Developers have taken many approaches over the years to developing web and enterprise applications on the Java platform. At the inception of the JVM, the groundbreaking servlet technology was introduced in an effort to bring dynamic content to web applications, and technology such as applets provided a way to deliver rich internet applications to a user’s desktop. Over time, developers created easier and more intuitive ways to work with servlets via frameworks such as JavaServer Pages (JSPs) and Apache Struts. Built on top of servlet technology, these solutions enabled developers to focus more on the front end than on boilerplate code.

The J2EE platform was introduced in 1999, providing a handful of APIs for the creation of enterprise-based applications. The APIs included JDBC, Enterprise JavaBeans (EJB), JSPs, and Java Message Service (JMS), to name a few. In the early era, J2EE was complex, because there were many configurations that needed to be made and the logic was difficult to follow. Configuration made the platform difficult to use.

As the years went on, J2EE evolved to include more APIs and the complexity level decreased. The platform name was changed in 2006 when Java EE 5 was introduced. This release was significant, because it included some great productivity boosters such as the changes made in EJB 3.0, and the JavaServer Faces (JSF) framework was introduced to the platform. The next two releases, Java EE 6 and Java EE 7, focused primarily on developer productivity and platform modernization. Each of these releases significantly reduced the complexity of the platform by incorporating technologies such as annotations, rather than requiring XML for configuration. Contexts and Dependency Injection (CDI) was introduced to the core of the platform in Java EE 6, providing an easy way to utilize contextual objects throughout an application. Developing and deploying modern applications within a single WAR file was then streamlined with the Java EE platform.

Although Java EE evolved significantly, rapid changes in technology placed demand on the platform to advance more quickly. The Java EE 8 release answered the call by providing APIs to facilitate using web services, a new security API, and improvements for deployment to container environments. However, the need to advance the platform at a more rapid pace remained an issue. To accommodate this requirement, Oracle open-sourced the platform through the Eclipse Foundation in 2017.

In this article, I take you through an introduction to the next evolution in the enterprise Java space: Jakarta EE. I explain the transfer of the specifications from Oracle to the Eclipse Foundation, and I demonstrate how to grab the latest code to get started with Jakarta EE.
When Oracle decided to open source the Java EE platform via the Eclipse Foundation, the Eclipse Enterprise for Java (EE4J) project was formed. This project is a base repository that resides on GitHub, and it is in place for the purpose of transitioning and housing the codebase, documentation, and Technology Compatibility Kits (TCKs) for each of the Java EE specifications. EE4J is not going to become the open platform; rather, it is a project that contains each of the specifications for the new platform. Oracle began to transfer the documentation, codebase, and TCK for each of the specifications to their respective EE4J projects during the second half of 2016. At the time of this writing (mid-2018), the transition is still underway, and significant work is being done to make the transition as seamless and timely as possible.

The name Jakarta EE was chosen for the new open-sourced platform that was once known as Java EE. That is, all EE4J projects that are transferred from Oracle will be combined to create the Jakarta EE platform. (The Jakarta name has significance in the Java community, because a project known as the Jakarta Project was operated as an umbrella project under the Apache Software Foundation for several years. The project was retired in 2011, because most of the subprojects had formed independent projects within the Apache Software Foundation, so the project team felt that no confusion would arise.)

**New Governance Process**

A key feature of Jakarta EE is a more frequent release cadence than Java EE had. To this end, the Jakarta EE platform will be governed by the Jakarta EE Working Group. This group will help to evolve and promote broad adoption of the technologies related to EE4J. As part of its mandate, the working group will provide vendor-neutral marketing, define and manage a new governance process for formalizing specifications, define compatibility and branding rules, encourage community participation, and establish a funding model that will enable the working group and community to operate on a sustainable basis.

There are five different classes of Jakarta EE Working Group membership: Strategic, Enterprise, Participant, Committer, and Guest. The Strategic, Enterprise, and Participant classes are geared toward organizations and provide different levels of participation in the process. Committer members are individuals who are able to contribute and commit code to the Eclipse Foundation projects. Guest members are organizations that are Associate members of the Eclipse Foundation; they are invited by the steering committee for a renewable year of membership in which they participate in specific aspects. Under the Jakarta EE Working Group, members will be able to help steer the direction of the platform or contribute directly by committing patches or adding enhancements to the EE4J projects that make up the platform.

**Getting Started with Jakarta EE**

Let's look at how to start building applications on the Jakarta EE platform. The Jakarta EE 8 release is due prior to the end of 2018. With that in mind, some of the information discussed in the following section will be relevant only to those of you who are trying Jakarta EE 8 prior to its official release.

In the next few sections, I cover how to obtain the libraries required to develop various services that will be used together to compose an example application. The individual services in the example application are actually separate projects that use only the EE4J projects that are required. In the end, I demonstrate how to deploy each of the services to a single instance of the “full” version of Payara Server to deliver the application. This setup can be used to deploy applications that are composed of multiple WAR files to a single server. However, it is more common for each WAR file to be deployed to separate containers as microservices. Either choice is valid and depends upon the requirements of your project.
Building the Sports Roster Application

A Jakarta EE application is built on standards, and it’s easy to develop. The example application will allow a sports team to register and query team members. The application I’m going to build utilizes a JSF 2.3 (Mojarra) front end, along with JAX-RS web services for communication with the Java Persistence API (JPA) that works with the database. The application is composed of three separate services, each of them Maven projects. The first project I demonstrate will be used to query the database, the second will register (insert or update to the database), and the third is a front-end user interface.

