01-2017 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Java ist die beste Wahl

Einfacher programmieren
Erste Schritte mit Kotlin

Security
Automatisierte Überprüfung von Sicherheitslücken

Leichter testen
Last- und Performance-Test verteilter Systeme

4 191978 304903 01

iJUG Verbund

Der Unterschied von Java EE zu anderen Enterprise Frameworks

Kotlin ist eine ausdrucksstarke Programmiersprache, um die Lesbarkeit in den Vordergrund zu stellen und möglichst wenig Tipparbeit erledigen zu müssen

Innerhalb von Enterprise-Anwendungen spielen Sicherheits-Aspekte eine wichtige Rolle

Java EE – das leichtgewichtigste Enterprise Framework?

Sebastian Daschner, Sebastian Daschner – IT-Beratung

Es war einmal eine Zeit, da galten J2EE und insbesondere die Application Server als zu aufgebläht und schwergewichtig. Es konnte ziemlich mühsam und entmutigend für Software-Entwickler sein, diese Technologie zu benutzen. Aber spätestens seitdem der Name zu Java EE geändert wurde, ist diese Annahme nicht mehr wahr. Wo genau liegt der Unterschied von Java EE zu anderen Enterprise Frameworks und was macht ein Framework überhaupt leichtgewichtig?

Einer der wichtigsten Aspekte bei der Auswahl einer Technologie ist der Entwicklungsprozess und die daraus resultierende Produktivität. Programmierer sollten möglichst viel Zeit mit dem Lösen von Problemen und Implementieren von Use-Cases verbringen. Denn genau das schafft am Ende des Tages den Mehrwert eines Produktes beziehungsweise den Profit einer Firma.

Das bedeutet wiederum, dass die Technologien und Methoden die Zeit, die Entwickler mit Warten auf Build-Prozesse, Tests, Deployments, aufwändigem Konfigurieren von Applikationen und Implementieren von nicht-Use-Case-relevanten „Klemptnerarbeiten" sowie Konfigurieren der Build-Umgebung und Abhängigkeiten der Applikation verbringen, minimieren sollten.

Standards

Einer der größten Vorteile, den Java EE gegenüber anderen Frameworks bietet, ist die Standardisierung der APIs. Standards klingen schwergängig und innovationsscheu – und genau das sind sie im Grunde auch, denn der Java Community Process (JCP) hält in den Java Specification Requests (JSR) fest, was sich über längere Zeit davor in der Industrie etabliert hat. Doch genau diese Standards bieten eine Reihe von Vorteilen.

Zusammenspiel der einzelnen Technologien

Die verschiedenen APIs innerhalb des Java-EE-Umbrellas wie zum Beispiel CDI, JPA, JAX-RS, JSONP oder Bean Validation spielen sehr gut zusammen und können out of the box miteinander kombiniert werden. Allen voran sorgt dafür CDI, das als „Kleber" zwischen Komponenten fungiert. Die JSRs beinhalten Voraussetzungen wie: Sobald eine bestimmte Technologie auf dem Container

zur Verfügung steht, muss sie auch mit der anderen Technologie nahtlos zusammenspielen.

Zwei Beispiele: JAX-RS unterstützt JSONP-Typen wie „JsonObject" als Request und Response Entities beziehungsweise integriert nahtlos Bean Validation – passende HTTP-Status-Codes im Falle einer fehlgeschlagenen Validierung inklusive. Validatoren wiederum können per „@Inject" CDI-managed-Beans injizieren und so weiter – Integration der Spezifikationen ist ein großes Augenmerk des Java-EE-Umbrellas.

Diese Ansätze schaffen ein Entwicklerfreundliches und produktives Arbeiten, da sich die Programmierer darauf verlassen können, dass der Application Server die Integrations- und Konfigurationsarbeit leistet. Der Fokus kann somit auf der eigentlichen Business-Logik liegen.

