User Guide

Demystifying Microservices for Jakarta EE Developers

David Heffelfinger

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

Contents

Microservices are a modern way to develop and design enterprise application where each “piece” is independently deployable to allow development teams to build new components without breaking the entire app. In this article we aim to cut through the hype, explaining microservices in terms that make sense to Java EE and Jakarta EE developers, while covering when a microservices architecture makes sense, as well as when it doesn’t.

What are Microservices?

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

What is Jakarta EE?

Java EE is a set of standard APIs for server-side Java development, including traditional enterprise applications and microservices.

In 2017, Oracle announced that it would be donating Java EE to the Eclipse Foundation, with one of the conditions of the transition being that the technology would have to be renamed. The community chose Jakarta EE as the new name for Java EE, under the reigns of the Eclipse Foundation.

Java EE / Jakarta EE’s main advantage is that, being a standard, there are multiple implementations, therefore code written against one implementation can be easily ported to another implementation with few or no changes.

Moving Java EE to a vendor independent foundation such as Eclipse, has the advantage that no one company or organization has too much influence over the standard. Additionally, previously organizations interested in certifying their products as “Java EE Certified” would have to pay a large amount of money to get access to a certification tool called the Technology Compatibility Kit (TCK), as part of the donation to the Eclipse Foundation, the TCK was open sourced and made freely available, which should open the door for additional Jakarta EE compatible products. Because of these and other advantages, the move of Java EE to the Eclipse Foundation was very well received by the Java community.

Can Jakarta EE do Microservices?

Some may think that Jakarta EE is “too heavyweight” for microservices development, but this is simply not the case. Because of this misconception of Jakarta EE being heavyweight, some may think that Jakarta EE may not be suitable for a microservices architecture, when in reality Jakarta EE fits microservices development well. Some time ago, Java EE applications were typically deployed to a “heavyweight” application server, creating the perception that a big, heavy application server is needed when working with Java EE applications. Nowadays, most Jakarta EE vendors offer lightweight runtimes which use very little memory or disk space, as is the case with Payara Micro.

Developing microservices with Jakarta EE involves writing standard Jakarta EE applications, while limiting yourself to a certain subset of Jakarta EE APIs, typically JAX-RS and JSON-B, CDI and, if interacting with a relational database, JPA. Jakarta 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 light-weight runtime as usual.

When using modern, embeddable Jakarta EE runtimes such as Payara Micro, frequently only one application is deployed to each instance of the runtime. With these modern Jakarta EE runtimes, several instances of the runtimes are often deployed to a server, frequently through containerization and orchestration tools (such as Docker and Kubernetes), making Jakarta EE very suitable for microservices development. Jakarta EE developers can certainly leverage their existing expertise to develop microservices compliant applications.

Can Jakarta EE be used to develop microservices then? The answer is a resounding “yes!”. Current crop of Jakarta EE runtimes are light weight and suitable for microservices development. Jakarta EE developers can leverage their expertise and deploy their code to one of these lightweight runtimes, such as Payara Micro.

MicroProfile

MicroProfile is a set of APIs that provide functionality commonly needed when developing micro-services, it is meant to be used in conjunction with Jakarta EE, providing functionality that augments Jakarta EE’s microservice capabilities. MicroProfile is a community collaboration between several companies and Java User Groups and, just like Jakarta EE, it is stewarded by the Eclipse Foundation.

The following table summarizes some of the APIs provided by MicroProfile:

MicroProfile API	Description
OpenTracing	Allows us to trace microservice invocations across server log files and allows log aggregation tools to analyze data of a request over different endpoints
Open API	Generates documentation for microservices and allows generation of client code capable of calling the generated OpenAPI endpoint.
REST Client	Provides functionality to easily develop clients for RESTful web services.
Config	Provides an easy way to configure applications via different sources such as config files, environment variables, etc.
Fault Tolerance	Provides functionality to built resilient microservices
Metrics	Provides a way for servers to export Monitoring data
JWT	Provides security by allowing integration with JSON Web Tokens (JWT)
Health	Provides a way for automated tools to check the status of our applica-tions and runtimes

Now that we've established that Jakarta EE is very much suitable for a microservices 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 microservices 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

ser- vices that may be need scaling, without having to waste effort on parts of the application that do not need to be scaled.

- **Polyglot Applications**, Microservices are typically developed as RESTful web services, and share data in standard JSON format, for this reason we are not locked to any particular language when developing individual microservices, our application could potentially be composed of microservices written in different programming languages. Microservices can be written in the language best suited for the task it must perform.

Microservices Disadvantages

Developing and deploying applications adhering to a microservices 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, there is a problem with microservice B, a local transaction in microservice B would roll back, then it would return an HTTP status code 500 (server error) to microservice A, microservice A could then use this HTTP status code as a signal to initiate a compensating transaction to bring the system back to its initial state.
- **Network latency**, since microservices rely on HTTP method calls for communications, due to network latency sometimes performance can suffer.
- **Potential for complex interdependencies**, while independent microservices tend to be simple, they are dependent on each other. A microservices 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 follow- ing a microservices architecture may make some incorrect assumptions such as network reliability, zero latency, infinite bandwidth, etc.

When developing a brand-new application from scratch, we should carefully evaluate our application requirements and weight them against the various advantages or disadvantages of a microservices architecture, then decide if implementing the new application by following a microservices archi- tecture would make sense. Migrating existing applications to microservices requires some consid- eration 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 disadvantages of a microservices architecture, but we need to consider the fact that a microservices migration 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 our users may be better served by keeping the existing architecture.

If we decide that migrating to a microservices 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 microservices architecture.

Iterative Refactoring

One approach we can use to refactor an existing application to a microservices architecture is to identify an existing component from an existing traditional architecture and refactor it as a micro-service, then refactor code utilizing this module to invoke your new microservice, then pick a new component and refactor it as a microservice, so on and so forth, until the whole application has been migrated to a microservices architecture. This approach allows us to iteratively refactor existing traditional applications into microservices, while mitigating the risk of not implementing new functionality while the application is being redesigned. By following this approach, we would eventually end up with our application completely refactored to a microservices architecture.

Partial Refactoring

In some cases, it may not be necessary or practical to migrate an existing application completely into a microservices architecture. If portions of the existing application provide functionality that may useful to other applications, this functionality may be refactored as microservices. Once we do this, our new microservice can be used by other applications, regarding 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

Additionally, existing applications deployed to production rarely remain unchanged, new requirements and enhancement requests come from the users all the time. Instead of completely migrating an existing application to microservices, 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.

Microservices Examples using Payara Micro

Now that we have given a brief introduction to microservices, we are ready to show an example microservices application written using Jakarta EE. Our example application should be very familiar to most Jakarta 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 pattern, with our DAO developed as a microservice as well.

Actually, 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. Payara Micro supports Java and Jakarta EE 8 web profile applications with some extensions for full profile features.

Our application will be developed as three modules, first a microservices client, followed by a microservices 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 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 https://github.com/payara/Payara-Examples/tree/master/payara-micro/demystifying-microservices-example.

