CHAPTER 1

Java Development, Enterprise Needs

In a corporate environment, a programming language and software platform like Java has to fulfill a couple of needs that are important to running a business. It has to be able to connect to one or more databases, reliably establish communication with other IT-based systems in the same company or connected businesses, and be powerful enough to consistently handle input and perform calculations based on both input and database data, as well as present the appropriate output to clients. As a cross-concern, security also plays an important role: an authentication process needs to be established that forces users to identify themselves, and an authorization needs to be achieved to limit the amount of resources a particular user is allowed to access. In addition, activities need to be logged for technical maintenance and audit purposes, and the platform should be able to present monitoring data for technical sanity checks and performance-related investigations.

For all of these elements to work in a desired way, a language and platform must be stable with respect to future changes and enhancements. This has to happen such that new language and platform versions can be appropriately handled by the IT staff. Jakarta EE follows this trail and by that largely augments its usefulness for corporate environments.

In this chapter, we will talk about standardization issues that help Jakarta EE to achieve its goals. And we will deal with licensing and the relationship of Jakarta EE to other technology stacks. The chapter closes with a short survey about Java 8 as a platform and as a programming language.

© Peter Späth 2019
P. Späth, *Beginning Jakarta EE*, https://doi.org/10.1007/978-1-4842-5079-2_1

Standardized Specifications

Specifications are important—they tell us what a software can do and how it does it, and they keep track of new versions. The main specification we use in this book reads Jakarta EE 8, and it includes sub-technologies also closely described by exact version numbers. We give a list here and a short description of what each technology does. If you don't understand it yet, don't worry. We will give thorough introductions to most of them in the course of this book. Note that the list is not exhaustive—it does not include some more advanced APIs, which we won't cover in this beginning Jakarta EE book.

- **Enterprise Java Beans (EJB)—Version 3.2**

  EJBs represent entry points for business logic. Each EJB plays the role of a *component* in an overall Jakarta EE architecture and signs itself responsible for a dedicated business task. EJBs allow one to add security, transactional features, JPA features for communication with databases, and web services functionality, and they can also be entry points for messaging (JMS; see later bullet item).

- **Java Server Faces (JSF)—Version 2.3**

  JSF is the dedicated web front-end technology to be used for browser access. It superseded JSPs (Java Server Pages), although the latter is still part of the Jakarta EE specification. In this book, we will concentrate on JSF for front-end work. JSFs usually communicate over EJBs with the business logic.

- **Unified Expression Language (EL)—Version 3.0**

  An important means for JSF pages to communicate with the application logic.

- **RESTful Web Services (JAX-RS)—Version 2.1**

  REST (REpresentational State Transfer) is about the original HTTP protocol, which defines reading and writing resources. It recently gained increased attention for single-page web applications, where the front-end page flow gets completely handled by JavaScript running in the browser.

- **JSON Processing (JSON-P)—Version 1.1**

  JSON (JavaScript Object Notation) is a lean data format that is particularly useful if a considerable amount of the presentation logic gets handled by JavaScript running in the browser.

- **JSON Binding (JSON-B)—Version 1.0**

  This technology simplifies the mapping between JSON data and Java classes.

- **Web Sockets—Version 1.1**

  Provides a full-duplex communication between web clients (browsers) and the Jakarta EE server. Other than "normal" access via HTTP, web sockets allow the server to send messages to a browser client as well!

- **JPA—Version 2.2**

  The Java Persistence API. Provides high-level access to databases.

- **Java EE Security API—Version 1.0**

  A new security API that didn't exist prior to Jakarta EE 8. It includes an HTTP authentication mechanism and an identity store abstraction for validating user credentials and group memberships, and also provides a security-context API to programmatically handle security.

- **Java Messaging Service (JMS)—Version 2.0**

  This is about messaging, which means messages can be produced and consumed asynchronously. A message sender produces and issues a message and can instantaneously continue its work even when the message gets consumed later.

- **Java Transaction API (JTA)—Version 1.2**

  JTA makes sure that processes that combine several worksteps acting as a unit can be committed or rolled back as a whole. This can become tricky if distributed partners are involved. JTA helps a lot here to ensure transactionality, even for more complex systems.

- **Servlets—Version 4.0**

  Servlets are the underlying technology for server–browser communication. You usually configure them only once at the beginning of a project. We describe servlets where necessary to get other technologies to run.

- **Context and Dependency Injection (CDI)—Version 2.0**

  CDI allows one to bind contexts to elements that are governed by a dedicated lifecycle. In addition, it injects dependencies into objects, which simplifies class associations. We will use CDI to connect JSF elements to the application logic.

