Java EE—the Most Lightweight Enterprise Framework?

community.oracle.com/docs/DOC-1008823

A long time ago J2EE and, especially, application servers were considered to be too bloated and "heavyweight." It could be quite tedious and discouraging for developers to use that technology for developing applications. But since the name of the J2EE framework changed to Java EE, that assumption is not true anymore. How does Java EE compare to other enterprise frameworks and what criteria makes a framework *lightweight*?

When choosing a technology, one of the most important aspects to consider is the developers' productivity during the development process. Engineers should spend as much time as possible implementing use cases and revenue-generating features, because this is what will move a company towards its goals.

The chosen technology and methods should minimize the time developers spend waiting for builds, tests, and deployments; configuring applications; implementing plumbing that is not relevant for business use cases; and configuring the build environment and the external dependencies. But the majority of available technologies do not do this.

Why Standards?

One of the biggest advantages Java EE has compared to other frameworks is the standardization of the used APIs. Standards might sound boring and not innovative enough—and, in essence, this is true, because the Java Specification Requests (JSRs) carve into stone what has been well proven in the past within the industry. But using these standards has a couple of advantages.

Integration of Specifications

The specific APIs within the Java EE umbrella—such as Contexts and Dependency Injection (CDI), JAX-RS, JSON Processing (JSR 353), and Bean Validation—work together really well and are meant to be combined with each other seamlessly. Best of all, CDI is used as a "glue" between components of an application. The specification contains wording such as *"If the container does support specification A and B, then A has to integrate and work well with B seamlessly."*

For example JAX-RS supports JSONP types such as `JsonObject` being used as request or response entities, and it supports calling Bean Validation functionality—including the correct HTTP status codes if the validation fails (see Listing 1).

```
@Path("duke")
public class DukeResource {
    @GET
    public JsonObject getDuke() {
        return Json.createObjectBuilder()
                .add("name", "Duke")
                .build();
    }
    @POST
    public void create(@Valid @NotPlayedYet Game game) {
        // game object has been validated at this point
    }
}
```

Listing 1. JSONP and Bean Validation integration of JAX-RS

Using JSONP types implies that the content type will be `application/json` and that the HTTP status code `400 Bad Request` will be sent if the validation fails. This is all done without needing to write any configuration code.

As another example is that CDI enables developers to inject any beans and user-defined objects into Java EE managed components via `@Inject`. See Listing 2 for a bean validation `Validator` that uses another CDI managed bean straightaway.

```
public class GameNotPlayedValidator implements ConstraintValidator<NotPlayedYet,
Game> {
    @Inject
    GameHistory history;
    public void initialize(NotPlayedYet constraint) {
        // no initialization needed
    }

    public boolean isValid(Game game, ConstraintValidatorContext context) {
        return !history.exists(game);
    }
}
```

Listing 2. CDI integration of bean validation

Integration is a main aspect of the specifications and enables a straight-forward developer experience. The developers can rely on the application server doing the integration and configuration work and can instead focus on the application's business logic.

Convention-over-Configuration Driven Development

Because of the convention-over-configuration driven approaches of Java EE, most real-world applications don't need a lot of configuration. The days of cumbersome XML descriptors are over. For a simple Java EE application, you don't need a single XML file.

Thanks to declarative annotations, a simple annotated plain old Java object (POJO) handles HTTP requests (`@Path`), or serves as an Enterprise JavaBeans (EJB) bean (`@Stateless`), respectively—including transactions, monitoring, or interceptors. In the past, these approaches have proven themselves very well in various frameworks and have been standardized in Java EE.

XML descriptors can still be used for deployment-time configuration if there is a need for this, but convention-over-configuration helps maximize developer productivity.

External Dependencies

The minority of real-world enterprise projects work without any extra dependencies shipped in the deployment artifact. But the justification for these dependencies is mainly driven by technology—such as including logging or entity mapping frameworks or common libraries such as Apache

Commons or Google Guava—not by use cases.

Java EE 7—especially when used together with Java 8—comes with enough functionality to cover the majority of use cases without any other dependencies. What isn't contained out of the box can mostly be realized with a minimal amount of code, for example, injectable configuration via CDI producers, circuit breakers via interceptors (have a look at Adam Bien's open source libraries), or sophisticated collection operations via Java 8 lambdas and streams.

Of course, you could argue not to reinvent the wheel here. But actually it doesn't make a lot of sense to include megabytes of external dependencies into the deployment artifact just to save a couple of self-written lines of code.

And experience shows that the biggest problems are not the directly introduced dependencies but rather the transitive ones. The transitive dependencies very often collide with already existing versions of libraries on the application server and cause challenging conflicts. At the end of the day, developers spend more time managing those conflicts than the time it would take to implement the small features into the project. This is mainly true for the cases with technology-driven, not use-case-driven, dependencies.