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 () {
                // click on button submit
                $("#submitBtn").on('click', function () {
                    var customerData = $("#customerForm").serializeArray();

                    var customerDataJsonObj = objectifyForm(customerData);

                    $.ajax({
                        headers: {
                            'Content-Type': 'application/json'
                        },
                        crossDomain: true,
                        dataType: "json",
                        type: "POST",
                        url: "http://localhost:8180/CrudController/
                             webresources/customercontroller/",
                        data: JSON.stringify(customerDataJsonObj)
                    }).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");
                    });
                });
            });
            function objectifyForm(formArray) {//serialize data function
                var returnJsonObj = {};
                for (var i = 0; i < formArray.length; i++){
                  returnJsonObj[formArray[i]['name']] = formArray[i]['value'];
                }
                return returnJsonObj;
            }
        </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 as follows:

```
[{"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, the corresponding "value" attributes match the user entered values. For convenience, we wrote a JavaScript function to convert the array to a single JSON object, we pass the string representation of the resulting JSON object as the value of the of the data attribute of the Ajax settings object.

Since the generated HTTP request will be sent to a different instance of Payara Micro, 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 object has a port of `8180`, 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 payara-micro.jar --port 8080 --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, 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.

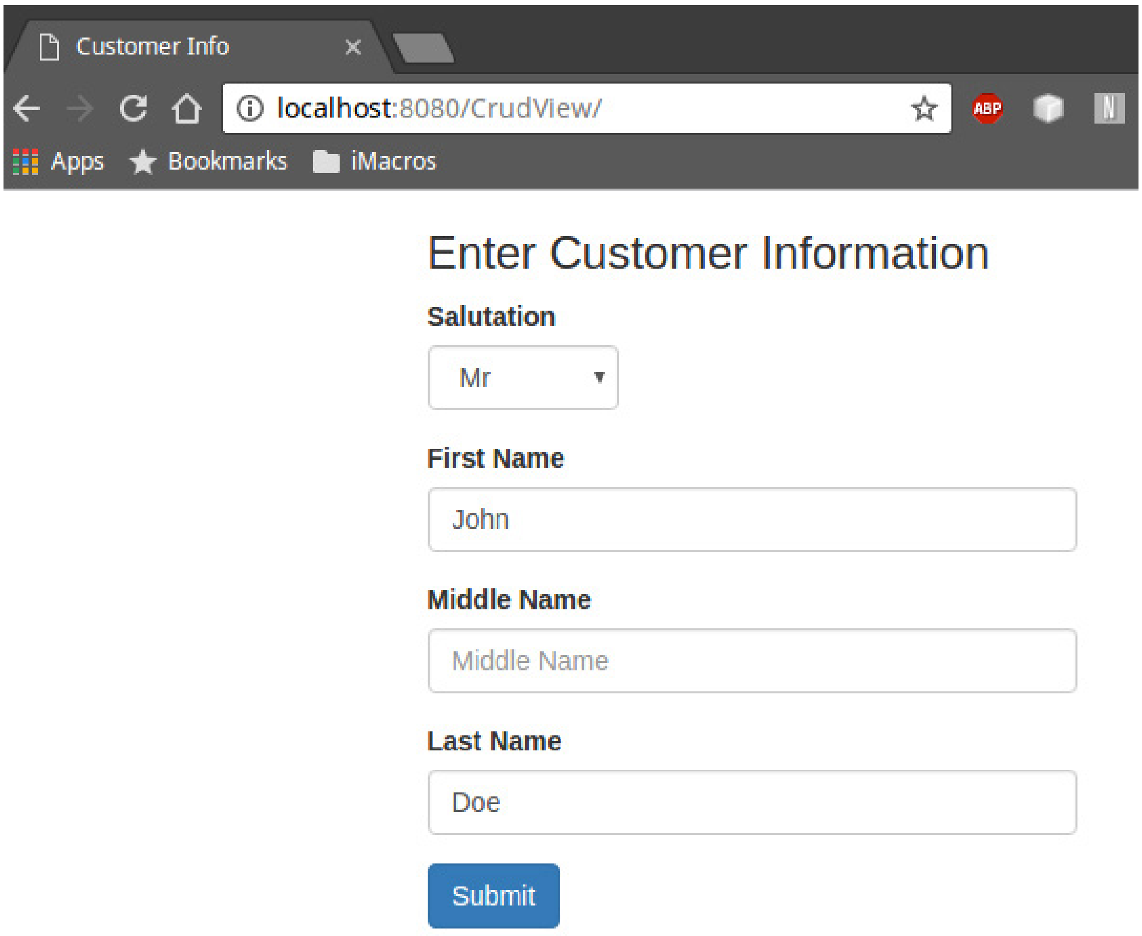
The --deploy command line argument is used to specify the artifact we want to deploy, in our case it is a war file containing the HTML5 page serving as the user interface of our example application.

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

```
[2019-03-05T17:34:07.079-0500] [] [INFO] [AS-WEB-GLUE-00172] [javax.enterprise.
web] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1551825247079]
[levelValue: 800] Loading application [CrudView] at [/CrudView]

[2019-03-05T17:34:07.151-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1551825247151] [levelValue: 800] [[

{
    "Instance Configuration": {
        "Host": "pop-os.localdomain",
        "Http Port(s)": "8080",
        "Https Port(s)": "",
        "Instance Name": "payara-micro",
        "Instance Group": "no-cluster",
        "Deployed": [
            {
                "Name": "CrudView",
                "Type": "war",
                "Context Root": "/CrudView"
            }
        ]
    }
}]]

[2019-03-05T17:34:07.155-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1551825247155] [levelValue: 800] [[

Payara Micro URLs:
http://pop-os.localdomain:8080/CrudView
]]

[2019-03-05T17:34:07.156-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1
_ThreadName=main] [timeMillis: 1551825247156] [levelValue: 800] Payara Micro
5.191 #badassmicrofish (build 94) ready in 7,882 (ms)
```

As we can see, Payara Micro's output provides lots of useful configuration information, including the URL of the application we just deployed, ready for copying and pasting.

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 {

    @Inject
    @RestClient
    private CustomerPersistenceClient customerPersistenceClient;

    @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(Customer customer) throws
      URISyntaxException {

        Response response = null;
        Response persistenceServiceResponse;

        try {
            persistenceServiceResponse =
              customerPersistenceClient.create(customer);
            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();
            }
        } catch (Exception e) {
            LOG.log(Level.SEVERE,
              "Exception while processing request", e);
            response = Response.serverError().
                    header("Access-Control-Allow-Origin",
                      "http://localhost:8080").build();
        }
        return response;
    }
```

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 accepts an instance of a Customer class as a parameter, since the properties of our Customer object match the names and values of the JSON string sent by the client, the Customer object is automatically populated from the JSON string, with no additional effort on our part.

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.

Our `CustomerController` service takes advantage of the MicroProfile REST client API, at the class level, we inject an instance of *`CustomerPersistenceClient`*, a custom class serving as a client to our Persistence service.

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.

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

```
package fish.payara.crudcontroller.restclient;
//Imports omitted
@Path("/webresources/customerpersistence")
@RegisterRestClient
public interface CustomerPersistenceClient {
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response create(Customer customer);

}
```

As we can see, our client code is an interface, we didn’t have to write the actual client code implementation, the actual code is generated by the MicroProfile REST client API dynamically at deploy time! In order for this to work properly, the interface needs to be annotated with the `@RegisterRestClient` annotation, and the value of its `@Path` annotation needs to match the relative path of the web service we are invoking. Additionally, method in our interface need to match the signature of the Web Service endpoint they are invoking.