- **JavaMail—Version 1.6**

  This provides facilities for reading and sending email. This is just an API; for an implementation, you can, for example, use Oracle's reference implementation: `https://javaee.github.io/javamail/`.

- **Bean Validation—Version 2.0**

  This allows for restricting method call parameters to comply with certain value predicates.

- **Interceptors—Version 1.2**

  Interceptors allow you to wrap method calls into invocations of interceptor classes. While this can be done by programmatic method calls as well, interceptors allow you to do that in a declarative way. You usually use interceptors for crosscutting concerns, like logging, security issues, monitoring, and the like.

- **Batch Processing—Version 1.0**

  This handles jobs that need to be started based on some scheduling.

- **Java Server Pages (JSP)—Version 2.3**

  JSPs can be used to establish a page flow in a server–browser communication. JSP is an older technology, but you still can use

it if you like. You should, however, favor JSFs over JSPs, and in this book we don't handle JSPs.

- **JSP Standard Tag Library (JSTL)—Version 1.2**

  This is used in conjunction with JSPs for page elements. You *could* use it for JSFs as well, but you should avoid it, since confusing side effects are likely to show up if you combine them. In this book, we won't talk a lot about JSTL.

Jakarta EE runs on top of the Java Standard Edition (SE), so you can always use any classes and interfaces of the Java SE if you program for Jakarta EE. A couple of technologies included within the Java SE, however, play a prominent role in Jakarta EE, as follows:

- **JDBC—Version 4.0**

  An access API for databases. All major database vendors provide JDBC drivers for their product. You *could* use it, but you shouldn't. Use the higher-level JPA technology instead. You'll get in contact once in a while, because JPA uses JDBC under the hood.

- **Java Naming and Directory Interface (JNDI)**

  In a Jakarta EE 8 environment, objects will be accessed by other objects in a rather loose way. In modern enterprise edition applications, this usually happens via CDI, more precisely via dependency injection. Under the hood, however, a lookup service plays a role, governed by JNDI. In former times, you'd have to directly use JNDI interfaces to programmatically fetch dependent objects. You could use JNDI also for Jakarta EE 8, but you normally don't have to.

- **Java API for XML Processing (JAXP)—Version 1.6**

  This is a general-purpose XML processing API. You can access XML data either via DOM (complete XML tree in memory), SAX (event-based XML parsing), or StAX (see the following bulleted item). This is just an API; normally you'd have to also add an implementation, but the Jakarta EE server does this automatically for you.

- **Streaming API for XML (StAX)—Version 1.0**

  This is used for streaming access to XML data. *Streaming* here means you serially access XML elements on explicit demand (pull parsing).

- **Java XML Binding (JAXB)—Version 2.2**

  JAXB is for connecting XML elements to Java classes.

- **XML Web Services (JAX-WS)—Version 2.2**

  Web services are for remotely connecting components using XML as a messaging format.

- **JMX—Version 2.0**

  JMX is a communication technology you can use to monitor components of a running Jakarta EE application. It is up to the server implementation as to which information gets available for JMX monitoring, but you can add monitoring capabilities to your own components.

The specifications get handled by a community process, and there will be tests that have to be passed if a vendor wants to be allowed to say its server product conforms to a certain version of Jakarta EE (or one of its predecessors, JEE or J2EE). It is not necessary to study this process if you want to understand Jakarta EE to the level we cover in this book, but if you are interested, the corresponding online resources give you much information about it. As a start, enter "java community process jcp" or "java eclipse ee.next working group" in your favorite search engine.

Multi-tiered Applications

In a corporate environment especially, it is common practice to modularize applications. On a higher level, the modules usually get called *layers,* and if there is more than one layer the application architecture is referred to as multi-layered or multi-tiered architecture.

So far, we've been talking about the client–server model, which is the most common example of a two-tiered architecture. For web applications and applications with dedicated client applications instead of browsers, it is, however, more appropriate to consider a three-tier architecture, which consists of the following elements:

- **Client applications**

  Browsers or specialized programs running on client machines and containing only input and presentation logic.

- **Application server**

  A server like Jakarta EE responsible for calculating and delivering data to the presentation layer.

- **Data source**

  A layer that holds the data. Most probably this is a database.

In a multi-tiered or multi-layered model, each layer depends only on the layer underneath it. So, in a three-tiered model the application tier depends on the data tier, and the presentation tier depends on the application tier. See Figure 1-1.

