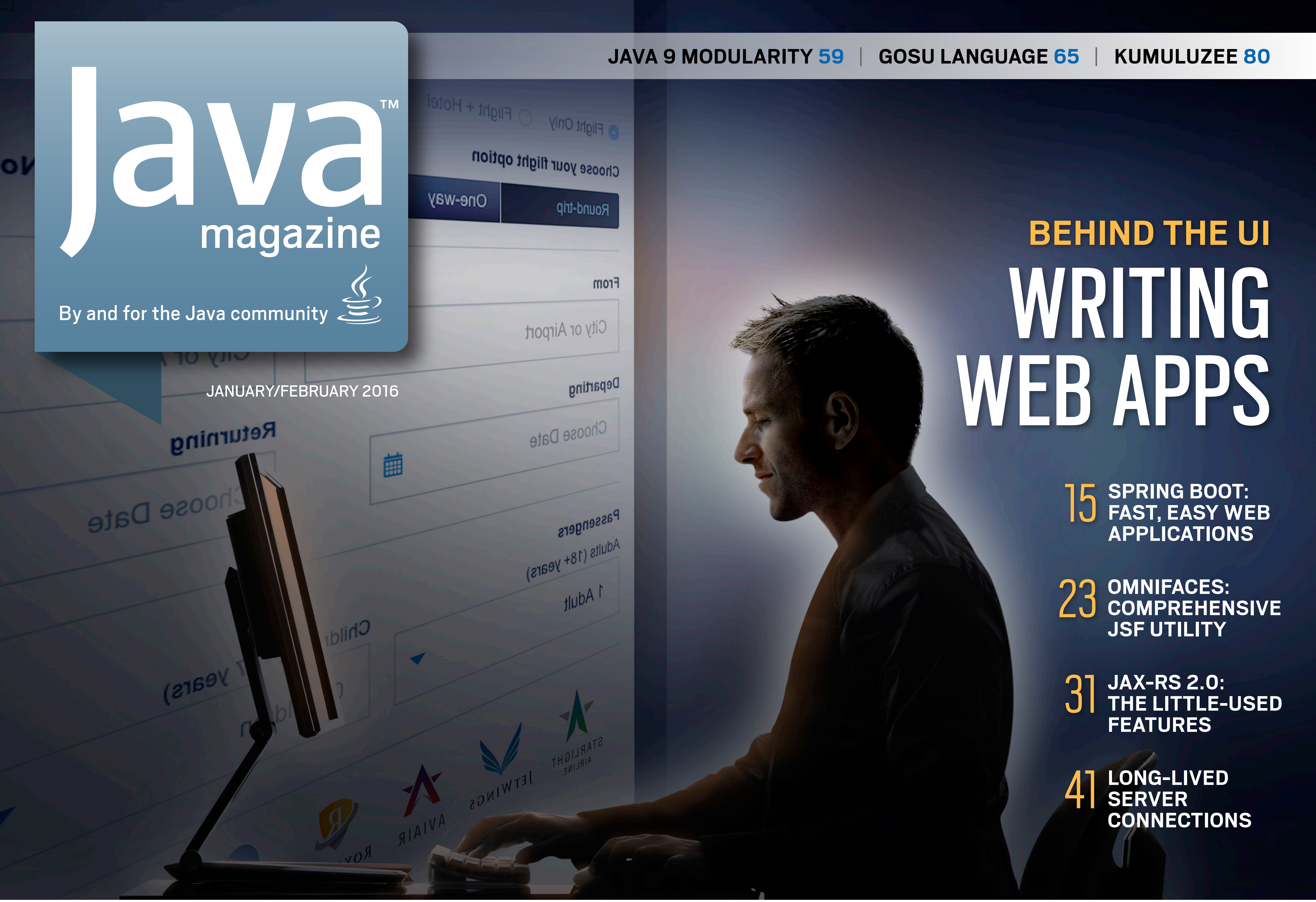
Java™ magazine

By and for the Java community

JANUARY/FEBRUARY 2016

BEHIND THE UI

WRITING WEB APPS



ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

COVER ART BY I-HUA CHEN

EDITORIAL
Editor in Chief
Andrew Binstock
Managing Editor
Claire Breen
Copy Editors
Karen Perkins, Jim Donahue
Section Development
Michelle Kovac
Technical Reviewers
Stephen Chin, Reza Rahman

DESIGN
Senior Creative Director
Francisco G Delgadillo
Design Director
Richard Merchán
Senior Designer
Arianna Pucherelli
Designer
Jaime Ferrand
Senior Production Manager
Sheila Brennan
Production Designer
Kathy Cygnarowicz

PUBLISHING
Publisher
Jennifer Hamilton +1.650.506.3794
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES
Tom Cometa tom.cometa@oracle.com
Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201
Mailing-List Rentals
Contact your sales representative.