The base URL of the web service is specified in a configuration file, the logic to read it is provided by the MicroProfile Config API. The file needs to be named `microprofile-config.properties` and needs to be placed in the `META-INF` directory of the war file containing our code.

```
fish.payara.crudcontroller.restclient.CustomerPersistenceClient/mp-rest/
url=http://localhost:8280/CrudPersistence
```

By convention, the property name for our REST client interface is the fully qualified name of the client interface, followed by “`/mp-rest/url=`”, then the actual value for the base URL of the web service we are calling.

We can deploy our controller service to Payara Micro by issuing a command similar to the following:

```
java -jar payara-micro.jar --port 8180 --noCluster -deploy /path/to/
CrudController.war
```

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

```
[2019-03-05T17:33:43.485-0500] [] [INFO] [AS-WEB-GLUE-00172] [javax.enterprise.
web] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1551825223485]
[levelValue: 800] Loading application [CrudController] at [/CrudController]

[2019-03-05T17:33:43.549-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1551825223549] [levelValue: 800] [[

{
    "Instance Configuration": {
        "Host": "pop-os.localdomain",
        "Http Port(s)": "8180",
        "Https Port(s)": "",
        "Instance Name": "payara-micro",
        "Instance Group": "no-cluster",
        "Deployed": [
            {
                "Name": "CrudController",
                "Type": "war",
                "Context Root": "/CrudController"
            }
        ]
    }
}]]

[2019-03-05T17:33:43.557-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1551825223557] [levelValue: 800] [[

Payara Micro URLs:
http://pop-os.localdomain:8180/CrudController

'CrudController' REST Endpoints:
GET     /CrudController/webresources/application.wadl
OPTIONS /CrudController/webresources/customercontroller
POST    /CrudController/webresources/customercontroller

]]

[2019-03-05T17:33:43.557-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1
_ThreadName=main] [timeMillis: 1551825223557] [levelValue: 800] Payara Micro
5.191 #badassmicrofish (build 94) ready in 7,922 (ms)
```

Since in this case we implemented some RESTful web services using JAX-RS, Payara Micro includes the endpoints for our services in its output.

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();
    }
}
```

Just like in our Controller service, 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, this method simply invokes a `create()` method on a class implementing the DAO design pattern. Our persistence service returns an HTTP response 201 (Created), if everything goes well, if the DAO's `create()` method throws an exception, then our service returns an HTTP error 500 (Internal Server Error).

Notice that the signature of our `create()` method matches the signature of the method of the same name in our `CustomerPersistenceClient` interface, this is necessary so that the MicroProfile REST client API can successfully generate the REST client code of our service.

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.

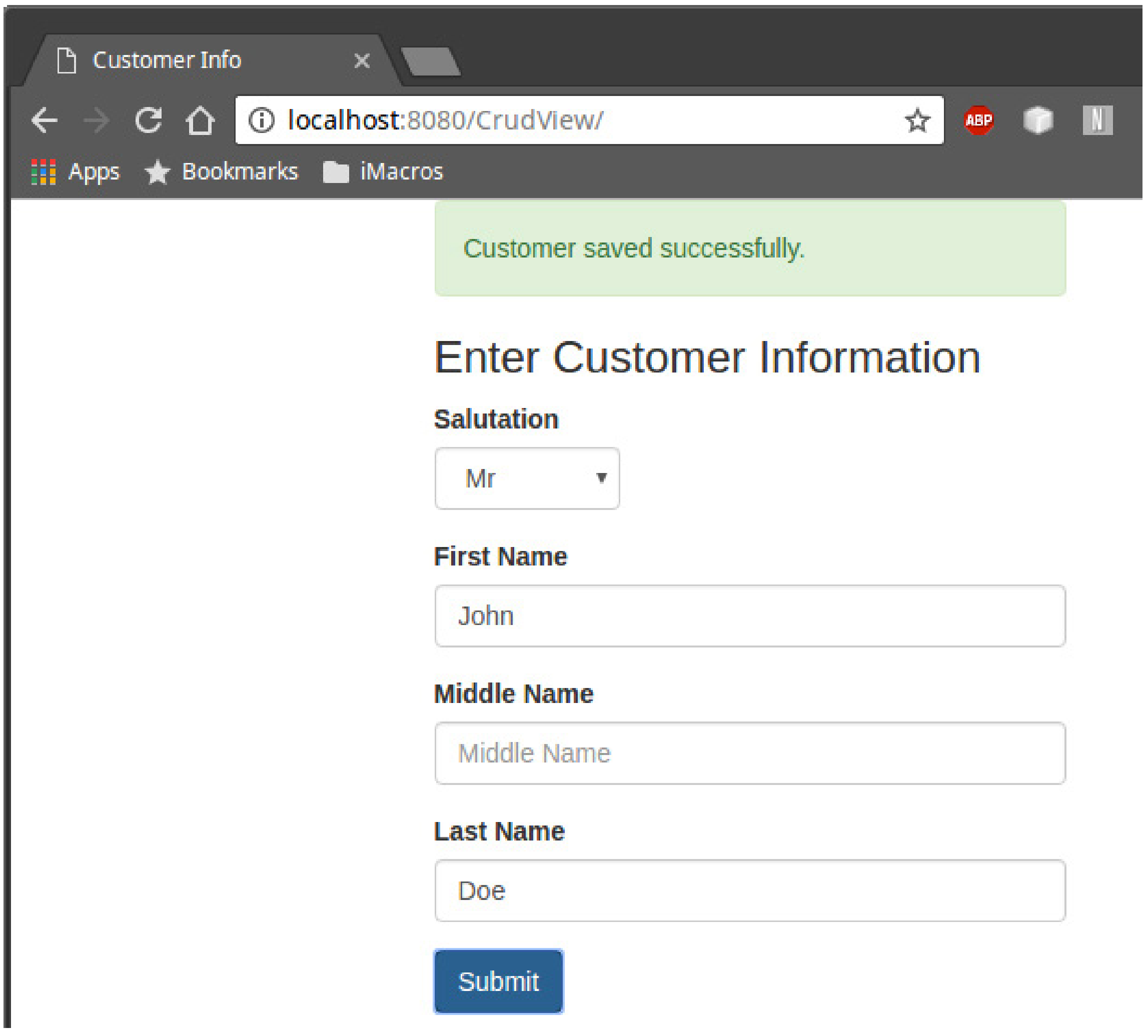
```
java -jar payara-micro.jar --port 8280 --noCluster -deploy /path/to/
CrudPersistence.war
```

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

```
[2019-03-05T19:11:05.630-0500] [] [INFO] [AS-WEB-GLUE-00172] [javax.enterprise.
web] [tid: _ThreadID=1 _ThreadName=main] [timeMillis: 1551831065630]
[levelValue: 800] Loading application [CrudPersistence] at [/CrudPersistence]

[2019-03-05T19:11:05.672-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1551831065672] [levelValue: 800] [[

{
    "Instance Configuration": {
        "Host": "pop-os.localdomain",
        "Http Port(s)": "8280",
        "Https Port(s)": "",
        "Instance Name": "payara-micro",
        "Instance Group": "no-cluster",
        "Deployed": [
            {
                "Name": "CrudPersistence",
                "Type": "war",
                "Context Root": "/CrudPersistence"
            }
        ]
    }
}]]