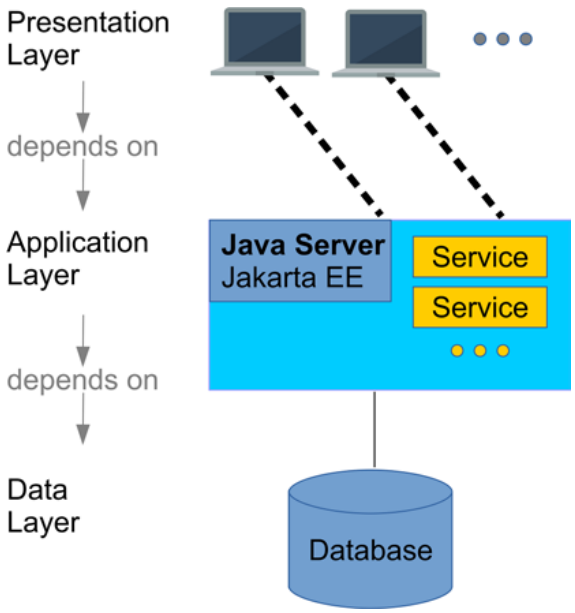

Figure 1-1. *Three-tiered model*

There are other models with a different tier demarcation, or even four and more tiers. For our aim, it is best to think of a three-tiered model as just stated.

Why Jakarta EE?

The Java enterprise edition was initially developed by Sun Microsystems and had the name J2EE. In 2006, the naming and versioning schema was changed to JEE, and after J2EE version 1.4 came JEE version 5. Since then, major updates have happened, and versions JEE 6, JEE 7, and JEE 8 were released. In 2010, Sun Microsystems was acquired by Oracle Corp. Under Oracle Corp., the versions JEE 7 and JEE 8 were released. In 2017, Oracle Corp. submitted Java EE to the Eclipse Foundation, and there the name of JEE 8 was changed to Jakarta EE 8.

In the beginning of 2019, the transition from JEE 8 to Jakarta EE 8 was still ongoing. So, depending on when you read this book, it could be that for online research on Jakarta EE 8 you have to consult pages about both JEE 8 and Jakarta EE 8. This is something you should keep in mind. To not complicate things in this book, we will only talk about Jakarta EE.

Exercise 1

Which of the following is/are true?

1. Jakarta EE 8 gets maintained exclusively by a single company.
2. Jakarta EE 8 does not depend on the Java standard edition (JSE).
3. Jakarta EE 8 is a successor of Jakarta EE7.
4. A multi-tiered model describes a modularization using independent modules.
5. The access to a database could be handled exclusively by a dedicated single tier.

Jakarta EE Servers and Licensing

When this book was written, there were not many Jakarta EE 8 servers released. There are basically the following:

- Glassfish Server, Open Source Edition, from Oracle Corp.
- WildFly Server, from Red Hat

- JBoss Enterprise Application Platform, from Red Hat
- Websphere Application Server Liberty, from IBM
- Open Liberty, from IBM

These servers have different licensing models. Glassfish, WildFly, and Open Liberty are free. This means you can use them without charge both for development purposes and production. To run the JBoss Enterprise Application Platform a subscription is required, although the sources are open. Websphere Application Server Liberty is proprietary.

In this book, we will talk about the Glassfish server, open source edition, version 5.1. Due to the nature of Jakarta EE 8, a transition to other servers is always possible, although you would have to spend a considerable amount of time changing the administration workflow.

Note If you target a proprietary server, it is generally not recommended to start development with a different product from a different vendor. You should at least try to develop with a free variant of the same server, or try to get a developer license. To learn Jakarta EE 8, using Glassfish first and only later switching to a different product or vendor is a reasonable approach.

Excursion to Microservices

Microservices are currently en vogue. They describe an architecture model where each module is responsible for just a single fine-grained task. While it is not this book's goal to introduce microservices, nothing prevents us from following microservice architecture paradigms, as follows:

- Each microservice handles just one identifiable and easy-to-grasp business task.
- Microservices are loosely coupled. Each microservice may easily be replaced by a new version.
- When releasing a new version of a microservice, the old version should be made available for some time to allow for transition.

- Microservices must be well isolated from other microservices. That means each microservice should be functional as independently from other microservices as possible.
- Each microservice may provide its own user interface. This could be a web front end, for example.
- Communication between different microservices should happen in a lean message format, like, for example, JSON.
- Microservices should be stateless to avoid complex state handling.
- If combined with Jakarta EE, each microservice gets deployed using its own deployment artifact. Under certain circumstances, a single microservice might be running in its own server instance. It could be possible, for example, to run microservices all in one server instance, or to scatter them over many different servers running on different network nodes.
- Microservices often use lean REST interfaces for communicating with other microservices.

We won't describe microservices explicitly in this book, but if it fits your purpose you can tailor your Jakarta EE application to adhere to these microservices paradigms.