Deklarative, von Convention-over-Configuration getriebene Entwicklung

Der Convention-over-Configuration-Ansatz von Java EE sorgt dafür, dass das Gros der Anwendungsfälle ohne jegliche Konfiguration auskommt. Die Tage von Unmengen an „xml"-Deskriptoren sind vorbei. Für die einfachste Java-EE-Anwendung ist keine einzige „.xml"-Datei mehr notwendig.

Dank der deklarativen Annotations wird ein einfaches, annotiertes POJO zum HTTP Endpoint („@Path") beziehungsweise zur Enterprise Bean („@Stateless") inklusive Transaktionen, Monitoring und Interzeptoren. Diese Ansätze haben sich in der Vergangenheit in vielen Frameworks etabliert und wurden in Java EE zum Standard erhoben.

Externe Abhängigkeiten

Die wenigsten Enterprise-Projekte in der realen Welt kommen ganz ohne im Deployment-Artefakt mitgelieferte externe Bibliotheken aus. Dazu gehören jedoch erfahrungsgemäß eher technisch- als Use-Case-begründete Abhängigkeiten, allen voran Logging- oder Entity-Mapping-Frameworks oder Common Purpose Libraries wie Apache Commons oder Google Guava.

Java EE 7 – insbesondere in Verwendung mit Java 8 – liefert jedoch ausreichend APIs und Funktionalitäten mit, um die gängigen Use-Cases im Enterprise-Umfeld abzudecken. Was nicht out of the box vorhanden ist, kann meist mit minimalem Code gelöst werden, zum Beispiel benötigte, injizierte Configuration per CDI-Producer, Circuit Breaker per Interzeptoren oder aufwändige Collection-Operationen mit Java-8-Lambdas und -Streams.

Natürlich könnte man an dieser Stelle argumentieren, das Rad nicht neu erfinden zu müssen. Er ergibt jedoch wenig Sinn, für eine Handvoll eingesparter Lines of Code Megabyte an Code-Bibliotheken in das Projekt und Deployment-Artefakt zu ziehen. Dabei liegt erfahrungsgemäß das größte Problem noch nicht einmal in der Größe der damit eingeführten direkten Abhängigkeiten, sondern in den transitiven Abhängigkeiten. Letztere kollidieren in vielen Fällen mit anderen im Deployment-Artefakt oder im Application Server vorhandenen Versionen und sorgen für aufwändig zu lösende Konflikte.

Im Endeffekt verbringen die Entwickler mehr Zeit mit dem Konfigurieren und Managen von Abhängigkeiten und eventuellem Ausschließen etwa per Maven „<exclude>"-Direktive als mit der Programmierung und dem Test des Features. Das gilt selbstverständlich in erster Linie für die einfacheren (und doch meisten) Fälle und erfahrungsgemäß dann, wenn Abhängigkeiten technisch und nicht Use-Case-begründet sind.

Ein Beispiel: In den wenigsten Fällen sieht die Business-Anforderung einer Applikation ein bestimmtes Logging-Framework vor, jedoch durchaus eine Integrations-Bibliothek eines externen Systems. Der Unterschied und die Legitimation, ob eine externe Abhängigkeit benutzt werden soll, liegt – neben der Komplexität – also an der fachlichen Notwendigkeit der Lösung. Es empfiehlt sich generell, wenn möglich auf Dependencies zu verzichten, vor der Einführung die Vor- und Nachteile gut abzuwägen und in allen Fällen den Dependency-Tree des Projektes im Auge zu behalten.

Programmieren gegen APIs

Genauso wie die Empfehlung, auf Java-Code-Ebene wenn möglich gegen vorhandene Interfaces statt Implementierungen zu entwickeln, hängen Java-EE-Applikationen im Idealfall nur vom API und nicht von der Container-Implementierung ab. Das minimiert Fehler und Risiken und gewährleistet eine Portabilität zwischen Implementierungen. Auch wenn in Projekten die Portabilität meist nicht in Anspruch genommen wird oder, falls doch, nicht unbedingt reibungslos vonstattengeht, ist sie eher als Rettungsboot denn als Garantie zu verstehen. Gerade für langfristige Projekte im Enterprise-Umfeld ist es unabdingbar, sich auf eine abwärtskompatible, möglichst portable Technologie verlassen zu können. Damit erreicht man langfristig die geringeren Wartungs- beziehungsweise Portierungsaufwände.