Consider that each service may be deployed to separate containers, such as Apache Tomcat, that do not bundle the full Jakarta EE stack. For that reason, I list only the dependencies in the POM that each service requires, rather than listing the entire stack. For this article, each of the services can be deployed to a single instance of GlassFish 5 or another application server or servlet container.

Setting Up the Environment

There are plenty of IDEs you can use for developing a Jakarta EE application. The main issue at the time of this writing is that none of them offers direct support for Jakarta EE, per se. However, most of them have complete support for Java EE 8. Because the initial release of Jakarta EE is aligned with Java EE 8, it is possible to make use of the IDE support for Java EE when developing Jakarta EE applications. Most of the major IDEs eventually will contain direct support for Jakarta EE.

The IDE that I use for this article is a release candidate of Apache NetBeans 9.0, built from the codebase. The Java EE plugins can be installed after the IDE has been built.

Obtaining Projects for Use Within Services

EE4J hosts the various projects that make up the Jakarta EE platform. As the Jakarta EE platform begins to have full releases, there will be Maven coordinates or a location where different variations of the Jakarta EE platform can be downloaded and added to a project. The platform will likely be available in its entirety or as separate APIs. Prior to the official release of Jakarta EE, the EE4J projects need to be downloaded or added as Maven dependencies separately. You can use a couple of paths to obtain the specifications or APIs that are required: download the codebases and build them yourself, or utilize the Maven Central repository to pull them into the project.

To download and build an EE4J API, visit its corresponding EE4J GitHub project page and find the respective GitHub repository, clone the codebase of the project, and (typically) use Maven to build the API. Follow the procedures on the respective project’s GitHub repository homepage. See the Mojarra project for a good example, because it contains lots of documentation on how to build and use it in resulting projects. Most of the projects simply require the project to be built by issuing `mvn clean install`, and the dependency JAR will be produced. The resulting JAR can then be added to a local Maven repository, or it can be added directly to a project.

For the examples in this article, I use Maven Central by adding the dependencies to project POM files. Note that Maven Central might not yet have true Jakarta EE platform APIs registered within it, so some of the APIs still point to the Java EE 8 dependency.

Developing the Services

I created each of the services for the example as Maven projects in Apache NetBeans by selecting New Project > Maven > Web Application. The projects are named as follows: SportsTeamQueryService, SportsTeamRegistrationService, and SportsTeamUIService.
The following lines of SQL can be used to create the database table (Apache Derby) for the example. You can add the table to the sample Derby schema for convenience. If you are using GlassFish 5 or Payara Server 5 to deploy the example, the Derby sample schema should already be registered as a JDBC connection.

```
cREATE TABLE team_roster (
  id                   numeric,
  first_name           varchar(150),
  last_name            varchar(150),
  position             varchar(150),
  registration_date    date,
  PRIMARY KEY (ID)
);
```

**Query service.** `SportsTeamQueryService` is the most straightforward, because it involves only a few lines of code to produce—especially if you're using an IDE. Listing 1 shows the POM for this service, because the service uses the EJB, JPA, and JAX-RS Jakarta EE specifications. (All the code—including the listings discussed but not inserted in this article—is available from my [GitHub repository](https://github.com).

A persistence unit is required so the service can connect to the database (Listing 2). The service uses an entity class for persistence. To create the `TeamRoster` entity class using Apache NetBeans, select New > Entity Classes from Database, and then select the database table that was created in the previous section. Listing 3 shows the codebase for the `TeamRoster` entity class.

A JAX-RS service can be used to query the database. To enable JAX-RS within the service, create an `ApplicationConfig` class with the contents of Listing 4.

**Listing 4.**

```java
package org.javamagazine.sportsteamqueryservice.service;

import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("rest")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

    private void addRestResourceClasses(Set<Class<?>> resources) {
        resources.add(TeamRosterFacadeREST.class);
    }
}
```

The `@javax.ws.rs.ApplicationPath` annotation is used to define the URL path to use as an entry point for each of the service’s RESTful web services. All resource classes that are used as a RESTful service must be registered within the `ApplicationConfig`.

The REST service class is named `TeamRosterFacadeREST`, as shown in Listing 5.

**Listing 5.**

```java
@javax.ejb.Stateless
@Path("teamrosterqueryservice")
```
This class is annotated with \texttt{javax.ejb.Stateless}, marking it as a stateless session bean. The \texttt{javax.ws.rs.Path} annotation defines the URL path that can be used to access the class service endpoints. The \texttt{TeamRosterFacadeREST} class contains two service methods, \texttt{find()} and \texttt{findAll()}, to find a \texttt{TeamRoster} or a \texttt{List} of \texttt{TeamRoster}, respectively.