[2019-03-05T19:11:05.679-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1 _
ThreadName=main] [timeMillis: 1551831065679] [levelValue: 800] [[

Payara Micro URLs:
http://pop-os.localdomain:8280/CrudPersistence

'CrudPersistence' REST Endpoints:
GET   /CrudPersistence/webresources/application.wadl
POST  /CrudPersistence/webresources/customerpersistence

]]

[2019-03-05T19:11:05.679-0500] [] [INFO] [] [PayaraMicro] [tid: _ThreadID=1
_ThreadName=main] [timeMillis: 1551831065679] [levelValue: 800] Payara Micro
5.191 #badassmicrofish (build 94) ready in 11,718 (ms)
```

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

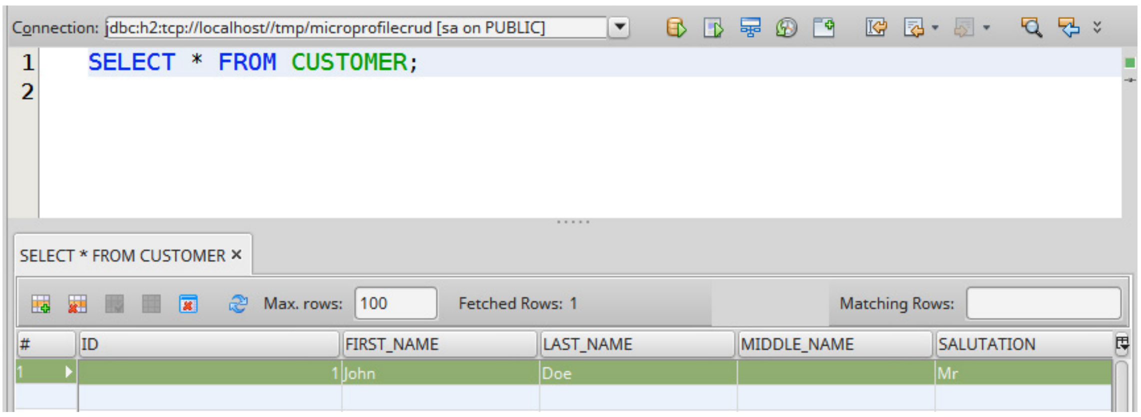
Salutation: Mr

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

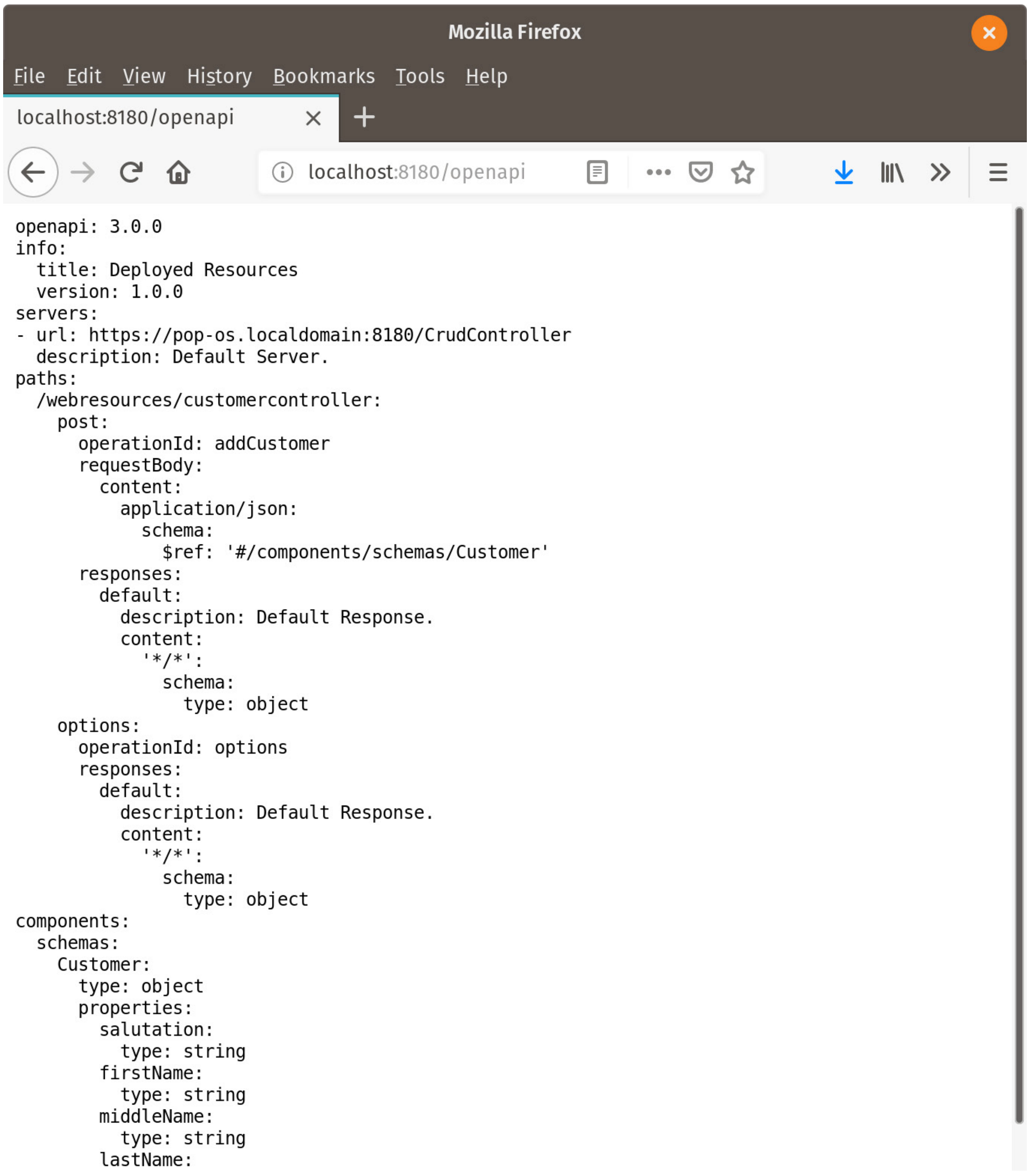
As shown by our example code, developing applications following a microservices architecture in Jakarta EE is very simple, it doesn't require any special knowledge, microservices are developed using standard Jakarta EE APIs and deployed to a lightweight runtime. If you are familiar with Jakarta EE, it is likely that very little if any of the example code we showed is new to you.

Free Functionality Provided by MicroProfile

Just by including the MicroProfile dependency for our application, we get a lot of functionality for free. The MicroProfile Open API can automatically generate documentation for our microservices, the Open Tracing API can generate tracing information in server logs to allow correlating micro-service invocations, and the Health API provides a health endpoint so that tools can automatically check application health. All three APIs allow us to optionally customize their functionality to meet our specific requirements, but the basic functionality is available without us having to write a single line of code.

Free Documentation by MicroProfile Open API

The MicroProfile Open API will provide URL documenting our microservices, without us having to write a single extra line of code, Open API will scan our JAX-RS annotations, and generate documen-tation, all we need to do is point the browser to the /openapi context root of our instance of Payara Micro. For example, the instance of Payara Micro where we deployed our Controller microservice is listening on port 8180, to view the generated documentation, we would point our browser to http://localhost:8180/openapi.