RESOURCES
Oracle Products
+1.800.367.8674 (US/Canada)
Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION
If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION
Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE
java@halldata.com **Phone** +1.847.763.9635

PRIVACY
Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

TILEN FAGANEL AND
MATJAZ B. JURIC

KumuluzEE: Building Microservices with Java EE

Develop self-contained microservices with standard Java EE APIs using the open source KumuluzEE framework.

In this article, we explore how to develop microservices with standard Java EE APIs. Microservices have become an important architectural style for developing complex applications composed of small, independent services deployed as individual processes. These services communicate through well-defined APIs. Each microservice is self-contained; is responsible for a single task; is accessible through a lightweight API, such as REST; and has its own lifecycle. Microservices result in a highly decoupled, modular, and reusable architecture for composite applications.

Developing microservices in Java means that not only is the application designed around REST services, but also that each service, the front end, and other components are deployed separately and independently. In other words, instead of packing all the services, business logic, front end, and other modules into a single EAR file and deploying it as a monolith, microservices use individual archives (such as JAR files) for each component. The KumuluzEE framework, as we'll see shortly, automates the tasks related to the configuration and deployment of microservices, specifically on Java EE. [**Note:** KumuluzEE won the Duke's Choice Award at the 2015 JavaOne conference. —*Ed.*]

Microservices with Java EE

Suppose we want to create an online train-reservation service people can use to plan their route and buy their tickets. For brevity, we are going to look at a simplified version of this example; it will allow only viewing routes and booking a route via a simple user interface.

Using the traditional approach, we would create a monolithic EAR package that would include our business logic, services, and one or more WAR packages containing the front end. We would deploy the EAR file to an application server, such as Oracle WebLogic Server or GlassFish.

In contrast, using the microservice architecture, we start by separating the responsibilities. Routes, the booking service (reservations), and the UI become three separately developed, configured, and deployed microservices. They are stateless and communicate through REST interfaces—although we could also use SOAP or remote method invocation (RMI). In this way, we create microservices that are domain-defined and follow the "single responsibility" principle with regard to explicit interfaces. We have created a highly modular, decoupled architecture, where each service is responsible for a single, dedicated piece of functionality. Also, each microservice has its own lifecycle. The proposed architecture using microservices is shown in Figure 1.

In the remainder of this article, we use this simplified example. But keep in mind that microservices are best suited for complex applications and systems.

