JAKARTA EE

Java for the enterprise: What to expect in Jakarta EE 10

The release is only a year away. Here's what to expect.

by Arjan Tijms

March 26, 2021

Last year, Java EE completed its transfer to the Eclipse Foundation and adopted a new name, Jakarta EE. While this is a great achievement in and of itself, perhaps the most interesting part of that is that it's now finally time to start looking forward.

As a quick recap, **Table 1** shows key historic and future Jakarta EE dates, some of which are tentative. There are some changes from the version of the table I presented in an article in February 2020.

Release	Date	Comment
Java EE 8	August 31, 2017	Final JCP/Oracle release
GlassFish 5.1	January 29, 2019	Verification build from transferred source
Jakarta EE 8	September 10, 2019	Initial Jakarta release; APIs under the Eclipse Foundation Technology Compatibility Kit (EF TCK) license following the Jakarta EE Specification Process (JESP)
GlassFish 6.0.0-RC2	November 1, 2020	Compatible implementation to go along with Jakarta EE 9 release
Jakarta EE 9	November 22, 2020	Ecosystem/tooling release
GlassFish 6.0.0	December 2020/January 2021	Final release of GlassFish 6.0.0
GlassFish 6.1.0	~Q2 2021*	Compatible implementation to go along with Jakarta EE 9.1 release; JDK 11 compatible
Jakarta EE 9.1	~Q2 2021*	JDK 11 release
Jakarta EE 10	~Q1 2022*	First new feature release
* tentative dates		

Table 1. The history and latest release projections for Java EE and Jakarta EE

Comparing the table shown in the previous article to this one, you can see that the JDK 11 compatibility theme moved from Jakarta EE 9 to Jakarta EE 9.1, which is still to be released this year.

While this obviously takes some time away from Jakarta EE 10, planning for that latter release has started to some degree nevertheless, and some of the individual specifications and API projects have started their discussions. Note that everything presented in this article is preliminary and represents the current state of what is thought to be the direction in which Jakarta EE 10 will be heading, but it provides no guarantees that any of this will actually end up in Jakarta EE 10.

It's all about CDI alignment

One of the topics that is likely to be adopted for the Jakarta EE 10 overall theme might be "Contexts and Dependency Injection (CDI) alignment," that is, closing the gap between Enterprise JavaBeans (EJB) and CDI. From roadmaps, to discussions among vendors, to wishes from the community, this often comes out on top.

Historically Jakarta EE has used different component models for many of its constituent specifications. Java Server Faces (JSF), now called Jakarta Server Faces, had its own managed beans as did, for example, the REST (JAX-RS), Java Servlet, and EJB specifications. For vendors this meant implementing similar things multiple times over, every time in a slightly different way, while for developers it meant learning similar things multiple times over—and especially wondering why certain things can't be combined in their applications.

For instance, an interceptor can't be applied to a `Servlet` method, while `@RolesAllowed` doesn't work on either a Servlet method or a JSF-managed bean. To fix these issues, a single platform-wide component model was introduced in Java EE 6: CDI. The CDI API fully focuses on being a standalone component model with several well-defined services such as interceptors and decorators.

Jakarta Transactions (JTA) was one of the first APIs to start this alignment process by providing a CDI-compatible interceptor, `@Transactional`, and scope, `@TransactionScope`, in Java EE 7.

JSF followed right away by introducing new scopes such as `@FlowScoped` and a CDI version of the existing `@ViewScoped` in Java EE 7. Quite a few other things such as `@Asynchronous`, `@Lock`, `@Startup`/`@DependsOn`, and `@Schedule` were, unfortunately, not included as CDI versions in Java EE 7. Sadly, those didn't even appear in Java EE 8, though that version did introduce Java EE Security (now Jakarta Security), which is built on top of CDI. That release also delivered JSF 2.3, which provided CDI-based injection and expression language lookup of

a large number of artifacts. Additionally, JSF 2.3 officially deprecated its own managed bean system in favor of using CDI beans.

Jakarta EE is expected to pick up the pace again, providing CDI versions of those enterprise beans and common annotations, as well as upgrading and enhancing the existing CDI support in several Jakarta APIs.