```
openapi: 3.0.0
info:
  title: Deployed Resources
  version: 1.0.0
servers:
- url: https://pop-os.localdomain:8180/CrudController
  description: Default Server.
paths:
  /webresources/customercontroller:
    post:
      operationId: addCustomer
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Customer'
      responses:
        default:
          description: Default Response.
          content:
            '*/*':
              schema:
                type: object
    options:
      operationId: options
      responses:
        default:
          description: Default Response.
          content:
            '*/*':
              schema:
                type: object
components:
  schemas:
    Customer:
      type: object
      properties:
        salutation:
          type: string
        firstName:
          type: string
        middleName:
          type: string
        lastName:
```

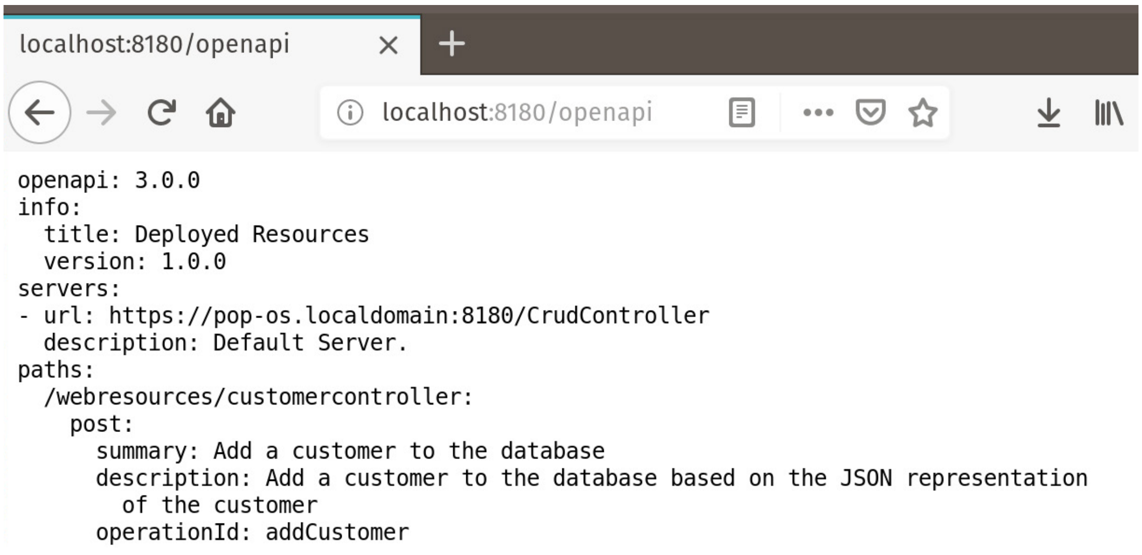
The generated documentation provides useful information about our microservice, such as a description of the operations in our microservice, along with a schema of the expected request body, and the response, all of this without us having to add a single line of code to our application.

Customizing API Documentation

We can easily customize the generated documentation by using annotations provided by MicroProfile, for example, if we wanted to add a summary and detailed description for the "addCustomer" operation provided by our `CrudController` class, we could simply annotate it with the `@Operation` annotation, as follows:

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Operation(summary = "Add a customer to the database",
            description = "Add a customer to the database based"
                    + " on the JSON representation of the customer")
    public Response addCustomer(Customer customer) throws
        URISyntaxException {

        Response response = null;
        Response persistenceServiceResponse;
        …
    }
```

The generated documentation would then be updated accordingly.



```
openapi: 3.0.0
info:
  title: Deployed Resources
  version: 1.0.0
servers:
- url: https://pop-os.localdomain:8180/CrudController
  description: Default Server.
paths:
  /webresources/customercontroller:
    post:
      summary: Add a customer to the database
      description: Add a customer to the database based on the JSON representation
        of the customer
      operationId: addCustomer
```

Powerful Request Tracing by MicroProfile OpenTracing

When debugging microservices, it is not always clear which microservice in our application may be causing an issue. Since microservices are deployed independently, each micro service has its own log file. The OpenTracing API generates output that allows us to correlate micro services invocations with one another, which greatly helps debugging complete operations. To this effect, the MicroProfile OpenTracing API automatically generates trace data from all corresponding services in a format compatible any OpenTracing collector service or tool.

Payara Micro's implementation of MicroProfile OpenTracing API uses an internal service called the Request Tracing Service to generate trace data in real time. Once this service is enabled, trace data will be generated and broadcasted to any compatible OpenTracing collectors. You can enable this service when launching a new instance from the command line like this:

```
java -jar payara-micro.jar –enableRequestTracing --requestTracingThresholdValue
25 --requestTracingThresholdUnit MICROSECONDS --deploy /path/to/CrudController.
war
```

As seen in the command sample, not only is the request tracing service is enabled, but it is configured to gather trace data of all operations that exceed 25 microseconds. By default, tracing data will be printed out to the instance's log output:

```
[2019-03-14T05:11:24.039-0400] [] [INFO] [] [fish.payara.nucleus.notification.
log.LogNotifierService] [tid: _ThreadID=30 _ThreadName=http-thread-pool::http-
listener(2)] [timeMillis: 1552554684039] [levelValue: 800] [[
Request execution time: 1003(ms) exceeded the acceptable threshold -
{
  "traceSpans": [
    {
      "operationName": "processContainerRequest",
      "spanContext": {
        "spanId": "721fb7d3-cbfb-4548-9ea4-eb124d107d24",
        "traceId": "5f4c72fb-a791-4a9f-9a5d-8a78fa600a1d"
      },
      "startTime": "2019-03-14T05:11:23.034-04:00[America/New_York]",
      "endTime": "2019-03-14T05:11:24.037-04:00[America/New_York]",
      "traceDuration": "1003000000",
      "spanTags": [{ "Server": "server" }, { "Domain": "domain1" }]
    },
    {
      "operationName": "processWebserviceRequest",
      "spanContext": {
        "spanId": "f849ed14-d546-4e82-8955-a2871ff8083b",
        "traceId": "5f4c72fb-a791-4a9f-9a5d-8a78fa600a1d"
      },
      "startTime": "2019-03-14T05:11:23.036-04:00[America/New_York]",
      "endTime": "2019-03-14T05:11:24.036-04:00[America/New_York]",
      "traceDuration": "1000000000",
      "spanTags": [
        { "referer": "[http://localhost:8080/CrudView/]" },
        { "content-length": "[71]" },
        { "accept-language": "[en-US,en;q=0.5]" },
        { "origin": "[http://localhost:8080]" },
        { "Method": "POST" },
        {
          "URL": "http://localhost:8180/CrudController/webresources/
customercontroller/"
        },
        { "accept": "[application/json, text/javascript, */*;q=0.01]" },
        { "ResponseStatus": "200" },
        { "host": "[localhost:8180]" },
        { "content-type": "[application/json]" },
        { "connection": "[keep-alive]" },
        { "accept-encoding": "[gzip, deflate]" },
        {
          "user-agent": "[Mozilla/5.0 (X11; Ubuntu; Linux x86_64;rv:65.0)
Gecko/20100101 Firefox/65.0]"
        }
      ],
      "references": [
        {
          "spanContext": {
            "spanId": "721fb7d3-cbfb-4548-9ea4-eb124d107d24",
            "traceId": "5f4c72fb-a791-4a9f-9a5d-8a78fa600a1d"
          },
          "relationshipType": "ChildOf"
        }
      ]
    },
    {
      "operationName": "POST:fish.payara.crudcontroller.service.
CustomerControllerService.addCustomer",
      "spanContext": {
        "spanId": "1694-5b76-794d-44e3-8052-e91e2e572f6f",
        "traceId": "5f4c72fb-a791-4a9f-9a5d-8a78fa600a1d"
      },
      "startTime": "2019-03-14T05:11:23.040-04:00[America/New_York]",
      "endTime": "2019-03-14T05:11:24.032-04:00[America/New_York]",
      "traceDuration": "992000000",
      "spanTags": [
        { "component": "jaxrs" },
        { "span.kind": "server" },
        {
          "http.url": "http://localhost:8180/CrudController/webresources/
customercontroller/"
        },
        { "http.method": "POST" }
      ],
      "references": [
        {
          "spanContext": {
            "spanId": "f849ed14-d546-4e82-8955-a2871ff8083b",
            "traceId": "5f4c72fb-a791-4a9f-9a5d-8a78fa600a1d"
          },
          "relationshipType": "ChildOf"
        }
      ]
    }
  ]
}
```