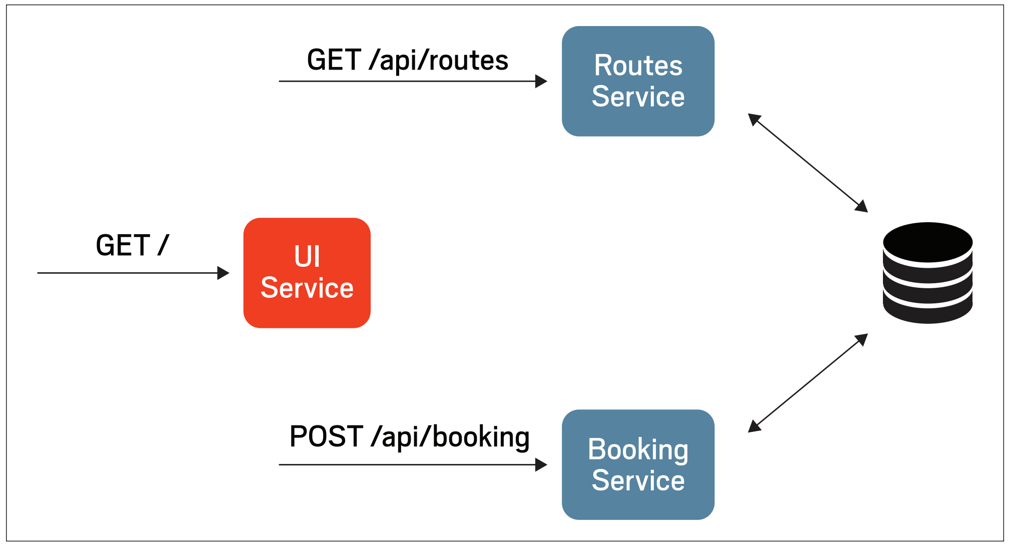


Figure 1. Overview of the proposed microservice architecture

Advantages. Let us now explore the advantages of microservices compared with a monolithic approach. The most obvious advantage is that microservices lead to a more flexible, decoupled architecture. Each microservice can be developed independently, which simplifies lifecycle and change management. For example, if we need to upgrade the booking service, we do not have to redeploy the whole application. Every microservice is a separate project with a separate repository and deployment configuration. We can roll out new features iteratively, thus increasing agility.

If, over time, we would like to upgrade to newer versions of Java, we can upgrade each microservice separately. For complex applications, this is a huge benefit, because it is often difficult to upgrade complex Java EE applications (to a new version of the app server or to new releases of Java EE and Java SE). Even better, microservices allow us to gradually incorporate new technologies.

Last but not least, microservices allow for much better scalability. For example, the route calculation service might get higher traffic than the booking service, because users might use the route calculation more frequently than they book a reservation. With microservices, we can scale the booking service independently of the rest of the application. This approach is a perfect fit for cloud and PaaS (platform as a service) environments, where elastic scalability can be configured easily. Scaling a microservice application in a Docker environment becomes a breeze. In other words, a microservices architecture is a good match for cloud-enabled applications.

To fully exploit these benefits, microservices need to be stateless. Every resource (such as a database, object storage, and so on) that a microservice uses should be configurable separately (usually via a connection string or environment parameters).

Disadvantages. Microservices do not come without drawbacks. Setting up and configuring Java EE projects to accompany this kind of architecture is often not trivial. Actually, it can require a lot of manual configuration and tracking of dependencies, and it can lead to significant operational complexity. Deployment becomes more complex, as does testing.

KumuluzEE

KumuluzEE addresses some of these drawbacks. It automates the tasks related to configuration and deployment of microservices in a seamless way. At its most basic, KumuluzEE collects each microservice together with the Java EE APIs' runtime into a simple, standalone package (JAR file), providing a minimal footprint by including only those APIs (and their runtimes) that are actually used. This enables developers to build standalone, stateless, self-contained microservices and package them in an efficient way without the overhead of including the entire Java EE application server in each microservice. Microservices created this way can be executed directly from the JRE with a minimal footprint and quick startup and shutdown times.

Getting Started

We'll now create our microservices using KumuluzEE, standard Java EE, and Maven. First, we create three Maven projects—each containing its own microservice. For brevity's sake, we will create them in the same repository. We will create a `routes` project for the route calculation service, a `bookings` project for the booking service, and a `ui` project for the front end. We will also add a module that will hold our JPA entities (`models`) and a module that will hold our `utils`, because they will be shared with our three microservices. **Note:** In real-world projects, it is generally recommended to use a separate repository for each microservice so that they are treated as separate entities and have separate versioning and revision history as well as separate deployment.

We will also create the topmost `pom.xml` file. You can look at the project structure and the `pom.xml` file by downloading the source code for this project from GitHub.

To create our microservice projects, we will use Maven archetype generation (`mvn -B archetype:generate`) for all modules. This will generate the three projects we require.

Adding KumuluzEE

Now we need to add the appropriate dependencies to the KumuluzEE framework. KumuluzEE is completely modular, which means that apart from the core functionality each Java EE component is packaged as a separate module and must be included explicitly as a dependency in order to be used. KumuluzEE will automatically detect which modules are included in the classpath and properly configure them.

We include KumuluzEE by defining the appropriate dependency in Maven. It is recommended that we define a property with the current version of KumuluzEE and use it with every dependency:

```
<properties>
    <kumuluzee.version>
        2.0.0
    </kumuluzee.version>
</properties>
```

Adding the Front-End UI Service

Let's start with the `ui` microservice and include the core KumuluzEE module for bootstrapping the logic and configuration. It provides the `com.kumuluz.ee.EeApplication` class with the main method that will bootstrap our app. We will also include the servlet and the HTTP server. We will use a Jetty servlet implementation, which is known for its high performance and small footprint. The `./ui/pom.xml` file contains this:

```
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-core</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-servlet-jetty</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
```

We must also include the `maven-dependency-plugin` in our `pom.xml` file, which will copy all our dependencies together with the classes. So we add the following to the `./ui/pom.xml` file:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.10</version>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

This is the bare minimum required to run a microservice with plain servlets and static files. To try it out, we can add a simple `index.xhtml` HTML file. KumuluzEE will use the `webapp` folder at the root of the `resource` folder to look for configuration data and files. This is where we should put the files.

