



//table of contents /



COVER ART BY I-HUA CHEN

23

OMNIFACES: MAKING JSF A LOT EASIER

By Anghel Leonard Solve many day-to-day JSF problems with a single, integrated utility library.

31

JAX-RS 2.0: USING ALL THE GOODNESS

By Abhishek Gupta
Understanding the
client API, filters,
interceptors, and other
useful REST features

41

LONG POLLING WITH ASYNCHRONOUS SERVLETS

By Henry Naftulin
The reliable workhorse
of client/server
communications is
the easy-to-use
fallback when other
methods don't work.

47

PUSHING DATA IN BOTH DIRECTIONS WITH WEBSOCKETS

By Danny Coward
Build on WebSockets'
long-lasting connections to create a simple chat app

03

From the Editor

The Rise and Fall of Languages in 2015

06

Letters to the Editor

Comments, questions, suggestions, and kudos

08

Events

Upcoming Java conferences and events

<u>11</u>

Java Books

Reviews of two Java tutorials

<u>59</u>

Path to Java 9

An Early Look at Java 9 Modules

By Ben Evans

Preparing for modularity—the biggest change by far in the next release of Java

65

JVM Languages

Gosu: A Modern, Down-to-Earth Language for the JVM

By Scott McKinney

A low-ceremony language used in enterprise apps offers an extraordinarily flexible, yet static, type system.

<u>72</u>

Containers

Using Multiple Docker Containers

By Arun Gupta

Assemble a cluster of Docker containers and run a Java EE app—without a lot of housekeeping.

<u>80</u>

Microservices

KumuluzEE: Building Microservices with Java EE

By Tilen Faganel and Matjaz B. Juric Develop self-contained microservices with standard Java EE APIs using the open source KumuluzEE framework.

89

Fix This

By Simon Roberts
Our latest code challenges

40

Java Proposals of Interest

JEP 254: Compact Strings

71

User Groups

Manchester Java Community

93

Contact Us

Have a comment? Suggestion? Want to submit an article proposal? Here's how.







01

EDITORIAL

Editor in Chief

Andrew Binstock

Managing Editor

Claire Breen

Copy Editors

Karen Perkins, Jim Donahue

Section Development

Michelle Kovac

Technical Reviewers

Stephen Chin, Reza Rahman

DESIGN

Senior Creative Director

Francisco G Delgadillo

Design Director

Richard Merchán

Senior Designer

Arianna Pucherelli

Designer

Jaime Ferrand

Senior Production Manager

Sheila Brennan

Production Designer

Kathy Cygnarowicz

PUBLISHING

Publisher

Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director

Karin Kinnear +1.650.506.1985

Audience Development Manager

Jennifer Kurtz

ADVERTISING SALES

Tom Cometa tom.cometa@oracle.com

Advertising Sales Assistant

<u>Cindy Elhaj</u> +1.626.396.9400 x 201

Mailing-List Rentals

Contact your sales representative.

RESOURCES

Oracle Products

+1.800.367.8674 (US/Canada)

Oracle Services

+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Copyright © 2016, Oracle and/or its affiliates. All Rights Reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the editors. JAVA MAGAZINE IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. Opinions expressed by authors, editors, and interviewees—even if they are Oracle employees—do not reflect the views of Oracle. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Java Magazine is published bimonthly at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.









7 Billion Devices Run Java

ATMs, Smartcards, POS Terminals, Blu-ray Players, Set Top Boxes, Multifunction Printers, PCs, Servers, Routers, Switches, Parking Meters, Smart Meters, Lottery Systems, Airplane Systems, IoT Gateways, Programmable Logic Controllers, Optical Sensors, Wireless M2M Modules, Access Control Systems, Medical Devices, Building Controls, Automobiles...



#1 Development Platform



//from the editor /



The Rise and Fall of Languages in 2015

An unusually good year for Java and JVM languages

or the last <u>few years</u>, in my first editorial of the new year, I've looked at how programming languages fared during the previous calendar year. Despite the general perception that changes in language popularity are slow-moving affairs, programming languages often suffer sudden declines (Objective-C) or enjoy unexpected surges (Java, as I'll explain in a moment). The most widely used measure of popularity is the TIOBE Index, which counts searches for languages and normalizes the results to a percentage of the total number of searches. Whether web searches are an accurate proxy for popularity is a point of some contention; however, the TIOBE Index has one significant advantage: it provides data for the index going back 15 years. This makes it possible to identify multiyear trends easily.

Good analysis of language popularity necessarily relies on additional sources. I also rely on

data extracted from GitHub, the popular host for open source and private projects; Open Hub, which surveys all active open source projects; and Google Trends. Each measure quantifies different things, and it's important to look at the data in context before determining what useful information it provides.

By most measures, Java had a banner year.

TIOBE just named Java Language of the Year for
2015 because it enjoyed the greatest jump of any
language in terms of percentage of searches.

TIOBE attributes this surge to the use of Java on
Android. I believe this is true, but it's only part of
the story. The rapid adoption of Java 8 certainly
contributed, too. Java's surge definitively lays to
rest the trend of click-bait articles inquiring "Is
Java Dead?" in which pundits invariably concluded
after lots of explanation that it's not dead.

GitHub is a good way to measure popularity



//from the editor /

among younger programmers. Here, too, Java is hot. It currently sits behind only JavaScript in popularity. This represents a remarkable ascent. GitHub initially rose to popularity in the Ruby community. In 2008, Java was the seventh most popular language on GitHub; Ruby was first. Java's ascent of five positions in the intervening years is unmatched. During the same period, no programming language has managed to rise more than two slots. I expect its popularity to continue due to Java's ubiquity on the cloud (every major cloud provider supports it) and its central role in the Internet of Things.

Among third-party JVM languages, the only two entrants to make it into the TIOBE top 50 or to be ranked by Open Hub are Groovy (#17) and Scala (#30). On TIOBE, Groovy had a banner year. It's hard to know the cause of this, although progress in the chief complaint against it performance—has surely helped. In open source projects, Scala has the upper hand in popularity. This suggests that Groovy is more popular in business contexts, which is a transition that Scala must make in the next few

years in order to move out of the margins. I'm curious to see if it will cross this chasm. Several years ago, the biggest complaints about Scala were the binary incompatibility of new releases, slow compile times, and language complexity. Today, the last two concerns remain important obstacles. Meanwhile, languages such as Kotlin, which is viewed by many as a direct competitor, are putting pressure on Scala. So is Java indirectly. Scala's claim to fame is that it enables developers to mix object-oriented (OO) and functional paradigms. But Java 8 introduced functional programming elements, which, while far more modest than Scala's, might induce businesses thinking of looking to Scala for its functional-OO hybrid qualities to stay put.

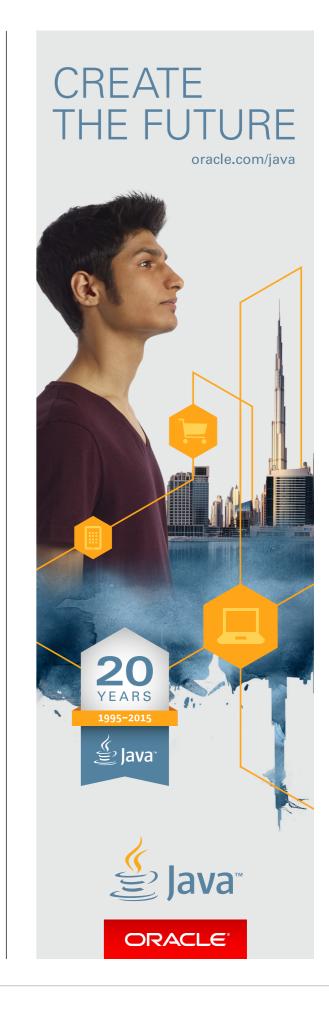
Developers who prefer the functional paradigm will be pleased to know that TIOBE expects that Clojure, with its Lisp-like syntax and currently sitting in third place among JVM languages, will soon advance to the honor roll of top 50 languages. Meanwhile, other functional languages, such as Haskell and Erlang, both broke into the top 40 spots.

In non-JVM languages, perhaps

the most interesting trend, which appears in multiple indexes, is that JavaScript's popularity appears to have peaked. A few years ago, its ubiquity on front ends (both web and mobile) and the advent of Node.js suggested that it might become the new universal programming language (see Atwood's Law). But limitations of the language make it difficult to use on large-scale projects. The result has been the growth of JavaScript transpiling alternatives such as Dart, CoffeeScript, and TypeScript. Of these, I personally most admire TypeScript, which also appears to be gaining the most traction among developers. I'll be curious to see whether the approval of the ECMAScript 6 standard (also known as ECMAScript 2015) in June of last year will improve JavaScript's popularity. We'll check in next year.

Language features and projections of future language adoption are among the most enjoyable discussions in programming. Let me know if you have different views from mine.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy



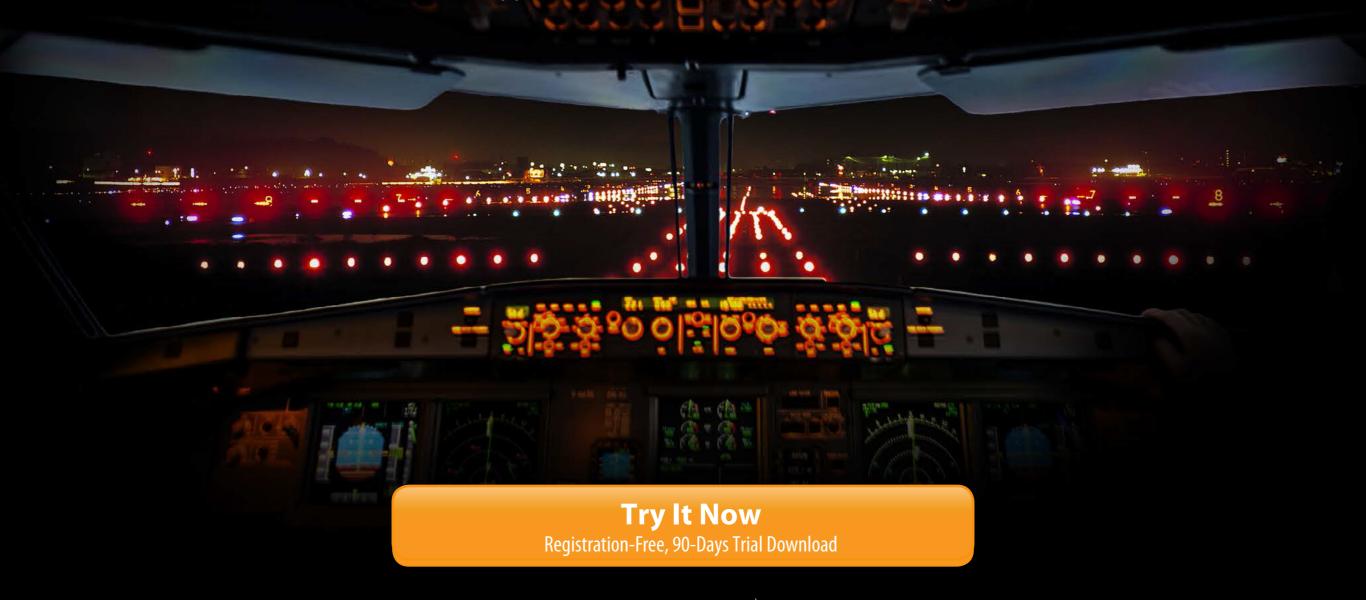






AOT Compilation Has Come to Java 8

Excelsior JET 11 supports Java SE 8 and JavaFX 8 on all desktop platforms.







//letters to the editor /



SEPTEMBER/OCTOBER 2015

More JavaFX and Introductory Articles

Java Magazine used to run articles for developers new to Java in every issue. In addition, there were occasional articles on JavaFX. Is it possible to continue publishing such articles?

—Marius Claassen

Editor Andrew Binstock responds:
Thanks for your note. Readers' suggestions about what to cover more frequently are very helpful to us.
We will be resuming the beginner series in the next issue, and it should become a regular feature. Michael Kölling—the author of the previous series on introductory topics—will be the author. His focus will be on language features, especially the dark corners where unexpected or unusual behaviors are found.

You might have seen in the September/October issue that we covered TestFX, a way of testing JavaFX apps. And we do have a couple of additional JavaFX articles in the pipeline. If we see an increase in demand for JavaFX articles, we will cover the topic even more.

How Effective Is Static Analysis?

Thank you for your editorial on the value of static analysis ("The

Unreasonable Effectiveness of Static Analysis," September/ October 2015 issue, page 3). You were kind enough to quote some of the statistics I published earlier. Let me make some comments and add some more information.

Static analysis is among the most effective forms of defect. removal, and also fairly inexpensive and fairly rapid. It can be used for both development and also for removing latent bugs in legacy applications. As you pointed out, false positives are much lower today than they were 10 years ago. I think static analysis should become a standard software quality method that is used on just about 100 percent of all software applications, with one caveat—the tool is simply not available for many languages except the 30 or so most popular ones.

However, its effectiveness is undeniable. Table 1 shows the approximate defect removal efficiency (DRE) values for a sample of pretest removal and for test stages. The DRE metric was developed by IBM circa 1973 as a tool for validating the effectiveness of inspections. It is a simple metric. If developers find 90 bugs and a customer reports 10 bugs in the first three months of use, then

the DRE would be 90 percent.

The current US average for DRE is about 92 percent, but top projects can get up to 99.5 percent. This average value is based on both pretest removal and the six normal test stages used

TECHNIQUE	DEFECT REMOVAL RATE
FORMAL INSPECTIONS	87%
STATIC ANALYSIS	55%
EXTERNAL BETA TEST > 1,000 CLIENTS	52%
SYSTEM TEST—CERTIFIED TESTERS	46%
INFORMAL PEER REVIEWS	45%
SYSTEM TEST—UNCERTIFIED DEVELOPERS	36%
FUNCTION TEST	35%
COMPONENT TEST	32%
UNIT TEST	30%
EXTERNAL BETA TEST < 1,000 CLIENTS	28%
DESK CHECK	27%
PAIR PROGRAMMING	22%
ACCEPTANCE TEST	17%
REGRESSION TEST	14%
PERFORMANCE TEST	12%

Table 1. Defect removal efficiency (in descending order) for pretest and test techniques







for most software (unit test, function test, regression test, performance test, system test, and acceptance test). Of course, some critical software—such as medical devices and weapons systems—might use more than 12 test stages and, therefore, exceed 99 percent in DRE.

DRE is only half of the IBM quality metric set. The other half was "defect potential" or the sum of the bugs found in many sources. Today, defect potential is measured using function point metrics because "lines of code" isn't a useful measure for requirements and design bugs. The current US average is about 4.5 bugs per function point, which represents the total of all bugs found in requirements, architecture, design, code, user documents, and "bad fixes" or new bugs in bug repairs.

Once again, thank you for a good article. I hope this additional information will be helpful.

—Capers Jones VP and CTO, Namcook Analytics LLC

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, please indicate this in your message. Please write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.









//events/



<u>JavaLand</u> MARCH 8–10 BRÜHL, GERMANY

This annual conference is a gathering of Java enthusiasts, developers, architects, strategists, and project administrators. Session topics for 2016 include containers and microservices, core Java and JVM languages, enterprise Java and the cloud, front end and mobile, IDEs and tools, and the Internet of Things (IoT). After lectures on the first day of the conference, attendees get exclusive use of Phantasialand and its rides and attractions.

Voxxed Days Berlin

JANUARY 27–29
BERLIN, GERMANY
Sharing the Devoxx philosophy that content comes first,
Voxxed Days events see both internationally renowned and local speakers converge. Berlin topics include Java component design with Spring 4.3; microservices with Java, Spring Boot, and Spring Cloud; and simple REST APIs with Dropwizard and Swagger.

Topconf Linz

FEBRUARY 1—3 LINZ. AUSTRIA

This conference focuses on new ways to manage mobile, Java, cloud, front end, security, and more. Workshops include sketching web and app interfaces, microservices and event sourcing with Spring Boot, and software management in a lean and agile world.

Jfokus

FEBRUARY 8–10
STOCKHOLM, SWEDEN
Jfokus has run for eight years
and is the largest annual
Java developer conference
in Sweden. Conference topics
include Java SE and Java EE,
the inner mechanics of the
JVM, front end and web,
mobile, continuous delivery
and DevOps, the IoT, cloud and
big data, future and trends,
alternative JVM languages,
and agile development.

DevNexus 2016

FEBRUARY 15–17
ATLANTA, GEORGIA
DevNexus is a conference
drawing 1,700 developers, with
6 workshops, 12 tracks, and 120

presentations. Featured tracks include HTML5 and JavaScript, Java SE/Java EE/Spring, and data and integration.

Apache Hadoop Innovation Summit

FEBRUARY 18–19
SAN DIEGO, CALIFORNIA
With presentations from more
than 25 hands-on industry
speakers, topics covered will
include MapReduce and Spark,
building privacy-protected
data systems, scalable data
curation, best practices, and
architectural considerations for
Hadoop applications.

Mobile World Congress

FEBRUARY 22–25
BARCELONA, SPAIN
This industry-leading event
focuses on in-depth analysis of
present and future trends in the
mobile industry. The 2016 MWC
conference program features
tracks on the IoT, smart cities,
digital finance, and more.

Embedded World 2016

FEBRUARY 23–25
NUREMBERG, GERMANY
The 14th annual gathering of embedded system developers





PHOTOGRAPH BY RACHEL TITIRIGIA/FLICKR

//events/



will explore the latest developments, define trends, and once again present the key areas of focus for future developments. This is where hardware, software, and system development engineers come together to turn the next milestones of the IoT into reality.

ConFoo

FEBRUARY 24–26
MONTREAL, QUEBEC, CANADA
ConFoo is a multitechnology conference for web developers, featuring about 150 presentations by popular international speakers. Past sessions have included

Testing Java EE Applications Using Arquillian, by Reza Rahman, and Hybrid Mobile Development with Apache Cordova and Java EE 7, by Ryan Cuprak.

Riga Dev Day

MARCH 2–4 RIGA. LATVIA

This event is a joint project by Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Day focuses on 25 of the most-relevant topics and technologies for that audience. Tracks include JVM and web development, databases, DevOps, and case studies.

QCon London

MARCH 7–9
LONDON, ENGLAND
QCon is designed for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams.
Topics include what to expect in Java 9 and Spring 5, containers in production, microservices for mega-architectures, full-stack JavaScript, and data science and machine learning methods.

EclipseCon 2016

MARCH 7–10
RESTON, VIRGINIA
EclipseCon is all about community. Contributors, adopters, extenders, service providers, consumers, and business and research organizations gather to share their expertise and learn from each other. Topics this year include an introduction to the Eclipse Che next-generation Java IDE, hawkBit and software updates for the IoT, a faster index for Java, and Java 9 support in Eclipse.

jDays

MARCH 8–9
GOTHENBURG, SWEDEN
jDays is a Java developer conference covering Java/Java EE, architecture, security, DevOps, cloud and microservices, testing, JavaScript, IoT trends, methodologies, and tools.

CITCON

with it.

LATE MARCH (DATE TO BE ANNOUNCED)
PERTH, AUSTRALIA
CITCON, the Continuous Integration and Testing Conference, is a worldwide series of free "Open Spaces" events for developertesters, tester-developers, and anyone else with an interest in continuous integration and the type of testing that goes along

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event at least four months in advance at javamag_us@oracle.com. We'll include as many as space permits.



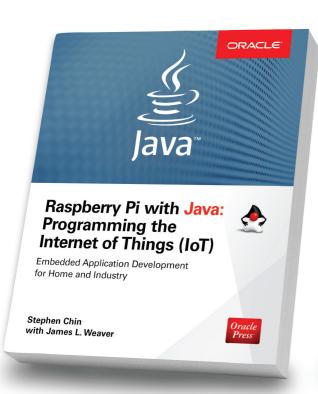


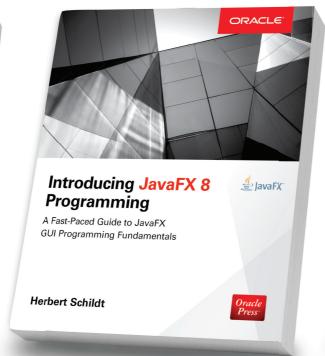


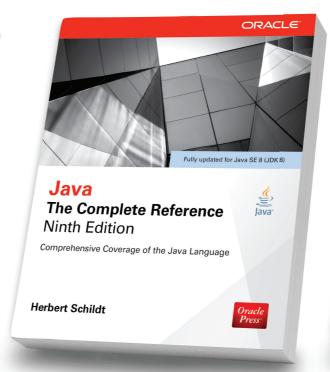


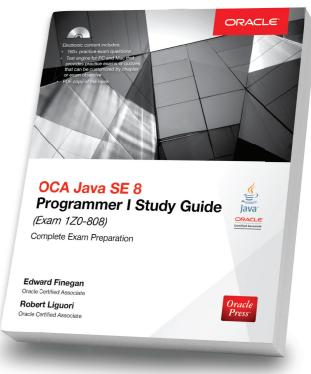
Your Destination for Java Expertise

Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.









Raspberry Pi with Java: Programming the Internet of Things (IoT)

Stephen Chin, James Weaver

Use Raspberry Pi with Java to create innovative devices that power the internet of things.

Introducing JavaFX 8 Programming

Herbert Schildt

Learn how to develop dynamic JavaFX GUI applications quickly and easily.

Java: The Complete Reference, **Ninth Edition**

Herbert Schildt

Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.

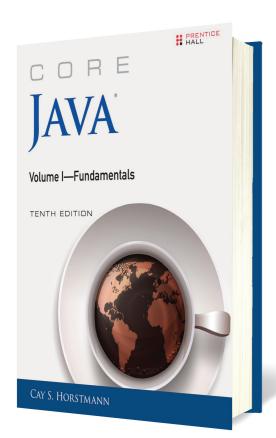
OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)

Edward Finegan, Robert Liguori

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.



//java books /



CORE JAVA VOLUME I—FUNDAMENTALS, 10TH EDITION

By Cay S. Horstmann Prentice-Hall

A common question asked by developers new to Java is which book is the best introduction to the language. The answer really depends on what exactly they're looking for. If it's a detailed explanation of the language that explains every aspect and idiom and can serve as a reference later on, then the book to get is Core Java, which I review next. If, however, they would rather not work through hundreds of pages but would prefer a hands-on experience in which they write small programs that quickly teach the language in a series of graduated projects, then the latest entry in the Murach's Beginning Java series, which I examine in the second part of this review, is the book I recommend.

Core Java announces its historical success by noting that this volume is the 10th edition. Not many books are popular enough to warrant the publisher

releasing 10 editions. But Core Java provides such a deep explanation of the language that most readers will want the book to be always close at hand. This tome, at an impressive 1,040 pages, is only Volume I of a two-volume set. (Volume II will be published shortly. If the division of coverage is similar to earlier editions, it will weigh in at about 1,000 pages as well.) Between the two books, there is no aspect of the language or its principal libraries that is not explained in detail.

Volume I presents everything a mainstream programmer will need to know about Java 8: the basics of the language (including generics, lambdas, and other advanced features), collections, concurrency, and, curiously, graphics programming with Swing. In addition, there is a lengthy section on building and deploying Java programs. (Volume II is scheduled to cover

Java 8 streams, XML and JSON processing, JNI, JNDI, networking, databases and JDBC, and so on.) Essentially, Volume I is the pure language and Volume II is key libraries as well as lesserused language features.

What makes Core Java the definitive work on the language is more than its vast scope—it is the quality of the presentation. Explanations are supremely clear, code examples are meticulous and highly approachable, and the pacing and sequencing are spot on. In addition, if a topic requires background to make the Java examples easier to understand, the author, the well-regarded Cay Horstmann, has no difficulty spending several pages on the conceptual issues before starting in on Javaspecific details. This patience and his preference for complete information is particularly noticeable in the chapters that deal with Java objects. There





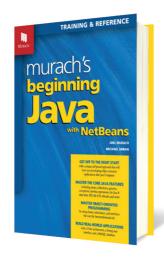


//java books /

Horstmann first explains how Java implements object orientation, the choices and trade-offs it makes, and how those affect the way the language is written and used today. In these explanations (and many others), Horstmann relies on his long experience with the language. Using his background, he frequently explains how certain features are based in Java's early history and how the needs of the time informed the reasoning for various design choices. This kind of full, detailed information makes it much easier to understand subtle aspects of the language and use them as intended. (Note that if this much detail is more information than you're looking for in a language tutorial, Horstmann has an abridged version, Core Java for the Impatient, which weighs in at just over 500 pages. I reviewed this book in the May/June 2015 issue.)