Jakarta EE Applications and the Cloud

There is an ongoing discussion about whether enterprise applications should be running on something that is considered a monolithic Jakarta EE server, or in a cloud environment, which basically means following a microservices architecture and having the infrastructure for running applications get outsourced to a cloud. If you consider them opposite poles, there are good reasons to favor one over the other. Some of the reasons are technical, some stem from marketing perspectives, and some target licensing and maintenance issues. Instead of contributing to this almost religious discussion, I leave the final decision to the reader. A couple of points that could be taken into account are as follows:

- A cloud is not utterly new from a technical perspective; the services infrastructure gets handled by a cloud product, which could be run by a third-party company. It still follows the venerable client–server paradigm.
- Jakarta EE servers are nowadays more lightweight than they used to be a single instance has an infrastructure overhead of less than 100 MB of memory. This is small compared to what modern servers can provide. A RAM of 64 GB capacity, common today, allows for hundreds of Jakarta EE instances to run on one computer, and it is even possible to switch off certain unneeded parts of a Jakarta EE server to further reduce the memory footprint.
- Cloud applications presumably are better scalable compared to monolithic Jakarta EE applications.
- If you rely on cloud infrastructures provided by other companies, you have to be aware that your business data get handled by foreign companies. This requires a big amount of trust, and in the worst case you lose control over valuable business resources.
- If you use clouds provided by other companies, you outsource technical know-how. This is an advantage since you don't have to provide appropriate human resources yourself, but you also give away control and risk a vendor lock-in.

If control over your own applications and your own data is important, having your own Jakarta EE infrastructure might be the way to go. You could even consider running your own company cloud either with or without the participation of Jakarta EE. In this book, we won't cover cloud issues, but you are free to tailor your applications to mimic cloud-like behavior from an infrastructure perspective.

Exercise 2

True or false?

1. Jakarta EE 8 follows a microservices architecture.
2. To run Jakarta EE 8 you need cloud access.

The Java Standard Edition JSE 8

In this book, we talk about the Jakarta EE 8 server, which entirely runs on and depends on Java. Java was invented in 1991 but was first publicly released under version 1.0 by Sun Microsystems in the year 1996. Over the twenty-three years since then, Java has played an important role as both a language and a runtime environment or platform. There are several reasons why Java became so successful, as follows:

- The same Java program can run on different operating systems.
- Java runs in a sandboxed environment. This improves execution security.
- Java can be easily extended by custom libraries.
- The Java language was extended only slowly. While a slow evolution means new and helpful language constructs are often missing from the most current language version, it helps developers to easily keep track of new features and thoroughly perform transitions to new Java versions in longer-running projects. Furthermore, with only a small number of exceptions, Java versions were backward-compatible.
- Java includes a garbage collector, which automatically cleans up unused memory.

Since 1998 and the major rebranding as Java2, the platform was made available in different configurations, as follows:

- The standard edition J2SE for running on a desktop. Further separated into JRE (Java runtime environment) for just running Java, and JDK (Java development kit) for compiling and running Java.
- The micro edition J2ME for mobile and embedded devices
- The enterprise edition J2EE with enterprise features added to J2SE. Each J2EE configuration includes a complete J2SE installation.

For marketing purposes, the “2” was removed in 2006, and the configurations since then got named JSE (or JDK, which is JSE plus development tools), JME, and JEE, respectively. In 2018, JEE was moved to the Eclipse Foundation and renamed Jakarta EE. The Java language substantially changed in the transition from Java 7 to Java 8. We will be using all modern features of Java 8 for our explanations and code examples.

So, if we talk about Jakarta EE, a complete set of the standard edition is included. Knowledge of the SE is a requirement for this book; the author, however, tries to explain complicated constructs.

Java, of course, gets developed further. While the latest version of Jakarta EE was 8 while writing this book, and the underlying Java standard edition version was 8 as well, the latest Java SE (JSE) version you could download was 11. We won't be talking about Java SE versions 9 or higher in this book.

The Java 8 Language

While knowledge of the Java standard edition (JSE) version 8 is considered a prerequisite for this book, for readers who are only partly familiar with Java 8, the following new features are worth investigating before moving on to the next chapters:

- Functional interfaces
- Lambda calculus (unnamed functions)
- The streams API for working with collections and maps
- The new date and time API

We will be using these where appropriate for the examples we are going to describe in this book.

Exercise 3

True or false?

1. Jakarta EE 8 can be run with Java exchanged by C++.
2. Java is a language, *and* it is a platform.
3. Jakarta EE 8 applications can be developed with a programming language other than Java.