To try it out, run `maven package`. Then, you can start the microservice using the following command:

```
$ java -cp ui/target/classes:ui/target/dependency/* \
    com.kumuluz.ee.EeApplication
```

Go to `http://localhost:8080/` in your browser and you should see the content of the `index.xhtml` HTML file. Of course, you can use JSP, use a servlet, or add JSF support and use it for the front end. We will extend the front end shortly.

Defining the JPA Module

We follow the procedure described above for each microservice. Because each microservice is its own project, you can customize it as much as you need.

Before we start developing the routes and booking services, we will define the JPA module with entities. To include JPA, we add the EclipseLink JPA implementation. We will also add the database driver. In our case, we will use the PostgreSQL database. However, you are free to use any database. Place the required dependencies in `./models/pom.xml`:

```
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-jpa-eclipselink
    </artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
```

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.4-1201-jdbc41</version>
</dependency>
```

Next, we will add the `persistence.xml` file and entity classes that will be shared with both our microservices even though the microservices will be run separately. This file is the same as what you would develop in a traditional Java EE application. You can look at the provided sample source code to see an example. Notice that the values defined in `persistence.xml` for the database URL, username, and password can be overwritten by setting the `DATABASE_URL`, `DATABASE_USER`, and `DATABASE_PASS` environment variables, respectively. Doing this is useful for easy configuration in Docker-style environments.

We are now ready to write the entity classes. We use standard JPA, which does not require any modifications for KumuluzEE. The example JPA entity classes for `Route` and `Booking` are shown next. The first file, `.../models/Route.java`, contains the following:

```
@Entity
@NamedQuery(name="Route.findAll",
            query="SELECT r FROM Route r")
public class Route {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Integer id;

    private String name;

    private String start;

    private String end;
    @Temporal(TemporalType.TIMESTAMP)
    private Date duration;

    @ManyToMany(mappedBy = "routes")
    private List<Booking> bookings;

    // ... Omitted getters/setters
}
```

The booking file, `.../models/Booking.java`, contains this:

```
@Entity
@NamedQuery(name="Booking.findAll",
            query="SELECT b FROM Booking b")
public class Booking {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Integer id;

    @Temporal(TemporalType.TIMESTAMP)
    private Date orderDate;

    @ManyToMany
    private List<Route> routes;

    // ... Omitted getters/setters
}
```

Implementing the Routes and Booking Services

We are now ready to implement the routes and booking services. We will need JAX-RS for the REST interfaces. We will use Jersey as implemented in KumuluzEE. We will also add CDI, because we want to use the `EntityManager` injection using `@PersistenceContext`. However, we could just as well do without CDI to keep our microservice even lighter. We add the following dependencies to `./booking/pom.xml`:

```
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-jax-rs-jersey</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-cdi-weld</artifactId>
    <version>${kumuluzee.version}</version>
</dependency>
```

Now, let's implement the booking REST service. We'll create a booking resource that contains two methods: `createBooking(Booking)` and `getBooking()`. Make sure you add the correct dependencies, as described earlier. Also make sure you add the `models` module to every microservice for access to JPA entities, whose REST service will be used. The code is:

```
@ApplicationPath("/")
public class BookingApplication extends
    javax.ws.rs.core.Application {
}
```

Then in `BookingsResource.java` we have this:

```
@Path("/bookings")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@RequestScoped
public class BookingsResource {
    @PersistenceContext(unitName = "trains")
    private EntityManager em;

    @GET
    public Response getBookings() {

        List<Booking> bookings = em.createNamedQuery(
            "Booking.findAll", Booking.class)
            .getResultList();

        return Response.ok(bookings).build();
    }

...