I am a big fan of detailed language tutorials that can serve as definitive references once I've mastered the basics. No other book on Java better fits this mandate than *Core Java*. This has been true for at least the last 10 years, possibly longer. If you feel that the advent of Java 8 suggests it's time to update your language tutorial

and reference, then this is *the* book to get, without any doubt at all.



MURACH'S BEGINNING JAVA WITH NETBEANS

By Joel Murach and Michael Urban Mike Murach & Associates

All Murach books, including this title, use a unique approach to instruction. They deliver a radical implementation of the notion that the best way to learn a new technology is to use it. So, rather than offering extensive tutorials that give you the full background information (as in the previously reviewed title), Murach books provide you only the material needed to master coding the immediate next step. These steps are typically presented in two-page chunks: the left page explaining the topic, the right page presenting the implementation in code,

with accompanying notes about the code itself. Turn the page, and you start a new two-page chunk. Needless to say, these short lessons build on each other so that after a chapter of them, you've done a fair amount of coding and have a working familiarity with the chapter's theme.

This is an entirely hands-on, pragmatic approach that works as follows, when explaining, for example, how to use arrays in Java. The topic comprises these two-page lessons: how to create an array, how to assign values to an array, how to use loops with arrays, how to use enhanced for loops with arrays, how to work with two-dimensional arrays, how to use the Array class, how to copy and compare arrays. Eighteen pages and you're good to go.

The benefit of this approach is obvious, and it makes the Murach books very attractive for teachers as well as for students who just want to get up and running quickly. The books also work as reference volumes. As can be seen from the previous list of topics, they can double as collections of useful, albeit elementary, recipes. (Notice the consistent use of "How to do x" as the template for each lesson.)

This hands-on design also

means that it's considerably easier to jump around and pick up the bits of knowledge you might need for solving a specific problem—without needing to learn the entire language. This approach works particularly well in Murach's books on HTML5 and CSS. It also works with languages, but not quite as well, because language features rarely can be studied in complete isolation.

While this volume is an introduction to the language, I feel that intermediate topics are given exposure that is too short. For example, in this volume, lambdas get only eight pages of coverage, which is plainly insufficient. To its credit, though, the book correctly identifies drawbacks of lambdas, which are rarely mentioned in other treatments: lambdas are difficult to debug, difficult to understand in a stack trace, and for those unfamiliar with them, difficult to understand in code.

This is not an introduction to programming with NetBeans; rather, it's a tutorial on Java that incidentally uses NetBeans. For teachers of Java boot camps or quick language intensives and for developers who need to get up and running quickly, this tutorial is the one to use. —Andrew Binstock







XRebel



THE LIGHTWEIGHT JAVA PROFILER

TRY IT FREE NOW!

Get a free t-shirt!



Building Modern Web Apps



his issue of Java Magazine focuses on development of web apps. Had this issue appeared, say, two years ago, the content would have been completely different. In those days, the topic would have mandated extensive coverage of those innumerable Gordian knots of interconnected services, Java frameworks. But today, in this issue, we cover only one framework, Spring Boot (page 15). This is partly due to the fact that Spring has essentially won the framework wars (at least from the perspective of popularity and adoption), and to the fact that the nature of web apps has changed substantially. REST services have replaced in many ways the services delivered formerly by frameworks. As REST continues its ascendance, the role of the JAX-RS 2.0 spec (page 31) has become more important than ever.

Not all connections to remote servers, however, are suited to REST's simple commands. Some connections need to be long-lasting, and for this problem, there are at least two important solutions: the WebSocket protocol (page 47) as found in Java EE and the more widely supported long-polling approach (page 41), both of which are remarkably straightforward to use. If the greater use of simple services is the emerging pattern in web apps, it's also the emerging pattern at the metalevel. Large applications of all kinds—web or not—are being decomposed into smaller services, which are then orchestrated to deliver the intended functionality. This microservices orientation is being driven via the agility conferred by smaller containers such as Docker (page 72). And ironically enough—new frameworks, such as KumuluzEE (page 80), are gaining traction to aid in this simplification.

Finally, for those of us working with tried-and-true JavaServer Faces (JSF), this issue includes a detailed overview and tutorial of OmniFaces (page 23), the wide-ranging JSF utility library that deservedly won the 2015 Duke's Choice Award.









JOSH LONG

First Steps with Spring Boot

The new Spring framework brings long-awaited ease of development to web apps.

Spring Boot is an opinionated way to quickly build web applications intended for production. It pulls together and integrates lots of technologies including Spring and Java EE. And it makes them accessible through a simple UI that enables you to mix and match components and quickly assemble the pieces, to which you then add your application's specific knowledge. This approach has gained popularity because it greatly reduces the amount of boilerplate code and housekeeping tasks. In this article, I show the steps involved in setting up a simple application that includes a modestly

complex UI and a persistence layer. In addition, it contains tests—in the way that modern software development practices dictate.

Getting Started

There are many ways to get started with Spring Boot. My favorite is Spring Initializr, which is a web service you can use directly or from wizards inside your favorite IDEs (NetBeans; IntelliJ IDEA 14 Ultimate; or the Spring Tool Suite, which is an Eclipse distribution). Whatever your approach, in Spring Initializr you'll be shown a menu of checkboxes. Choose the type of workloads and technologies you'd like to work with.

You can select what type of packaging you'd like: JAR file or WAR file. And you can choose a language: Java or Groovy. You can also specify which version of the JVM you'd like to use: 1.6 through 1.8. Whenever possible, choose 1.8 because the other two versions have already reached official end of life.

This article looks briefly at lots of different technologies. The goal is to highlight just how easy it is to pull them together into a useful application. For our example, choose these technologies from the wizards: Web, JPA (the Java Persistence API), REST repositories, Vaadin (the UI library),

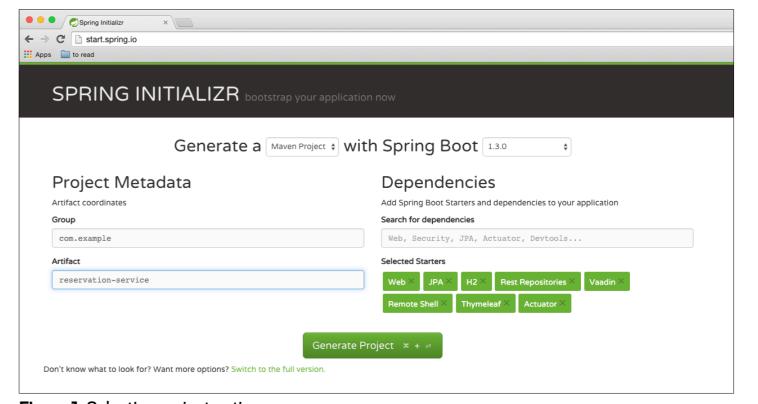


Figure 1. Selecting project options





Actuator (Spring Boot's web services component), Remote Shell, H2 (the database), and Thymeleaf (a templating engine). See Figure 1. Then click Generate Project to download a zip file that you can open in your favorite IDE.

When you open the project, you'll see that there's a simple public static void main method and an empty unit test. You'll also find two empty directories: static and templates in the src/main/resources directory. Templates that are processed on the server side—such as FreeMarker, Velocity, and Thymeleaf—live in the templates directory. Assets that will be sent to the client directly—such as JavaScript, images, CSS, and so forth—live in the static directory. In Figure 2, I've shown the directory complete with a sample Thymeleaf template (reservations.html) and a simple hi.html file to be sent to an HTTP client unprocessed.

Open the Maven build file, pom.xml. You'll see that it contains dependencies that correspond to checkboxes selected in Spring Initializr. The various starter-* dependencies are opinionated dependencies: they don't contain code; instead they just bring in other dependencies. For example, org .springframework.boot:spring-boot-starter-data-jpa brings in everything you're likely to need to get started with JPA: the latest and greatest JPA 2 specification types, the latest Hibernate implementation, Spring's support JPA, ORM, JDBC, and so on.

You don't have to worry about managing version ranges or lining up dependencies across common libraries. Maven has the ability to import or inherit build configuration from other build artifacts. Spring leverages this feature and exposes a useful parent build that defines version ranges and other needed items such as dependencies like the Servlet API, Hibernate, and so forth. There are two benefits to this approach: you can omit version declarations for the dependencies that are predefined for you, and you can update to the latest version of Spring Boot when a new version rolls around by simply overriding the version of the parent build

that you're using. All the associated dependencies will be upgraded in lockstep automatically.

The Data Speaks for Itself

Let's create a simple JPA entity, Reservation.java, that we can manipulate in our application.

```
package com.example;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

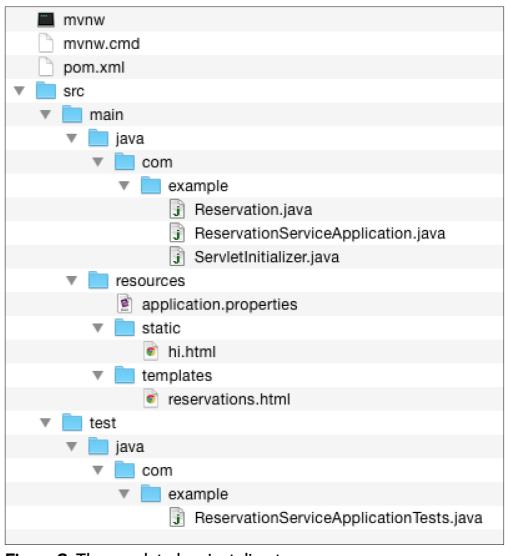


Figure 2. The populated project directory





```
@Entity
public class Reservation {
    @Id
    @GeneratedValue
    private Long id;
    private String reservationName;
    public Reservation(String reservationName) {
        this.reservationName = reservationName;
    Reservation() {
    @Override
    public String toString() {
        return "Reservation{" + "id=" + id +
                ", reservationName='" +
                reservationName + '\'' +
                '}';
    public Long getId() {
        return id;
    public String getReservationName() {
        return reservationName;
```

The Reservation entity is trivial; it's just meant to demonstrate that we can use the full power of JPA. We'll use Spring Data JPA, one module of the larger Spring Data project, which features similar support for other persistence technologies such as Oracle Database, MongoDB, Neo4j, Cassandra, and others. It enables us to declaratively define a repository object,

based on interface method name conventions, such as this:

```
interface ReservationRepository extends
    JpaRepository<Reservation, Long> {
    Collection<Reservation>
        findByReservationName(String rn);
}
```

With this in place, we've also defined an @Bean provider method in the ReservationServiceApplication class. Because it's annotated with @SpringBootApplication, the ReservationServiceApplication class is also an @Configuration class where @Bean provider methods may live. Any @Configuration class may use defined beans as return values from provider methods. This is an alternative to annotating the component itself with stereotype annotations like @Component, @RestController, and so on. The returned object, of type CommandLineRunner, will be run when the Spring Boot application starts up. Therefore, this is an ideal place to insert some sample data for this demonstration:

```
@Bean
CommandLineRunner runner(ReservationRepository rr) {
    return args -> Stream.of(
        "Julia", "Mia", "Phil", "Dave", "Pieter",
        "Bridget", "Stéphane", "Josh", "Jennifer")
        .forEach(n -> rr.save(new Reservation(n)));
}
```

The Web

We can then create a REST API using Spring MVC (as well as the Jersey JAX-RS implementation or Ratpack, for which there are checkboxes in Spring Initializr). Here's our code:

```
@RestController
class ReservationRestController {
```







Bring up http://localhost:8080/reservations and you'll see the results returned in JSON, as in Figure 3.

This is a respectable first cut at a REST API, but what I really want to do is map the state transitions of my entities—turning creation, querying, deletion, and updates into HTTP verbs. So, let's remove the manual REST API we just created and instead let the ReservationRepository that we created earlier—which knows how to create, read, update, query, and delete Reservation entities—handle this for us with Spring Data REST.

We'll simply revise the depository code, adding the requisite @RepositoryRestResource and @RestResource annotations.

Bring up http://localhost:8080/reservations again and you'll see the same information as before along with extra links. These links are metadata that are embedded in the resource response. They provide a menu of paths that a client may take that are related to the payload. You can add custom links using ResourceProcessor implementations. Scroll through the results. Each Reservation contains a link to itself, aptly named self. Scroll further and you'll see the links to the search resource, where you can find more links still, including one that exports our earlier custom finder method, findByReservationName, as a search resource. Now, without any a priori knowledge, a REST client can start at / and interactively navigate through the resources using the hypermedia (which in HTTP parlance refers to the previous links) provided by the links and the HTTP OPTIONS verb to understand what's possible.

Now, let's add some web application functionality. We'll stand up a Spring MVC controller that processes an HTTP



Figure 3. Returned results as a JSON file





request, establishes model data, and then forwards it to a Thymeleaf template for rendering.

A request to http://localhost:8080/reservations
.mvc ultimately renders an HTML template in the src/main/
resources/templates directory. It looks like this:

The result of this is shown in Figure 4.

Many applications might simply deliver REST APIs and serve a client-side JavaScript application that connects to those endpoints. The assets for such an application would live in the src/main/resources/static directory. They will not be processed.

If you're trying to build a data-driven user interface quickly, you might use a UI component-based approach such as the well-regarded, open source <u>Vaadin Framework</u>. Earlier, I selected the **Vaadin** checkbox in Spring Initializr. That selection brings in a third-party dependency to support an easy Spring Boot integration with Vaadin. Vaadin builds on top of <u>GWT</u>; UI components consist of fast, responsive transpiled client-side JavaScript. But in Vaadin apps, business state lives on the server. Here's a simple

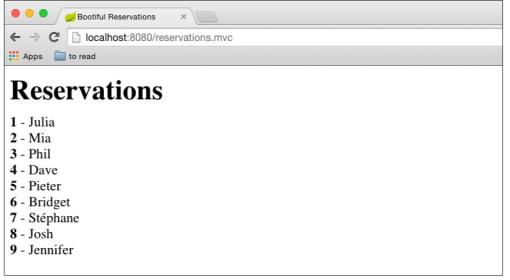


Figure 4. The reservations application so far





data grid to present the Reservation data, using Vaadin components.

```
@SpringUI(path = "ui")
@Theme("valo")
class ReservationUI extends UI {
    private final ReservationRepository
                  reservationRepository;
    @Autowired
    public ReservationUI(ReservationRepository
                         reservationRepository) {
        this.reservationRepository =
            reservationRepository;
    @Override
    protected void init(VaadinRequest request) {
        Grid table = new Grid();
        BeanItemContainer<Reservation> container =
            new BeanItemContainer<>(
              Reservation.class,
              this.reservationRepository.findAll());
        table.setContainerDataSource(container);
        table.setSizeFull();
        setContent(table);
```

And that's it! Run the application by going to http://localhost:8080/ui and you'll see the screen in Figure 5.

Testing

Naturally, we should test all the parts of the application. The Spring MVC test framework makes short work of exercising our HTTP endpoints, thanks to a fluid, built-in domain-specific language (DSL).



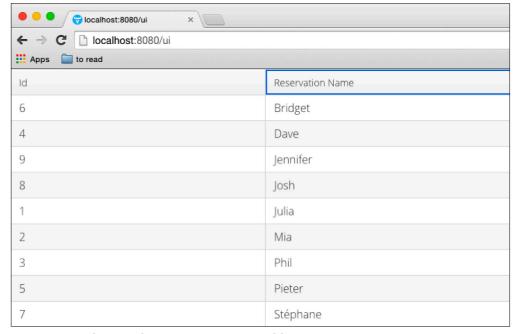


Figure 5. The application using a UI library





```
@Test
public void contextLoads()
    throws Exception {
        this.mockMvc.perform(
            get("/reservations")
            .accept(this.mediaType))
            .andExpect(content().
                  contentType(this.mediaType))
                 .andExpect(status().isOk());
    }
}
```

This code uses the Spring MVC test framework, which is a custom test runner that works with JUnit or TestNG. This test framework is ideal for pseudo integration tests. Here, I use it to stand up our Spring Boot application completely without actually connecting to a socket. The Spring MVC test framework dispatches calls through the whole Spring MVC machinery and actually exercises all the web components, but does so without sitting on a socket somewhere listening for actual TCP packets coming in as HTTP. This means that you don't have to stand up a container just to test the framework machinery or the components responsible for processing requests.

Production

The application is now interesting, but being interesting just isn't enough. We need to get the software deployed. If you've ever read Michael Nygard's excellent book, *Release It!*, then you know that the last mile between "code complete" and "production ready" can be a long mile indeed! Talk to enough organizations and you'll realize that there's a litany of nonfunctional requirements, usually buried in some wiki page somewhere, that each organization requires its applications to satisfy before code can be moved to production.

These aren't exciting things, either. They're not differen-

tiators. What does the application's environment look like? How do you capture growth metrics? Does your application provide an endpoint to answer questions about its health? How do you identify which version of the service is running? Which HTTP resources are exposed to traffic? These questions, and many more besides, are answered thanks to automatically created endpoints provided by the Spring Boot Actuator, which is a module in Spring Boot designed to support operationalizing applications for production. Spring Boot Actuator helps reduce or remove the list of nonfunctional requirements that commonly gate our ability to move code to production. It furnishes management endpoints such as /health, /beans, /trace, /mappings, /metrics, /env, and many more. These endpoints are useful and easily customized. Let's change the management endpoint's prefix with a property specified in the application.properties file, where, along with application.yml, Spring Boot will look to find configuration keys and values:

management.context-path=/admin

With this property, users will need to go to /admin/health, /admin/env, and so on. If I add spring-boot-starter-security to the CLASSPATH, I'll get the default HTTP BASIC authentication prompt with a default username and password that will be printed on the console. I can delegate the HTTP BASIC authentication challenge to a Spring Security AuthenticationProvider instead by configuring the appropriate Spring Security options.

The health endpoint is interesting, because it measures the health of the system. It automatically tells us whatever it can about the technologies in play. It knows about the file system, a DataSource connection pool, JavaMail, JMS, Cassandra, Elasticsearch, Solr, MongoDB, RabbitMQ, Redis, and many more technologies. We can augment this data with a semantic HealthIndicator, if we want to:







The kind of data it retrieves is shown in Figure 6.

The application automatically collects metrics for common things—for example, requested HTTP resource counts, how much memory is available, how many classes loaded, and so on. In addition, any component in the application can inject a

```
← → C | localhost:8080/admin/health
Apps 🛅 to read
    status: "UP",
  - custom: {
        status: "I <3 Java!"</pre>
    },
  - diskSpace: {
        status: "UP",
        total: 499082485760,
        free: 124751495168,
        threshold: 10485760
    },
  - db: {
        status: "UP",
        database: "H2",
        hello: 1
    },
  - links: {
      - self: {
            href: "http://localhost:8080/admin/health"
    }
```

Figure 6. Data from monitoring the application's health

CounterService to emit semantic metrics. If you configure a MetricReporter from the excellent <u>DropWizard Metrics</u>, or simply use one of Spring Boot's built-in MetricWriter implementations, you can collect and graph the metrics from your applications in a single, integrated dashboard such as Graphite, StatsD, and OpenTSDB.

Running the application is easy, too. You can choose to use the self-contained JAR-based deployment by selecting Jar for the packaging in Spring Initializr. Or you can deploy into any servlet container such as Apache Tomcat/TomEE, Oracle WebLogic or GlassFish, Payara Micro, IBM WebSphere, or Red Hat's WildFly application server by selecting .war for the packaging in Spring Initializr. Modern cloud environments such as CloudFoundry, Heroku, Google App Engine, and OpenShift can also run Spring Boot applications.

Conclusion

This short tutorial has just barely scratched the surface. Spring Boot can integrate many other technologies, including many from Java EE: JAX-RS, JMS, the Servlet API, JTA, and JDBC, as well as all the components from the Spring ecosystem. It also provides the basis for Spring Cloud, which is a set of components designed to support the creation of distributed systems based on microservices. Spring Boot is an opinionated way to get going by leveraging default configurations for common tasks. These configurations are often defined and provided by the industry experts who lead the technologies being configured. To get the bits and start your journey, check out Spring Initializr. </article>

Josh Long (@starbuxman) is the Spring developer advocate at Pivotal, a Java Champion, the author of five books on programming (including the upcoming *Cloud Native Java*), and the author of three best–selling LiveLessons videos. He is also a longtime open source contributor.









ANGHEL LEONARD

OmniFaces: Making JSF a Lot Easier

Solve many day-to-day JSF problems with a single, integrated utility library.

mniFaces is a utility library for JSF 2 developed by two members of the JSF Expert Group: Bauke Scholtz and Arjan Tijms. Its main purpose is to give JSF developers a solution for day-by-day JSF-related problems and questions. OmniFaces can be used in both JSF implementations, Mojarra and MyFaces, and it works with existing JSF libraries (such as PrimeFaces, RichFaces, ICEfaces, OpenFaces, and others). In previous writing, I defined OmniFaces as "a utility library for JSF and a comprehensive compendium of programming techniques, design patterns and recipes for JSF developers." [OmniFaces won a 2015 Duke's Choice Award from the Java community. —Ed.]

The OmniFaces project started in March 2012, and the first release (OmniFaces 1.0) took place on June 1 of that year. The latest stable release is OmniFaces 2.2, released in late 2015. According to the project founders, the original motivation for the library was that the lead developers realized they each had built libraries of routines that covered the same problems and that they had separate but similar libraries they'd developed at work. It occurred to them that a single, integrated library with no duplication of solutions would be ideal. This led to OmniFaces. Currently, OmniFaces is on the road to version 2.3 and, in the future, to version 3.0.

What Does OmniFaces Offer?

Today OmniFaces offers a significant number of utility components, validators (including cross-field validation), converters, resource handlers, view handlers, tag handlers, exception handlers (such as an Ajax exception handler), extensionless URL support, event listeners, CDI artifacts (for example, view scope, cookie and init parameters injection, and eager beans), and so on. In addition, it contains more than 300 utility methods and functions. Via these utilities, your JSF code will become less verbose, more loosely coupled, cleaner, and easier to debug. Everything is available for testing (live demo) in the OmniFaces Showcase.

Table 1 gives a quick overview of the contents and abilities of OmniFaces. While using OmniFaces components, you can develop new families of components, extend existing components, suppress or add specific functionality, write custom renderers, and perform other useful actions.

In addition to the capabilities shown here, OmniFaces ships with features that support CDI, FacesViews, filters, managed beans, render kits, scripts, and functions.

How OmniFaces Differs from Other JSF Libraries

To understand the difference, it is important to understand the niche that OmniFaces occupies among the JSF libraries. In Figure 1, you can see the location of OmniFaces on the JSF map.

OmniFaces is not in the same category as PrimeFaces, RichFaces, or BootsFaces. While PrimeFaces and RichFaces can be considered mutually exclusive, this is not the case with OmniFaces. Because it is a utility library, it can be employed with or without PrimeFaces, RichFaces, and BootsFaces, rather than as an alternative to these libraries.







23

COMPONENTS (PARTIAL LIST).		
A SERVER-SIDE CACHE OF RENDERED MARKUP		
LOADS IMAGES FROM byte[]/InputStream/SVG(OPTIONAL AS DATA URI)		
PROVIDES FULL CONTROL OVER THE MARKUP OF A TREE HIERARCHY		
GENERATES A JAVASCRIPT FUNCTION IN THE GLOBAL JAVASCRIPT SCOPE		
DEFERS THE LOADING OF THE GIVEN SCRIPT RESOURCE TO THE WINDOW LOAD EVENT		
KEEPS VIEW OR REQUEST PARAMETERS IN THE REQUEST URL AFTER A POST-BACK		
IGNORES VALIDATION FAILURES (INVOKE ACTION PHASE WILL BE EXECUTED ANYWAY)		
HIGHLIGHTS ALL INVALID UIInput COMPONENTS AND THE ASSOCIATED LABELS		
MULTIPLE for OPTIONS, SINGLE MESSAGE, HTML ESCAPING, ITERATION MARKUP CONTROL		
OLATOR AND CROSS-FIELD VALIDATORS, WHICH ARE NOT SUPPORTED BY DEFAULT IN JSF).		
ALLOWS A LABEL TO APPEAR IN THE MIDDLE OF A BEAN VALIDATION MESSAGE		
SOLVES THE required="true" ATTRIBUTE AND UISelectBoolean ISSUE		
VALIDATES ONLY WHEN THE SUBMITTED VALUE IS REALLY CHANGED		
VALIDATES IF ALL UIInputs HAVE BEEN FILLED OUT		
VALIDATES IF ALL OR NONE OF THE UIInputs HAVE BEEN FILLED OUT		
VALIDATES IF ALL UIInputs HAVE THE SAME VALUE		
VALIDATES MULTIPLE FIELDS BY A CUSTOM VALIDATOR METHOD		
PROVIDES BEAN VALIDATION ON A PER-UICommand/UIInput COMPONENT BASIS, AS WELL AS VALIDATING A GIVEN BEAN AT THE CLASS LEVEL		
RFUL FOR CAPTURING AND TREATING "SILENT" EXCEPTIONS IN A FRIENDLY MANNER).		
ADDS EVERY EXCEPTION AS A GLOBAL FATAL FACES MESSAGE		
HANDLES AS NON-AJAX EXCEPTIONS ANY EXCEPTIONS THAT OCCUR DURING AJAX REQUESTS		
EXTENDS AND IMPROVES THE STANDARD PARTIAL VIEW CONTEXT		

Table 1. List of principal components and handlers in OmniFaces (continued on next page)

Including OmniFaces in JSF Applications

OmniFaces can be added in a JSF application as a JAR file or as a Maven dependency. Practically, it is merely a matter of putting the OmniFaces JAR file into the /WEB-INF/lib directory.

Maven users can add OmniFaces by adding the corresponding Maven coordinates to the pom.xml file. In the following example, I have added the dependency for OmniFaces 2.2:

```
<dependency>
  <groupId>org.omnifaces</groupId>
  <artifactId>omnifaces</artifactId>
  <version>2.2</version>
</dependency>
```

In the JSF application below, I've added it just ahead of the PrimeFaces library and Java EE 7:

```
<dependencies>
<dependency>
 <groupId>org.omnifaces
 <artifactId>omnifaces</artifactId>
 <version>2.2
</dependency>
 <dependency>
 <groupId>org.primefaces/groupId>
 <artifactId>primefaces</artifactId>
 <version>5.2</version>
</dependency>
 <dependency>
 <groupId>javax
 <artifactId>javaee-api</artifactId>
 <version>7.0</version>
 <type>provided</type>
</dependency>
</dependencies>
```







TAG HANDLERS.	
converter	PROVIDES SUPPORT FOR DEFERRED VALUE EXPRESSIONS IN ALL CONVERTER ATTRIBUTES
RestorableView	RE-CREATES THE ENTIRE VIEW WHENEVER THE VIEW HAS BEEN EXPIRED
importConstants	MAPS CONSTANTS OF THE GIVEN FQN OF A TYPE IN THE REQUEST SCOPE
importFunctions	MAPS FUNCTIONS OF THE GIVEN FQN OF A TYPE IN THE "FACELET SCOPE"
validator	PROVIDES SUPPORT FOR DEFERRED VALUE EXPRESSIONS IN ALL VALIDATOR ATTRIBUTES
viewParamValidationFailed	HANDLES VIEW PARAMETER VALIDATION FAILURE VIA REDIRECT OR HTTP ERROR CODE
CONVERTERS (THESE ADVANCED CONVERTI	ERS CLOSE GAPS IN JSF).
ListIndexConverter	PERFORMS CONVERSION BASED ON THE POSITION (INDEX) OF THE SELECTED ITEM IN THE LIST
ListConverter	WORKS DIRECTLY VIA A List OF ENTITIES (CANNOT USE ListIndexConverter)
SelectItemsConverter	AUTOMATICALLY CONVERTS < f: selectItems > ENTITIES
SelectItemsIndexConverter	SAME AS SelectItemsConverter, BUT DOES CONVERSION BASED ON THE POSITION (INDEX)
ValueChangeConverter	DOES CONVERSION ONLY WHEN THE SUBMITTED VALUE HAS CHANGED
GenericEnumConverter	USED IN UISelectMany COMPONENTS WHOSE VALUE HAS BEEN BOUND TO A List <e> PROPERT WHERE E IS AN ENUM</e>
RESOURCE HANDLERS (HANDLING CDNS, M	ULTIPLE RESOURCES IN A SINGLE CALL, AND SO ON).
CDNResourceHandler	PROVIDES CDN URLS INSTEAD OF THE DEFAULT LOCAL URLS FOR JSF RESOURCES
CombinedResourceHandler	CREATES A COMBINED RESOURCE FOR ALL SCRIPTS AND ONE FOR ALL STYLESHEETS
UnmappedResourceHandler	MAPS JSF RESOURCES ON A URL PATTERN OF /javax.faces.resource/*
EVENT LISTENERS.	
InvokeActionEventListener	WORKS WITH NEW <f:event>TYPES preInvokeAction AND postInvokeAction</f:event>
ResetInputAjaxActionListener	RESETS INPUT FIELDS THAT ARE NOT EXECUTED DURING AN AJAX SUBMIT, BUT WHICH ARE RENDERED/ UPDATED DURING AN AJAX RESPONSE
VIEW HANDLERS (A USEFUL AND CORRECT (JSE CASE FOR VIEW HANDLERS).
NoAutoGeneratedIdViewHandler	DOESN'T ALLOW AUTO-GENERATED IDS (j_id AND SO ON)

Table 1. List of principal components and handlers in OmniFaces (continued from previous page)

For OmniFaces SNAPSHOTS, the numbering pattern is *major_version.minor_version*-SNAPSHOT:

```
<dependency>
  <groupId>org.omnifaces</groupId>
  <artifactId>omnifaces</artifactId>
  <version>2.3-SNAPSHOT</version>
  </dependency>
```

Note: Users of outdated environments who can't or won't use CDI should use OmniFaces version 1.11 instead. It doesn't contain anything from CDI, so while it works, it doesn't offer OmniFaces' full range of solutions.

The OmniFaces artifacts are available in the following XML namespaces:

```
xmlns:o="http://omnifaces.org/ui"
xmlns:of="http://omnifaces.org/functions"
```

Using OmniFaces

Probably, the best way to demonstrate OmniFaces artifacts is to explore several examples. For the rest of this article, I'll solve common problems of JSF applications by using OmniFaces. Additional examples can be found https://examplescommons.org/learning-nc-12

Combine script and stylesheet resources to improve page loading. As a JSF developer, you should be familiar with the mechanism of loading resources in JSF. Basically, for each resource, JSF will fire a separate GET request. This is a time-consuming task, especially if there is a large number of resources to load.

The <u>OmniFaces CombinedResourceHandler</u> considerably improves page loading. This is







achieved by removing from the UIViewRoot all the separate script and stylesheet resources that have the target attribute set to "head" and combining them for all scripts (and separately combining them for all stylesheets). So, instead of firing a GET request per resource, the browser will fire a single GET request for all scripts and another one for all stylesheets. Further, the CombinedResourceHandler is responsible for sequentially returning all the resources combined in these requests.

Visually speaking, something like this:

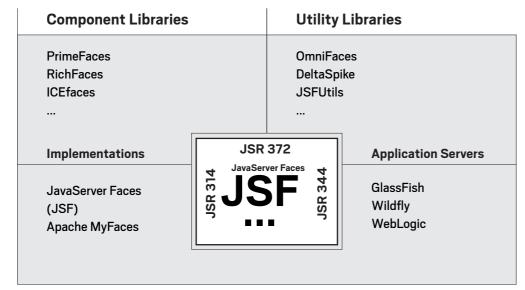


Figure 1. Where OmniFaces fits

```
name="js/custom2.js" target="head"/>
<!-- deferred scripts -->
<o:deferredScript library="default"
     name="js/js2.js"/>
<o:deferredScript library="default"</pre>
     name="js/js2.js"/>
becomes this:
<!-- for stylesheets -->
<link type="text/css" rel="stylesheet"</pre>
href="/MyApp/faces/javax.faces.resource/...css?
ln=omnifaces.combined&v=1433659201062" />
<!-- for scripts -->
<script type="text/javascript"</pre>
src="/MyApp/faces/javax.faces.resource/...js?
ln=omnifaces.combined&v=1433659204389">
<!-- for deferred scripts -->
<script type="text/javascript">
OmniFaces.DeferredScript.add(
    '/MyApp/faces/javax.faces.resource/...js?
     ln=omnifaces.combined&v=1433659201450');
</script>
 [The . . . in these resources indicates an app-specific string
of characters, removed for legibility. —Ed.]
 To get it to run, this handler needs to be registered, as fol-
lows, in faces-config.xml:
<application>
<resource-handler>
org.omnifaces.resourcehandler.CombinedResource
Handler
```





```
</resource-handler>
</application>
```

More details about configuring this handler are available in the <u>OmniFaces Showcase</u>. You might also be interested in <u>CDNResourceHandler</u> and <u>UnmappedResourceHandler</u>.

The select-items converter saves you from writing custom converters for UISelectItems. Typically, <f:selectItems> (or UISelectItems) points to a collection that provides the selectable items with values (objects), which can be instances of SelectItem or any other Java object. These instances are then encapsulated by JSF in SelectItem instances.

To render each selectable item, JSF uses a string representation of the selectable item's value (for example, it might invoke the toString() method of the object). While the rendering process of the selectable items works smoothly, the conversion of the submitted strings to corresponding values can be accomplished only if we indicate a custom converter, which might need to do the job based on possibly expensive service/DAO operations.

This problem is solved by OmniFaces, which provides a general custom converter (SelectItemsIndexConverter) capable of automatically converting the submitted strings to corresponding values based on the position of the submitted string (index) in the list of values.

In order to exemplify this case, first we need the base object (value). Let's use Player:

```
public class Player implements Serializable {
  private Long id;
  private String name;
  private String residence;
  private int age;

// constructor with id, name, residence and age
```

```
// getter and setter
// override equals() and hashCode()
 A managed bean contains a List<Player>:
@Named
@ViewScoped
public class PlayerBean implements Serializable {
 private static final long serialVersionUID = 1L;
 private Player selected;
private List<Player> topfive;
 // populate the list
// getters and setters
 Finally, we expose to the user the List<Player> via a
selection component (such as <h:selectOneMenu>):
<h:form>
<h:outputLabel for="atp" value="Players: " />
 <h:selectOneMenu id="atp"
      value="#{playerBean.selected}">
  <f:selectItem itemLabel="Choose player"</pre>
      noSelectionOption="true" />
 <f:selectItems value="#{playerBean.topfive}"</pre>
      var="t" itemLabel="#{t.name},#{t.age}"
      itemValue="#{t}" />
 </h:selectOneMenu>
 <h:commandButton value="Select"/>
</h:form>
Selected: <h:outputText</pre>
              value="#{playerBean.selected.name},
              #{playerBean.selected.age}"/>
```







This will be rendered as expected, but, if we select and submit a Player string representation, then an error will occur. This is a conversion error indicating that JSF cannot automatically convert the submitted string into the selection. This generates the message: Player: Conversion Error setting value '3, ANDY MURRAY, London, 27' for 'null Converter'.

Normally, at this point you must write a custom converter especially for this case only, but OmniFaces provides a general custom converter named SelectItems

IndexConverter. Once you specify this converter (via omnifaces.SelectItemsIndexConverter), it automatically converts the submitted strings into the corresponding values (objects):

```
<h:selectOneMenu id="atp"
  value="#{playerBean.selected}"
  converter=
     "omnifaces.SelectItemsIndexConverter">
     ...
</h:selectOneMenu>
```

It's simple to use, isn't it? In addition, OmniFaces provides an alternative to this approach, which converts the submitted strings to corresponding values based on the toString() implementation of those values. This <u>converter</u> uses the ID omnifaces.SelectItemsConverter.

Provide full control over the markup of a tree hierarchy by declaring the appropriate JSF components or HTML elements in the markup. By default, JSF does not provide any built-in artifacts dedicated to exposing or controlling the markup in a hierarchical tree structure, but such components are quite popular among JSF developers.

The OmniFaces <u>tree</u> component provides a powerful and extremely versatile hierarchical tree structure solution that can be obtained via a TreeModel (with default implementa-

tions: ListTreeModel and SortedTreeModel) and four tags: the main one, <o:tree> (which encapsulates the entire tree structure), <o:treeNode> (a single tree node within a parent <o:tree>), <o:treeNodeItem> (a single child tree node within a parent <o:treeNode>), and <o:treeInsertChildren> (which indicates the place to insert the children of the current child tree node recursively by an <o:treeNode> associated with the children's level in the same parent <o:tree>).

Let's see a simple example, and let's start with the model.

TreeBean uses ListTreeModel, which is a TreeModel
implementation that holds the tree children in an ArrayList:

```
@Named @ViewScoped
public class TreeBean implements Serializable {
private static final long serialVersionUID = 1L;
private TreeModel<String> tree;
@PostConstruct
public void init() {
 tree = new ListTreeModel<>();
 tree.addChild("Components Showcase")
      .addChild("OutputFamily")
      .addChild("ResourceInclude")
      .addChild("example 1")
      .addChild("source code")
    .getParent().getParent()
      .addChild("example 2")
      .addChild("source code")
    .getParent().getParent().getParent()
      .addChild("Conditional Comment")
      .addChild("...")
    .getParent().getParent().getParent()
      .addChild("UtilFamily")
      .addChild("MoveComponent")
    .getParent()
     .addChild("...");
```







The output is shown in Figure 2.

The full application is available in the Tree app in the <u>down-loadable</u> code.

Use the full Ajax exception handler. Starting with JSF 2, there is a generic API that enables developers to write a global exception handler. This is very helpful, especially when we need to signal "silent" exceptions (such as Ajax exceptions) that are not reported to the user or are reported in a very discreet manner. As you probably know, exceptions that occur during Ajax requests are not treated the same as exceptions that occur during non-Ajax requests. By default, most of them are invisible to the client. Because users do not receive any feedback about the success of Ajax requests, users might become confused and restart the application, resend the request, or take other actions.

The OmniFaces <u>FullAjaxExceptionHandler</u> enables developers to handle exceptions that occur during Ajax

requests by using error pages configured in web.xml (or web-fragment.xml). To exploit the FullAjax ExceptionHandler, we need to explicitly configure the FullAjaxExceptionHandlerFactory in faces-config.xml.

```
<factory>
  <exception-handler-factory>
  org.omnifaces.exceptionhandler
    .FullAjaxExceptionHandlerFactory
  </exception-handler-factory>
  </factory>
```

[In the snippet above, lines 3 and 4 should be entered as a single line. —*Ed.*]

It is good practice to provide at least a fallback error page whenever there is no match with any of the declared specific exceptions. Now, Ajax exceptions will not occur imperceptibly for the user, because they will be reported in the configured error page. The user will now know that something went wrong via messages in the error page and can act accordingly. So, you must at least have either

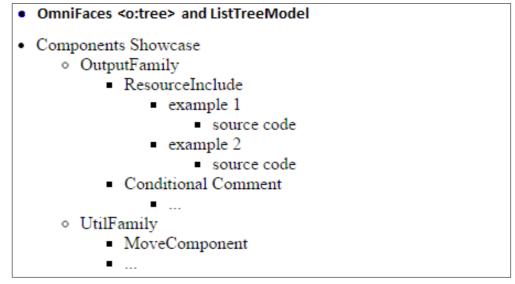


Figure 2. Simple tree output





```
<error-page>
    <error-code>500</error-code>
    <location>/WEB-INF/errorpages/500.xhtml
    </location>
</error-page>

or this:

<error-page>
    <exception-type>java.lang.Throwable
    </exception-type>
    <location>/WEB-INF/errorpages/500.xhtml
    </location>
</error-page>
</error-page>
```

Reference an image provided as an InputStream, byte[], or data URI. Most JSF novices know how to use the built-in <h:graphicImage> tag with the name and library attributes to load an image from the well-known /resources folder or from an external URL using the value or url attributes. (If not, check out this article.) But what should be done when the image is not present at the end of a URL, for example, when it comes from a database, a web service, or a WebSocket? In such cases, the image usually comes as a byte array, InputStream, BLOB, or other form. In fact, what about an image as a data URI?

OmniFaces provides the <u>GraphicImage</u> component, which is an elegant solution for loading an image that comes as a byte array or <u>InputStream</u>, and it is capable of extracting a data URI also. The content type will be guessed based on the content header (it supports the JPEG, PNG, GIF, ICO, SVG, BMP, and TIFF formats). In addition, OmniFaces supports a "last modified" time stamp, which improves browser caching. In addition, SVG images can be loaded via modes, including the ability to show only a part of an SVG image (via viewBox). There are many examples of displaying images provided from an external resource, such as a serv-

let, that can be handled elegantly by OmniFaces.

Deal with a ViewExpiredException. A common issue in JSF consists of dealing with a ViewExpiredException. When a user session expires (such as when a session times out, a logical view was removed from the logical views map, or a session was programmatically invalidated), a ViewExpiredException occurs.

OmniFaces provides a tag handler named <u>Enable</u>
<u>RestorableView</u>, which is capable of "swallowing" a
<u>ViewExpiredException</u> and restoring the view. I am here
talking about session restoring, not about session recovery,
which is a different challenge in JSF.

This tag handler can be used in a page by placing the <o:enableRestorableView/> tag in <f:metadata>. This allows us to instruct the view handler to re-create the entire view whenever the view has been expired. You simply write it like this:

```
<f:metadata>
<o:enableRestorableView/>
</f:metadata>
```

Conclusion

OmniFaces contains many other useful artifacts for JSF developers. In this article, you saw just a small part of what OmniFaces is capable of. As development continues, OmniFaces is becoming a more mature and powerful JSF utility library, thanks to the work of Bauke Scholtz and Arjan Tijms, who scan the JSF community for newly reported issues to solve. In this way, OmniFaces is always up to date on current issues. As expected, OmniFaces is open source under the Apache License 2.0. Enjoy! </article>

Anghel Leonard is a senior Java developer with many years of experience in Java SE, Java EE, and frameworks. He has written numerous articles about Java as well as several books, including *Mastering OmniFaces* (Glasnevin Publishing).









ABHISHEK GUPTA

JAX-RS 2.0: Using All the Goodness

A new client API, filters, interceptors, and other useful REST features

he JAX-RS framework is a set of APIs for building applications and services that implement RESTful principles. It provides a simple yet powerful programming model in which plain old Java objects (POJOs) can be decorated with annotations to expose their services and some of the business logic over HTTP, the ubiquitous web protocol.

JAX-RS was standardized as a JCP specification JSR 311 with a 1.0 release in 2008 and also was made part of the Java EE 6 specification JSR 316. JAX-RS 2.0 (JSR 339) came into being in 2013 and is an integral part of the Java EE 7 specification (JSR 342). Work on the next edition, JAX-RS 2.1 (JSR 370), is in progress and is slated to be released as part of the Java EE 8 platform.

Some of the widely used JAX-RS implementations are <u>Jersey</u> (which is also the reference implementation), <u>RESTEasy</u>, and <u>Apache CXF</u>. All these implementations provide additional useful features beyond basic JAX-RS specification compliance and support.

In this article, I focus on the latest features in JAX-RS 2.0. These include a new client API, filters, interceptors, new ways to handle exceptions, and advances in asynchronous programming, among many other useful additions.

A Brand-New Client API

Prior to the addition of a full-fledged client API, developers had to resort to using third-party implementations or interacting with the HTTPUrlConnection API in the JDK to interact

with HTTP-oriented (REST) services. The client API (part of the javax.ws.rs.client package) is fairly compact, lean, and fluent. Let's look at some of its classes and interfaces.

A ClientBuilder enables you to initiate the invocation process by providing an entry point via its overloaded newClient methods and the build method. The Client helps create a WebTarget instance with the help of overloaded target methods. WebTarget is a representation of the URI endpoint for HTTP request invocation. It helps configure various attributes such as query, matrix, and path parameters and exposes overloaded request methods to obtain an instance of Invocation.Builder, which is responsible for further building the HTTP request and configuring attributes such as headers, cookies, and cache

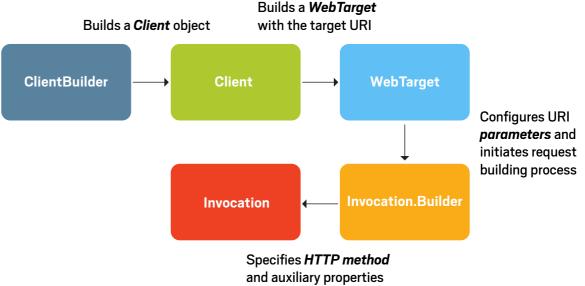


Figure 1. The JAX-RS 2.x client







control along with content negotiation parameters such as media types, language, and encoding. Finally, the Client helps obtain an instance of the Invocation object by using one of its buildXXX methods. An instance of Invocation encapsulates an HTTP request and allows synchronous and asynchronous request submission via overloaded versions of the invoke and submit methods, respectively. Figure 1 illustrates this sequence.

Let's look at an example:

```
Client client = ClientBuilder.newClient();
WebTarget webTarget =
    client.target("http://service.com/user")
        .queryParam("card", "4275391126915480");
Invocation.Builder builder =
    webTarget.request("text/plain");
Invocation invocation =
    builder.header("testheader","testvalue")
        .buildGet();
Response response = invocation.invoke();
```

Now, let's dissect this code snippet to gain a better understanding of what's going on.

An instance of Client, obtained via the Client
Builder class, is used to specify the target URI. An
instance of WebTarget is created as a result; it is further used to specify the expected response/media type
(equivalent to an Accept HTTP header) and associated
URI (path and query) parameters. This creates an instance
of Invocation.Builder, which further builds a complete
HTTP GET request. The Invocation instance is used to
deliver the request to the server.

The Configurable interface. The Client, ClientBuilder, WebTarget, and Invocation objects implement the javax.ws.rs.core.Configurable interface. This allows them to define custom JAX-RS components such as filters, interceptors, entity providers (message readers and writers),

and so on. This is made possible using the overloaded versions of the register method, which I'll discuss shortly. Note that this API is applicable to server-side JAX-RS components (filters, interceptors, and so on) as well.

Filters

Filters and interceptors (discussed later) are another bigticket feature in JAX-RS 2.0. They provide aspect-oriented programming (AOP)-like capabilities within JAX-RS applications and enable developers to implement cross-cutting, application-specific concerns, which ideally should not be

Server-side response filters are similar to their counterpart (request filters) in terms of their utility (the read and mutate aspects of the response, for example, HTTP headers) and the programming model (for example, being executed as a chain in a user-defined or default order).

sprinkled all over the business logic. These capabilities include authentication, authorization, request/response validation, logging, and so forth. The AOP-based programming model involves interposing in methods of JAX-RS resource classes and dealing with (or mutating) components of HTTP request/response headers, request URIs, and the invoked HTTP method (GET, POST, and the others).

Server-side request filters. Server-side request filters act on incoming HTTP requests from the clients. They are called prior to JAX-RS resource method invocation, thus providing an opportunity to act on certain characteristics of the incoming HTTP requests.

To implement a server-side request filter, implement the javax.ws.rs.container.ContainerRequestFilter interface, which is an extension provided by JAX-RS. An instance of the javax.ws.rs.container.Container







32

RequestContext interface is seamlessly injected by the container into the filter method of ContainerRequest Filter. It is a mutable object (on purpose) and exposes methods to access and modify HTTP request components.

A JAX-RS request-processing pipeline involves dispatching an HTTP request to the appropriate Java method in the resource classes based on a matching algorithm implemented by the JAX-RS provider. Server-side request filters take this into account and are divided into two categories: prematching and postmatching.

Prematching filters are executed before the incoming HTTP request is mapped/dispatched to a Java method. A prematching filter can be configured easily by using the javax.ws.rs.container.PreMatching annotation on the class. Note that the javax.ws.rs.ext.Provider annotation on the implementation class, as shown next, is required for the JAX-RS runtime to recognize the filter as a JAX-RS filter.

}

Postmatching filters are executed by the JAX-RS container only after the completion of the method dispatch/matching process. Unlike prematching filters, these filters do not need an explicit annotation. As a result, filter classes without the <code>@PreMatching</code> annotation are assumed to be postmatching by default. The following code shows an example in use:

A JAX-RS application can have multiple filters (forming a chain-like structure), which are executed in a developer-defined order (more on this shortly) or in a default container-driven order. However, it is possible to break the chain of processing by throwing an exception from the filter implementation logic or by calling the abortWith method. In either case, the subsequent filters in the chain are not invoked.

Server-side response filters. A server-side response filter is invoked by the runtime after a response is generated by the JAX-RS resource method before dispatching the response to the caller/client. Response filters are similar to their counterpart (request filters) in terms of their utility (the read and







33

mutate aspects of the response—for example, HTTP headers) and the programming model (for example, being executed as a chain in a developer-defined or default order).

Setting up a server-side response filter is as simple as providing an implementation for the javax.ws.rs.container.ContainerResponseFilter interface. Just like its server-side request counterpart, the injection of ContainerResponseContext into the filter method of the ContainerResponseFilter interface is taken care of by the JAX-RS runtime. Server-side response filters also need to be annotated with the javax.ws.rs.ext.Provider annotation in order for the JAX-RS runtime to recognize them. An example of this follows:

Client-side filters, predictably, include request and response filters, which I'll quickly summarize.

Client-side request filters. These are invoked after the HTTP request has been constructed but prior to the request being dispatched to the server. This type of filter provides an opportunity to mutate the properties of the HTTP request (such as headers and cookies). To implement a client-side request filter, implement the extension interface provided by javax.ws.rs.client.ClientRequestFilter.

Client-side response filters. Client-side response filters are invoked after the HTTP response has been received from the server but before it is dispatched to the caller/client for processing. Like client-side request filters, these filters provide an opportunity to mutate the properties of the HTTP response. To use a client-side response filter, implement the extension interface provided by javax.ws.rs.client.ClientResponseFilter, as shown next.

Interceptors

Interceptors are similar to filters in the sense that they are also used to mutate HTTP requests and responses, but the major difference lies in the fact that interceptors are used primarily to manipulate HTTP message payloads. They are divided into two categories, javax.ws.rs.ext.ReaderInterceptor and javax.ws.rs.ext.WriterInterceptor, which are used to process HTTP requests and responses, respectively. Note that, unlike filters, the same set of interceptors is applicable on the server side and the client side.

Server-side interceptors need to be annotated with @Provider for the JAX-RS runtime to detect them. Client-side interceptors need to be registered with ClientBuilder, Client, or WebTarget class or interface (using the register method).







ReaderInterceptor. This type of interceptor is a contract (extension interface) provided by the JAX-RS API. On the server side, reader interceptors act on HTTP payloads sent by the client, while on the client side, reader interceptors are supposed to act on (read and mutate) the request payload prior to it being sent to the server side.

WriterInterceptor. This type of interceptor on the server side acts on HTTP payloads produced by the resource methods, while on the client side, a writer interceptor is supposed to act on (read and mutate) the payload sent by the server prior to it being dispatched to the caller. This code shows the server-side variant in operation:

```
public interface WriterInterceptor{
    public void aroundWriteFrom(
        WriterInterceptorContext writerCtx)
    throws IOException, WebApplicationException;
}
```

JAX-RS interceptors are invoked in a chain-like fashion (similar to filters) and are triggered only when entity providers(javax.ws.rs .MessageBodyReader and javax.ws.rs .MessageBodyWriter) are required to convert HTTP messages to and from their Java object representations. Both ReaderInterceptor and WriterInterceptor wrap around MessageBodyReader and MessageBodyWriter, respectively. Hence, they are executed in the same call stack.

JAX-RS 2.0 includes a brand-new API for asynchronous processing that includes server-side as well as client-side counterparts. Being asynchronous inherently implies request processing on a different thread than the thread that initiated the request.

Binding Strategies for Filters and Interceptors

JAX-RS defines multiple ways that filters and interceptors can be bound to their target components. For server-side filters, there is the global default behavior plus named binding and dynamic binding. Let's look at these briefly.

Global default behavior. By default, JAX-RS filters and interceptors are bound to all the methods of the resource classes in an application. That is, both request (prematching and postmatching) and response filters will be invoked whenever any resource method is invoked in response to an HTTP request by the client. This convention can be overridden by using named binding or dynamic binding.

Named binding. To handle filter and interceptor scoping in a fine-grained manner (on a per-resource class or per-method basis), you can leverage the <code>@NameBinding</code> annotation. Doing this involves several steps:

• **Step 1.** Define a custom annotation with the @NameBinding annotation.

• **Step 2.** Apply the custom annotation to the filter or interceptor.

• **Step 3.** Apply the same annotation to the required resource





class or method. If it is applied to a class, the filter or interceptor will be bound to all the resource methods of the class.

Dynamic binding. JAX-RS provides the DynamicFeature interface to help bind filters and interceptors dynamically at runtime. (They can be used in tandem with the more static way of binding, which is made possible by using the @NameBinding annotation just discussed.)

```
public interface DynamicFeature {
public void configure(
    ResourceInfo resInfo,
    FeatureContext ctx);
}
```

The injected instance of the ResourceInfo interface helps you choose the resource method in a dynamic fashion by exposing various methods, and the FeatureContext interface allows you to register the filter or interceptor once the resource method has been selected. The following code shows an example:

Binding and Registering Client-Side Filters and Interceptors

We saw that a (class-level) @Provider annotation is required for server-side filters. On the client side, filters and interceptors are registered using ClientBuilder, Client, or WebTarget interfaces, all of which implement the javax.ws.rs.core.Configurable interface, which provides multiple overloaded versions of the register method.

Ordering Filters and Interceptors

The @Priority annotation can be used to define the order of execution of filters and interceptors. It accepts a numerical value that is interpreted differently by request and response filters and interceptors.

The request filters (ContainerRequestFilter and ClientRequestFilter) and interceptors (Reader Interceptor and WriterInterceptor) are executed in ascending order of their priorities. In other words, the lesser-valued @Priority annotated providers are executed first.

The response filters (ContainerResponseFilter and ClientResponseFilter) are executed in exactly the opposite (that is, descending) order. In the absence of a @Priority annotation, a default value is assumed. It is defined by the USER constant in javax.ws.rs.Priorities (this equals 5,000). Priorities of client-side components can be set using the register method of interfaces implementing Configurable.







Support for Asynchronous Processing

JAX-RS 2.0 includes a brand-new API for asynchronous processing that includes server-side as well as client-side counterparts. Being asynchronous inherently implies request processing on a different thread than the thread that initiated the request. From a client perspective, asynchronous processing prevents blocking the request thread, because no time is spent waiting for a response from the server. Similarly, asynchronous processing on the server side involves the suspension of the original request thread and the initiation of request processing on a different thread, thereby freeing up the original server-side thread to accept other incoming requests. The end result of asynchronous processing (if it is leveraged correctly) is scalability, responsiveness, and greater throughput.

Server-side asynchronous processing. The server-side asynchronous programming model is mainly powered by the following API abstractions: the @javax.ws.rs.container. Suspended annotation and the javax.ws.rs.container. AsyncResponse interface.

An instance of AsyncResponse can be transparently injected as a method parameter (of a JAX-RS resource class) by annotating it with @Suspended. This instance serves as a callback object to interact with the caller/client and perform operations such as response delivery (postrequest processing completion), request cancellation, error propagation, and so forth.

```
@GET
@Path("{id}")
public void search(
    @Suspended AsyncResponse asyncResp,
    @PathParam("id") String id){
    //launching search in a new thread
    new Thread(){
        public void run(){
        //execute search op and resume
```

```
UserInfo user = //obtain via search...
    asyncResp.resume(user);
}
}.start();
}
```

To propagate responses and exceptions, use the overloaded versions of the resume method on the AsyncResponse interface to return responses or exceptions back to the client.

To handle request processing timeouts, configure a timeout after which an HTTP 503 error response is returned to the client. This can be achieved by configuring a timeout threshold. By default, a timeout triggers an HTTP 503 error response. However, this can be overridden by registering a javax.ws.rs.container.TimeoutHandler implementation.

javax.ws.rs.container.CompletionCallback represents a callback interface whose implementation can be registered in order to execute business logic after the completion of the request. It is possible to terminate request processing by using overloaded versions of the cancel method. This results in an HTTP 503 error response being sent to the client. Client-side asynchronous processing. The JAX-RS API allows asynchronous request invocations using the Invocation and the AsyncInvoker interfaces. The Invocation interface handles asynchronous submission via overloaded versions of the submit method, while the AsyncInvoker interface supports asynchronous invocation with dedicated methods (get(), post(), put(), and so on) for standard HTTP actions: GET, PUT, POST, DELETE, HEAD, TRACE, and OPTIONS.

Once registered, an implementation of the Invocation Callback will automatically be executed once the asynchronous request is processed. It provides the ability to account for scenarios that cause exceptions or that execute successfully. In the event that a Future object is obtained and no callback has been registered, manually poll it in order







to interact with the response. isDone, get, and cancel are some of the methods that can be invoked. The following illustrates response handling via a callback and a Future object:

```
Invocation.Builder builder1 = //...
AsyncInvoker invoker = builder1.async();
//obtain a Future
Future<Response> future1 = invoker.get();
//create another builder
Invocation.Builder builder2 = ...
Invocation invocation = builder2.buildGet();
//provide a callback
Future<Response> future2 =
  invocation.submit(
    new InvocationCallback<Customer>(){
      public void completed(Customer cust){
        System.out.println(
          "Customer ID:" + cust.getID());
      public void failed(Throwable t){
        System.out.println(
          "Unable to fetch Cust details: " +
          t.getMessage());
});
```

Enhanced Exception Handling

Before looking at the other new features in JAX-RS 2.0, let's review some of the existing exception-handling capabilities provided by the framework for robust exception handling. The javax.ws.rs.core.Response object allows you to wrap an HTTP error state and return it to the caller. A javax.ws.rs.WebApplicationException (unchecked exception) can be used as a bridge between the native busi-

ness- or domain-specific exceptions and an equivalent HTTP error response. A javax.ws.rs.ext.ExceptionMapper represents a contract (interface) for a provider that maps Java exceptions to javax.ws.rs.core.Response objects. It can be thought of as an enhanced version of a javax.ws.rs.WebApplicationException and helps embrace the "don't repeat yourself" (DRY) principle for exception handling by allowing you to define flexible mappings between business-logic exceptions and the desired HTTP response, for example:

```
public class BookNotFoundMapper implements
   ExceptionMapper<BookNotFoundException>{
     @Override
     Response toResponse(
         BookNotFoundException bnfe){
         return Response.status(404).build();
     }
}
```

JAX-RS 2.0 has been supplemented with unchecked

EXCEPTION	HTTP ERROR CODE
BadRequestException	400
ForbiddenException	403
InternalServerErrorException	500
NotAcceptableException	406
NotAllowedException	405
NotAuthorizedException	401
NotFoundException	404
NotSupportedException	415
ServiceUnavailableException	503

Table 1. How exception classes map to HTTP error codes





exceptions that inherit from <code>javax.ws.rs.Web</code>
ApplicationException. This design relieves you from having to build and throw a <code>WebApplicationException</code> with explicit HTTP error information. The new exceptions are intuitively named, and each of them maps to a specific HTTP error scenario (by default). This means that throwing any of these exceptions from your resource classes will result in a predefined (as per mapping) HTTP error response being sent to the client; that is, the client would receive an HTTP 403 in the event that you throw a <code>NotAuthorizedException</code> (detailed mapping in <code>Table 1</code>). The exceptions' behavior can also be modified at runtime by using the <code>ExceptionMapper</code> to return a different <code>Response</code> object. <code>Figure 2</code> shows the new exception hierarchy.

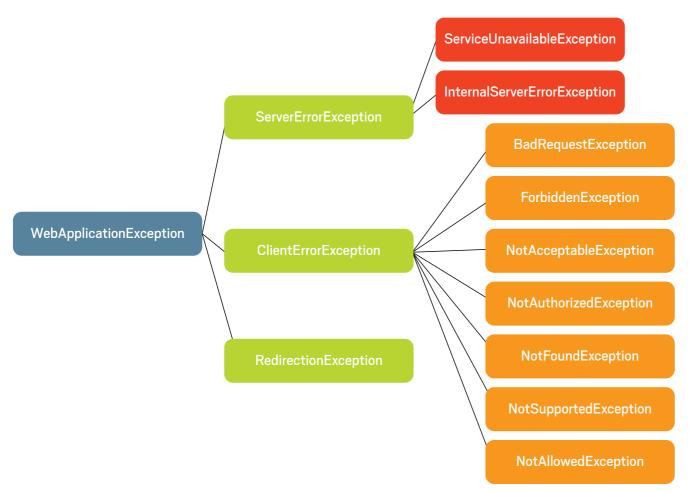


Figure 2. The hierarchy of JAX-RS 2.0 exceptions

A mapping of these exception classes to the corresponding HTTP error codes is shown in **Table 1**.

Other Notable Enhancements

Apart from the big-ticket features, several other useful enhancements were introduced as part of JAX-RS 2.0, which you should use if they fit your application.

@BeanParam. This annotation can be used to inject a custom POJO or bean whose instances can be annotated with various Param annotations such as @HeaderParam, @CookieParam, @PathParam, @QueryParam, and so on. As shown below, it provides a convenient way to capture HTTP URI parameters with the help of simple POJOs instead of injecting individual components from the HTTP request into JAX-RS resources:

```
public class CustomerSearchRequest{
    @QueryParam("id")
    private String userid;

@HeaderParam("Accept")
    private String accept;

@CookieParam("lastAccessed")
    private Date lastAccessed;
    //getters to fetch the values
}
```

The JAX-RS runtime injects an instance of the @javax .ws.rs.BeanParam annotated method argument (or an instance variable).

@ConstrainedTo. There are certain provider components in JAX-RS that are applicable to the server side and the client side. Interceptors are a good example where both ReaderInterceptor and WriterInterceptor can be applied on both the server side and the client side. The @javax.ws.rs.ConstrainedTo annotation can applied on a provider class in order to (explicitly) restrict its contextual







//java proposals of interest /

usage to either the client side or the server side.

ParamConverter. The ParamConverter interface comes in handy when handling custom objects or POJOs that do not satisfy the prerequisites for automatic injection (having a public constructor that takes a String argument or having a static valueOf method that takes a String argument and returns an instance of the same type as the POJO). An implementation of this interface can be used to provide custom logic for conversion of String parameters obtained via the Param annotations (@QueryParam, @PathParam, and so on) into custom POJOs.

Conclusion

The JAX-RS framework serves as a solid platform for building REST-oriented applications by providing a much needed abstraction on top of HTTP. The 2.0 release is a major overhaul with significant additions to the API. If you're not using the full range of options JAX-RS offers you, chances are good you're carrying around handwritten code that would benefit from being replaced by these capabilities. </article>

Abhishek Gupta is a Java EE developer, an architect, and a consultant specializing in the Oracle Identity Governance (middleware) product stack. His other areas of interest include scalable architecture, distributed caching technologies, and design patterns.

learn more

<u>Latest JAX-RS 2.0 specification</u> <u>Jersey</u>

Roy Fielding's seminal paper on REST

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 254: Compact Strings

JEP 254 proposes changing the internal representation of strings inside the JVM. As most readers surely know, strings are stored using UTF-16, which uses two bytes per character. This proposal suggests using a more compact, one-byte-per-character representation internally: "Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most String objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal char arrays of such String objects is going unused," says the JEP proposal.

Changing to the more compact form would not affect existing code or any APIs; it would be a purely internal change inside the JVM and not visible to programmers.

Interestingly, the information on the JEP's web page reveals that a string compression feature was tested in Java 6. It converted String.value to an Object that pointed either to an array of 7-bit characters or an array of regular Java characters. That feature, though, was removed subsequently.







Long Polling with Asynchronous Servlets

HENRY NAFTULIN

The reliable workhorse of client/server communications is the easy-to-use fallback when other methods don't work.

Not too long ago, a large gap existed between desktop applications and web applications. If you go back 10 years, desktop applications were noticeably more responsive, had far better user interfaces, and overall presented a much better user experience. A major reason why web-based applications were far behind in user experience was because they were not as responsive to server state changes as their desktop counterparts, and users had to update a whole page to get new data on the screen. Back then, in order to compete with desktop apps, companies that produced web applications used a variety of different tactics. They used applets, Adobe Flash applications, Comet, or similar frameworks that were popular at the time, while others used raw Ajax calls. Even with all these technologies, web apps couldn't compete with desktop software for usability.

Nowadays, web applications are expected to be interactive, have a stellar-looking UI, and do almost as much as their desktop counterparts. With rich front-end UI frameworks such as Bootstrap and processing frameworks such as jQuery and Angular JS, it is much easier to build applications that are easy on the eyes and can quickly respond to user input notifying a server about changes made in the UI.

But what about propagating the changes from the server to the web clients? After all, given today's cutting-edge technology, we are used to and expect almost instantaneous UI feedback to anything that is happening to the information on the server. In this article I explore one such solution—long polling—and briefly take a look at its alternatives.

Long Polling and Its Alternatives

For a long time, web applications were developed in an *n*-tiered (usually three-tiered) architecture, in which the client initiated requests for data from the server. There was no way for the server to push data to the client without the client requesting an update. Yet in many applications, a change on the server needs to be propagated to the clients in a timely manner. To work around this limitation, a technique known as *long polling* can be used.

Long polling is a technique used to push updates to a web client. After a client requests new information, the server holds the request until new data becomes available. Once the server receives new data, it completes the client's response by sending the data to the client. At this point the server can either keep the connection open while sending further updates to the client as they happen, or close the connection right away. In the latter case, once the client receives the response from the server and the connection is closed, the client immediately sends another request for an update and the operation is repeated.







Nowadays, web applications are expected to be interactive, have a stellar-looking UI, and do almost as much as their desktop counterparts.

Long polling has several variants. The simplest version is achieved with the client polling the server in a specified interval. Upon receipt of such a request, the server responds immediately. It either sends the latest data to the client or informs the client that it has no new data to share. This simple polling technique works for some applications for which updates are infrequent and displaying stale data is not problematic. Another version of long polling, the one I discuss in this article, is used when the server holds onto the client request and does not respond until the data that the client has asked for is available.

The principal alternatives to long polling are WebSocket and Server–Sent Events (SSE). WebSocket is currently a widely used alternative. It is a standard protocol providing a full–duplex communication channel over a single TCP connection. One of the biggest advantages of WebSocket is that it can drastically reduce the amount of network traffic between the server and the clients. The disadvantages are that not all browsers support it, and the older network routers that are optimized for the HTTP protocol can cache or close your WebSocket connection. This is why some connection libraries upgrade to the WebSocket protocol if it's available and fall back to the long-polling solution if it's not. [An article about using WebSocket for a project similar to the one in this article can be found on page 47 of this issue. —Ed.]

SSE is another standard technology in which a browser receives automatic updates from a server via an HTTP connection. It is designed to be efficient while pushing data to clients. The protocol has automatic reconnection and other useful features built in, such as automatic tracking of the last–seen message. Still, not all browsers support it, and in such cases the web solutions are usually programmed to fall back to using long–polling methods.

As such, long polling remains a significant player and a reliable solution. It is useful to know how it works and how to implement it efficiently.

Long Polling Before Servlet 3.0

Before the Servlet 3.0 specification, there were two serverthreading models: thread per connection and thread per request. In the thread-per-connection model, a thread is associated with every TCP/IP connection, and the server can scale to a very high number of requests per second when these requests come from the same set of clients. However, this model exhibits scalability issues. The reason is that for most websites, the users initiate an action and then the connection stays mostly idle while users read the pages and decide what to do next. Hence, the threads tied to a connection are sitting idle. To improve scalability, the web servers can use a threadper-request model. In this model, after servicing the request, the thread can be reused to service a request from a different client. This model allows for much greater scaling of the user base at the minor expense of increased time servicing each request. This expense is due to the thread scheduling that takes place. All major web servers today use the thread-perrequest model.

Yet, with long polling, the difference in scalability between thread per connection and thread per request is blurred. This is because each request must wait for data to be available on the server before generating the response. Waiting in the servlet is inefficient, because the server thread that could be used to service another request is blocked. Prior to Servlet 3.0, this resulted in poor scalability as users were added to the application.

Servlet 3.0 Changes Long Polling

Servlet 3.0 introduced *asynchronous processing*, which is a way for servers to process requests, particularly those in which a long-running operation such as a remote call or an application event must happen before a response can be generated. Prior to the Servlet 3.0 specification, a servlet would block a response thread and hold on to other limited resources while it waited for a response to be generated. With asynchronous







processing enabled, we can use a different thread to process the request and dispatch a response to the user. This change frees the original servlet request thread right away, so that it is returned to the servlet container to service other requests from other users.

Servlet 3.0 makes coding asynchronous processing simple by introducing AsyncContext—an execution context for asynchronous operations. AsyncContext encapsulates servlet request and response and lets you work with them outside of the original servlet processing thread. To use AsyncContext, you first need to indicate your intent to the servlet container—for example, by adding asyncSupported=true to your servlet annotation. Then, to put a request into asynchronous mode, you need to create an instance of AsyncContext inside the servlet's service call method. This can be done with

```
AsyncContext asyncContext =
    request.startAsync(request, response);
```

At this point, the asynchronous request processing can be delegated to a different thread or put in a queue for processing later. Because servlet request and response objects are encapsulated with AsyncContext, they are available to any thread and are not tied to the original servlet thread. This allows the original servlet thread to finish the service call without waiting for the asynchronous response to be completed and to be available to service requests from other clients.

A Simple Example of Asynchronous Long Polling

Let's look at the simple example of a web chat application that demonstrates the advantages of asynchronous servlet processing. In this application, the user types a username and a message and then clicks Send (see Figure 1). This message will then appear on all the browsers polling the chat URL.

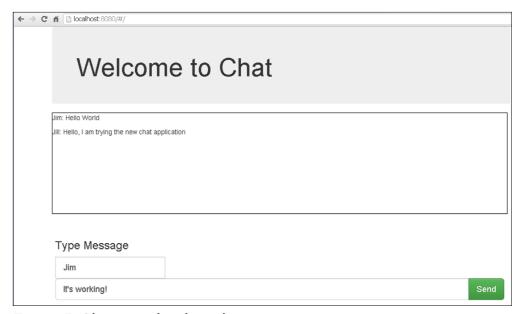


Figure 1. Chat app that broadcasts incoming messages

The code in **Listing 1** shows the key communication portions of the chat app.

Listing 1.