See Listing 3 for a simple Java EE 7 project Maven project object model (POM) file—inspired by Adam Bien's Java EE 7 Essentials Archetype.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.sebastian-daschner</groupId>
 <artifactId>game-of-duke</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>

 <dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
 </dependencies>

 <build>
    <finalName>game-of-duke</finalName>
 </build>

 <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

Listing 3. Java EE 7 Maven POM file

Of course, sometimes applications do need to integrate libraries that are crucial to fulfill the purpose of the software. However, these dependencies need to be justified by the business requirements. In general, it makes a lot of sense and saves time and effort to minimize external production libraries.

For testing dependencies, it's a different story, because libraries—such as JUnit, Mockito or, in some cases, Arquillian—are crucial to be included. But again, it makes sense to keep an eye on the list of test dependencies as well.

Thin Deployment Artifacts

Due to the fact that the application server knows about the Java EE API, that API doesn't have to be included in the deployment artifact. Only the business logic is included—with a minimum of glue code and cross-cutting-concerns.

Therefore these kilobyte-sized artifacts make it possible to have very short build times, because the build process doesn't have to copy a lot of things. This can make a difference of several seconds on each and every build. If you sum up all that extra time that is spent by developers and the continuous integration (CI) server, it makes quite a difference. The more often the project is being built—and this is especially true for continuous delivery (CD) scenarios—the bigger that impact is.

Besides short build times, small deployment artifacts also ensure short publish and deployment times. The moving parts are minimal in all cases, due to the fact that the implementation is already contained in the runtime.

The Ideal Framework for Docker

This is exactly the reason why Java EE is the perfect framework to be used in container technologies such as Docker. Docker images are based on layers and when an image is built, the base image already contains the operating system, the Java runtime, and the application. So the only thing that is added on each and every build is the last kilobyte-thin layer of the deployment artifact. This saves time and storage—not only on each build but also when images are versioned or shipped—compared to fat WAR or standalone JAR approaches.

No matter at which stage, having thin deployment artifacts enables very fast and productive deployment pipelines.

Modern Application Servers

J2EE application servers were the embodiment of heavyweight software in terms of start and deployment times, installation sizes, and resource footprints. But in the new world of Java EE, this isn't true anymore.

All modern Java EE 7 application servers, such as WildFly, Payara, WebSphere Liberty, Profile and TomEE, start and deploy in a few seconds. Due to internal, comprehensive modularity, they're able to load only needed components and deploy the thin application artifacts as quickly as possible.

The installation sizes and the footprint nowadays are very reasonable. An application server doesn't consume much more than a simple servlet container but then comes with full-fledged Java EE capabilities. Funnily, a running browser instance nowadays consumes more memory.

Having said that, it is possible and reasonable to deploy only one application per server—either in a container or on premises. With that "one application per application server per container" approach, you then have a very productive and yet flexible solution for modern microservice architectures.

Packaging

When it comes to packaging, there is no reason to still go with EAR files. The approach of having the whole application on a single, dedicated server requires you to have all components on that environment anyway, and doing so saves further build and deployment time. Besides that, this also avoids class loading hierarchy issues that EAR files tend to cause.

In the majority of cloud and microservice deployments, standalone JAR packages are used. These contain both the application and the runtime implementation. In the Java EE world, this approach can be realized using vendor-specific toolchains, for example WildFly Swarm, Payara Micro, or TomEE Embedded.

However, because of the reasons stated above, I highly recommend separating the business logic from the runtime if possible. This means packaging the application in a WAR file that contains only the application's code.

In my opinion, standalone JAR files are a helpful workaround if it isn't possible to control the installation or operation processes due to company "political" issues rather than technical reasons. Then shipping everything that's needed in the deployment artifact and requiring only a Java runtime can work around quite a few non-technical problems.

Recommendation for a Productive Development Process

One of the most productive solutions for enterprise projects is the following:

- Using Java EE 7 and Java 8 only with the API being provided

- Building a kilobyte-sized WAR file containing only the business logic plus minimal plumbing (such as JAX-RS resources or JPA)
- Building a Docker image—adding only a WAR file to a base image containing the configured application server
- Shipping via a CD pipeline that deploys the application using containers

Conclusion

The days of "heavyweight Java EE" are certainly over. The APIs contained in the Java EE umbrella offer a productive and enjoyable developer experience and seamless integration within the standards. In particular, the approach of separating the application code from the runtime enables fast and productive development processes.

With the new MicroProfile initiative that has been initiated by several vendors, it will be possible in the future to further shrink the needed components of Java EE.