Here are several changes you should expect in the Jakarta EE 10 specs.

Jakarta Server Faces

The next version of JSF will be JSF 4.0. Its own major theme will be removing legacy functionality that has already been deprecated. Plus, legacy features that haven't been deprecated before will be deprecated and likely removed in a future release.

For example, the native expression language that JSF still includes will be removed. This was deprecated over 15 years ago but is still there. That expression language shows up in a number of API types, for example, here in `ActionSource`:

```
public interface ActionSource {
    MethodBinding getAction();
    void setAction(MethodBinding action);
    // other methods omitted for brevity
}
```

All methods referencing types from the native expression language, such as `MethodBinding`, will be removed.

Support for Jakarta Server Pages (JSP) as a view declaration language will be removed as well, meaning Facelets will remain as the only default view language. Corresponding with the potential overall Jakarta EE 10 theme, the native managed bean system will be completely removed, making CDI beans the designated bean type for JSF.

Finally, some of the extension tags will be removed, such as `composite:extension`. These were related to the big plans JSF designers once had for visual editors, such as those that existed for Microsoft Visual Basic. These plans never came to fruition, and despite some attempts, most of it was withdrawn. Some remnants of these plans, however, remained in JSF and will now finally be removed.

You can expect some new small features and refinements in the API, for instance, default methods in the `PhaseListener` interface, use of suppliers in several places, adding generics that were still not present, and small utility methods helpful for component libraries. One example: There will be a `release()` method on `FacesContext` as part of `PrimeFaces`.

As for bigger features, a prototype is currently in the works to add a simple REST lifecycle to JSF. This is not intended as a full-featured REST framework, but instead it is to simplify the use case where JSF applications now use a view action in combination with an empty page. This looks as follows:

```
@RequestScoped
public class RestBean {

   @Inject FacesContext context;

   @RestPath("/helloWorld")
   public String helloWorld() {
        return "Hello World! Postback is " +
   }
}
```

Another feature being looked at is supporting extensionless URLs by default or by using a single configuration option. JSF 2.3 provided basic support for this by officially supporting exact mapping, and JSF 4.0 may expand on this support. Thus, a URL such as http://localhost:8080/foo.xhtml (the current default) will be accessible via http://localhost:8080/foo as well.

Scopes have always played an important role in JSF, and one of the things the team is looking forward to is adding a new scope, `@ClientWindowScoped`, which builds on the `Client Id` feature that was introduced in JSF 2.2 as a base facility but was not expanded upon in JSF 2.3.

The Jakarta Security API

Jakarta Security was a new API in Java EE 8. It came out of the box with three authentication mechanisms: `Basic`, `Form`, and a variant on `Form` that's best for working with JSF.

For the version in Jakarta EE 10, the plan is to add new authentication mechanisms. High on the list are at least `Client-Cert` and `Digest`, to make Jakarta Security a full replacement for authentication mechanisms provided by Java Servlet, and to add new methods to support OpenID, OAuth, and JSON Web Token (JWT). The latter is an especially interesting case, because during the Java EE transfer, JWT had already been added to MicroProfile. It's an open question how to deal with this.

Supporting the CDI-alignment theme, the Jakarta Security wish list includes CDI-based alternatives for the common annotations `@RolesAllowed` and `@RunAs`, including, perhaps, supporting the existing annotations. Currently in Jakarta EE, `@RolesAllowed` is supported only by EJB, where it throws an exception if access is denied to a bean method. However, in MicroProfile or, more precisely in JWT, it's implied that `@RolesAllowed` triggers a mandatory authentication mechanism invocation when access is initially denied to a

Jakarta REST resource method. Jakarta Security should cover both cases and define those well.

A major new feature being considered for Jakarta Security is that of user-friendly authentication modules, thereby enabling custom authorization rules. There's some history here. One of the main interfaces in Jakarta Security is the `HttpAuthenticationMechanism`, which is effectively an HTTP-specific and CDI-enabled ease-of-use layer on top of the lower-level `ServerAuthModule` from Jakarta Authentication.

By the way, there is a Jakarta Authorization feature that provides low-level portable authorization modules. However, due to the way modules must be created and installed, modules are not really suitable for use in ordinary applications. Let's hope Jakarta Security provides a similar CDI-enabled ease-of-use layer.