```
@WebServlet(urlPatterns="/chatApi",
            asyncSupported=true,
            loadOnStartup = 1)
public class AsyncChatServletApi
             extends HttpServlet {
 private static final int
     NUM WORKER THREADS = 10;
 Lock lock = new ReentrantLock();
 LinkedList<AsyncContext> asyncContexts =
                  new LinkedList<>();
 private AsyncListener listener =
                  new ChatAsyncListener();
 @Override
 public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                  throws IOException {
     AsyncContext asyncContext =
```







43

```
request.startAsync(request, response);
    asyncContext.setTimeout(-1);
    asyncContext.addListener(listener);
    try {
        lock.lock();
        asyncContexts.addFirst(asyncContext);
    } finally {
      lock.unlock();
@Override
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
                 throws IOException {
    final String message = getMessage(request);
    Collection<AsyncContext> localCopy;
    try {
        lock.lock();
        localCopy= asyncContexts;
        asyncContexts = new LinkedList<>();
    } finally {
        lock.unlock();
    for(AsyncContext asyncContext : localCopy) {
        Runnable runnable =
        new RunnableWriter(asyncContext, message);
        executor.execute(runnable);
    localCopy.clear();
private String getMessage(...) { ... }
class RunnableWriter implements Runnable {
  private final AsyncContext asyncContext;
  private final String message;
```

```
public RunnableWriter(..) { ... }

@Override
public void run() {
    try(PrintWriter writer =
        asyncContext.getResponse().getWriter()) {
        writer.println(message);
        writer.flush();
        asyncContext.complete();
    } catch(Exception e) {
        ...
    }
    }
}
```

This code example implements an asynchronous servlet that is tied to the /chatApi path. A collection is created in which the multiple asynchronous contexts are stored. It contains servlet contexts that are waiting to receive a chat message once it is available. So instead of tying up servlet processing threads to wait until someone sends a chat message, the AsyncContext is created and stored right away to be processed at a later time, when a new chat message becomes available.

Two REST APIs are used: doGet and doPost, which are shown here as the first two methods in the AsyncChat ServletApi class. doGet subscribes a user to receive a new chat message, while doPost pushes a new chat message to all waiting clients. Specifically, when the user hits a get URL, an asynchronous context is created by calling

```
AsyncContext asyncContext =
    request.startAsync(request, response);
```

Following that, the asynchronous context is saved in a list of contexts. This is a list representing clients that are waiting to get the next chat message. At this point, the server thread







44

is done with processing this request, and it can be reused for processing other requests.

When a user issues a doPost request posting a chat message, the message is retrieved. This message is then written to each of the asynchronous contexts, and the processing is completed by calling

asyncContext.complete();

Because the servlet can be called by multiple threads, the processing thread must be made thread-safe. In particular, three scenarios should be addressed:

- Two post requests are processed simultaneously.
- A get request is received while a post request is being processed.
- Two get requests are processed simultaneously.

 When two posts occur simultaneously or a get and a post occur simultaneously, all clients waiting for a message should receive it (that is, messages are not dropped). This can be achieved by synchronizing the copying context into a local collection, and reinitializing the member context collection so that it can accumulate new requests. Choosing to synchronize the addition of an element to a collection in the doGet method and synchronizing creation of a local copy of a request collection in doPost takes care of both scenarios.

In the case of two get requests being processed simultaneously, both contexts need to be stored for future processing. This can be achieved by having a thread-safe collection. Alternatively, it is possible to synchronize the block of code that adds an element, as shown in **Listing 1**.

Now, let's examine the timeouts. If we were to set a non-infinite timeout on the asynchronous context, and the timeout is reached, clients receive the response "Server returned HTTP response code: 500 for URL: http://localhost:8080/chat-Api." If the timeout is not set explicitly, the request inherits the default timeout from the server settings. If the timeout

is set as 0 or negative, as shown in Listing 1, the server never times out the request (although a web client can). To handle timeouts on the server, you can extend AsyncListener and implement custom code in the onTimeout callback event. For instance, changing the server response to send status code 408 (the HTTP code for request timeout) along with the message "Request timeout, no chat messages so far, please try again" can be achieved by using the code in Listing 2.

Listing 2.

```
public class ChatAsyncListener
                 implements AsyncListener {
  @Override
  public void onComplete(AsyncEvent event) { }
  @Override
 public void onTimeout(AsyncEvent event) {
    AsyncContext asyncContext
            = event.getAsyncContext();
   HttpServletResponse response
            = (HttpServletResponse)
                 asyncContext.getResponse();
    response.sendError(
          HttpServletResponse.SC_REQUEST_TIMEOUT,
          "Request timeout, no chat messages so far,"
          + " please try again.");
    asyncContext.complete();
  @Override
  public void onError(AsyncEvent event) { }
  @Override
 public void onStartAsync(AsyncEvent event) { }
```







Performance

In a usual chat application, all clients are listening for new messages, while only a few of the clients are writing chat messages. Hence, for this application, it makes sense to concentrate on what happens if we start increasing the number of clients. To test performance, I started with 64 clients and steadily increased the number of clients to 2,048, while measuring the time it takes them to get a new chat message. Table 1 presents the results of this test, and Figure 2 shows them graphically.

It is not essential for our analysis to know the specific time it takes to perform a get operation, but it is important to see the trends as the number of clients increases. The average response time does not change much with the number of clients until we hit around 1,024 clients, and stays right around 500 milliseconds—the time it takes for our producer thread to produce a new message. As the number of clients increases beyond 1,024, we see some degradation in performance mainly because it takes time to process posting of the chat message to the clients.

Another interesting point this test demonstrates is that the average time it takes to consume a message is slightly below the rate at which messages are produced. It indicates that

		NUMBER OF CLIENTS					
		64	128	256	512	1,024	2,048
RUN #1	RESPONSE	483.28	492.63	521.00	485.24	504.31	616.75
	STDEV	1.39	3.31	26.01	10.55	15.57	37.35
RUN #2	RESPONSE	499.80	490.04	494.45	501.02	546.78	575.85
	STDEV	0.98	3.15	3.87	19.84	26.85	35.03
RUN #3	RESPONSE	469.90	495.57	489.43	485.57	544.37	659.75
	STDEV	0.69	1.53	6.79	7.87	36.46	29.13

Table 1. Results of three runs sending messages to multiple clients (time in ms)

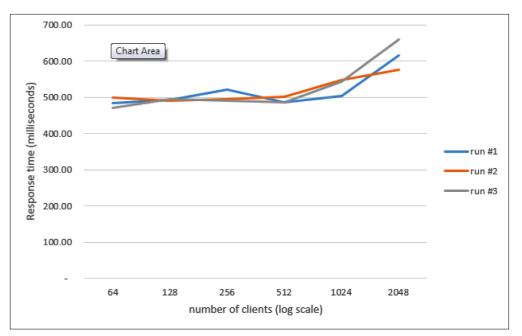


Figure 2. Response time as the number of clients increases

processing a message and resubscribing does take time, and there is a slight chance for the client to miss a new message.

To enhance this setup to work in an environment where missing messages is not a viable option, a client can pass a token to the server indicating the last message it has seen. If the client has already received the latest message available, the server puts the thread on asynchronous wait for a new message. Otherwise, the server responds with all messages produced since the last message the client received.

Conclusion

In this article, I've explained what long polling is and demonstrated how simple it is to use. With the universal support it enjoys, long polling can be used anytime a long-lived connection between client and server is needed. </article>

Henry Naftulin has been designing Java EE distributed systems for 15 years. He is currently leading development of proprietary short-side preorder services for one of the largest financial companies in the United States.







46



DANNY COWARD

Pushing Data in Both Directions with WebSockets

Using WebSockets' long-lasting connections to build a simple chat app

n Part 1 of this article, I introduced WebSockets. I observed that the base WebSocket protocol gives us two native formats to work with: text and binary. This works well for very basic applications that exchange only simple information between client and server. For example, in the Clock application in that article, the only data that is exchanged during the WebSocket messaging interaction is the formatted time string broadcast from the server endpoint and the stop string sent by the client to end the updates. But as soon as an application has anything more complicated to send or receive over a WebSocket connection, it will find itself seeking a structure into which to put the information. As Java developers, we are used to dealing with application data in the form of objects: either from classes from the standard Java APIs or from Java classes that we create ourselves. This means that if you stick with the lowest-level messaging facilities of the Java WebSocket API and want to program using objects that are not strings or byte arrays for your messages, you need to write code that converts your objects into either strings or byte arrays and vice versa. Let's see how that's done.

Fortunately, the Java WebSocket API gives us some support in this task of encoding objects to WebSocket messages and decoding WebSocket messages into objects.

First, the Java WebSocket API attempts to convert incoming messages into any Java primitive type (or its class equiv-

alent) that you request. This means you can declare a message handling method of the form

```
@OnMessage
public void handleCounter(int newvalue) {...}

or

@OnMessage
public void handleBoolean(Boolean b) {...}
```

and the Java WebSocket implementation attempts to convert any incoming message into the Java primitive parameter type you declare.

Equivalently, the RemoteEndpoint.Basic methods for sending include a general-purpose

```
public void sendObject(Object message)
    throws IOException, EncodeException
```

method, into which you can pass any Java primitive or its class equivalent, and the Java WebSocket implementation converts the value into the string equivalent for you.

This only gets you so far. Often, you want higher-level, highly structured objects to represent the messages in your application. In order to handle custom objects in your message handling methods, you must provide, along with the







endpoint, a WebSocket Decoder implementation, which the runtime uses to convert the incoming message into an instance of the custom object type. To handle custom objects in your send methods, you must provide a Web-Socket Encoder implementation that the runtime will use to convert instances of the custom object into a native WebSocket message. We can summarize this kind of scheme in Figure 1.

Figure 1 shows endpoints exchanging strings with the client at the top, and other endpoints using an encoder and a decoder for converting Foo objects into WebSocket text messages and vice versa.

There is a family of <code>javax.websocket.Decoder</code> and <code>javax.websocket.Encoder</code> interfaces in the Java Web-Socket API to choose from, depending on what kind of conversion you wish to make. For example, to implement a <code>Decoder</code> that converts text messages into instances of a custom developer class called <code>Foo</code>, you would implement the interface <code>Decoder.Text<T></code> using <code>Foo</code> as the generic type, which would require you to implement this method:

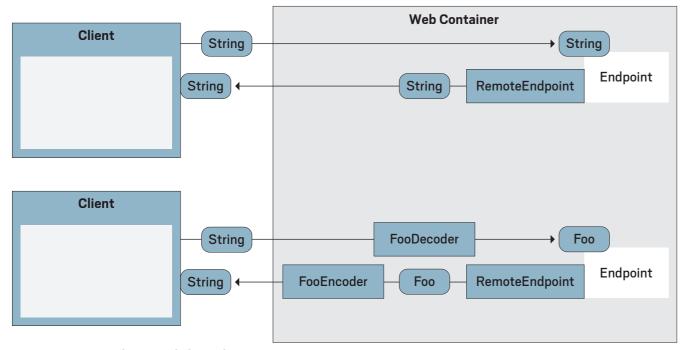


Figure 1. Encoders and decoders

public void sendObject(Object message) throwsIOException, EncodeException

This is the workhorse method of the decoder and would be called each time a new text message came in to produce an instance of the Foo class. The runtime would then be able to pass this class into the message handling method of your endpoint.

There are sibling **Decoder** classes for decoding binary WebSocket messages and WebSocket messages that arrive in the form of a blocking I/O stream (which is also supported).

To implement an Encoder that converts instances of a custom developer class Foo into a WebSocket text message, you would implement the Encoder. Text<T> interface using Foo as the generic type. This would require you to implement this method:

public String encode(Foo foo) throws EncodeException

This does the work of converting Foo instances into strings, which are needed by the Java WebSocket runtime if you call the RemoteEndpoint's sendObject() method (discussed previously), passing in an instance of the class Foo. Like Decoders, there are Encoder siblings for converting custom objects into binary messages and for writing custom objects to blocking I/O streams in order to send the message.

This scheme is easy to wire into an endpoint if you want to use it, as we saw in the definitions for <code>@ClientEndpoint</code> and <code>@ServerEndpoint</code>. You can simply list the decoder and encoder implementations you want the endpoint to use in the <code>decoders()</code> and <code>encoders()</code> attributes, respectively.

If you configure your own encoders or decoders for the Java primitive types, they will override the runtime's default encoders or decoders for those types, just as you would expect.







Message Processing Modes

So far, we have only discussed sending and receiving Web-Socket messages one entire message at a time. Although many applications retain this simple model for message processing because they define only small messages in their application protocol, some applications will deal with large WebSocket messages, perhaps transmitting photographs or large documents. The Java WebSocket API provides several processing modes suited to handling larger messages gracefully and efficiently.

Receiving large messages. The Java WebSocket API has two additional modes for receiving messages that are suited to situations when you know the message will be large. The first mode exposes the endpoint to a blocking I/O API that the endpoint can use to consume the message, either java .io.Reader for text messages or java.io.InputStream for binary messages. To use this mode, instead of using either a String or ByteBuffer parameter in your message handling method, you would use a Reader or InputStream. For example,

The second mode allows for a kind of elementary chinking API, where the WebSocket message is passed to the message handler method in small pieces together with a boolean flag telling you whether there are more pieces yet to come in order to complete the message. Of course, the message pieces arrive in order, and there is no interleaving of other messages. To

use this mode, the message handler method adds a boolean parameter. For example,

In this mode, the size of the chunks depends on several factors relating to the peer that sends the message and the configuration of the Java WebSocket runtime. All you know is that you will receive the whole message in a number of pieces.

Modes for sending messages. As you might expect, given the symmetry of the WebSocket protocol, there are equivalent modes for sending messages in the Java WebSocket API suited to large message sizes. In addition to sending a message all in one piece, as we have seen so far, you can send messages to a blocking I/O stream, either java.io.Writer or java.io.OutputStream depending on whether the message is text or binary. These are, of course, additional methods on the RemoteEndpoint.Basic interface that you obtain from the Session object:

```
public Writer getSendWriter() throws IOException
and
public OutputStream getSendStream()
```

throws IOException

The second mode is the chunking mode, but in reverse, for sending rather than receiving. Again, an endpoint can send messages in this mode by calling either of the following methods of RemoteEndpoint.Basic,







```
public void sendText(
    String partialTextMessage, boolean isLast)
    throws IOException

or

public void sentBinary(
    ByteBuffer partialBinaryMessage,
    boolean isLast)
    throws IOException
```

depending on the type of message you wish to send. Asynchronous sending of messages. Receipt of WebSocket messages is always asynchronous. An endpoint typically has no idea when messages are going to arrive; they just appear whenever the peer chooses. Now, all the methods of the RemoteEndpoint.Basic interface for sending messages (most of which we have seen) are synchronous sends. In simple terms, what this means is that the send() method calls always block until the message has been transmitted. This is fine for small messages, but if the message is large, a WebSocket may well have better things to do than wait for it to send, such as messaging someone else, repainting a user interface, or focusing more resources on processing incoming messages. For such endpoints, the RemoteEndpoint.Async, obtainable from the Session object, as is the RemoteEndpoint.Basic, contains send() methods that take a whole message as a parameter (in various forms). It returns immediately, and before the message passed in is actually sent. For example, when sending a large text message, you might want to use the

```
public void sendText(
     String textMessage, SendHandler handler)
```

method. The method returns immediately, and the Send Handler that you pass in to this method receives a call-

back when the message is actually transmitted. In this way, you know the message was sent, but you don't have to wait around until it does so. Or you may want to check in periodically on the progress of an asynchronous message send. For example, you might choose the method

in which case the method returns immediately and before the message is transmitted. You can query the Future object you obtain in return for the status of the message sent, and even cancel transmission if you change your mind.

There are binary message equivalents for these methods, as you might expect.

Before we leave the topic of sending messages in the Java WebSocket API, it's worth pointing out that the WebSocket protocol has no built-in notion of delivery guarantee. In other words, when you send a message, you don't know for sure whether it was received by the client. If you receive an error in your error handler methods, that's usually a sure sign that the message was not delivered properly. But if there is no error, the message still may not have been properly delivered. It is possible to build interactions yourself in Java WebSockets, wherein for important messages you have the peer send you an acknowledgement of receipt. But, unlike other messaging protocols, such as JMS, there is no inherent guarantee of delivery.

Path Mapping

In the Clock example, there was one endpoint and it was mapped to a single relative URI in the URI space of the web application. The client that connected to this endpoint did so by choosing a URL that was exactly that of the URI to the web application, plus the URI of the endpoint. This is an example of exact path mapping in the Java WebSocket API.







50

In general, an endpoint is accessible at

```
<ws or wss>://<hostname>:<port>/
<web-app-context-path>/<websocket-path>?
<query-string>
```

where <websocket-path> is the value attribute of the @ServerEndpoint annotation and query-string is an optional query string.

When the <websocket-path> is a URI, as it is in the ClockServer endpoint, the only request URI that will connect to the endpoint is the one that matches it exactly.

The Java WebSocket API also allows server endpoints to be mapped to level 1 URI templates. URI templates are a fancy way of saying that one or more segments of the URI can be substituted with variables. For example,

```
/airlines/{service-class}
```

is a URI template with a single variable called service-class.

The Java WebSocket API allows incoming request URIs to match an endpoint using a URI template path mapping if and only if the request URI is a valid expansion of the URI template. For example,

```
/airlines/coach
/airlines/first
/airlines/business
```

are all valid expansions of the URI template

```
/airlines/{service-class}
```

with variable service-class equal to coach, first, and business, respectively.

URI templates can be very useful in a WebSocket application, because the template variable values are available within the endpoint that matches the request URI. In any of the lifecycle methods of a server endpoint, you can add as many String parameters annotated with the <code>@PathParam</code> annotation to obtain the value of the variable path segments in the match. Continuing this example, suppose we had the server endpoint shown in Listing 1.

```
Listing 1. A Booking notifier endpoint
```

```
@ServerEndpoint("/air1ines/{service-class}")
public class MyBookingNotifier {

@OnOpen
public void initializeUpdates(Session session,
     @PathParam("service-class") String sClass) {
     if {"first".equals(sClass)) {
         // open champagne
     } else if ("business".equals(sClass)) {
         // heated nuts
     } else {
         // don't bang your head on our aircraft
     }
   }
}
```

This would yield different levels of service, depending on which request URI a client connects with.

Accessing path information at runtime. An endpoint has full access to all of its path information at runtime. First, it can always obtain the path under which the WebSocket implementation has published it. Using the ServerEndpointConfig.getPath() call for the endpoint holds this information, which you can easily access wherever you can get hold of the ServerEndpointConfig instance, such as we see in Listing 2.







Listing 2. An endpoint accessing its own path mapping

```
@ServerEndpoint("/travel/hotels/{stars}")
public class HotelBookingService {
  public void handleConnection{
    Session s, EndpointConfig config) {
      String myPath =
            ((ServerEndpointConfig) config).getPath();
            // myPath is "/travel/hotels/{stars}"
            ...
  }
}
```

This approach will work equally well for exact URImapped endpoints.

The second piece of information you may wish to access at runtime from within an endpoint is the URI with which the client to your endpoint connected. This information is available in a variety of forms, as we shall see later, but the workhorse method that contains all the information is the

Session.getRequestURI()

method. This gives you the URI path relative to the web server root of the WebSocket implementation. Notice that this includes the context root of the web application that the WebSocket is part of. So, in our hotel booking example, if it is deployed in a web application with context root /customer/services and a client has connected to the HotelBookingService endpoint with the URI

```
ws://fun.org/customer/services/
    travel/hotels/3
```

then the request URI the endpoint receives by calling getRequestURI() is

```
/customer/services/travel/hotels/3
```

Two more methods on the Session object parse out further information from this request URI when the request URI includes a query string. So let's take a look at query strings. Query strings and request parameters. As we saw earlier, following the URI path to a WebSocket endpoint is the optional query string

```
<ws or wss>://<host:name>:<port:>/
    <web-app-context-path>/<websocket-path>?
    <query-string>
```

Query strings in URIs originally became popular in common gateway interface (CGI) applications. The path portion of a URI locates the CGI program (often /cgi-bin), and the query string appended after the URI path supplies a list of parameters to the CGI program to qualify the request. The query string is also commonly used when posting data using an HTML form. For example, in a web application, in the HTML code

clicking the Submit button produces an HTTP request to the URI

```
/form-processor?user=Jared
```

relative to the page holding the HTML code and where the input field contains the text <code>Jared</code>. Depending on the nature of the web resource located at the URI path <code>/form-processor</code>, the query string <code>user=Jared</code> can be used to determine what kind of response should be made.







For example, if the resource at form-processor is a Java servlet, the Java servlet can retrieve the query string from the HttpServletRequest using the getQueryString() API call.

In a similar spirit, query strings can be used in the URIs that are employed to connect to WebSocket endpoints created with the Java WebSocket API. The Java WebSocket API does not use a query string sent as part of the request URI of an opening handshake request to determine the endpoint to which it might match. In other words, whether or not a request URI contains a query string makes no difference to whether it matches a server endpoint's published path. Additionally, query strings are ignored in paths used to publish endpoints.

Just as CGI programs did and other kinds of web components do, WebSocket endpoints can use the query string to further configure a connection that a client is making. Because the WebSocket implementation essentially ignores the value of the query string on an incoming request, any logic that uses the value of the query string is purely inside the WebSocket component. The main methods that you can use to retrieve the value of the query string are all on the Session object

```
public String getQueryString()
```

which returns the whole query string (everything after the? character) and

```
public Map<String,List<String>>
   getRequest:ParameterMap()
```

which gives you a data structure with all the request parameters parsed from the query string. You'll notice that the values of the map are lists of strings; this is because a query string may have two parameters of the same name but different values. For example, you might connect to the HotelBookingService endpoint using the URI

```
ws://fun.org/customer/
    services/travel/hotels/4?
    showpics=thumbnails&
    description=short
```

In this case, the query string is showpics=thumbnails& description=short, and to obtain the request parameters from the endpoint, you might do something like what's shown in Listing 3.

where the values of pictureType and textMode would be thumbnails and short, respectively.

You can also get the query string from the request URI. In the Java WebSocket API, the Session.getRequestURI call always includes both the URI path and the query string.

Deployment of Server Endpoints

Deployment of Java WebSocket endpoints on the Java EE web container follows the rule that easy things are easy. When you package a Java class that has been annotated with @ServerEndpoint into a WAR file, the Java WebSocket





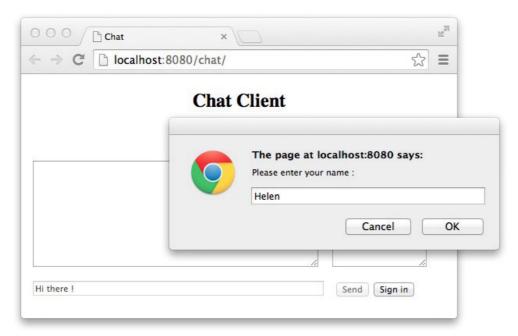


Figure 2. Logging in to chat

implementation scans the WAR file and finds all such classes and deploys them. This means there is nothing special you have to do in order to get your server endpoints deployed except package them in the WAR file. However, you may wish to more tightly control which of a collection of server endpoints gets deployed in a WAR file. In this case, you may provide an implementation of the Java WebSocket API interface javax.websocket.ServerApplicationConfig, which allows you to filter which of the endpoints get deployed.

The Chat Application

A good way to test a push technology is to build an application that has frequent, asynchronous updates to make to a number of interested clients. Such is the case with a Chat application. Let's take a look in some detail at how to apply what we have learned about the Java WebSocket API to build a simple chat application.

Figure 2 shows the main window of the Chat application, which prompts for a username when you sign in.

Several people can be chatting all at once, typing their messages in the text field at the bottom and clicking the Send

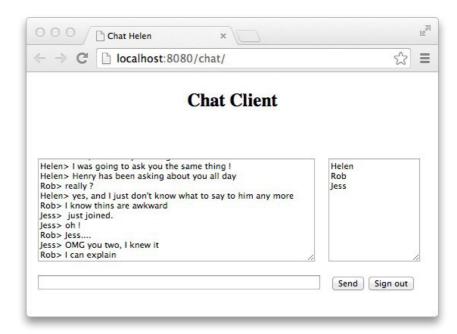


Figure 3. Chat in full flow

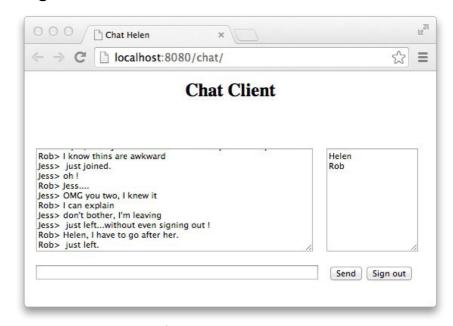


Figure 4. Leaving chat

button. You can see the active chatters on the right side and the shared transcript recording everyone's messages in the middle and left. In **Figure 3**, there is an uncomfortable triad of people chatting.

In Figure 4, we can see that one of the chatters left rather suddenly, and the other has left slightly more gracefully, leaving just one chatter in the room.







Before we look at the code in detail, let's get the big picture of how this application is built. The web page uses the JavaScript WebSocket client to send and receive all the chat messages. There is a single ChatServer Java WebSocket endpoint on the web server, which is handling all chat messages from multiple clients, keeping track of those clients that are actively chatting, maintaining the transcript, and broadcasting updates to all connected clients whenever someone enters or leaves the chat room and whenever any one of them sends a message to the group. The application uses custom objects with WebSocket Encoders and Decoders to model all the chat messages.

Let's look at the **ChatServer** endpoint in **Listing 4**. [Due to its length, the listing should be downloaded from this issue's download area. —*Ed.*]

There is a lot to notice in this code. First, notice that this is a server endpoint that is mapped to the relative URI /chat-server. The endpoint uses an encoder and a decoder class, ChatEncoder and ChatDecoder, respectively.

The best way to look at Java WebSocket endpoints for the first time is to look at the lifecycle methods: These, as you know, are the methods annotated by @OnOpen, @OnMessage, @OnError, and @OnClose. We can see by looking at the ChatServer class in this way that the first thing the ChatServer WebSocket does when a new client connects is to set up instance variables that reference the chat transcript, the session, and the EndpointConfig. Remember that there is a new instance of the endpoint for each client that connects. So each chatter in the chat room will have a unique chat server instance associated with it. There is always a single EndpointConfig per logical WebSocket endpoint, so the endpointConfig instance variable on each instance of the ChatServer points to the single shared instance of the EndpointConfig class. This instance is a singleton, and it holds a user map that can hold an arbitrary application state. Thus, it is a good place to hold global state in an application.

There is always a unique session object per client connection, so each ChatServer instance points to its own Session instance representing the client to which the instance is associated by following the code to the Transcript class, as shown in Listing 5.

Listing 5. The Transcript class