Da das Java-EE-API den Application Servern bekannt ist, muss es außerdem nicht beim Deployment mitgeliefert werden. Im Artefakt ausgeliefert wird nur die tatsächliche Businesslogik – mit nur geringem Anteil an „Glue Code" und Cross-Cutting-Concerns.

Schlanke Deployment-Artefakte

Da in einem modernen Java EE-Projekt das API nur als „provided Dependency" verwendet wird, landen im Deployment-Artefakt auch nur die projekteigenen Klassen. Das sorgt für sehr schlanke Artefakte im Kilobyte-Bereich, die wiederum schnelle Build-Zeiten ermöglichen, da der Build-Prozess ohne ständiges Kopieren der Abhängigkeiten auskommt. Der Unterschied kann in der Praxis viele Megabyte und im Build einige Sekunden ausmachen. Summiert man die Zeit, die alle Entwickler beziehungsweise die Continuous Integration Server im Lauf der Projektzeit dadurch mehr aufwenden, dann werden die Unterschiede deutlich. Je häufiger man das Projekt zusammenbaut – gerade im Hinblick auf Continuous Delivery – desto größer diese Ersparnis.

Abgesehen von Build-Zeiten sorgen schlanke Artefakte auch für schnellere Deployment- beziehungsweise Publish-Zeiten. In allen Fällen sind die Moving Parts gering, da die Implementierung des Frameworks eben schon auf dem Application Server vorhanden ist und nicht jedes Mal mit ausgeliefert wird.

Das ideale Framework für Docker

Für moderne, containerbasierte Umgebungen bietet Java EE aus diesem Grund die ideale Struktur. Das Base-Image enthält schon das Betriebssystem, die Java-Runtime und

den Application Server; das Zusammenbauen des Container-Images fügt nur noch die letzte, wenig Kilobyte schlanke Image-Schicht des Deployment-Artefakts hinzu. Diese Zeit- und Speicherplatz-Ersparnis im Vergleich zu Fat-war- oder Standalone-jar-Ansätzen betrifft sowohl das Zusammenbauen als auch das Ausliefern und Versionieren der Images.

Moderne Applikationsserver

J2EE Application Server waren der Inbegriff schwergewichtiger Software in Hinblick auf Start- und Deployment-Zeiten, Installationsgröße und Ressourcen-Footprint. Doch seit der neuen Welt von Java EE entspricht diese Annahme nicht mehr der Wahrheit.

Alle modernen Java EE 7 Application Server wie WildFly, Payara, WebSphere Liberty Profile oder TomEE starten und deployen in wenigen Sekunden. Dafür sorgen fortgeschrittene Modul-Systeme, die nur die benötigten Komponenten initialisieren, und die Tatsache, dass die Deployment-Artefakte sehr schlank gehalten sind.

Die Installationsgröße und der Footprint halten sich sehr in Grenzen. Ein Application Server verbraucht nicht viel mehr Speicher als ein Servlet Container und liefert dann aber schon alle benötigten Enterprise-Funktionalitäten mit. So ist es mit den heutigen Servern möglich und sinnvoll, statt mehrere Deployments auf einem Server einfach mehrere Server mit jeweils nur einer Applikation laufen zu lassen – entweder in einem Container oder als herkömmliche Installation.

Packaging

Was das Packaging angeht, gibt es keinen wirklichen Grund, der noch für „ear"-Archive spricht. Diese Archive beinhalten die Web- und Enterprise-Komponenten noch einmal verpackt als „.war"- beziehungsweise „.ejb-jar"-Archiv. Da der Ansatz, die komplette Applikation auf einem einzigen, dedizierten Applikationsserver laufen zu lassen, voraussetzt, dass ohnehin alle Komponenten vorhanden sein müssen, lässt sich durch den Verzicht auf „.ear"-Archive und die zusätzliche Verpackung Build- und Deployment-Zeit einsparen. Davon abgesehen vermeidet man damit alle Arten von Classloader-Hierarchie-Fehlern, da alle Klassen in einem Kontext vorhanden sind.

