JOHAN VOS

BIO

Listen to author ***Johan Vos*** *discuss the productivity gains offered in Java EE 7.*

Java EE 7

Boosting Developer Productivity with Java EE 7

Increase your productivity using the updated specifications and the new APIs in Java EE 7.

Java EE 7 is the long-awaited major overhaul of Java EE 6. With each release of Java EE, new features are added and existing specifications are enhanced. Java EE 7 builds on top of the success of Java EE 6 and continues to focus on increasing developer productivity.

The Java EE umbrella contains a number of specifications. JSR 342 describes the overall container requirements, and it lists the technologies that should be supported in a Java EE platform. The Java EE Platform Specification project also has a dedicated wiki. The Java EE specification document explicitly mentions ease of development as one of the key goals:

> *"Java EE 7 also continues the 'ease of development' focus of Java EE 5 and Java EE 6. Most notably, Java EE 7 includes a revised and greatly simplified JMS 2.0 API. Ease of development encompasses ease of configuration as well."*

The early versions of the Java EE specification were often criticized as being very complex for developers to work with. The balance between complexity and simplicity was clearly more in the direction of complexity.

With Java EE 5, focus shifted toward ease of development. By leveraging annotations and by providing default behavior and default configuration, it enabled a drastic reduction in lines of code and lines of XML configuration.

The Java EE 5 requirements for simplicity are twofold:

- Defaults should be applicable to most applications and useful for most developers.
- It should always be possible to change the defaults (by providing annotations or descriptors).

By meeting these requirements, the Java EE specification adhered to the convention-over-configuration design paradigm. For behavior that follows the convention, no additional code or XML should be required. If developers want nonconventional behavior, they should be able to achieve that by writing code or declaring XML. This clearly lowered the barrier to Java EE development.

The Java EE 6 specification made reasonable progress in the same direction and added a number of new APIs (for example, JAX-RS, describing the Java REST APIs).

Java EE 7 simplifies the default development and configuration even more. The list of changes in the technologies that were already part of Java EE 6 is huge, but this article highlights a few changes in existing specifications that will help developers increase their productivity:

- JPA 2.1 (JSR 338) standardizes schema generation. Developers can rely on JPA to create tables, indexes, generators, and so on.
- JPA 2.1 also adds support for stored procedures.

MAKING IT SIMPLER

Java EE 7 simplifies the default development and configuration even more.

PHOTOGRAPH BY TON HENDRIKS

- JPA 2.1 requires containers to provide a default datasource.
- Servlets 3.1 (JSR 340) allows for asynchronous I/O based on NIO2.
- JavaServer Faces (JSF) 2.2 (JSR 344) adds support for HTML5.
- JSF 2.2 now contains a file upload component.
- Bean Validation 1.1 (JSR 349) allows for method-level constraints.
- Contexts and Dependency Injection (CDI) 1.1 (JSR 346) allows developers to specify the orders of the interceptors. In addition, by using the @Vetoed annotation, types or packages can be prevented from being considered by CDI.
- Expression Language (EL) 3.0 (JSR 341) supports lambda expressions.

But Java EE 7 is more than a maintenance release. It also adds a few additional APIs, including the following:

- JSR 352: Batch processing
- JSR 353: Java API for JSON processing
- JSR 356: WebSocket
- JSR 236: Concurrency

The new JSRs were added to the platform because they are relevant to today's Java EE developers. The landscape in enterprise computing is always evolving, and the Java EE standard should evolve as well. More specifically, the Java EE specifications should allow enterprise developers to take advantage of the latest and greatest technologies, without having to write lots of boilerplate code and without running into potential vendor locks—hence the need for standardization.

Without exception, the new JSRs target real developer needs, and they allow developers to boost their productivity and focus on their core business.

Batch Processing

Many real-world business applications have a need for some means of *batch processing*. Occasionally or periodically, data has to be processed in a noninteractive way, and the processing can take a long time. A typical example is a job that runs once a day and calculates an inventory and generates reports.

It was already possible to achieve this in the Java EE world, but developers had to come up with their own code, or they had to use existing third-party (nonstandardized) frameworks. JSR 352 now describes the involved entities and their relations. The terminology and concepts are very familiar to users of existing batch frameworks.