By design, trace data is structured as a set of **spans** (each one identified with a unique ID), and each span represents a separate service unit/method of the complete trace. OpenTracing collectors will collate all spans by matching the corresponding `traceId` that is part of the context data of each span. Now, a good question you might ask is, how useful is this trace data? The answer is simple: A lot! the MicroProfile OpenTracing runtime gathers information about each span start/end times, duration, HTTP invocation and response data, etc.

However, reviewing trace data in a plain text log is not a sustainable solution, right? In a real-world scenario, distributed tracing systems like Jaeger or Zipkin are used to gather and visualize trace date in real-time to allow proper debugging of our microservice operations. Jaeger is one of the most popular solutions in the market due to its simplicity and extensibility, so we'll be using this tool to showcase how trace data is gathered automatically.

To integrate Jaeger with Payara Micro, it is necessary to provide a valid i`o.opentracing.Tracer` component implementation that delegates the trace data collecting to Jaeger. This implementation must have a no-args constructor as well.

For the sake of simplicity, there's an open source project named [ecosystem-jaeger-tracing] which already has a straightforward implementation of a Jaeger wrapper class. To use it, clone the project locally and build it. By default, all trace data will be logged under the `jaeger-test` *service name. We'll need the resulting* `jaeger-tracer-lib-1.0-jar-with-dependencies` *artifact.*

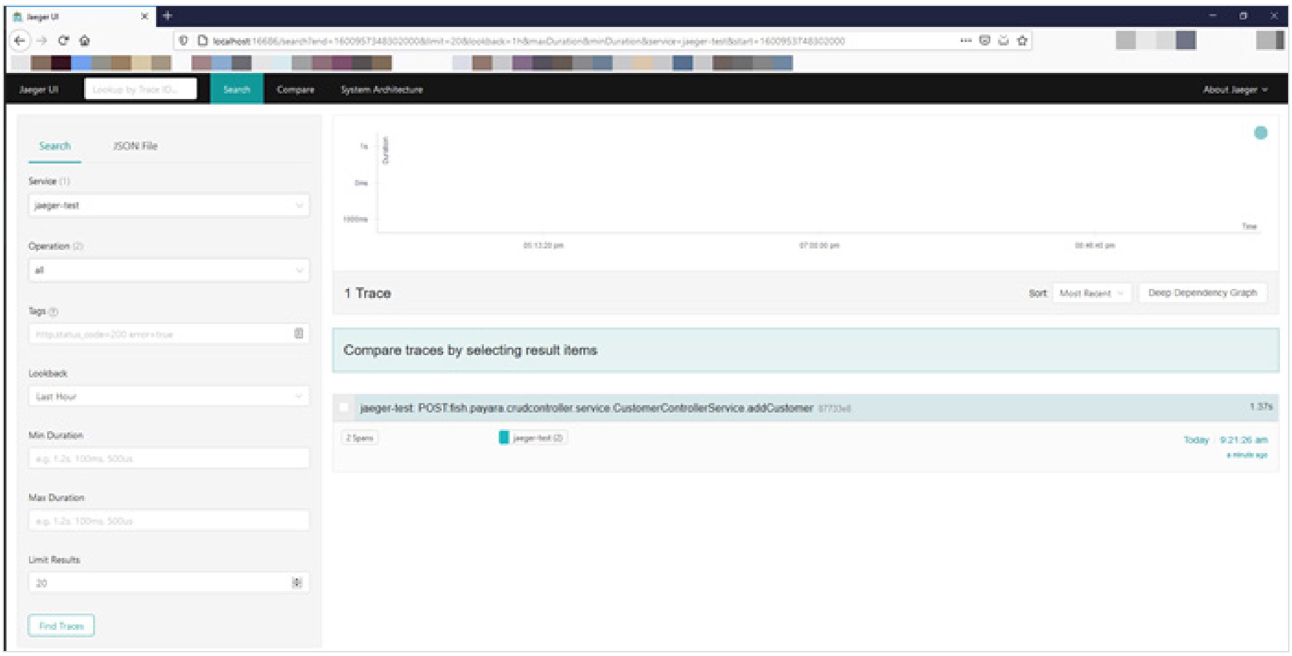
First, let's run Jaeger locally. Jaeger is a collection of applications, so the best way to run it for testing purposes is to spawn a new docker container:

```
docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 14250:14250 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.19
```

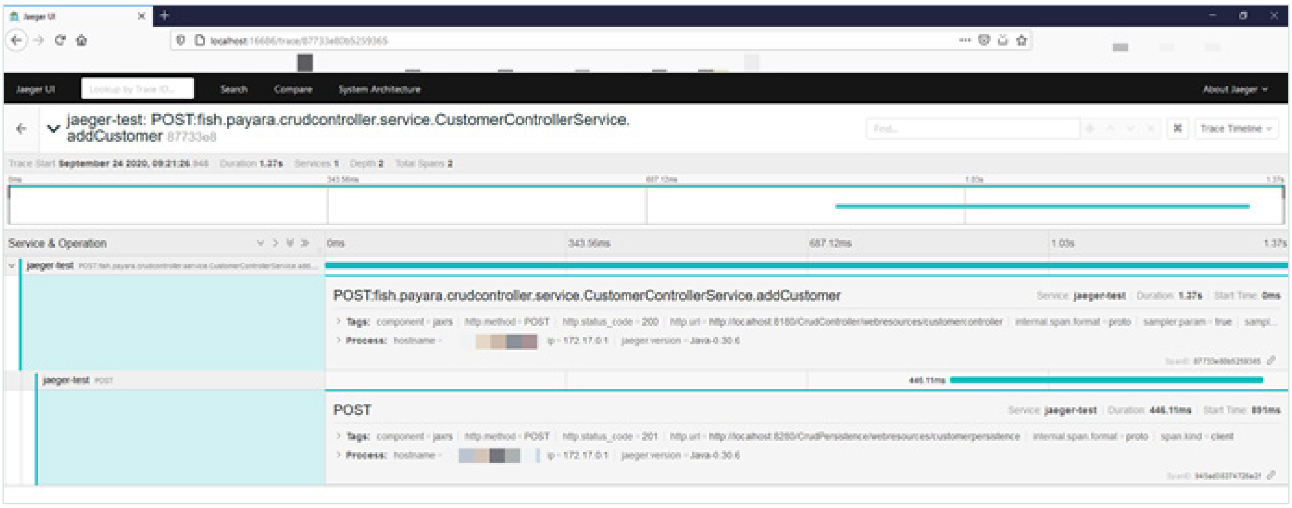
This will spawn a new docker container prepared with all Jaeger processes and ready to collect trace data. Next, we'll restart our `CrudController` service to both enable request tracing and configure the corresponding Jaeger `OpenTracing` wrapper implementation as an added library as well:

```
java -jar payara-micro.jar -noCluster -port 8180 -deploy /path/to/
CrudController.war -enablerequesttracing -addLibs /path/to/jaeger-tracer-lib-
1.0-jar-with-dependencies.jar
```