    @POST
    public Response createBooking(Booking b) {
        em.getTransaction().begin();
        em.persist(b);
        em.getTransaction().commit();
        return Response.status(
                  Response.Status.CREATED)
                       .entity(b).build();
    }
}
```

As you can see, with KumuluzEE the microservice implementation does not require any modifications and is exactly the same as with any other Java EE application. You can implement the remaining microservices the same way or look at the provided sample code.

Handling Service Discovery

Let's suppose that we are using JavaServer Faces (JSF) for the front-end UI module. From there, we will be calling our microservices via REST to receive or save the actual data. The main problem with this is that we need to know the exact URLs of the microservices. We could pass the URLs via environment variables. However, that approach would require manual updating every time anything changes. And in the cloud, these changes can be quite frequent.

A better solution would be to dynamically query the address of the requested microservice. We will use Apache ZooKeeper for service discovery, but we could also use any similar tool. In short, ZooKeeper is a centralized service for maintaining configuration information. We will use it to store the correct URL to each microservice. We add a helper class, `ServiceRegistry`, in our `utils` module to handle the dynamic registering, unregistering, and retrieval of endpoints with ZooKeeper:

```
@ApplicationScoped
public class ServiceRegistry  {

    private final CuratorFramework zookeeper;
    private final
        ConcurrentHashMap<String, String> zonePaths;

    @Inject
    public ZooKeeperServiceRegistry()
           throws IOException {
        try {
            String zookeeperUri =
                System.getenv("ZOOKEEPER_URI");
            zookeeper = CuratorFrameworkFactory
                .newClient(zookeeperUri,
                     new RetryNTimes(5, 1000));
            zookeeper.start();
            zonePaths = new ConcurrentHashMap<>();
        } catch (IOException) {
          ...
        }
    }

    public void registerService(
            String name, String uri) {
        try {
            String node = "/services/" + name;
            if (zookeeper.checkExists()
                  .forPath(node) == null) {
                zookeeper.create()
                    .creatingParentsIfNeeded()
                    .forPath(node);
            }
            String nodePath =
              zookeeper.create().withMode(
                    CreateMode.EPHEMERAL_SEQUENTIAL)
                    .forPath(node + “/_”,
                             uri.getBytes());
                zonePaths.put(uri, nodePath);
            } catch (Exception ex) {
                ...
            }
        }

        // ... Similar for the other methods.
        // See full example available for download.
}
```

The ZOOKEEPER_URL environment variable should contain our ZooKeeper URL. We will be starting a ZooKeeper service alongside our microservices using Docker, which is discussed in the next section. Also make sure that the utils package is added as a dependency to every microservice.

We can now inject the service above into our microservices to register its URL upon startup and unregister it upon shutdown. One way to do this is to add a bean to our booking microservice to handle this task dynamically:

```
@ApplicationScoped
public class BookingService  {

    @Inject
    ServiceRegistry services;

    private String serviceName = "trains-booking";
    private String endpointURI;

    public BookingService() {
        endpointURI =
        System.getenv("BASE_URI");
    }

    @PostConstruct
    public void registerService() {
        services.registerService(
            serviceName, endpointURI);
    }

    @PreDestroy
    public void unregisterService() {
        services.unregisterService(
            serviceName, endpointURI);
    }
}
```

We use the @PostConstruct, @PreDestroy, and @ApplicationScoped annotations to make sure we register and unregister the service only once per lifecycle. The BASE_URI environment variable should contain the microservice’s public URI. We can now create a similar bean for each of our microservices.

Injecting the Microservices at the Front End

We can use the just-described approach for injection on the front end to connect to the REST services. We can inject our ServiceDiscovery bean to our JSF backing bean to retrieve the URL of a microservice we want to invoke, as shown below:

```
@Model
public class BookingsBean  {

    @Inject
    ServiceRegistry services;

    public List getAllBookings() {
        return ClientBuilder.newClient()
            .target(services.discoverServiceURI(
                     "trains-booking"))
            .path("bookings")
            .request().get(List.class);
    }
}
```

Updating the JSF Front End

Finally, let's update our `index.xhtml` file in the `ui` module to use the created bean and to list the available bookings:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>All available bookings</title>
  </h:head>

  <h:body>
    <h:dataTable
       value="#{bookingsBean.allBookings}"
        var="booking">
      <h:column>
        <f:facet name="header">Booking id</f:facet>
        #{booking.id}
      </h:column>

      <h:column>
        <f:facet name="header">Date of order
        </f:facet>
        #{booking.orderDate}
      </h:column>

    </h:dataTable>

  </h:body>
</html>
```

And we're done! We have created three microservices that will run independently of each other. They communicate through REST interfaces and share common resources such as JPA entities. Of course, we could use HTML5 with AngularJS or ReactJS for the front end as well.