In den neuen Microservices- und Cloud-Ansätzen werden Anwendungen vermehrt als Standalone-jar, das sowohl die Applikation als auch die Laufzeit beinhaltet, ausgeliefert. Das ist in der Java-EE-Welt dank Technologien wie WildFly Swarm oder TomEE Embedded möglich.

Aus den genannten Gründen ist es jedoch sinnvoll, wenn möglich die Geschäftslogik, also die Anwendung, von der Implementierung zu trennen. Standalone-jars sind nach Meinung des Autors genau dann ein hilfreicher Workaround, wenn kein Einfluss auf die vorhandene Infrastruktur genommen werden kann, etwa darauf, welche Versionen von Applikationsservern zu Verfügung stehen, oder wenn die Installations- und Operations-Prozesse zu schwergängig sind. In diesen Fällen setzen die Standalone-jars nur eine Java-Runtime voraus und liefern den Rest selbst mit.

Dennoch ist die zu empfehlende Art des Packagings das „.war"-Archiv, das idealerweise nur die Applikation ohne externe Bibliotheken enthält. Ein sehr gutes Beispiel liefert der Maven Java EE 7 Essentials Archetype von Adam Bien.

Fazit

Die Tage von schwergewichtigem J2EE sind schon länger vorbei. Das API des Java-EE-Umbrella bietet ein sehr gutes Entwicklungs-Erlebnis und nahtlose Integration der vorhandenen Standards. Gerade der Ansatz, die Applikationen gegen die APIs zu programmieren und die Implementierung separat zu halten, sorgt in der Praxis für gut performende Entwicklungs- und Auslieferungsprozesse. Mit dem heutigen, modernen Java EE 7 zusammen mit Java 8 haben Entwickler daher eine produktive und ausgereifte Technologie zum Realisieren von Enterprise-Software an der Hand.

Sebastian Daschner
mail@sebastian-daschner.com

Sebastian Daschner arbeitet als freiberuflicher Java-Consultant, Software-Entwickler und -Architekt und programmiert begeistert mit Java (EE). Er nimmt am Java Community Process teil, ist in der JSR 370 Expert Group vertreten und entwickelt an diversen Open-Source-Projekten. Er ist Java Champion und arbeitet seit mehr als sieben Jahren mit Java. Sebastian Daschner evangelisiert Java- und Programmier-Themen unter „https://blog.sebastian-daschner.com" sowie auf Twitter unter „@DaschnerS". Wenn er nicht gerade mit Java arbeitet, bereist er auch gerne die Welt – entweder per Flugzeug oder Motorrad.

Interessenverbund der Java User Groups e.V. (iJUG) vereint bereits 31 Java User Groups

Nach der Übernahme von Sun durch Oracle im Jahr 2009 haben sieben Anwendergruppen den iJUG gegründet. Der Interessenverbund ist auf mittlerweile 31 Mitglieder gewachsen und vertritt damit mehr als 20.000 Entwickler aus Deutschland, Österreich und der Schweiz (siehe Seite 66). Ziel ist die umfassende Vertretung der gemeinsamen Interessen der Java User Groups sowie der Java-Anwender im deutschsprachigen Raum. Ein großer Erfolg des iJUG ist die Durchführung der JavaLand-Konferenz, die bereits im dritten Jahr im Phantasialand in Brühl stattfindet und mittlerweile zu den größten Java-Konferenzen in Europa zählt. Hinzu kommt die Herausgabe der Zeitschrift Java aktuell, die bereits nach einem Jahr einen führenden Platz unter den deutschsprachigen Java-Magazinen erreicht hat. Durch regelmäßige Öffentlichkeitsarbeit und in zahlreichen Gesprächen mit Vertretern von Oracle konnte der iJUG etliche Verbesserungen für die Java-Community erzielen. Themen waren unter anderem Sicherheitslücken in Java, Probleme beim Java-Update sowie Aufforderungen an Oracle hinsichtlich der Weiterentwicklung von Java FX und Java EE.