Using JSR 352, Java EE developers can define a *job* using XML. **Listing 1** shows how a job is composed of steps containing chunks.

LISTING 1 LISTING 2

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee">
  <step id="myStep" >
    <chunk item-count="3">
      <reader ref="myItemReader"></reader>
      <processor ref="myItemProcessor"></processor>
      <writer ref="myItemWriter"></writer>
    </chunk>
  </step>
</job>
```

Download all listings in this issue as text

The typical job execution is a process that reads data, processes data, and writes results. Java classes are used to read, process, and write data. In the XML example in **Listing 1**, three Java classes are involved, which are shown in **Listing 2**.

By using JSR 352, developers don't have to worry about the infrastructure required for batch processing. They can focus on their domain-specific issues instead.

JSON

Many Web services use the JSON format for receiving requests and sending responses. Also, a large number of Web frameworks (Java and non-Java) rely on the JSON format. As a consequence, many

Java EE applications need to be able to parse or generate JSON code. A number of JSON parsers and generators are available, but using them implies that developers have to create hard dependencies on nonstandard interfaces.

JSR 353 standardizes the concepts of JSON parsing. Using JSR 353 APIs, developers can use standardized APIs in the package javax.json (and in the subpackages javax.json.spi and javax.json.stream) rather than using proprietary APIs. The underlying implementation is shielded from the developer, and it is expected that suppliers of JSON parsers will implement the standard APIs.

COMPLEXITY WON
The early versions of the Java EE specification were often criticized as being very complex for developers to work with. **The balance between complexity and simplicity** was clearly more in the direction of complexity.

The JSON processing APIs have some similarities with the XML processing APIs. This is important, because many applications have to deal with both XML and JSON. Having a similar approach for both formats makes developers' lives easier.

Similar to XML processing, there are two ways to process JSON data: stream-based processing and document-based processing.

Using the stream-based approach, a JSON document is read and processed event by event, as shown in **Listing 3**. The document-based approach allows developers to browse through the whole JSON document—similar to using the Document Object Model (DOM) for browsing XML documents (see **Listing 4**).

LISTING 3 LISTING 4 / LISTING 5