The service can be used by calling the URL \url{http://localhost:8080/SportsTeamQueryService/rest/teamrosterqueryservice} or a \texttt{TeamRoster} ID can be passed as a path parameter by appending it to the end of the URL.

\textbf{Registration service.} The \texttt{SportsTeamRegistrationService} exposes a web service capable of registering new players to a sports team. The service is nearly identical to the \texttt{SportsTeamQueryService}, with the exception of the \texttt{TeamRosterFacadeREST} class (Listing 6), which contains different RESTful web service methods. It contains a \texttt{@GET} method named \texttt{countRoster()} that returns a count of \texttt{TeamRoster} objects, and a \texttt{@POST} method named \texttt{addPlayer()} that accepts parameters of type \texttt{@FormParam} for a player’s first name, last name, and position. The method creates a new \texttt{TeamRoster} object; sets the values passed in as parameters; and persists the object to the database, using an \texttt{EntityManager}. Finally, the method returns a \texttt{javax.ws.rs.core.Response} to indicate success or failure.

To register a player, a \texttt{javax.ws.rs.core.Form} can be sent via a JAX-RS client to the \texttt{addPlayer} web service. The following section demonstrates how to do so via a JSF user interface.

\textbf{User interface service.} The \texttt{SportsTeamUIService} drives the user interface, using the JSF framework. The service also requires the JAX-RS client dependency to query the \texttt{SportsTeamQueryService} web services and to register new players by using the \texttt{SportsTeamRegistrationService}. The POM file is shown in Listing 7. The UI service could use any web framework, but here I chose to use JSF. Therefore, the web views are created as XHTML files, and

```java
public class TeamRosterFacadeREST {
    @PersistenceContext(unitName = "SportsTeamQueryServicePU")
    private EntityManager em;

    public TeamRosterFacadeREST() {
    }

    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML,
               MediaType.APPLICATION_JSON})
    public TeamRoster find(
            @PathParam("id") BigDecimal id) {
        return (TeamRoster)
                em.createQuery("select object(o) from TeamRoster o
                                .setParameter("id", id)
                                .setHint("javax.persistence.cache.retrieve.mode",
                                        CacheRetrieveMode.BYPASS)
                                .getSingleResult();
    }

    @GET
    @Produces({MediaType.APPLICATION_XML,
               MediaType.APPLICATION_JSON})
    public List<TeamRoster> findAll() {
        return  em.createQuery("select object(o) from TeamRoster o
                                .setHint("javax.persistence.cache.retrieve.mode",
                                        CacheRetrieveMode.BYPASS)
                                .getResultList();
    }

    protected EntityManager getEntityManager() {
        return em;
    }
}
```
CDI controller classes are used to communicate between the front end and the back end.

The main view, the index.xhtml file (shown in Listing 8), simply uses a `DataTable` to list each of the registered players, and it also contains a form for registering new players. The backing beans behind index.xhtml are CDI controllers named `TeamRegistrationController` (shown in Listing 9) and `TeamQueryController` (shown in Listing 10).

The controller is annotated with `@RequestScoped`, which means that a new instance of the controller is constructed with each request. Therefore, each time the index.xhtml view is visited, the controller is constructed and the `getTeamRosterList()` method is invoked, returning the full list of players. A JAX-RS client is employed to query the `SportsTeamQueryService` to populate the list of players.

The `teamRoster` field of the controller populates and constructs new players (`TeamRoster` objects). Clicking the Register button in the view (Figure 1) calls on the `registerPlayer()` controller method. This method creates a new JAX-RS client to initiate a call to the `teamroster-registrationservice.addPlayer` web service. A new `javax.ws.rs.core.Form` is created, and the appropriate fields for a `TeamRoster` object are populated from the data entered into the view. The JAX-RS response is returned from the service call to indicate success or failure. Note that in a real-world scenario, a security API should be used to secure the web service, employing technology such as JSON Web Tokens.

![Figure 1. The user interface of the SportsTeamRegistrationService](image-url)

### Deployment Options and Configurations

Jakarta EE is a cloud platform, which means that applications can be deployed to almost any container. As previously mentioned, the three services demonstrated in this article can be deployed to a single application server container, or each of them can be deployed to separate containers such as Docker containers.

In situations where you use separate containers, it might be helpful to use a configuration API such as the one offered by MicroProfile for handling dynamic configurations such as web service URIs. The initial release of Jakarta EE does not contain a configuration API, but one may be added later. It also makes sense to use a standardized security solution such as the Java EE Security API or JSON Web Tokens for securing web service endpoints.

### Roadmap

Jakarta EE is a new, open platform, which means it has the potential to evolve at a much faster pace than its predecessor, Java EE. The initial release of Jakarta EE will be at parity with Java EE 8. However, later releases of Jakarta EE will introduce new features, and possibly new APIs, into the platform. Throughout its evolution, the Java EE platform...
Josh Juneau

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to Oracle Technology Network and Java Magazine and has written several books for Apress about Java and Java EE. Juneau was a JCP Expert Group member for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, leader for the CJUG OSS Initiative, and a regular voice on the JavaPubHouse Off_heap podcast.

Share this Page

This article was originally published in the September/October 2018 issue of Java Magazine.