A prototype for this functionality was developed all the way back in 2016, but it was not incorporated in Jakarta Security 1.0 due to lack of time to properly evaluate it. For instance, bridging role checking to an external service instead of assigning all roles when a caller is authenticated would look like the following:

```
@ApplicationScoped
public class MyAuthorizationModule {

    @Inject
    SecurityConstraints securityConstraints

    @Inject
    MyService service;

    @PostAuthenticate
    @PreAuthorize
    @ByRole
    public Boolean myLogic(
        Caller caller, Permission requestedPe

        return securityConstraints.getRequire
                .stream()
                .anyMatch(role -> service.isI
    }

}
```

The authorization module is called by the container to check whether a caller can access a protected URL such as https://localhost:8080/myapp/admin/foo, or in response to `HttpServletRequest.isCallerInRole()`, or following a `@RolesAllowed` annotation.

As part of Jakarta Security, the lower-level Jakarta Authentication and Jakarta Authorization APIs may get some smaller updates as well. These APIs (technically service provider interfaces, or SPIs) are not directly aimed at application developers; the goal is to extend them somewhat and adding clarifications to help higher-levels APIs. For Jakarta

Authorization, an important new feature planned is to allow low-level authorization modules to be installed per application—and allow that to be done by the application. Currently this can be done only at the server level.

The Jakarta Servlet API

Jakarta Servlet is the quintessential API in Jakarta EE. Over time it has been adapted to support the somewhat lesser known Jakarta Managed Beans 2.0 specification, meaning that in Jakarta EE, a servlet is a managed bean. In practice this means some CDI features are supported, such as `@Inject`, but for instance scopes or CDI-style interceptor bindings are not supported.

To align Jakarta Servlet further with CDI is difficult. More than most other APIs in Jakarta EE, Jakarta Servlet has a huge active user base that uses it separately from Jakarta EE, and there are several vendors that exclusively focus on this user base.

So far, the proposals for further alignment vary between multiple options. One is to include a Jakarta EE–specific chapter in the Jakarta Servlet specification that says that in a Jakarta EE environment, servlets should be full CDI beans. This would require no API changes, which is a plus, but it would still require the traditional `Servlet` base class to be extended, which by default delegates all HTTP methods to a single `service()` method. This, for instance, is not ideal for security interceptors.

A potential solution is to make all the methods from the `Servlet` base interface into default methods, so that in a Jakarta EE environment you could write something like the following:

```
@RequestScoped
@WebServlet("/foo/bar")
public class MyBean implements Servlet {

    public void doGet(HttpServletRequest req,
        // ...
    }
}
```

Another proposal is to change nothing in the API but to specify that if a servlet is treated as a CDI bean, and the container detects (for example) that the `service()` method has not been overridden, the `doGet()` methods are called directly. Such a CDI bean would then almost look like a regular servlet:

```
@RequestScoped
@WebServlet("/foo/bar")
public class MyBean extends HttpServlet {

    public void doGet(HttpServletRequest req,
```

```
            // ...
        }
    }
```

Another CDI-alignment issue concerns the additional built-in beans for `HttpServletRequest`, `HttpSession`, and `ServletContext`, which are now defined by the CDI specification. Conceptually those don't belong in the CDI spec, and for this reason alone it would be better if they were moved to the Jakarta EE part of the Jakarta Servlet spec. Practically, the injected `HttpServletRequest` is the most troublesome because it doesn't define which `HttpServletRequest` is injected. `ServerAuthModules` and `Filters` can wrap it and after forwarding to another servlet, there's another version of the request coming into view. Most implementations today inject `HttpServletRequest` in the state in which it entered the request pipeline, and this is often not what applications expect. A Jakarta Servlet native version of `HttpServletRequest` could provide the actual current request.

At the other end of the spectrum of alignment, there's the issue in Jakarta EE that Jakarta REST, which listens to HTTP requests as well, technically does not depend on Jakarta Servlet. In a Jakarta EE environment, it always practically depends on Jakarta Servlet, but in other environments this doesn't need to be the case. To align these two, the idea has been expressed to extract from Jakarta Servlet a low-level flexible HTTP API, on which both Jakarta Servlet and Jakarta REST could be based in Jakarta EE and, potentially, in other frameworks. In practice, this separation already takes place. For example, in GlassFish this is implemented by Grizzly, and in Tomcat there's Coyote.