As soon as our microservice is ready, test the customer create form again. After the operation is completed, let's head to Jaeger's UI located at http://localhost:16686/. Let's select the `jaeger-test` service and click the Find Traces button. The following results will be shown:



You can see that there is one trace for the `addCustomer` method that corresponds to two separate spans, which you can inspect by clicking on the trace:


Jaeger UI
Search
Compare
System Architecture
About Jaeger
jaeger-test: POST:fish.payara.crudcontroller.service.CustomerControllerService.addCustomer 87733e8
Trace Timeline
Trace Start September 24 2020, 09:21:26.948 Duration 1.37s Services 1 Depth 2 Total Spans 2
Service & Operation
0ms
343.56ms
687.12ms
1.03s
1.37s
jaeger-test
POST:fish.payara.crudcontroller.service.CustomerControllerService.addCustomer
Service: jaeger-test Duration: 1.37s Start Time: 0ms
Tags: component = jaxrs | http.method = POST | http.status_code = 200 | http.url = http://localhost:8180/CrudController/webresources/customercontroller | internal.span.format = proto | sampler.param = true
Process: hostname = | ip = 172.17.0.1 | jaeger.version = Java-0.30.6
jaeger-test POST
446.11ms
POST
Service: jaeger-test Duration: 446.11ms Start Time: 891ms
Tags: component = jaxrs | http.method = POST | http.status_code = 201 | http.url = http://localhost:8280/CrudPersistence/webresources/customerpersistence | internal.span.format = proto | span.kind = client
Process: hostname = | ip = 172.17.0.1 | jaeger.version = Java-0.30.6

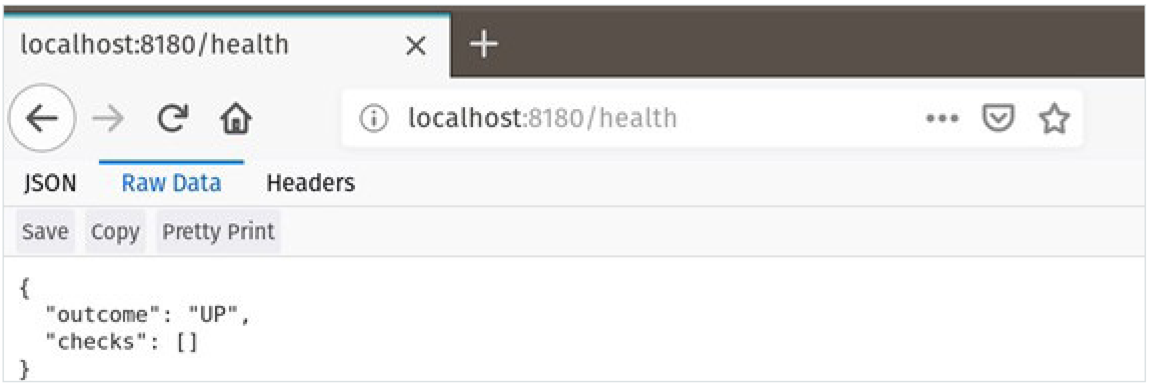
The first span contains information about the complete scope of the `addCustomer` method, while the second span represents the call to the `CustomerPersistenceService` located in our second microservice. You can quickly glance information about the HTTP methods called, their duration, responses, which processes were affected and so much more!

Free Health Check via the MicroProfile Health API

Many cloud providers and container orchestration tools periodically check the health of deployed applications via pre-determined endpoints, these tools can then automatically discard an "unhealthy" container and start a new one, or automatically manage "unhealthy" applications in a way that doesn't disrupt overall user experience.

The MicroProfile Health API provides two health endpoints ready to be used by these orchestration tools, which are easy to configure and available to any applications developed with this API. These endpoints allow services to find out if an application is effectively **UP** or **DOWN**. The API allows the definition of 2 probes:

- *Readiness* probes, which define whether an application is ready to process requests or not. This should help complex environment orchestrators to define a proper dependency order.
- Liveness probes, which establishes if the application is running. When the probe fails, the application's runtime can be either restarted or terminated.



```
{
  "outcome": "UP",
  "checks": []
}
```

Customizing Health Checks

By default, MicroProfile Health reports if an application is running properly, however it has no way of knowing if any resources our application depends on are up and running.

For example, even if our database is not available, by default the API would report a status of UP for our `CrudPersistence` service. If we want to add a check for database availability, we need to add custom code. To this effect, we'll define a liveness check:

```
package fish.payara.crudpersistence.healthcheck.database;
```

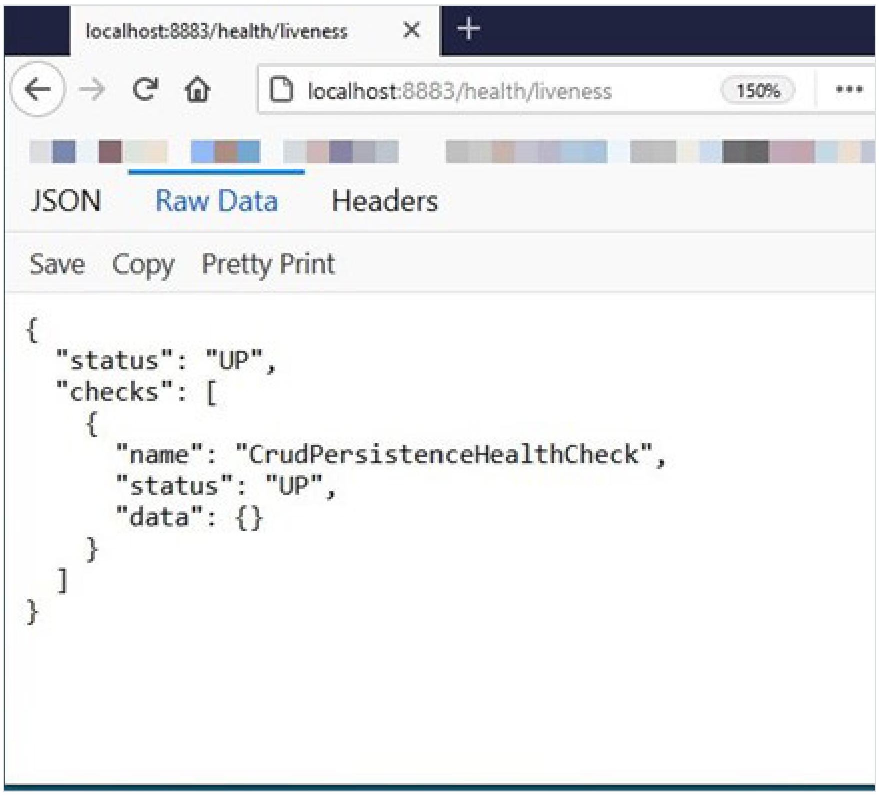
```
//Imports omitted for brevity
@Liveness
@ApplicationScoped
```

```
public class CrudPersistenceHealthCheck implements HealthCheck {
      @Inject
      private CrudDao crudDao;
      @Override
      public HealthCheckResponse call(){
            boolean valid;
            try {
                  valid = crudDao.checkDatabaseConnection();
            } catch (Throwable e) {
                  valid = false;
            }

            if (valid) {
                  return HealthCheckResponse.named(
                        CrudPersistenceHealthCheck.class.getSimpleName()).
                        up().build();
            } else {
                  return HealthCheckResponse.named(
                        CrudPersistenceHealthCheck.class.getSimpleName()).
                        down().build();
            }
      }
}
```