Deploying and Running Using Docker

We can now build and run our microservices. To demonstrate this, we will use Docker. First we need to create a `Dockerfile`, which will contain the steps to create a Docker image that will build each microservice and run it. Because KumuluzEE runs a microservice as a normal, single-process, standalone JAR file with a minimal footprint (and does not require a separate app server instance for each microservice), it fits perfectly with how Docker operates. The `Dockerfile` is shown next. Its `./routes/Dockerfile` file is in the downloaded files.

```
FROM java:openjdk-8u45-jdk
MAINTAINER info@kumuluz.com
RUN apt-get update -qq && \
    apt-get install -y wget git
RUN wget [...maven binaries...] && \
         [...install maven...]
RUN mkdir /app
WORKDIR /app
ADD . /app
RUN mvn clean package -Pdeploy
ENV JAVA_ENV=PRODUCTION
EXPOSE 8080
CMD ["java", "-server", "-cp",
     "ui/target/classes:ui/target/dependency/*",
      "com.kumuluz.ee.EeApplication"]
```

As we can see, the `Dockerfile` is pretty simple and straightforward. It installs the required dependencies, builds the microservice, and supplies the command to run it. In our case, we can use the same `Dockerfile` for every microservice, except for the final command, which specifies which microservice is going to run in a particular Docker container.

Next, we need to build a Docker image for every microservice that we have:

```
$ docker build -t trains/ui \
     -f ui/Dockerfile .
$ docker build -t trains/routes \
     -f routes/Dockerfile .
$ docker build -t trains/bookings \
     -f bookings/Dockerfile .
```

Once built, these images can be run anywhere Docker or the Docker API is used, as well as anywhere Docker Swarm, Kubernetes, various PaaS providers, and many such services are used. To test the images, we can run them locally. We also need to start a ZooKeeper instance by using an existing image. Don't forget to pass the required environment variables to our microservices (using the -e switch). The following example uses 172.17.42.1 as the Docker host, because that is the default. Adjust as needed.

```
$ docker run -p 2181:2181 -d fabric8/zookeeper
$ docker run --name ui -p 3000:8080 -d \
   -e BASE_URI='http://172.17.42.1:3000' \
   -e ZOOKEEPER_URI=172.17.42.1:2181 trains/ui
$ docker run --name routes -p 3001:8080 -d \
   -e BASE_URI='http://172.17.42.1:3001' \
   -e ZOOKEEPER_URI=172.17.42.1:2181 trains/routes
$ docker run --name bookings -p 3002:8080 -d \
   -e BASE_URI='http://172.17.42.1:3002' \
   -e ZOOKEEPER_URI=172.17.42.1:2181 trains/bookings
```

We can now browse our microservice application at http://localhost:3000 or browse our REST services for the routes and bookings modules at http://localhost:3001 and http://localhost:3002, respectively.

Conclusion

In this article, we have seen how to develop microservices with standard Java EE using the open source KumuluzEE framework. As demonstrated, the KumuluzEE framework automates tasks related to the configuration and deployment of microservices. With simple dependency definitions in Maven, it allows us to create self-contained, standalone JAR files, which contain the required execution environment. Therefore, they can be executed within a standard JRE without requiring an application server. This way, microservices with KumuluzEE provide a minimal footprint and are a good fit for executing in cloud, PaaS, and Docker-style environments, which was demonstrated with this example. We also saw how to handle service discovery using ZooKeeper.

KumuluzEE is an enabler for a microservices architecture in Java EE. The major benefit is that it allows us to use standard Java EE APIs to develop microservices. This way we can also port existing applications to microservices. Although at first blush, microservices might look odd to traditional app server developers, many technologists believe microservices represent the future of Java in the cloud. Feel free to download KumuluzEE and try it out. </article>

Tilen Faganel is lead software architect at Sunesis. He is the lead developer of the KumuluzEE framework for Java, which won the 2015 Duke's Choice Award for best Java innovation.

Matjaz B. Juric, PhD, is a Java Champion and Oracle ACE Director who has authored or coauthored more than 15 books on Java, BPM, and SOA and published in several magazines and journals.

learn more

Getting Started with KumuluzEE