraMicro] [tid:
_ThreadID=1 _ThreadName=main] [timeMillis: 1484872445830] [levelValue: 800] Deployed 1
archives
```

Now that we have successfully deployed our controller service, we are ready to go through the final component of our application - the persistence service.

The persistence service

Our persistence service is a JAX-RS RESTful web service; it is a thin wrapper over a class implementing the DAO design pattern.

```
package fish.payara.crudpersistence.service;


//imports omitted for brevity
@ApplicationScoped
@Path("customerpersistence")
public class CustomerPersistenceService {
    @Context
    private UriInfo uriInfo;
    @Inject
    private CrudDao customerDao;


    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response create(Customer customer) {
        try {
            customerDao.create(customer);
        } catch (Exception e) {
            return Response.serverError().build();
        }
        return Response.created(uriInfo.getAbsolutePath()).build();
    }
}
```

In this case, since the client code invoking our service is developed in Java, there is no need to convert the JSON string we receive to Java code, this is done automatically under the covers. Our `create()` method is invoked when the controller service sends an HTTP POST request to the persistence service, invoking the `create()` method of a class implementing the DAO design pattern. Our persistence service returns an HTTP response 201, if everything goes well, and returns an HTTP 500 error (Internal Server Error) of the DAO's create method throws an exception.

Our DAO is implemented as a CDI managed bean, using JPA to insert data into the database.

```
package fish.payara.crudpersistence.dao;

//imports omitted for brevity

@ApplicationScoped

@Transactional

public class CrudDao {

    @PersistenceContext(unitName = "CustomerPersistenceUnit")

    private EntityManager em;


    public void create(Customer customer) {

        em.persist(customer);

    }

}
```

Our DAO couldn't be much simpler, it implements a single method that invokes the `persist()` method on an injected instance of `EntityManager`.

In our persistence service project, the `Customer` class is a trivial JPA entity

We now deploy our persistence service as usual.

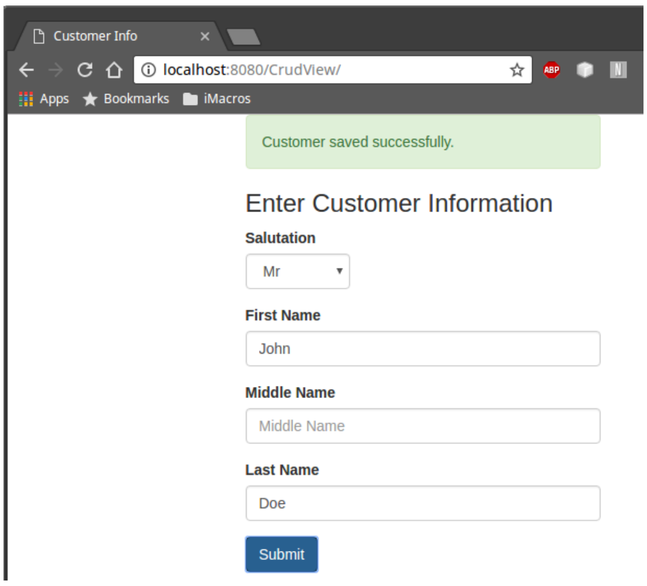
```
java -jar payara-micro-4.1.1.164.jar --port 8280 --noCluster --deploy
/path/to//CrudPersistence.war
```

Examining Payara Micro's output we can see that our persistence service was deployed successfully.

```
[2017-01-21T13:13:20.503-0500] [Payara Micro 4.1] [INFO] [AS-WEB-GLUE-00172] [javax.enterprise.web] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1485022400503] [levelValue: 800] Loading application [CrudPersistence] at [/CrudPersistence]

[2017-01-21T13:13:20.573-0500] [Payara Micro 4.1] [INFO] [] [javax.enterprise.system.core] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1485022400573] [levelValue: 800] CrudPersistence was successfully deployed in 2,606 milliseconds.