Besides these alignment issues, there are a number of more native features in the pipeline, with the most important one being RFC 6265: state-management cookies with SameSite behavior.

Other small enhancements to Jakarta Servlet include distinguishing between the query string and POST body parameters, as well as gaining an easier-to-use `HttpServletRequestWrapper` such that only a minimal amount of work has to be done to override the URL.

The Jakarta REST API

Like JSF, Jakarta REST has its own native managed bean system. Because Jakarta REST was introduced together with CDI in Java EE 6, it had some alignment facilities from the get-go, but nevertheless Jakarta REST uses its own injection annotations (specifically `@Context`) and its own rules around these.

Just like JSF 4.0, Jakarta REST 4.0 will drop its own managed bean system and its own injection annotations. This means that moving forward, Jakarta REST resources will be only CDI

beans. An intermediate version, Jakarta REST 3.1, is planned, which will formally deprecate this managed bean system and will allow at least class-level injection of the artifacts currently injected using `@Context` via `@Inject`. This release will likely also deprecate the use of Java Architecture for XML Binding (JAXB) in the API, specifically by deprecating `Link.JaxbLink` and `Link.JaxbAdapter`.

In addition to the switch over to CDI, there will be a number of smaller features introduced. For instance, parameters annotated with `@CookieParam`, `@FormParam`, `@HeaderParam`, `@MatrixParam`, and `@QueryParam` can now also have an array type. In earlier versions of Jakarta EE, they could use only a type of `List`, `Set`, or `SortedSet`. For instance, now you can code the following:

```
@Path("/users")
public class UserResource {
    @GET
    public Response getUsers(@QueryParam("ord
        return …
    }
}
```

Another addition is a default exception mapper that implements `ExceptionMapper<Throwable>` and sets the response to status 500 unless the exception is a `WebApplicationException`. In that case, the mapper sends the embedded response and its own status code.

The Jakarta Concurrency API

The Java EE Concurrency API was first created in 2003, but it was then stalled for many years, only to be released in Java EE 7, seemingly under some time constraints.

The API shows its age a little by still strongly adhering to the container-managed principle. This practically means that the configuration of the concurrency resources is supposed to be done in a proprietary way using specific tools of the Jakarta EE server (for instance, an admin GUI, a CLI, or an XML file inside a server folder).

While this principle may have been the norm in 2003, the world moved on in the years that the Concurrency API lay dormant. More common, concurrency evolved to a hybrid model where resources can be defined and configured by either the server or the application. Therefore, a long overdue addition to the Jakarta Concurrency API is a `@ManagedExecutorServiceDefinition`—just like `@LdapIdentityStoreDefinition` and `@DataSourceDefinition`—which allows applications to define and configure their own managed executor.

By the way, the Jakarta Concurrency API is very important for the CDI-alignment story, because nearly all the things that are still very useful and available only in EJB are related to concurrency. This concerns specifically the following annotations:

- `@Asynchronous`
- `@Lock` and `@AccessTimeout`
- `@Schedule` and `@Timeout`
- `@Stateless`

`@Asynchronous` in EJB is pretty basic, so a newer version could go a little beyond those basics. One proposal is to optionally allow a managed thread pool to be specified on which the annotated method will be executed. That way with two such pools, you can avoid a certain type of deadlock for cooperating asynchronous methods. As with JWT for Jakarta Security, here too a potential difficulty is that MicroProfile has already introduced a CDI-based `@Asynchronous` (in the Fault Tolerance API, which is a little unexpected perhaps).

`@Stateless` itself will not be directly transferred into the Jakarta Concurrency API. One implied aspect is that `@Stateless` beans are pooled, and a single-bean instance is defined to handle only a single call at the same time. Together, these two beans would form a natural way to throttle concurrency. Discussions around this led to a proposed `@Pooled` or `@MaxConcurrency` annotation for the new version of the Jakarta Concurrency API.