```
JsonParserFactory factory = Json.createParserFactory();
 JsonParser parser = factory.createParser(...);
while (parser.hasNext()) {
Event event = parser.next();
}
```

Download all listings in this issue as text

WebSocket

Java EE developers are increasingly confronted with Web clients or client applications that require full-duplex bidirectional communication. This request is often driven by HTML5–based Websites that want to leverage the WebSocket protocol, as described in the IETF RFC 6455. Also, applications that are not Web–based (for example, JavaFX applications) often benefit from the advantages provided by the WebSocket protocol.

As part of the Java EE 7 platform, developers are able to use JSR 356, Java API for WebSocket. This JSR provides Java EE developers with an easy-to-use annotations-based framework for managing WebSockets. Without this JSR, developers would need to write lots of boilerplate code in order to register a WebSocket endpoint or make a connection to another WebSocket endpoint.

All the internal details about the WebSocket protocol are provided by implementations of JSR 356. This saves a huge amount of time. Registering a WebSocket endpoint and accepting incoming messages can be done as shown in **Listing 5**.

From the code in **Listing 5**, it should be clear that the bulk of the work—including the lifecycle management of the WebSockets—is done by the implementation provided by the Java EE platform.

When a client makes a connection to a WebSocket annotated with @ServerEndpoint("/myEndpoint"), the method annotated with @OnOpen is called and a Session object is passed.

Each time a message is received, the method annotated with @OnMessage is called. When the connection gets closed, the method annotated with @OnClose is called.

For developers, using the WebSocket API is almost as simple as implementing the application-specific logic and annotating the methods with the correct annotations. More WebSocket examples can be found here.

Concurrency

A complaint often heard before Java EE 7 was that the concurrency tools developed for Java SE cannot be used in Java EE implementations. In the Java EE world, all resources are managed by containers. Threads and executors, as in the java.util.concurrent package, are not managed by containers. It is, therefore, dangerous or even forbidden to use them in an enterprise application.

This situation made life for the average Java developer pretty difficult, because a developer is used to working with the concurrency tools when developing a Java SE application, but can't use the same concepts when developing a Java EE application. JSR 236 now provides the infrastructure for using the traditional concurrency concepts in a container-managed environment.

Building on the java.util.concurrent.ExecutorService, the ManagedExecutorService provides a container-managed ExecutorService, which is safe to use in Java EE applications. A ManagedExecutorService implementation can be obtained using JNDI and easily inserted, as shown in **Listing 6**.

EVER EVOLVING

The landscape in enterprise computing is always evolving, and the Java EE standard should evolve as well.

Other Upgrades in Java EE 7

Apart from the new JSRs, a number of existing specifications got a major upgrade in Java EE 7. Most notably, the specification for Java Message Service (JMS) has been changed from 1.1 to 2.0. The JAX-RS specification, which was introduced in Java EE 6, was also updated to version 2.0 and is now included in Web Profile.

JMS. The Java EE 5 and Java EE 6 specifications made it much easier to use most of the enterprise resources, including Enterprise JavaBeans (EJB) and persistence resources. The messaging component, however, had not changed since 2003, when version 1.1 was released. Compared to other Java EE 6 technologies, using JMS 1.1 required lots of boilerplate code and configuration.

JMS 2.0 now provides the same simplifications provided for the other components. By default, sending a message or processing a message requires less code and less configuration. But more-flexible and more-complex configurations are still possible using either annotations or descriptors.

Basically, JMS is about sending messages from one software component to another. This is nontrivial, and it requires a number of intermediate steps provided by different actors. In JMS 1.1, most of these steps had to be implemented by the Java developer. In JMS 2.0, most of the steps are implemented by the JMS implementation.

A typical usage of JMS 1.1 would inject a ConnectionFactory and a ConnectionQueue and create a Connection, a Session, a MessageProducer, and a Message. In JMS 2.0, this becomes much simpler by default. A JMSContext is created, and this context is used to send messages, as shown in **Listing 7**.

JAX-RS. The Java API for RESTful Web Services is one of the fastest-evolving APIs in the Java EE landscape. This is, of course, due to the massive adoption of REST-based Web services and the increasing number of applications that consume those services.

JAX-RS 2.0 fine-tunes the server-side specifications for REST APIs and adds a client part. Today's enterprise applications often make their services available via REST APIs, but they also need to consume other (external) Web services. In order to do this with Java EE 6, developers needed to either create their own boilerplate code or use a proprietary framework.

LISTING 6 LISTING 7 / LISTING 8

```
ManagedExecutorService mes = (ManagedExecutorService)ctx.lookup(
"java:comp/env/concurrent/ThreadPool");
```

Download all listings in this issue as text

JAX-RS specifies a javax.ws.rs.client package that can easily be used for querying external Web services. A POST request to a fictive server called my.server.com can be done as shown in **Listing 8**.

Note the use of method chaining, which is very convenient and productive when a (potentially) large number of methods have to be called.

The JAX-RS 2.0 specification also allows the result of a client request to be java.util.concurrent.Future, thereby enabling asynchronous requests. This can simply be done by adding the async() method call on the Target instance. Asynchronous processing is also possible on the server side by adding the @Suspended Asyncresponse signature in the processing method.

Conclusion

The Java EE specifications need to strike a difficult balance between business needs and developer productivity. Starting with Java EE 5, developer productivity became more of a priority and changes to support increased productivity were done without jeopardizing business needs.

Java EE 7 follows this path by providing some new features that are very important in today's business environments and by simplifying some existing specifications. The Java EE 7 specifications, thereby, achieve two goals:

- It is very easy to use the default behavior with very few lines of code.
- If the default behavior is not what is needed, it is still possible to configure the behavior.

The majority of existing application servers will likely implement the Java EE 7 specifications. The GlassFish project serves as a Reference Implementation for the various specifications. `</article>`

LEARN MORE

- Java EE 7 specification
- "Batch Applications in Java EE 7—Understanding JSR 352 Concepts"
- "JSON-P: Java API for JSON Processing"
- "JMS 2.0 Early Draft—Simplified API Sample Code"
- JMS 2.0 blog