As can be seen in the example, to write custom health checks, we need to write an application scoped CDI bean, this bean must be annotated with either the `@Liveness` or `@Readiness` annotation and must implement the `HealthCheck` interface. This interface has a single method named `call()`, which takes no arguments and returns an instance of `HealthCheckResponse`, which indicates the outcome of the health probe in question.

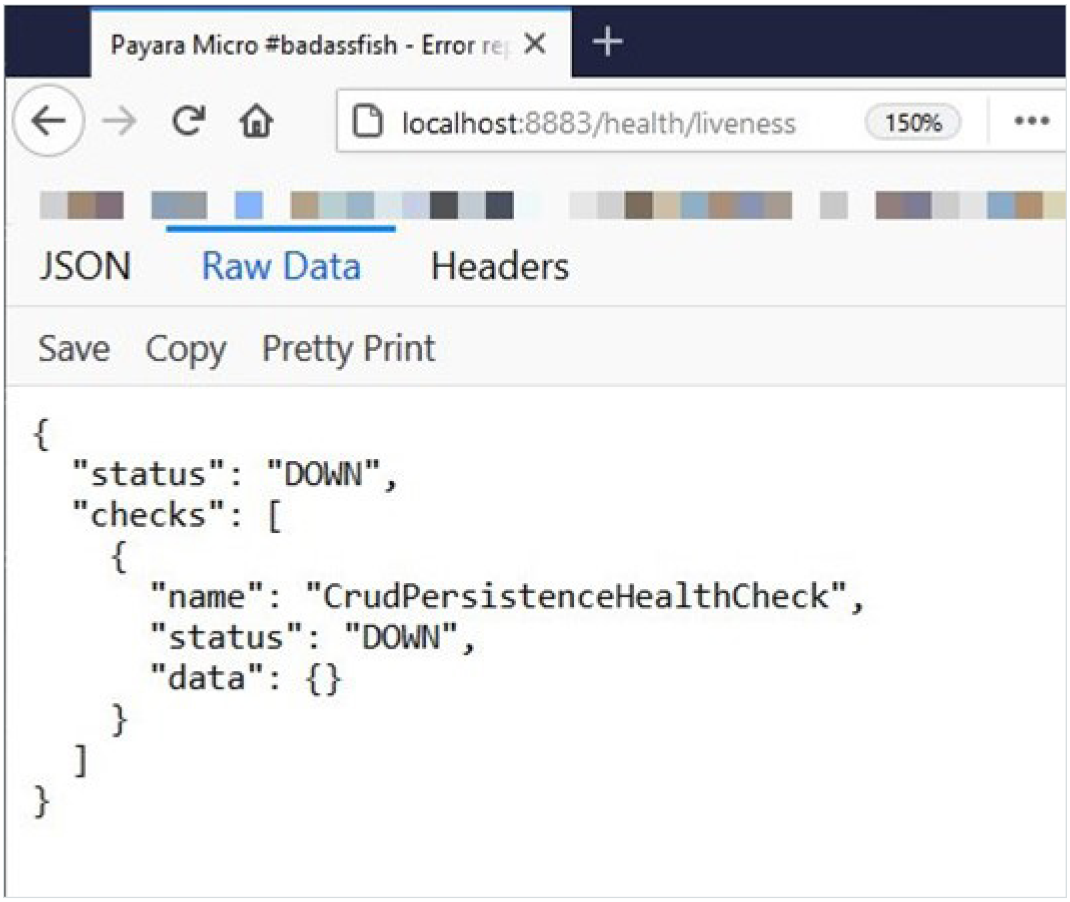
In our example, we added a method to our DAO which checks that a valid connection to the database can be established. If this is the case, the response is built with a positive **UP** status, and if not, the **DOWN** status is used instead.

After deploying the application, we can check if the database is up by checking the `/health/liveness` endpoint of the Payara Micro instance where we deployed our persistence microservice:


localhost:8883/health/liveness
localhost:8883/health/liveness
150%
JSON
Raw Data
Headers
Save
Copy
Pretty Print

```
{
  "status": "UP",
  "checks": [
    {
      "name": "CrudPersistenceHealthCheck",
      "status": "UP",
      "data": {}
    }
  ]
}
```

If we manually stop the database, then check the same endpoint again, it correctly reports that the database is down.


Payara Micro #badassfish - Error re
localhost:8883/health/liveness
150%
JSON
Raw Data
Headers
Save
Copy
Pretty Print

```
{
  "status": "DOWN",
  "checks": [
    {
      "name": "CrudPersistenceHealthCheck",
      "status": "DOWN",
      "data": {}
    }
  ]
}
```

Use Existing Java EE Knowledge to Develop Microservices and Deploy to Payara Micro

As we can see, Jakarta EE is quite suitable for microservices development. Jakarta EE developers can leverage their existing knowledge to develop a microservices architecture and deploy to modern, lightweight runtimes such as Payara Micro. Additionally, it isn't necessary to "throw the baby with the bath water" so to speak, when migrating to micro services. Traditional Jakarta EE applications can interact with microservices quite well, as well as can be refactored iteratively into a microservice-oriented architecture when it makes sense.

Whether tackling one of the previously mentioned scenarios, Jakarta EE developers can leverage their existing skills for the task at hand, simplifying the effort and cost needed for the overall transition.

About Payara Micro

Payara Micro Enterprise is the lightweight middleware platform of choice for containerized Jakarta EE (Java EE) application deployments. Less than 70MB, Payara Micro requires no installation, configuration, or code rewrites – so you can build and deploy a fully working app within minutes.

Compatible with Eclipse MicroProfile, Payara Micro is the microservices-ready version of Payara Server. You can run war files from the command line without any application server installation. Automatic and elastic clustering makes Payara Micro the platform of choice for running Jakarta EE (Java EE) applications in a modern virtualized infrastructure.

Payara Micro also comes with a Java API to embed and launch from your own Java applications.

Learn more about Payara Micro here: https://www.payara.fish/products/payara-micro/

About the Author

David Heffelfinger *is a Java Champion and Apache NetBeans committer, as well as 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 8 Application Development", "Java EE 7 with GlassFish 4 Application Server" and others.*

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.