A particular problem when doing concurrent programming in Jakarta EE is when, for example, an initial request thread holds a lot of contextual information, such as the current application for which the request is needed (for proper Java Naming and Directory Interface lookups), the authenticated identity, or the current active CDI scopes. When work starts in a new thread, some or all of that context needs to be transferred (propagated).

When the Jakarta Concurrency API was revived from initial work done in early 2000, the designers didn't take CDI into account. This has been a major hindrance ever since because nothing concerning scopes propagates now in a portable way. To solve this problem, an explicit context propagation API is in the works. This API has been prototyped under MicroProfile, with a stated goal that it is to be included in the Jakarta Concurrency API.

Variants of CDI

With Jakarta EE likely having CDI alignment as one of its main themes, the main new feature that is being planned for CDI itself is another variant of CDI. The specification has already been split into three parts: Core CDI, CDI in Java SE, and CDI in Jakarta EE. The new variant, called CDI-Lite, will focus on build-

time concerns, specifically detecting beans during build-time and providing a new kind of extension that can run during build-time.

There's some interesting history here, because this is how EJB 1.0 actually worked; there was no reflection, and skeletons, stubs, and proxies were all generated using tools at build-time. Because this was seen as a lot of hassle, newer versions of EJB built those automatically at runtime using reflection, an approach later adopted by CDI. With CDI now explicitly supporting build-time, it's gone full circle.

Plans for CDI-Lite are still greatly in flux, and it hasn't even been decided yet whether CDI-Lite will be a proper subset of its higher layer, but potentially the stack could look approximately like the following:

1. Jakarta CDI: A small set of key annotations, shared with Guice, HK2, and Spring, including `@Inject`, `@Named`, `@Qualifier`, and `@Scope`
2. Jakarta CDI Lite: Beans, qualifiers (behavior), scopes (behavior), stereotypes, and build-time portable extensions
3. Jakarta CDI Core: Alternatives, decorators, runtime portable extensions (potentially, the build-time extensions are excluded)
4. Jakarta CDI EE: Rules for EJB beans and servlet components, bean names, and scope in expression language, specifically including JSF and JSP, built-in beans for Jakarta Transaction, Jakarta Security, and Jakarta Servlet

While most focus has been on CDI-Lite until now, some features for the main CDI functionality are planned as well. Many of those are specifically for the overall CDI-alignment theme, meaning that they are intended to make it easier for other APIs in Jakarta EE to integrate with CDI.

One such proposal concerns the introduction of executable methods, which effectively lets arbitrary business methods in beans use parameter injection. (Note that CDI already supports this for some callback methods.) An example would be the following:

```
@RequestScoped
public class MyBean {
      String hello(@ConfigOption("foo") Strin
   }
}
```

A framework such as Jakarta REST or JSF, but of course also application code itself, could then execute this method in some way. Perhaps something like:

```
beanManager.execute(bean, method);
```

Some APIs building on CDI struggle because they have fewer options to define or use certain things than CDI itself has, making them a second class citizens. Two examples concern bean-defining annotations and built-in beans.

At the moment, only CDI itself defines which annotations are bean defining. To truly integrate other APIs, they should also be able to create bean-defining annotations. This is something the next version of CDI will likely take a look at.

As discussed above, the CDI spec defines several built-in beans, and so do APIs such as Jakarta Security, JSF and, soon, Jakarta REST. The way this is typically done is via a CDI extension, which programmatically adds a `Bean<T>` instance. These are low-level types, so they have to find their own decorators and generate a proxy to apply them.

Unfortunately, there's no portable API in CDI to find decorators and generate proxies, so many implementations of Jakarta APIs don't actually do this. The result is that such built-in beans are not decoratable and also can't be specialized, which can be quite problematic.

Built-in beans might also need the ability to obtain the current `InjectionPoint`. There's currently no well-defined portable way to obtain such an `InjectionPoint` from within a `Bean<T>` instance. Making this possible is proposed for the next version of CDI.

Another proposed feature gives interceptors in CDI access to their actual (nonbinding) annotation parameters. Currently there's no portable way to achieve this, so interceptors resort to looking at their target class and inspecting that. This works for interceptor annotations that are physically present on those classes, but it does not work for interceptors that have been dynamically added.