```
import java.util.ArrayList;
import java.util.List;
import javax.websocket.*;
public class Transcript {
  private List<String> messages =
    new ArrayList<>();
  private List<String> usernames =
    new ArrayList<>();
  private int maxLines;
  private static String
    TRANSCRIPT ATTRIBUTE NAME =
      "CHAT_TRANSCRIPT_AN";
  public static Transcript
    getTranscript(EndpointConfig ec) {
    if (!ec.getUserProperties().
        containsKey(TRANSCRIPT_ATTRIBUTE_NAME)) {
          ec.getUserProperties()
            .put(TRANSCRIPT_ATTRIBUTE_NAME,
                 new Transcript(20));
          return (Transcript) c.getUserProperties()
                   .get(TRANSCRIPT_ATTRIBUTE_NAME);
     }
     Transcript(int maxLines) {
         this.maxLines = maxLines;
     public String getLastUsername() {
         return usernames.get(usernames.size() -1);
```







```
public String getLastMessage() {
    return messages.get(messages.size() -1);
}

public void addEntry(
    string username, String message) {
    if (usernames.size() > maxLines) {
        usernames.remove(0);
        messages.remove(0);
    }

    usernames.add(username);
    messages.add(message);
}
```

We can see that there is a single transcript instance per EndpointConfig. In other words, there is a single Transcript instance and it is shared across all ChatServer instances. This is good because we need the transcript to show the group chat messages to all clients.

The most important method on the ChatServer is the message handling method, annotated with @OnMessage. You can see from its signature that it deals with ChatMessage objects rather than text or binary WebSocket messages, thanks to the ChatDecoder that it uses. The ChatDecoder it uses has already decoded the message into one of the subclasses of ChatMessage. In the interest of brevity, rather than listing all the ChatMessage subclasses, Table 1 is a sum-

CHATMESSAGE SUBCLASS	PURPOSE
ChatUpdateMessage	MESSAGE HOLDING A USERNAME AND A CHAT MESSAGE THAT USER SENT
NewUserMessage	MESSAGE HOLDING THE NAME OF A NEW USER SIGNING ON
UserListUpdateMessage	MESSAGE HOLDING A LIST OF THE NAMES OF THE CURRENT ACTIVE CHATTERS
UserSignoffMessage	MESSAGE HOLDING THE NAME OF A USER WHO HAS SIGNED OFF

Table 1. The ChatMessage subclasses

mary of the different ChatMessage subclasses and the function of each.

Now we can easily see that the ChatServer's message handling method, handleChatMessage(), which is called by the client every time a new chat-related action occurs, is designed to handle situations when a new user signs in, when a user posts a new chat message to the board, and when a user signs out.

Let's follow the code path when a ChatServer is notified that the user has posted a new chat message. This leads us from the handleChatMessage() method to the process ChatUpdate() method, which calls addMessage(), adding the new chat message to the shared transcript. Then it calls broadcastTranscriptUpdate(), as shown in Listing 6.

Listing 6. Broadcasting a new chat message

This method uses the very useful API call Session .getOpenSessions(), allowing one endpoint instance to gain a handle on all the open connections to the logical endpoint. In this case, the method uses that list of all the open







connections to broadcast the new chat message out to all the clients so that they can update their user interfaces with the latest chat message. Notice that the message that is sent is in the form of a ChatMessage—here, the ChatUpdateMessage. The ChatEncoder takes care of marshaling the

The encode() method is the meat of the class and turns the message instance into a string, ready for transmission back to the client.

ChatUpdateMessage

instance into a text message that is actually the one sent back to the client with the news of the new chat message contained within.

Because we did not look at the ChatDecoder when we were looking at incoming messages, let's pause to look at the ChatEncoder class, as shown in Listing 7.

Listing 7. The ChatEncoder class

```
import java.util.Iterator;
import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;
public class ChatEncoder implements
       Encoder.Text<ChatMessage> {
   public static final String SEPARATOR = ":";
    @Override
    public void init(EndpointConfig config) {}
    @Override
    public void destroy() {}
   @Override
    public String encode(ChatMessage cm)
           throws EncodeException {
      if (cm instanceof StructuredMessage) {
           String dataString = "";
```

You can see that the ChatEncoder class is required to implement the Encoder lifecycle methods init() and destroy(). Although this encoder does nothing with these callbacks from the container, other encoders may choose to initialize and destroy expensive resources in these lifecycle methods. The encode() method is the meat of the class and takes the message instance and turns it into a string, ready for transmission back to the client.

Returning now to the ChatServer class, we see from the handleChatMessage() method that this endpoint has a graceful way of dealing with clients that sign off in the proper way: by sending a UserSignoffMessage prior to closing the connection. It also has a graceful way of dealing with clients who simply close the connection unilaterally, perhaps by closing the browser or navigating away from the page. The @OnClose annotated endChatChannel() method broadcasts a message to all connected clients informing them







when someone leaves the chatroom without saying goodbye. Looking back at the screenshots, we can now see the difference between the manner in which Jess and Rob left the room.

Conclusion

In this two-part article, we have learned how to create a Java WebSocket endpoint. We have explored the basic concepts of the WebSocket protocol and what kinds of situations demand their true server push nature. We have looked at the lifecycle of a Java WebSocket endpoint, examined the main classes of the Java WebSocket API, and looked at encoding and decoding techniques, including the variety of messaging modes supported in the Java WebSocket API. We looked at how server endpoints are mapped to the URI space of a web application and how client requests are matched to endpoints therein. We concluded with a look at a Chat application that exercises many of the features of the Java WebSocket API. Armed with this knowledge, now you can easily build applications with long-lived connections. </article>

This article was adapted from the book Java EE 7: The Big Picture with kind permission from the publisher, Oracle Press.

Danny Coward is the principal software engineer at Liquid Robotics. Earlier, he was a member of the Java development team at Oracle (and previously at Sun Microsystems), where he worked extensively on WebSockets, among other responsibilities.

learn more

Oracle's Java WebSockets tutorial

Long polling, a WebSocket alternative









//path to java 9 /



BEN EVANS

An Early Look at Java 9 Modules

Preparing for modularity, a significant change that is the cornerstone of the next major release of Java

ntil now—that is, up through Java 8—the language's model for controlling access to code has been relatively straightforward:

- Classes are arranged into packages. Packages are globally visible and open for extension (with the exception of packages that start with *java* or *javax*). Packages are designed for the logical organization of code and nothing more.
- The unit of delivery for code is the JAR file. Regular (unprotected) packages can span multiple JAR files.
- Classes and methods can restrict access. For example, they can enforce that they are accessible only to instances of the same type or to other code in the same package. This is expressed by the familiar Java keywords for access control. This design has the advantage of being simple to understand and to reason about. However, it does lead to several issues related to access control. Of these, the most important problem is this: non-JDK libraries are powerless to prevent client code from creating additional classes in the libraries' packages. Creating such classes gives access to all of the protected and package-access classes (and methods) that are defined in the package.

This is sometimes referred to as the "shotgun privacy" problem, after this famous observation about the Perl programming language by its designer, Larry Wall: "Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun."

For Java, however, shotgun privacy represents an important issue. As the language and environment currently stand, there's

no way to control access across an entire package. Another way of saying this is that Java libraries want to be able to define a public API and know with certainty that clients of that API can't subvert it or directly couple to the package internals.

Many enterprise applications would very much like to enforce a tight level of access control, where the only accesses are through a known and enforced API. However, this level of control of APIs is not completely possible in Java 8 and below. This was one of the principal reasons that Oracle (and previously Sun) wanted to develop a set of technologies capable of implementing this level of safety. As became clear, though, work on this problem began to allow the possibility of solving other problems.

The resulting core goals of the upcoming JDK 9 modules system have become the following over time:

- Extend access control so public APIs can be fully enforced by introducing a concept of "sealing" access within a deployment unit.
- Facilitate modular deployment of code at a scale larger than a single package.
- Modularize the JDK itself so that the platform can improve JVM startup times, reduce resource consumption, remove packages that shouldn't be in the core, and remove or hide internal classes to reduce the attack surface of the platform. Oracle has sponsored a project within OpenJDK, called Project Jigsaw, whose mission is to deliver modularity for the Java environment. As the release date for JDK 9 starts to approach, code from Jigsaw has begun to be migrated into the mainline OpenJDK repositories. This means that at the

PHOTOGRAPH BY JOHN BLYTHE







time of this writing (late 2015), there are different binaries at various levels of completeness that support modules.

- The first Java 9 early access (EA) builds from Oracle that contain modules <u>are available</u>. However, the module functionality that is contained in these binaries is limited.
- To supplement the mainline JDK 9 builds and provide even earlier access to modular functionality, the Jigsaw project is also making binaries available.

For the rest of this article, the examples and discussion center on the Jigsaw binaries, because they are more complete than the JDK 9 mainline.

The Jigsaw Pieces

Project Jigsaw identified several subgoals in the course of a roadmap leading to full modularity. These were defined within the Java Enhancement Process (JEP) and included the following:

- Modularize the layout of the source code in the JDK (JEP 201).
- Modularize the structure of the binary runtime images (JEP 220).
- Disentangle the complex implementation dependencies between JDK packages.

This last goal was one of the most labor-intensive, but it was also necessary in order to make modules as independent from each other as possible. In turn, this goal allows the maximum possible flexibility for modules to be loaded and linked separately.

As an example of why this is needed, consider the <code>java.util.Properties</code> package, which contains a requirement to parse XML. This requirement greatly increased the footprint for any code that depended on Properties, because it pulled in Java's full XML libraries. To fix this, a small XML parser based on the original reference implementation (JSR 280) was added to remove the heavier dependencies.

Another example is the RMI-IIOP transport, which was separated out from the management module, so that the

remote management module (which uses JMX) no longer requires CORBA to be present.

One other minor change is also worth mentioning because it is rather unusual in Java's history. A small number of methods in java.util.logging.LogManager were removed from the public API, because they caused undesirable linkages between some modules and would have introduced considerable bloat.

This is one of the very few times that methods have been actually removed from the public API of the JDK (after all, even the highly dangerous Thread.stop is still present, despite having been deprecated since Java 1.1). Removing methods is a very rare event and shows the importance of a good modularity solution to the Java team.

A First Look at Modules

Let's have a look at how things have changed in the Java 9 (Jigsaw) EA release:

```
$ cd $JAVA_HOME
$ pwd
/java9/Contents/Home
$ 1s -1h
total 42640
                         3.2K COPYRIGHT
             1 10
                   143
             1 10
                   143
                          40B LICENSE
             1 10
                   143
                         158B README.html
             1 10
                   143
                         174K THIRDPARTYLIC...txt
drwxr-xr-x 46 10
                   143
                         1.5K bin
                         238B conf
             7 10
drwxr-xr-x
drwxr-xr-x
             9 10
                   143
                         306B db
drwxr-xr-x 12 10
                   143
                         408B demo
drwxr-xr-x 11 10
                   143
                         374B include
             1 10
                   143
                         127K jrt-fs.jar
                         2.8K lib
drwxr-xr-x 84 10
                   143
                         549B release
             1 10
                   143
           15 10
                   143
                         510B sample
             1 10
                   143
                          21M src.zip
-rw-r--r--
```







//path to java 9 /

[In order to fit, some minor file information has been removed from this listing. —Ed.]

There are a few things to see here, but let's start with the big news—the <code>jre</code> directory is gone. There is no more <code>rt.jar</code> to contain the whole of the JRE. Instead, the JRE is stored in the new <code>jimage</code> format. This is a container format that manages classes and resources needed by the modularized runtime.

Rather than traditional zip-based compression, jimage files are indexed to support fast lookup of classes and resources. The content region of a jimage contains all classes and resources for the image and is managed in terms of locations.

Resources can be located in the index by using a module path as a search key. The path format is /<module-name>/ <package>/<base-name>.<extension>. Paths are versioned, and the current path is "9.0"—this reflects the current release of the JVM.

The main jimage file is found in lib/modules and is called bootmodules.jimage. However, in addition to the jimage files there are also .jmod files—Java modules. These files are being used as a test bed for other module features outside of the efforts in .jimage to speed up access. As it stands now, .jmod files are still implemented as zip files, but the format will almost certainly change in future. The fact that there are two packaging formats shows how much of a moving target Java 9 modules really are at present.

The java.base.jmod file contains the minimal viable collection of classes for any nontrivial Java program to run. It contains the following packages (or subsets of them):

```
java.io
java.lang
java.math
java.net
java.nio
java.security
java.text
java.time
```

```
java.util
javax.crypto
javax.net
javax.security
```

The rest of the module consists of the implementation classes to support the base module. Of course, the real intent of Project Jigsaw is that Java developers also make use of the mechanism to produce modular libraries and applications of their own. Let's take a quick look into this world by considering a simple "Hello World" for modules.

A simple Java module. To create a very simple Java module, we need to write two parts, the first of which is a simple core class such as this:

```
package com.jdk9ex;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello Modules!");
    }
}
```

In Java 8 and earlier, this code could be compiled and run without further ceremony. However, to run correctly in modular Java, this needs to be supplemented with a new piece of metadata: a module-info.java file.

The intention is that this file will become a fairly sophisticated metadata provider and handle versioning and other important features of modules, but the currently shipping version in Jigsaw EA is quite primitive and is only intended to allow keen developers to start investigating module technology.

The simplest module looks like this (assuming our module for JDK 9 examples is named com.jdk9ex):

```
module com.jdk9ex { }
```







//path to java 9 /

Eagle-eyed developers will note that this means that module is a new keyword in Java 9. For this to compile, the directory structure needs to be laid out according to the new modular source rules. For our module com.jdk9ex, this structure is the following:

Note that the module name is independent of the normal conventions for Java packages that live inside the module. To compile this modular code, we need to tell javac that we're running a module-aware build:

```
$ javac -d modules/com.jdk9ex \
    src/com.jdk9ex/module-info.java \
    src/com.jdk9ex/com/jdk9ex/*
```

This produces a set of modular class files, but not a jimage. To execute this code, we also need to explicitly inform the JVM that modules are needed:

```
$ java -modulepath modules -m \
        com.jdk9ex/com.jdk9ex.Main
Hello Modules!
```

The basic "Hello World" example is an established pattern for programmers, and experimenting with the modules system and seeing how it works is highly recommended for developers who want to fully understand the current state of modularity for Java 9.

What could go wrong? Inevitably, some aspects of any new technology will cause problems for early adopters, so let's look at a couple of basic failure cases. In particular, if you fail

to build or invoke modular Java correctly, you might see a new sort of exception:

```
$ java -modulepath modules -m com.jdk9ex.Main
Error occurred during initialization of VM
java.lang.module.ResolutionException:
  Module com.jdk9ex.Main not found at
java.lang.module.Resolver.fail
   (java.base@9.0/Resolver.java:880)at
java.lang.module.Resolver.resolve
   (java.base@9.0/Resolver.java:193)at
java.lang.module.Resolver.resolve
   (java.base@9.0/Resolver.java:173)at
java.lang.module.Configuration.resolve
   (java.base@9.0/Configuration.java:229)at
jdk.internal.module.ModuleBootstrap.boot
  (java.base@9.0/ModuleBootstrap.java:174)at
java.lang.System.initPhase2
  (java.base@9.0/System.java:1242)
```

[Some lines have been wrapped to fit. —*Ed.*] This single stack trace shows that we have the following new items:

- Packages, including java.lang.module
- Exceptions, including ResolutionException
- Locations for code, such as java.base@9.0/ModuleBootstrap.java
- Entry points for startup, such as java.lang.System::initPhase2

From this we can also see that the new modular paths used for jimages are now directly present in stack traces.

Alternatively, we might try to access an internal implementation class. The code shown below compiles and runs on Java 8:

```
import sun.invoke.util.BytecodeName;
public class InternalsCheck {
```





```
public static void main(String[] args) {
    String bName =
        BytecodeName
        .toBytecodeName("java.lang.Object");
        System.out.println(bName);
    }
}
```

As expected, it produces the internal name for java.lang.Object. However, with modules, access to the implementation class is restricted, and this code will no longer even compile:

```
$ javac -d modules/com.jdk9ex \
    src/com.jdk9ex/module-info.java \
    src/com.jdk9ex/com/jdk9ex/!nternalsCheck.java:8:
    error: package sun.invoke.util does not exist import sun.invoke.util.BytecodeName;

src/com.jdk9ex/com/jdk9ex/InternalsCheck.java:12:
    error: cannot find symbol
        String bName = BytecodeName.
toBytecodeName("java.lang.Object");

    symbol: variable BytecodeName location: class InternalsCheck
2 errors
```

These errors demonstrate the strong guarantees of API integrity that are one of the goals of the Java module system. As it stands, javac has been upgraded to prevent access to the implementation class <code>BytecodeName</code>, but IDEs and other tools within the Java environment do not yet acknowledge this new reality—one more sign that there's a long way to go before JDK 9 is ready for the general population of developers.

The good news is that all of the major IDE vendors are actively working on new versions to fully support Java 9. For example, Oracle has announced that NetBeans 9 will be focused on JDK 9 support, and Eclipse's JDT now has all active development happening on a JDK 9 branch.

A word about Unsafe. From the point of view of the Java platform developers, especially those involved in the OpenJDK

Many popular Java libraries break the rules and access the internals of the platform. This is a big problem, because it leaves the JDK developers in a very difficult situation.

project, "shotgun privacy" is a significant problem. While the java and javax packages are protected by the "prohibited package" SecurityException mechanism, this does not apply to the internal implementation classes.

Java libraries and frameworks are not supposed to couple directly to these implementation details. The attitude of Java's stewards has always been that to do so is dangerous, and no attempt will be made to accommodate developers who break the rules and link to internal details.

Despite repeated admonitions, many popular Java libraries break the rules and access the internals of the platform. This is a big problem, because it leaves the JDK developers in a very difficult situation.

One choice is to concede that some internals have become a de facto part of the API, support them, and accept that it is now very difficult to make changes that prevent those internals from being accessed. Another choice is to change internals and, thus, affect or break libraries and frameworks that millions of production applications rely on.

The most famous, widely used, and dangerous example of this direct coupling is the sun.misc.Unsafe class. This class represents a major pressure point in the adoption of







63

//path to java 9 /

a modules system. If access to Unsafe were simply shut off in JDK 9, this would prevent almost all Java applications from upgrading, because basically any nontrivial application will use a library that relies, directly or indirectly, on functionality in Unsafe.

However, indefinite access to Unsafe risks making it into an unofficial, de facto standard, and removes the possibility of evolvAlthough the earlyaccess previews do not contain all expected functionality, there is still plenty for you to explore in the work in progress.

ing internal platform mechanisms. The middle path agreed to between Oracle and the Java community is that temporary access will be provided for Java 9, but it is expected that access will be removed (and replaced with standardized, supported equivalent functionality) in Java 10.

This means that Java developers will have an easier path to upgrading to Java 9, because existing frameworks will continue to work. Library and framework developers will need to migrate to the new mechanisms in time for Java 10, which is not likely to ship until 2018.

What will developers need to do differently? Java developers will need to adapt to a new way of packaging and deploying code to get the most out of the new functionality. The good news, though, is that there's no need to start doing that straightaway. The traditional methods using JAR files will continue to work until such time as teams are ready to adopt the modular way of working.

One simple approach (which teams can use even before Java 9 arrives) is to make use of the Compact Profiles that shipped as part of Java 8. This was a first step toward modularity and defines three separate profiles that are true subsets of the JRE. The smallest profile, compact1, is around 11 MB, which saves a significant amount of space.

Applications that can run in a smaller profile than the full JRE will have a much easier time converting to a modular world. In many cases, it will be worth doing some up-front work now to make life easier later. Also, understanding and documenting application dependencies is a worthwhile exercise in the reduction of technical debt.

Conclusion

The advent of modules is a large change in the Java environment, probably one of the biggest ever. Although the early-access previews do not contain all expected functionality, there is still plenty for you to explore in the work in progress. The release of Java 9 is months away, and the feedback of developers who have tried Java 9 EA in realworld tests is very welcome. The OpenJDK Adoption group is the ideal place for interested developers to contribute their feedback and find out what they can do to help move Java 9 modules forward. </article>

Ben Evans helps to run the London Java Community and represents the user community as a voting member on Java's governing body—the JCP Executive Committee. He is the author of *The Well-Grounded Java Developer*, the new edition of *Java in a Nutshell*, and the forthcoming *Optimizing Java*.

learn more

JDK 9 outreach

Using Modularity (video presentations from Devoxx)

The JDK 9 schedule of release

How version strings will appear in JDK 9







SCOTT MCKINNEY

Gosu: A Modern, Down-to-Earth Language for the JVM

A low-ceremony language used in large enterprise apps offers an extraordinarily flexible, yet static, type system.

<u>sosu</u> is about 13 years old. It began as a simple scripting language here at Guidewire Software late in 2002. We needed a straightforward language for our customers to customize our products. Our basic criteria for the language included that it be Java interoperable, statically typed, familiar, and easy to use. Static typing facilitates our larger goal to provide a safe and pleasurable user experience via deterministic tooling such as rich parser feedback, code completion, refactoring, usage searching, and so forth. We would much rather have pulled a language from the shelf than build our own, but at the time, there weren't any scripting languages available that came close to satisfying our needs. As a fledgling startup, we weren't about to compromise at this critical stage; Guidewire's success would be won at least in part on our customers' ability to tailor our products using our language and tools. As a result, Gosu came to life.

Since then, we've come a long, long way. With more customers came demand for more Gosu features, which over time led Gosu to become a powerful, general-purpose programming language. Thus, unlike many programming languages, nearly every step of Gosu's development is a result of true pragmatism—its features were developed and refined for broad usability and have been vetted in the field by hundreds of medium to very large, international corpo-

rations employing thousands of developers writing mission-critical software with Gosu.

Hello

Getting down to business, you are probably more interested in Gosu's capabilities and syntax than in its history, so let's get to it. Here is Gosu's "Hello, World!":

```
print("Hello, World!")
```

That's it. Because Gosu is inherently scriptable, there is no need for boilerplate code, such as a Main class or main method. But if you're feeling nostalgic, this also works:

```
class RunMe {
  static function main(args: String[]) {
    print("Hello, World!")
  }
}
```

This class prints "Hello, World!" to the console. The same holds for the former, simpler example. So in addition to conventional class files, Gosu supports *program* source files that are free-form scripting files. You can code up anything in a program, including statements, functions, whole classes, whatever—in any arrangement you like.







Scripting, experimenting, and learning. Programs are effectively scripts. That is to say, Gosu is both a comprehensive, general-purpose, static language and a concise scripting language. As I'll demonstrate shortly, Gosu's powerful type inference capabilities, its syntax, and the enhancements to Java's runtime library implement this dual mission. One aspect that really qualifies Gosu as a scripting language is that it executes directly from source—compilation to class files is optional. When running from source, Gosu compiles and loads classes on demand. There is no performance penalty aside from the one-time price of compilation (to conventional bytecode).

In addition to scripting, programs also promote learning and experimentation. To leverage this ability, our full-featured IntelliJ IDE plugin also provides a Scratchpad editor for quickly cranking out and executing experimental code. You can even execute Scratchpad scripts against a running project in the debugger, without interrupting execution at a breakpoint.

Class format. Gosu is foremost an imperative, object-oriented language with classes, interfaces, structures, annotations, enums, programs, templates, functions, and enhancements all as first-class types. In addition, Gosu's type system is uniquely designed for customization—you can plug in your own first-class types. I'll cover more of the type system as I go, but first let's have a look at the basic format of a Gosu class:

As you can see, the main difference between Gosu and Java syntax is the Pascal-style declarations where the name precedes the type—for example, var name: Type. Choosing this style mostly came from the desire for more-consistent syntax where a variable's type can be inferred. We're also happy with the Pascal style for achieving consistency with generic declarations and, in our view, for providing better overall readability.

After a quick scan of the code, you'll notice some similarities with Java. The package statement is the same, and the uses statement is the equivalent of Java's import. Inheritance with extends and implements is the same. The var keyword declares fields; they're private by default. Classes, constructors, functions, and properties are public by default. Semicolons are extinct—that is, they're optional but discouraged. Whitespace is insignificant as well; newlines have no meaning, except to us humans.

Some Features

Gosu's feature set is much too broad to cover in a single article. Instead let's look at some features that will pique your interest.

Properties. Think of properties as a formalization of Java's getter and setter methods. Gosu also provides shortcuts with properties to eliminate boilerplate code. For instance, here's how you can declare a mutable property for a field:

```
var _names: List<String> as Names
```

A primary difference between a get/set pair of methods and a property is that you reference a property consistently as a single identifier, not via separate method calls:

```
property get Names(): List<String> {
  return names.copy() // returns a copy
}
```







Null-safety. Gosu provides several operators to save you from NullPointer exceptions and from writing boilerplate code. So instead of writing this:

```
if(location != null) {
   return location.Address
}
else {
   return null
}
you can write this:
```

return location?.Address

Named arguments and optional parameters. You can provide default values for constructor and function parameters to make the parameters optional to the caller:

Both calls to printList() are valid. The first one uses the default separator value. You can also name arguments when calling a function:

```
function configure(lux=false, awd=false, ac=false)
configure(:awd = true)
```

Intervals. Interval (or Range) expressions are Gosu's way of expressing a contiguous set of numbers, dates, names, or anything with a Comparable interface; they are denoted by the .. operator:

```
var range = "A".."M"
print(range.contains("Jones")) // true
print(range.contains("Smith")) // false
```

In addition, if the elements in the interval also represent a sequence, you can iterate over it. Here's a simple way to print the numbers 1 to 10:

```
for(i in 1..10) {
  print(i)
}
```

You can also set the step and unit, if applicable:

```
// A biweekly date sequence
var span = (date1..date2).step(2).unit(WEEKS)
```

And you can reverse the order and express open or closed endpoints to the range:

```
10..1 // reverse order
start..|end // start inclusive, end exclusive
```

Delegation. As an alternative to class inheritance, Gosu provides support for composition in the form of interface delegation:

Basically, this delegate statement hooks up the ClipboardPartMixin with MyWindow's implementation of ClipboardPart. No other work is necessary; all ClipboardPart methods automatically forward to the delegate. No more boilerplate compositional code!







67

Blocks (aka closures). In appearance, Gosu blocks are similar to Java lambdas:

```
var list = {"Pascal", "Java", "Gosu"}
var lengths = list.map(\ e -> e.length)
```

As you can see, Gosu blocks begin with a \ instead of Java's (); otherwise the syntax is nearly identical. Under the hood, however, things are quite different. For starters, Gosu blocks are true closures. Basically, this means that they can access and modify any variables from the local scope. But what really distinguishes a block from a lambda is that it has its own type, the Function type. There is no need for functional interfaces (or SAMs) as a go-between in Gosu, because the Function type is a first-class type:

```
var capitalize(String):String = \s -> s.capitalize()
or
var capitalize = \s: String -> s.capitalize()
```

You can use Function types anywhere. For instance, you can use Function types directly in a function's parameter list:

```
function visitThings(consumer(Thing))
```

And you can use them with generics:

```
var listeners: Set<block(Event)>
```

Function types are 100 percent interoperable with Java 8 functional interfaces. You can pass a block to a functional interface and pass a functional interface to a Function type.

Generics. Gosu generics use array-style covariance by default. For instance, the following assignment is legal in Gosu:

```
var list: List<Object> = new ArrayList<String>()
```

While this is technically unsound, it works the way most programmers think. So, unlike Java, Gosu does not have wild-cards or any type of use-site variance. In our experience, use-site variance is a significant source of confusion, which leads to misuse and circumvention of generics via casting and other techniques. Fundamentally, type safety exists to help programmers read and write better code; it shouldn't get in the way.

Alternatively, if you don't want array-style variance for your generic class, you can make it fully typesafe using C# style in/out variance modifiers in your type variable declarations. For instance, typesafe Consumer and Producer interfaces can be defined like so:

```
interface Consumer<in T> {
  function apply(t: T)
}
interface Producer<out T> {
  function get(): T
}
```

Generally, stuff returned *out* of a function is covariant while stuff passed *in* to a function is contravariant. Gosu verifies usage of <code>in/out</code> type variables against a more sophisticated version of these basic rules. Note that if you mix a default generic type with an <code>in/out</code> type, Gosu *infers* the variance of the default type and checks whether it's compatible:

```
// error: T is used in an 'out' position in Callable
interface Listener<in T> extends Callable<T> {
  function someEvent(t: T)
}
```

Here Gosu infers the variance of Callable's T as "out" and reports the incompatibility. Gosu infers variance of both Gosu and Java types, including usage of wildcards inside Java types.

Another useful aspect of Gosu generics: type arguments persist at runtime. In technical speak, Gosu employs type





reification as opposed to Java's type erasure. This means that the following code does what you expect:

```
var foo = new Foo<String>()
print(typeof foo) // prints Foo<String>
```

This cannot be done in Java, because of the type erasure. **Enhancements.** An *enhancement* is a Gosu type that defines new functions and properties for an existing type. Enhancements live in separate source files such as classes; there is no need to import them. Here is a simple String enhancement:

```
enhancement MyStringEnhancement : String {
  function print() {
    print(this)
  }
}
```

Now you can tell a String to print itself:

```
"Echo".print()
```

Gosu provides a host of useful enhancements for most of the Java runtime library—tons of refreshing features for Collections, Iterable, String, and so on.

Structures. Sometimes we need a way to make a type satisfy an interface. For example, Java's Rectangle, Point, and Component classes all have X, Y coordinates accessible by identically named methods:

```
class Rectangle {
  public double getX();
  public double getY();
  ...
}
class Point {
  public double getX();
```

```
public double getY();
...
}

class Component {
  public int getX();
  public int getY();
  ...
}
```

Yet they do not share a common interface for accessing the coordinate. Using nominal typing, there is no way to extract an interface for these classes: we can't modify them to implement a Coordinate interface. This is where structural typing comes in handy:

The Coordinate structure is just like an interface, but Gosu determines assignability from its methods and properties instead of forcing a class to implement it nominally. So in this example, because Point satisfies the *structure* of Coordinate, it is considered structurally assignable to it.







This applies equally to Rectangle and Component.

Note that structures can be generic, too. We could have defined Coordinate like this:

```
structure Coordinate<out T extends Number> {
  property get X(): T
  property get Y(): T
}
```

Gosu infers the type arguments from the implementing classes. For instance, Point structurally implements Coordinate<Double>.

Internally, a structure is nearly identical to an interface; in the JVM it *is* an interface. A structure can be implemented nominally and enjoy the benefits of nominal typing and vice versa; a structure can extend nominal interfaces.

Also it's worth mentioning that a type can be structurally assignable from enhancement methods. This powerful technique can be used to connect and unify disparate systems.

Type inference. By now you may have gathered that Gosu infers types rather aggressively. Basically, if Gosu can determine the type of something from its surroundings, you don't have to enter the type. You can still examine an inferred type using the IDE if needed, but in most cases types get in the way when they can otherwise be inferred. For instance, the var statement infers its type from its initializer if present:

```
var name = "Fred"
```

There is no need to specify the type when Gosu can infer it from "Fred". This is the simplest and probably the most useful form of type inference. There are many more situations in which Gosu infers types, most notably with blocks and generics. You can learn more about type inference and the other features discussed in this article at the documentation site for Gosu.

Templates. Templates are first-class types in Gosu. A Template type is a text file with Gosu expressions and scripts embedded in it, delimited by standard JSP-style syntax. This Gosu file, SampleTemplate.gst, defines a simple example:

```
<%@ params(names: List<String>) %>
All Names: <% for(name in names) { %>
    * ${name}
    <% } %>
```

You can render the template like this:

```
var str =
   SampleTemplate.renderToString({"Joe", "John"})
print(str)
```

You can also use template syntax directly in Gosu String literals:

```
var age = 39
print("Dog years: ${age * 7}")
```

Custom types. No one likes code generation. It tends to be messy and hard to maintain, is difficult to manage with source control, often results in code bloat, and is commonly a challenge to keep in sync. Gosu offers a powerful alternative via its Open Type System. The type system abstracts the notion of a type in terms of public interfaces in our API that you can plug in to provide first-class custom types. All core Gosu types are implemented this way. We also provide custom types for Dynamic types, XML Schema Definition (XSD) types, and Properties types.

IDE and Building

The JVM may be Gosu's runtime, but IntelliJ IDEA is Gosu's universe. Gosu's IntelliJ plugin is packed with professional-grade features such as syntax/error highlighting, very smart







//user groups /

code completion, usage searching, "go to feature" navigation, refactoring, code formatting, code inspections, comprehensive debugger, incremental compiler, Scratchpad editing, and more. Download it for free.

We've also recently released full support for building Gosu projects with Maven and Gradle. We use these tools ourselves to build and distribute Gosu and supporting projects.

Open Source

Gosu is a very active, fully open source language, found on <u>GitHub</u>. There you'll see that we're also currently working on some interesting projects to support an ecosystem for Gosu. Notable among them is <u>SparkGS</u>, a simple yet powerful web application framework for Gosu.

Conclusion

Guidewire's internal need for a straightforward static language to build and drive our applications ultimately led to the development of a powerful, pragmatic language. We feel that Gosu is uniquely positioned as both a powerful modern OOP language and as a concise scripting language. You can write small scripts and quickly execute them from source or, as several enterprise customers have done, write Gosu applications that exceed 1 million lines of code. As a fully open source project and now with support for open tooling such as IntelliJ Community Edition, Maven, and Gradle, we invite you to try Gosu in one of your projects or for a fun afternoon of scripting. Enjoy! </article>

Scott McKinney is a senior staff engineer at Guidewire Software. He is the creator of Gosu and is currently a principal developer and lead on the Gosu team.

MANCHESTER JAVA COMMUNITY



Formed in May 2013 by Alison McGreavy, the Manchester Java Community (MJC) is the nexus for the city's Java community. Currently led by McGreavy, Debbie Roycroft, and Roberto Nerici, the MJC has grown from four members at the first meeting to more

than 300 members. Meetings are held typically once a month at venues such as <u>MadLab</u>, and draw on average 30 people. The members come mainly from the Manchester, UK, area, but some from as far away as Liverpool and Leeds. Sessions range from lightning talks and hands-on instruction to guest presentations. Over the last two years, topics have included core Java, Java frameworks, tooling, JVM languages, and the JVM itself. These sessions have encouraged homegrown speakers and have attracted some great guest speakers such as the London Java Community's Ben Evans, the Ceylon team from Red Hat, and Heinz Kabutz. The MJC is a member of the JCP and will also be running a Java 9 Hack event in January 2016.

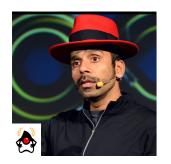
The MJC has great links with the wider programming community, both in Java events such as JavaOne and Devoxx and through the general tech community in Manchester. It is keen on developing more.

If you plan to be in the Manchester area and would like to participate in an MJC event, check out the <u>Manchester Java Community</u> on Meetup.com or on Twitter <u>@mcrjava</u>.









ARUN GUPTA

Using Multiple Docker Containers

Assemble a cluster of Docker containers and run a Java EE app—without a lot of housekeeping.

The <u>first article</u> in this two-part series explained the key concepts of Docker. It explained the first steps for getting started with Docker using Toolbox, how to build your first Docker image, how to package a Java application as a Docker image, and how to run the image as a container. Finally, a simple Java EE application was deployed on a WildFly application server.

This article explains how to deploy an application with multiple containers on multiple hosts. Specifically, I show how a Java EE application deployed on WildFly can perform CRUD operations using a database (here, Couchbase). Finally, I demonstrate Docker Compose for creating multicontainer applications, and I explain Docker Swarm for creating multihost application environments.

Docker Compose

Docker Compose is a tool for defining and running multicontainer Docker applications. It is installed along with Docker Toolbox, as explained in Part 1 of this series.

An application using Docker containers typically consists of multiple containers—for example, one for an application server, another for a database, possibly another for a messaging server, and so on. Each container typically starts with different configuration options such as port forwarding, environment variables, volume mapping, and so on.

With Docker Compose, there is no need to write shell scripts to start your containers. All containers are defined in a YAML configuration file using *services*. Each service consists of one or more containers. A Docker Compose script is used to start, stop, and restart the application plus all the services in that application and all the containers within each service. This approach is often helpful when starting a single container inside a larger service, because all the options can be specified beforehand in a file.

The command to start a Couchbase server that forwards port 8091 (administration port), port 8093 (query port), and some other ports in the container to similar ports on the host, as well as map a Couchbase log directory in the container to a directory on the host, would look like this:

[Enter the command above as a single line. —Ed.] Instead of remembering all the options for starting this container, you can define this command in a Docker Compose file. By default, this file is named docker-compose.yml in the current directory. The equivalent file for our command looks like this:

mycouchbase:

image: couchbase/server
volumes:
 - ~/couchbase:/opt/couchbase/var
ports:
 - 8091:8091







- 8092:8092 - 8093:8093 - 11210:11210

This file has one service, mycouchbase, which uses the prebuilt image couchbase/server defined by the image key. Volume mapping is done using the volumes key, and port forwarding is done using the ports key. The key name/values are analogous to their docker run counterpart, as shown on the previous page.

The complete syntax for the Compose file is explained at the <u>Docker site</u>.

Now, you can start the Couchbase database container by running the following command:

docker-compose up -d

This command starts the service mycouchbase (in the background because of the -d switch) and the Couchbase container in this service.

A list of common commands that can be used with Compose is shown in **Table 1**.

You can display the complete list of commands using the command docker-compose --help.

COMMAND	PURPOSE	
ир	CREATE AND START CONTAINERS	
stop	STOP SERVICES	
rm	REMOVE STOPPED CONTAINERS	
ps	LIST CONTAINERS	
logs	VIEW OUTPUT FROM CONTAINERS	
scale	SET THE NUMBER OF CONTAINERS FOR A SERVICE	

Table 1. Common commands used in Compose

Docker Compose becomes more relevant and interesting when multiple services need to be started for an application, which is generally the case. As an example, if your application consists of one WildFly service and one Couchbase service, then the Compose file would look like this:

```
mycouchbase:
 container name: "db"
 image: couchbase/server
 volumes:
    - ~/couchbase:/opt/couchbase/var
 ports:
    - 8091:8091
    - 8092:8092
    - 8093:8093
    - 11210:11210
mywildfly:
 image: arungupta/wildfly-admin
  environment:
    - COUCHBASE URI=db
 ports:
    - 8080:8080
    - 9990:9990
```

This application uses two services, mycouchbase and mywildfly. The mycouchbase service starts the Couchbase server. It has an additional attribute, container_name, that specifies a custom container name instead of a generated default name. The mywildfly service starts the WildFly application server. The default WildFly image exposes only port 8080 for the application to be accessed. The custom image, arungupta/wildfly-admin, starts WildFly such that the management interface is bound to all network interfaces. It also exposes port 9990, which can then be used for deploying applications. In addition, it also has an environment variable, COUCHBASE_URI, that is then used in applications deployed on WildFly to access the database. I'll provide more detail about this a little later (when the application environment is started).







The image key in the Compose file is employed to run a prebuilt image. Alternatively, a build key can be used to build an image and then run it. The value of this build key is the name of the directory in which the Dockerfile that defines the image definition exists.

For example, in our previous example, if the stock Couchbase image is used but a custom WildFly image needs to be built, then the updated service in docker-compose .yml is this:

```
mywildfly:
   image: .
   environment:
    - COUCHBASE_URI=db
   ports:
    - 8080:8080
    - 9990:9990
```

Notice that the only change here is the use of the **image** key. The directory location is relative to the Compose file.

Docker Compose provides two methods to share configuration: by providing multiple Compose files or by extending individual services using the extends keyword.

As mentioned earlier, the default filename for a Docker Compose file is docker-compose.yml. You can specify an alternative name using the -f switch, and multiple files using multiple -f switches. This option allows multiple Compose files to be created and targeted to different environments such as dev, test, continuous integration, and production. This option can also be used to deploy the same code but different databases in different environments. Overall, this ability reduces impedance mismatches between environments.

By default, Compose reads docker-compose.yml and an optional docker-compose.override.yml file. By convention, docker-compose.yml defines the base configuration and docker-compose.override.yml, if present, can override the base configuration.

Docker Swarm

Docker Swarm provides native clustering for Docker. It allows you to create and access a pool of Docker hosts using the full suite of Docker tools. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to scale transparently to multiple hosts. Figure 1 shows how Docker Swarm works in a typical setup.

The important elements include the following:

- Swarm manager. The swarm manager is a predefined Docker host. It is a single point for all administration. The swarm manager orchestrates and schedules containers for the entire cluster. You can configure the swarm manager for high availability.
- **Swarm nodes.** The containers are deployed on nodes that are additional Docker hosts. Each node runs a Docker Swarm agent that registers the referenced Docker daemon, monitors it, and updates the discovery back end with the node's status.
- Scheduler strategy. Different scheduler strategies—spread

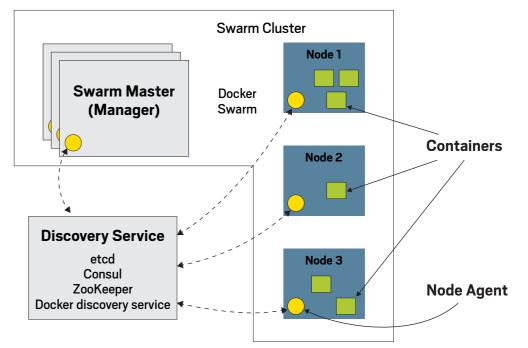


Figure 1. A typical setup using Docker Swarm





(the default), binpack, and random—can be applied to pick the best node to run your container. The default strategy optimizes the node for the least number of running containers.

- Node discovery service. The swarm manager talks to a hosted discovery service. This service maintains a list of IP addresses in the swarm cluster. The Docker hub hosts a discovery service that can be used during development. In production, the default discovery service is replaced by other services such as etcd, Consul, or ZooKeeper. You can even use a static file.
- Standard Docker API. Docker Swarm serves the standard Docker API. Thus, any tool that talks to a single Docker host will seamlessly scale to multiple hosts now. This means that if you were using shell scripts utilizing the Docker CLI to configure multiple Docker hosts, the same CLI can now talk to a Docker Swarm cluster.

Deploying a multicontainer Java application that uses Couchbase for a Docker Swarm cluster that spans multiple hosts requires the following steps:

- **1.** Create the discovery service.
- 2. Create the Docker Swarm cluster.
- **3.** Start the application environment.
- **4.** Configure the Couchbase server and load sample data.
- **5.** Deploy the application to the Docker Swarm cluster.

Docker Swarm takes care of the distribution of deployments across the nodes.

Figure 2 provides a conceptual overview of different components in the application:

- The cluster has two nodes: one master and another worker node.
- Docker Compose is used to start the application environment on the cluster.
- The discovery service is hosted on a separate machine outside the cluster.

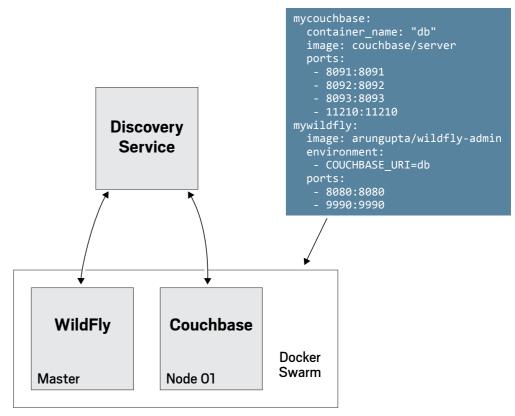


Figure 2. A simple service running on a Docker Swarm

Maven is used to configure Couchbase and deploy the application to WildFly. **Note**: This article uses WildFly and Couchbase. But you can easily use Tomcat, GlassFish, or any other application server. Similarly, you can use any other database server such as MySQL or JavaDB. Naturally, the image and the included application would need to be updated accordingly.

Let's dig into the details of how to set this up.

Create the Discovery Service

I will use <u>Consul</u> for the discovery service, although other similar tools, such as etcd and ZooKeeper, can be used instead.

First, create a new Docker machine that will host the discovery service:







docker-machine create -d=virtualbox consul-machine

Then connect to this machine:

```
eval $(docker-machine env consul-machine)
```

Run the Consul service using the following Compose file:

```
myconsul:
```

```
image: progrium/consul
restart: always
hostname: consul
ports:
    - 8500:8500
command: "-server -bootstrap"
```

Start this service using docker-compose up -d. You can verify the started container using the docker ps command described in the first part of this article.

Create the Docker Swarm Cluster

Docker Swarm is fully integrated with Docker Machine, and so it is the easiest way to get started. Create a swarm master and point to the Consul discovery service:

```
docker-machine create -d virtualbox
    --virtualbox-disk-size "5000" --swarm
    --swarm-master --swarm-discovery=
    "consul://$(docker-machine ip consul-machine)
    :8500" --engine-opt="cluster-store=consul://$(
    docker-machine ip consul-machine):8500"
    --engine-opt="cluster-advertise=eth1:2376"
    swarm-master
```

The following are the meanings of these switches:

- --swarm configures the machine with Docker Swarm.
- --swarm-master configures the created machine to be the swarm master.

- --swarm-discovery defines the address of the discovery service.
- --cluster-advertise advertises the machine on the network.
- --cluster-store designates a distributed key/value storage back end for the cluster.
- --virtualbox-disk-size sets the disk size for the created machine to 5 GB. This is required so that the WildFly and Couchbase images can be downloaded onto any of the nodes.

You can get more information about this machine by running the inspect command. If you run it, you'll see that the disk size is indeed 5 GB.

Now, create a new machine to join this cluster:

```
docker-machine create -d virtualbox
    --virtualbox-disk-size "5000" --swarm
    --swarm-discovery=
    "consul://$(docker-machine ip consul-machine):
    8500" --engine-opt="cluster-store=consul://$(
    docker-machine ip consul-machine):8500"
    --engine-opt="cluster-advertise=eth1:2376"
    swarm-node-01
```

Notice that no --swarm-master option is specified in this command, which ensures that the created machine is a *worker* node.

If you list all the created machines, you will see the contents of Table 2. The machines that are a part of the cluster have the cluster's name in the SWARM column; otherwise, that column is blank for machines that are not part of the cluster. For example, consul-machine is a standalone machine, whereas all the other machines are part of the swarm-master cluster. The swarm master is also identified by (master) in the SWARM column.

Connect to the swarm cluster and find some information about it:





NAME	ACTIVE	DRIVER	STATE	URL	SWARM
consul-machine	-	virtualbox	Running	tcp://192.168.99.100:2376	
swarm-master	*	virtualbox	Running	tcp://192.168.99.101:2376	swarm-master (master)
swarm-node-01	-	virtualbox	Running	tcp://192.168.99.102:2376	swarm-master

Table 2. The contents of the sample Docker Swarm

eval "\$(docker-machine env --swarm swarm-master)"
docker info

Note that --swarm is specified to connect to the swarm cluster. Otherwise, the command will connect to the swarm-master machine only.

The docker info command provides detailed output about the different nodes and the number of containers running on each node. The first part of the output provides a summary and is shown below:

docker info
Containers: 3
Images: 2

Role: primary Strategy: spread

Filters: health, port,

dependency, affinity, constraint

There are two nodes: one swarm master and one swarm worker node. There are three containers running in this cluster: one swarm agent on the master and each node, and there is an additional swarm-agent-master running on the master.

Start the Application Environment

The cluster is now ready, and our application can be deployed to it. This Java EE application deployed on WildFly will provide a CRUD/REST interface to the data in Couchbase.

I will use the Compose file shown earlier to start the application environment. But before I start, let's look at the current state of the network in the cluster.

By default, Docker creates three networks for each host, as shown in Table 3.

Docker allows you to create *bridge* and *overlay* networks. Bridge networks span a single host, and overlay networks span multiple hosts. The Docker Compose application can be started with the --x-networking switch. In this case, a bridge network is created for a single host and an overlay network is created for a swarm cluster.

Make sure you are connected to the cluster by running the following command:

eval "\$(docker-machine env --swarm swarm-master)".

I will use the Compose file shown earlier to start WildFly and Couchbase. In the WildFly service, the COUCHBASE_URI environment variable is set to db. And this is the name of the Couchbase container. Start the application with

NETWORK NAME	PURPOSE
bridge	DEFAULT NETWORK THAT CONTAINERS CONNECT TO
none	CONTAINER-SPECIFIC NETWORKING STACK
host	ADDS THE CONTAINER ON THE HOST'S NETWORKING STACK

 Table 3. Three networks created by Docker





the following command:

```
docker-compose --x-networking up -d
```

An overlay network, specific to the application, is created. You can verify this by running the following command:

```
docker network 1s
```

Three networks for each swarm node will be shown. In addition, the wildflycouchbasejavaee7 overlay network and the docker_gwbridge bridge network are created for containers to gain connectivity outside the cluster.

You can verify the WildFly and Couchbase containers by using the docker ps command. The truncated output of this command shows that the Couchbase server is running on the swarm-master machine and WildFly is running on the swarm-node-01 machine.

Configure the Couchbase Server and Install the Sample Data

When it is started as a Docker container, the Couchbase server needs to be configured so that it can be used for storing data. Fortunately, it provides a REST API that can do exactly that. This API can also be used to upload the sample data.

Use git to clone the online app, as shown next, and then use Maven for the installation. This application has a Maven profile that allows you to configure the Couchbase server using the REST API.