[2017-01-21T13:13:20.575-0500] [Payara Micro 4.1] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1485022400575] [levelValue: 800] No META-INF/deploy directory

[2017-01-21T13:13:20.575-0500] [Payara Micro 4.1] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1485022400575] [levelValue: 800] Deployed 1 archives
```

Now that we have deployed all three components of our application, we are ready to see it in action.

Once the user enters some data and clicks the submit button, we should see a “success” message at the top of our page.



Customer saved successfully.

Enter Customer Information

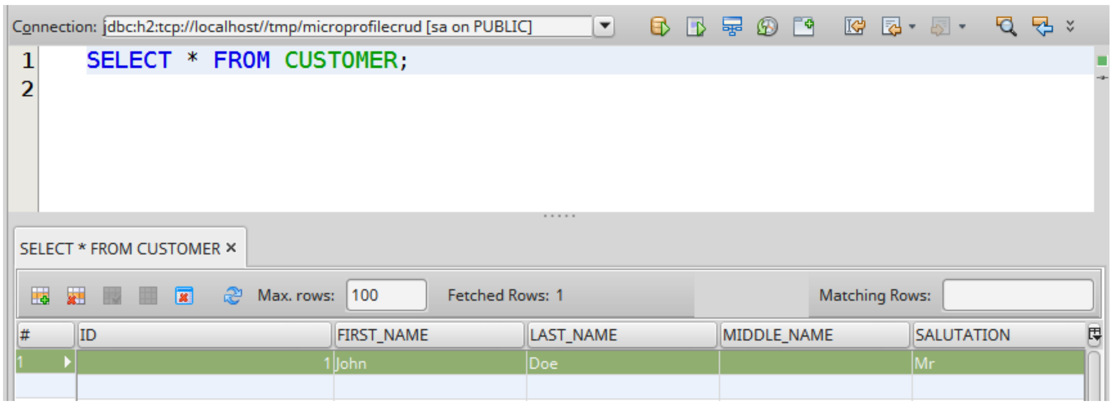
Salutation: Mr

First Name: John

Middle Name: Middle Name

Last Name: Doe

If we take a look at the database, we should see that the user-entered data was persisted successfully.



Connection: jdbc:h2:tcp://localhost//tmp/microprofilecrud [sa on PUBLIC]