There are a few other CDI proposals that have been discussed less but are nevertheless worth mentioning.

The first is the ability to easily apply interceptors to built-in beans. Interceptors are easy to apply to your own code, but they are more troublesome to add to existing types. Using a producer that's an `@Alternative` can use the `InterceptionFactory`, but then you need to get ahold of the type that the `@Alternative` overrides. This can be done using `BeanManager#getBeans` and some filtering, but it's quite verbose. It would be much easier if this overridden instance (the instance that would have been selected for a type if you didn't provide your alternative producer) could be injected directly.

The second issue concerns the programmatic API for obtaining bean instances. This API should provide the same expressive power to select instances that the declarative (injection) API has. At the moment, this is not the case for beans where the beans' producer or `Bean<T>` makes use of an `InjectionPoint`. As a

contrived example, consider the MicroProfile Config API, where a combination of the `ConfigProperty` qualifier and the name of the injected field is used to obtain the right configuration value. Via injection, this works as follows:

```
@Inject
@ConfigProperty
String foo;
```

The inputs to the selection mechanism here are `string`, `ConfigProperty`, and `foo`. The last part is something that can't be provided to the programmatic selection mechanism today. In a proposed feature for CDI, this would be possible and would look something like the following:

```
CDI.current()
      .select(
           String.class,
           new ConfigProperty.Literal(),
           injectionPoint().withMemberName("f
       .get();
```

Other Jakarta EE 10 APIs

Several other Jakarta EE APIs have pending new features that have not been actively discussed as candidates for inclusion in Jakarta EE 10.

For example, Jakarta Persistence has ideas around adding support for transforming Java Persistence Query Language queries to the Criteria API and the other way around, adding higher-level pagination support (the well-known filtering, sorting, and paging paradigm), adding support for specifying which data a fetch graph should not fetch (as opposed to specifying what it should fetch), and supporting some smaller things such as allowing empty collections as a parameter in an `in(…)` clause.

Likewise, Jakarta Messaging has a lot of pending new features. During the Java EE 8 cycle, a number of them had actually been worked on quite a bit for what was to become Messaging 2.1 (which was never released). Specifically features for the CDI-alignment story have been proposed, such as CDI Message Consumers (having a CDI bean listen to incoming messages) and replacing the string-based `activationConfig`, which is in practice a rather thin layer on top of the original XML format used to configure message-driven beans. Smaller features include being able to easily send JSON- or XML-based messages.

There are also various new APIs in the works, for instance, NoSQL and model-view-controller, which may target Jakarta EE 10. For years now, there has also been talk about including a caching and a configuration API in Jakarta EE. A configuration

API actually came to fruition but was developed in MicroProfile after an attempt for Jakarta EE was aborted during the Java EE 8 cycle.

Development of a temporary caching API started as early as 2001, in JSR 107: JCACHE. This was a candidate to include in Jakarta EE multiple times, but it never happened. Whether JCACHE will be transferred to Eclipse and finally be included in Jakarta EE 10 is a big question, and at this point, I don't know the answer.

Conclusion

This article gave just a glimpse into what you might expect for Jakarta EE 10. With about a year to go, Jakarta EE 9.1 is about to come out, discussions are just starting, and an official plan for Jakarta EE 10 still needs to be officially submitted. Nothing has been set in stone, and everything is still subject to change. Still, now you're up to date.

Dig deeper

- Transition from Java EE to Jakarta EE
- Get started with concurrency in Jakarta EE
- Jakarta EE 9 is released

Arjan Tijms

Arjan Tijms was a JSF (JSR 372) and Security API (JSR 375) EG member and is currently project lead for a number of Jakarta projects including Jakarta- Security, Authentication, Authorization, Faces, and Expression Language. He is the co-creator of the popular OmniFaces library for JSF that was a 2015 Duke's Choice Award winner and is the author of two books: *The Definitive Guide to JSF* and *Pro CDI 2* in Java EE 8. Arjan holds an MSc degree in computer science from the University of Leiden, The Netherlands. Follow Arjan on Twitter at @arjan_tijms.

Share this Page