```
SELECT * FROM CUSTOMER;
```

SELECT * FROM CUSTOMER

Max. rows: 100 Fetched Rows: 1 Matching Rows:

#	ID	FIRST_NAME	LAST_NAME	MIDDLE_NAME	SALUTATION
1	1	John	Doe		Mr

If you are familiar with Java EE, it is likely that very little if any of the example code we showed is new to you.

Now that we've seen how we can implement a microservice architecture by leveraging current Java EE APIs, let's take a look at what new Java EE features will be introduced in the near future to help develop Java EE microservices even easier.

As shown by our example code, developing applications following a microservice architecture in Java EE is very simple.

It doesn't require any special knowledge, microservices are developed using standard Java EE APIs and deployed to a lightweight application server.

JAVA EE 8, JAVA EE 9 AND MICROPROFILE

Java EE 8 is scheduled to be released in the Fall of 2017 (about 6 months from the time of writing), Java EE 9 is scheduled to be released a year after that, plus an independent microservices focused initiative called MicroProfile is already available.

Java EE 8

One welcome addition to Java EE 8 will be the inclusion of a brand new Java API for JSON Binding (JSON-B), which will provide a standard way of populating Java objects from JSON, and vice versa. Currently we need to rely on manually writing code to populate Java from JSON, or use a non-standard APIs to populate Java objects from JSON (i.e. Jackson, GSON, MOxy, etc).

Java EE 8 will include a new version of JAX-RS, namely JAX-RS 2.1, which will introduce support for the aforementioned JSON-B API, as well as Server Sent Events (SSE), improved CDI integration, non-blocking interceptors via the NIO API, and declarative security, among other new features.

Java EE 8 will include a new version of the Servlet API, namely Servlet 4.0, which will include support for the HTTP/2 protocol. HTTP/2 reduces network latency and allows server push (that is, data can be sent from the server to the client without having to wait for a client request).

New versions of CDI, Bean Validation will also be included with Java EE 8, which may come in handy regardless of if we are developing microservices or a traditionally designed application.

Java EE 9 will add even more new features and APIs that may be useful when developing microservices.

Java EE 9

Java EE 9 is expected to be released after Java SE 9, and as such Java EE 9 will be able to take advantage of Project Jigsaw, which provides modularity to the Java platform. Think of project Jigsaw as a standard way to declare library dependencies at the JVM level, similar to what we can do today with Maven.

The Java language has been around for 21 years, but technologies that were popular when Java was first released are not so popular today (CORBA, for instance). For backward compatibility purposes, modern versions of Java need to support these legacy technologies.

Project Jigsaw will allow Java application developers to pick and choose what technologies they need in their applications, eliminating the need to carry all this legacy baggage from 21 years ago.

Once project Jigsaw is released, Java EE library dependencies may be declared as project Jigsaw modules, and included only if the application being developed requires it. This will eliminate the need for the concept of Java EE profiles, as application developers will be able to freely include only necessary Java EE API's in their applications.

There are plans to modify JAX-RS in Java EE 9 to include first class support for the Circuit Breaker design pattern. This pattern allows an application to stop invoking a service after a predetermine number of attempts, returning instead a cached response or an error.

Other Java EE 9 features suitable for microservices include a new Event API , support for NoSQL databases and a new State Management API.

MicroProfile

Java EE 8 is scheduled to be released around October 2017, and Java EE 9 should be delivered about a year after that.

Although Java EE 8 and particularly 9 will offer better support for microservices, we as application developers don't need to wait until new Java EE versions are released to start developing microservice compliant Java EE applications. We can instead choose to utilize one of the several existing MicroProfile implementations, such as Payara Micro or other MicroProfile members' offerings. It is worth noting that the idea behind MicroProfile is not to compete directly with Java EE, but to come up with innovative ways of implementing microservices with Enterprise Java, which later may be incorporated into the Java EE standard.

Our example controller service uses the scaled down version of Payara Micro, which - as previously mentioned - is an implementation of the MicroProfile initiative.

The idea behind the MicroProfile initiative is to create innovative new microservice-oriented APIs such as distributed configuration and fault tolerance. The current initial version of the MicroProfile initiative currently supports JAX-RS (Java API for RESTful Web Services), CDI (Contexts and Dependency Injection), and JSON-P (the Java API for JSON processing). Most application server vendors have an embedded version of their products, employing these embedded application servers allows application developers to deploy their microservices as standalone modules.

Think of it this way - you are deploying your application server as part of the application. Potentially, multiple MicroProfile compliant microservices could be deployed to a single server, each one of which would have its own embedded copy of the application server.

When using an application server that supports MicroProfile, such as Payara Micro, application developers would again be primarily developing RESTful web services. It is worth repeating that MicroProfile currently only supports three Java EE APIs (JAX-RS, CDI and JSON-P) therefore application developers would be wise to try and limit themselves to these three APIs in order to keep their deployment artifacts small. This is not to say that other Java EE APIs cannot be used - most modern Java EE application servers either allow,

MicroProfile

The MicroProfile initiative (www.microprofile.io) is a community collaboration between several application server vendors (Payara Services, IBM, Red Hat and Tomitribe) plus the London Java Community and the Brazilian SouJava Java Users Group. The MicroProfile initiative became an Eclipse Foundation project in December 2016. Having an independent foundation being the steward prevents any single vendor from having too much control over the initiative.

or will allow in the near future, the creation of composable microruntimes, meaning that the application server can be configured to use only the Java EE APIs that our application needs.

It may be worth mentioning that microservices developed against Java EE APIs can be developed using any popular IDE such as NetBeans, Eclipse or IntelliJ IDEA, and standard build tools such as Maven or Gradle can be used, leveraging Java developer expertise using these tools.

CONCLUSION

As we can see, Java EE is very suitable for microservices development. Java EE developers can leverage their existing knowledge to develop a microservice architecture and deploy them to modern, lightweight application servers such as Payara Micro. Additionally, it isn't necessary to "throw the baby out with the bath water" so to speak, when migrating to microservices. Traditional Java EE applications can interact with microservices quite well, and can be refactored iteratively into a microservices architecture when it makes sense. Whether developing new applications following a microservice architecture, refactoring an existing application to microservices, or modifying existing applications to interact with microservices, Java EE developers can leverage their existing skills for the task at hand.

About the Author

David Heffelfinger is an independent consultant focusing on Java EE. He is a frequent speaker at Java conferences and is the author of several books on Java and related technologies, such as "Java EE 7 Development with NetBeans 8", "Java EE 7 with GlassFish 4 Application Server" and others and has over 20 years of software architecture, design and development experience.

David was named by TechBeacon as one of 39 Java leaders and experts to follow on Twitter. You can follow David on Twitter at @ensode

PAYARA MICRO – SMALL, SIMPLE, SERIOUS

Payara Micro enables you to run WAR files from the command line without any application server installation.

It is small, <70 MB in size, and incredibly simple to use. With its automatic and elastic clustering, Payara Micro is designed for running Java EE applications in a modern containerized / virtualized infrastructure, using automated provisioning tools like Chef, Ansible or Puppet.

That’s not all! Using the Hazelcast integration, each Payara Micro process will automagically cluster with other Payara Micro processes on the network, giving web session resilience and a fully distributed data cache using Payara’s JCache support.

Find out more & download Payara Micro from: www.payara.fish/payara_micro

sales@payara.fish

+44 207 754 0481

www.payara.fish