```
git clone https://github.com/
    arun-gupta/couchbase-javaee.git
mvn install -Pcouchbase -Ddocker.host=
    $(docker-machine ip swarm-master)
```

In this snippet, \$(docker-machine ip swarm-master)

will provide the IP address of the swarm master machine that was created earlier. You could use the actual IP address (which could be found using docker-machine ip swarm-master) instead of using \$(docker-machine ip swarm-master).

Deploy the Application to the Docker Swarm Cluster

Deploying the application requires us to specify the IP address of the host where WildFly is running and the username/password of a user in WildFly's administration realm. In our case, the IP address is that of the swarm-node-01 machine and the username/password values are the ones that are used to build the arungupta/wildfly-admin image.

The exact command (to be entered as a single line) will be

```
mvn install -Pwildfly -Dwildfly.hostname=
  $(docker-machine ip swarm-node-01)
  -Dwildfly.username=admin
  -Dwildfly.password=Admin#007
```

You might want to obtain the IP address of the machine running WildFly by using \$(docker-machine ip swarm-node-01) and then use the IP address instead.

Access the Application

Now that the WildFly and Couchbase services have started, it's time to access the application. You need to specify the IP address of the machine where WildFly is running. An easy way is to do this is to use the widely known command-line utility curl:

```
curl http://$(docker-machine ip swarm-node-01):8080/
    couchbase-javaee/resources/airline
```

[This command should be entered as a single line. —*Ed.*] This command prints the list of 10 airlines from the sample data that was configured earlier in Couchbase.

Alternatively, you can obtain the IP address of WildFly





using the docker-machine ip swarm-node-01 command and then access the application in your browser.

This application supports GET, PUT, POST, and DELETE APIs for resources in the Couchbase sample data. The complete list of REST APIs for this application is <u>documented online</u>.

Conclusion

This two-part series explained how to deploy a Java application on Docker. The first part introduced the basic concepts of Docker and the Docker Toolbox. I then discussed how to build an image and run a container. Those basics will apply to almost all Docker projects.

This second part introduced Docker Compose and Docker Swarm to help build multicontainer applications that can be deployed on multiple hosts. In the process, I touched on how Docker networking provides isolated environments for different applications. This should provide what you need to get started using multiple Docker containers in your application.

Docker nicely complements Java's "write once, run anywhere" mandate, because it enables "package once, deploy anywhere." The huge ecosystem that has built up around Docker in the past few years is driven by Docker's simplicity. With a little experimenting, I expect you will be equally excited about packaging Java applications using Docker. </article>

Arun Gupta (@arungupta) is the vice president of developer advocacy at Couchbase. He is also a Java Champion and a JUG leader. Years ago, he was a founding member of the Java EE team.

learn more

Getting Started with Docker









//microservices /





TILEN FAGANEL AND MATJAZ B. JURIC

KumuluzEE: Building Microservices with Java EE

Develop self-contained microservices with standard Java EE APIs using the open source KumuluzEE framework.

In this article, we explore how to develop microservices with standard Java EE APIs. Microservices have become an important architectural style for developing complex applications composed of small, independent services deployed as individual processes. These services communicate through well-defined APIs. Each microservice is self-contained; is responsible for a single task; is accessible through a lightweight API, such as REST; and has its own lifecycle. Microservices result in a highly decoupled, modular, and reusable architecture for composite applications.

Developing microservices in Java means that not only is the application designed around REST services, but also that each service, the front end, and other components are deployed separately and independently. In other words, instead of packing all the services, business logic, front end, and other modules into a single EAR file and deploying it as a monolith, microservices use individual archives (such as JAR files) for each component. The <u>KumuluzEE framework</u>, as we'll see shortly, automates the tasks related to the configuration and deployment of microservices, specifically on Java EE. [**Note**: KumuluzEE won the Duke's Choice Award at the 2015 JavaOne conference. —*Ed*.]

Microservices with Java EE

Suppose we want to create an online train-reservation ser-

vice people can use to plan their route and buy their tickets. For brevity, we are going to look at a simplified version of this example; it will allow only viewing routes and booking a route via a simple user interface.

Using the traditional approach, we would create a monolithic EAR package that would include our business logic, services, and one or more WAR packages containing the front end. We would deploy the EAR file to an application server, such as Oracle WebLogic Server or GlassFish.

In contrast, using the microservice architecture, we start by separating the responsibilities. Routes, the booking service (reservations), and the UI become three separately developed, configured, and deployed microservices. They are stateless and communicate through REST interfaces—although we could also use SOAP or remote method invocation (RMI). In this way, we create microservices that are domain-defined and follow the "single responsibility" principle with regard to explicit interfaces. We have created a highly modular, decoupled architecture, where each service is responsible for a single, dedicated piece of functionality. Also, each microservice has its own lifecycle. The proposed architecture using microservices is shown in Figure 1.

In the remainder of this article, we use this simplified example. But keep in mind that microservices are best suited for complex applications and systems.





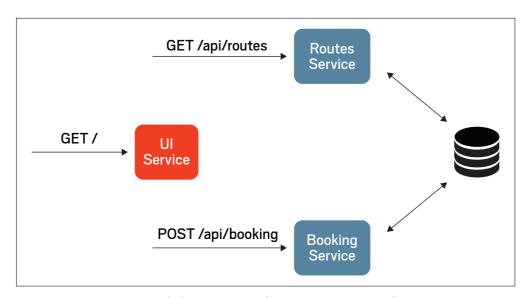


Figure 1. Overview of the proposed microservice architecture

Advantages. Let us now explore the advantages of microservices compared with a monolithic approach. The most obvious advantage is that microservices lead to a more flexible, decoupled architecture. Each microservice can be developed independently, which simplifies lifecycle and change management. For example, if we need to upgrade the booking service, we do not have to redeploy the whole application. Every microservice is a separate project with a separate repository and deployment configuration. We can roll out new features iteratively, thus increasing agility.

If, over time, we would like to upgrade to newer versions of Java, we can upgrade each microservice separately. For complex applications, this is a huge benefit, because it is often difficult to upgrade complex Java EE applications (to a new version of the app server or to new releases of Java EE and Java SE). Even better, microservices allow us to gradually incorporate new technologies.

Last but not least, microservices allow for much better scalability. For example, the route calculation service might get higher traffic than the booking service, because users might use the route calculation more frequently than they book a reservation. With microservices, we can scale the book-

ing service independently of the rest of the application. This approach is a perfect fit for cloud and PaaS (platform as a service) environments, where elastic scalability can be configured easily. Scaling a microservice application in a Docker environment becomes a breeze. In other words, a microservices architecture is a good match for cloud-enabled applications.

To fully exploit these benefits, microservices need to be stateless. Every resource (such as a database, object storage, and so on) that a microservice uses should be configurable separately (usually via a connection string or environment parameters).

Disadvantages. Microservices do not come without drawbacks. Setting up and configuring Java EE projects to accompany this kind of architecture is often not trivial. Actually, it can require a lot of manual configuration and tracking of dependencies, and it can lead to significant operational complexity. Deployment becomes more complex, as does testing.

KumuluzEE

KumuluzEE addresses some of these drawbacks. It automates the tasks related to configuration and deployment of microservices in a seamless way. At its most basic, KumuluzEE collects each microservice together with the Java EE APIs' runtime into a simple, standalone package (JAR file), providing a minimal footprint by including only those APIs (and their runtimes) that are actually used. This enables developers to build standalone, stateless, self-contained microservices and package them in an efficient way without the overhead of including the entire Java EE application server in each microservice. Microservices created this way can be executed directly from the JRE with a minimal footprint and quick startup and shutdown times.

Getting Started

We'll now create our microservices using KumuluzEE, standard Java EE, and Maven. First, we create three Maven







projects—each containing its own microservice. For brevity's sake, we will create them in the same repository. We will create a routes project for the route calculation service, a bookings project for the booking service, and a ui project for the front end. We will also add a module that will hold our JPA entities (models) and a module that will hold our utils, because they will be shared with our three microservices. Note: In real—world projects, it is generally recommended to use a separate repository for each microservice so that they are treated as separate entities and have separate versioning and revision history as well as separate deployment.

We will also create the topmost pom.xml file. You can look at the project structure and the pom.xml file by downloading the source code for this project from <u>GitHub</u>.

To create our microservice projects, we will use Maven archetype generation (mvn -B archetype:generate) for all modules. This will generate the three projects we require.

Adding KumuluzEE

Now we need to add the appropriate dependencies to the KumuluzEE framework. KumuluzEE is completely modular, which means that apart from the core functionality each Java EE component is packaged as a separate module and must be included explicitly as a dependency in order to be used. KumuluzEE will automatically detect which modules are included in the classpath and properly configure them.

We include KumuluzEE by defining the appropriate dependency in Maven. It is recommended that we define a property with the current version of KumuluzEE and use it with every dependency:

Adding the Front-End UI Service

Let's start with the ui microservice and include the core KumuluzEE module for bootstrapping the logic and configuration. It provides the com.kumuluz.ee.EeApplication class with the main method that will bootstrap our app. We will also include the servlet and the HTTP server. We will use a Jetty servlet implementation, which is known for its high performance and small footprint. The ./ui/pom.xml file contains this:

We must also include the maven-dependency-plugin in our pom.xml file, which will copy all our dependencies together with the classes. So we add the following to the ./ui/pom.xml file:







```
</executions>
</plugin>
```

This is the bare minimum required to run a microservice with plain servlets and static files. To try it out, we can add a simple index.xhtml HTML file. KumuluzEE will use the webapp folder at the root of the resource folder to look for configuration data and files. This is where we should put the files.

To try it out, run maven package. Then, you can start the microservice using the following command:

```
$ java -cp ui/target/classes:ui/target/dependency/* \
    com.kumuluz.ee.EeApplication
```

Go to http://localhost:8080/ in your browser and you should see the content of the index.xhtml HTML file. Of course, you can use JSP, use a servlet, or add JSF support and use it for the front end. We will extend the front end shortly.

Defining the JPA Module

We follow the procedure described above for each microservice. Because each microservice is its own project, you can customize it as much as you need.

Before we start developing the routes and booking services, we will define the JPA module with entities. To include JPA, we add the EclipseLink JPA implementation. We will also add the database driver. In our case, we will use the PostgreSQL database. However, you are free to use any database. Place the required dependencies in ./models/pom.xml:

Next, we will add the persistence.xml file and entity classes that will be shared with both our microservices even though the microservices will be run separately. This file is the same as what you would develop in a traditional Java EE application. You can look at the provided sample source code to see an example. Notice that the values defined in persistence.xml for the database URL, username, and password can be overwritten by setting the DATABASE_URL, DATABASE_USER, and DATABASE_PASS environment variables, respectively. Doing this is useful for easy configuration in Docker-style environments.

We are now ready to write the entity classes. We use standard JPA, which does not require any modifications for KumuluzEE. The example JPA entity classes for Route and Booking are shown next. The first file, .../models/Route.java, contains the following:







```
@Temporal(TemporalType.TIMESTAMP)
    private Date duration;
    @ManyToMany(mappedBy = "routes")
    private List<Booking> bookings;
    // ... Omitted getters/setters
 The booking file, .../models/Booking.java, contains
this:
@Entity
@NamedQuery(name="Booking.findAll",
            query="SELECT b FROM Booking b")
public class Booking {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Integer id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date orderDate;
    @ManyToMany
    private List<Route> routes;
    // ... Omitted getters/setters
```

Implementing the Routes and Booking Services

We are now ready to implement the routes and booking services. We will need JAX-RS for the REST interfaces. We will use <u>Jersey</u> as implemented in KumuluzEE. We will also add CDI, because we want to use the EntityManager injection using @PersistenceContext. However, we could just as well do without CDI to keep our microservice even lighter. We add the following dependencies to ./booking/pom.xml:

Now, let's implement the booking REST service. We'll create a booking resource that contains two methods: createBooking(Booking) and getBooking(). Make sure you add the correct dependencies, as described earlier. Also make sure you add the models module to every microservice for access to JPA entities, whose REST service will be used. The code is:

```
@ApplicationPath("/")
public class BookingApplication extends
    javax.ws.rs.core.Application {
}
```

Then in BookingsResource.java we have this:







As you can see, with KumuluzEE the microservice implementation does not require any modifications and is exactly the same as with any other Java EE application. You can implement the remaining microservices the same way or look at the provided sample code.

Handling Service Discovery

Let's suppose that we are using JavaServer Faces (JSF) for the front-end UI module. From there, we will be calling our microservices via REST to receive or save the actual data. The main problem with this is that we need to know the exact URLs of the microservices. We could pass the URLs via environment variables. However, that approach would require manual updating every time anything changes. And in the cloud, these changes can be quite frequent.

A better solution would be to dynamically query the address of the requested microservice. We will use <u>Apache ZooKeeper</u> for service discovery, but we could also use any similar tool. In short, ZooKeeper is a centralized service for maintaining configuration information. We will use it to store the

correct URL to each microservice. We add a helper class, ServiceRegistry, in our utils module to handle the dynamic registering, unregistering, and retrieval of endpoints with ZooKeeper:

```
@ApplicationScoped
public class ServiceRegistry {
    private final CuratorFramework zookeeper;
    private final
        ConcurrentHashMap<String, String> zonePaths;
   @Inject
   public ZooKeeperServiceRegistry()
           throws IOException {
       try {
            String zookeeperUri =
                System.getenv("ZOOKEEPER_URI");
            zookeeper = CuratorFrameworkFactory
                .newClient(zookeeperUri,
                     new RetryNTimes(5, 1000));
            zookeeper.start();
            zonePaths = new ConcurrentHashMap<>();
        } catch (IOException) {
   public void registerService(
            String name, String uri) {
       try {
            String node = "/services/" + name;
            if (zookeeper.checkExists()
                  .forPath(node) == null) {
                zookeeper.create()
                    .creatingParentsIfNeeded()
                    .forPath(node);
            String nodePath =
              zookeeper.create().withMode(
```







The ZOOKEEPER_URL environment variable should contain our ZooKeeper URL. We will be starting a ZooKeeper service alongside our microservices using Docker, which is discussed in the next section. Also make sure that the utils package is added as a dependency to every microservice.

We can now inject the service above into our microservices to register its URL upon startup and unregister it upon shutdown. One way to do this is to add a bean to our booking microservice to handle this task dynamically:

```
@ApplicationScoped
public class BookingService {
    @Inject
    ServiceRegistry services;

    private String serviceName = "trains-booking";
    private String endpointURI;

    public BookingService() {
        endpointURI =
            System.getenv("BASE_URI");
    }

    @PostConstruct
    public void registerService() {
```

We use the @PostConstruct, @PreDestroy, and @ApplicationScoped annotations to make sure we register and unregister the service only once per lifecycle. The BASE_URI environment variable should contain the microservice's public URI. We can now create a similar bean for each of our microservices.

Injecting the Microservices at the Front End

We can use the just-described approach for injection on the front end to connect to the REST services. We can inject our ServiceDiscovery bean to our JSF backing bean to retrieve the URL of a microservice we want to invoke, as shown below:







Updating the JSF Front End

Finally, let's update our index.xhtml file in the ui module to use the created bean and to list the available bookings:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html >
<html xmlns="http://www.w3.org/1999/xhtml"</pre>
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>All available bookings</title>
  </h:head>
  <h:body>
    <h:dataTable
       value="#{bookingsBean.allBookings}"
        var="booking">
      <h:column>
        <f:facet name="header">Booking id</f:facet>
        #{booking.id}
      </h:column>
      <h:column>
        <f:facet name="header">Date of order
        </f:facet>
        #{booking.orderDate}
      </h:column>
    </h:dataTable>
  </h:body>
</html>
```

And we're done! We have created three microservices that will run independently of each other. They communicate through REST interfaces and share common resources such as JPA entities. Of course, we could use HTML5 with AngularJS or ReactJS for the front end as well.

Deploying and Running Using Docker

We can now build and run our microservices. To demonstrate this, we will use Docker. First we need to create a Dockerfile, which will contain the steps to create a Docker image that will build each microservice and run it. Because KumuluzEE runs a microservice as a normal, single-process, standalone JAR file with a minimal footprint (and does not require a separate app server instance for each microservice), it fits perfectly with how Docker operates. The Dockerfile is shown next. Its ./routes/Dockerfile file is in the downloaded files.

```
FROM java:openjdk-8u45-jdk
MAINTAINER info@kumuluz.com
RUN apt-get update -qq && \
    apt-get install -y wget git
RUN wget [...maven binaries...] && \
    [...install maven...]
RUN mkdir /app
WORKDIR /app
ADD . /app
RUN mvn clean package -Pdeploy
ENV JAVA_ENV=PRODUCTION
EXPOSE 8080
CMD ["java", "-server", "-cp",
    "ui/target/classes:ui/target/dependency/*",
    "com.kumuluz.ee.EeApplication"]
```

As we can see, the Dockerfile is pretty simple and straightforward. It installs the required dependencies, builds the microservice, and supplies the command to run it. In our case, we can use the same Dockerfile for every microservice, except for the final command, which specifies which microservice is going to run in a particular Docker container.

Next, we need to build a Docker image for every microservice that we have:







```
$ docker build -t trains/ui \
    -f ui/Dockerfile .
$ docker build -t trains/routes \
    -f routes/Dockerfile .
$ docker build -t trains/bookings \
    -f bookings/Dockerfile .
```

Once built, these images can be run anywhere Docker or the Docker API is used, as well as anywhere Docker Swarm, Kubernetes, various PaaS providers, and many such services are used. To test the images, we can run them locally. We also need to start a ZooKeeper instance by using an existing image. Don't forget to pass the required environment variables to our microservices (using the -e switch). The following example uses 172.17.42.1 as the Docker host, because that is the default. Adjust as needed.

```
$ docker run -p 2181:2181 -d fabric8/zookeeper
$ docker run --name ui -p 3000:8080 -d \
    -e BASE_URI='http://172.17.42.1:3000' \
    -e ZOOKEEPER_URI=172.17.42.1:2181 trains/ui
$ docker run --name routes -p 3001:8080 -d \
    -e BASE_URI='http://172.17.42.1:3001' \
    -e ZOOKEEPER_URI=172.17.42.1:2181 trains/routes
$ docker run --name bookings -p 3002:8080 -d \
    -e BASE_URI='http://172.17.42.1:3002' \
    -e ZOOKEEPER_URI=172.17.42.1:2181 trains/bookings
```

We can now browse our microservice application at http://localhost:3000 or browse our REST services for the routes and bookings modules at http://localhost:3001 and http://localhost:3002, respectively.

Conclusion

In this article, we have seen how to develop microservices with standard Java EE using the open source KumuluzEE framework. As demonstrated, the KumuluzEE framework

automates tasks related to the configuration and deployment of microservices. With simple dependency definitions in Maven, it allows us to create self-contained, standalone JAR files, which contain the required execution environment. Therefore, they can be executed within a standard JRE without requiring an application server. This way, microservices with KumuluzEE provide a minimal footprint and are a good fit for executing in cloud, PaaS, and Docker-style environments, which was demonstrated with this example. We also saw how to handle service discovery using ZooKeeper.

KumuluzEE is an enabler for a microservices architecture in Java EE. The major benefit is that it allows us to use standard Java EE APIs to develop microservices. This way we can also port existing applications to microservices. Although at first blush, microservices might look odd to traditional app server developers, many technologists believe microservices represent the future of Java in the cloud. Feel free to download KumuluzEE and try it out. </article>

Tilen Faganel is lead software architect at Sunesis. He is the lead developer of the KumuluzEE framework for Java, which won the 2015 Duke's Choice Award for best Java innovation.

Matjaz B. Juric, PhD, is a Java Champion and Oracle ACE Director who has authored or coauthored more than 15 books on Java, BPM, and SOA and published in several magazines and journals.



Getting Started with KumuluzEE







Quiz Yourself

Test questions from the author of the Java certification tests

SIMON ROBERTS

[In order to expand this section and make it more useful, we asked Simon Roberts, an author of many of the certification tests, to develop questions and answers for us. Unlike the contents of previous columns, Simon provides detailed and in-depth explanations of why a given solution is correct. We hope you like this new expanded format. If so or if not, drop me a line. —Ed.]

These questions are intended to be at the same level of difficulty as the 1Zo-809 Programmer II exam, and they should serve at least two purposes. First, I hope that for many readers they might introduce something new, or clarify some subtle aspect of the Java language. Second, of course, I hope they'll give some insight into how exam questions might look, the kind of difficulty they might present, and how you might go about answering them.

Question 1. Given this code:

```
public class Outer {
  private int x = 99;

private class Inner {
  private int x = 100;

  public void show() {
    System.out.println(x); } // line n1
  public void showO() {
    System.out.println(Outer.this.x); } // line n2
  }

public void show(Inner i) {
```

```
System.out.println(i.x); } // line n3
public void show() {
    System.out.println(x); } // line n4

public static void main(String [] args) {
    Outer o = new Outer();
    Inner i = o.new Inner(); // line n5
    o.show(); o.show(i);
    i.show(); i.showO();
}
```

Which two, taken individually, are true? Choose two.

- **a.** At line n2, the expression Outer.this.x has the value 99.
- **b.** Outer.this.x at line n2 fails because x in the Outer class is inaccessible from Inner.
- c. The expression Outer.this.x at line n2 is a syntax error that should read super.x.
- **d.** The expression i.x at line n3 has the value 100.
- e. The access to i.x at line n3 fails because x in the Inner class is inaccessible from Outer.

```
Question 2. Given this interface: public interface Clothing {
```

```
public interface Clothing {
    int getSize();
    String getColor();
}
```

What changes must be made to the following class to allow its use as a generic container of clothing items?

```
public class Pair { // line n1
  private E left, right; // line n2
```





```
public Pair(E left, E right) {
    this.left = left;
    this.right = right;
  public boolean isMatched() {
    return left != null && right != null
      && left.getSize() == right.getSize()
      && left.getColor().equals(right.getColor());
a. Modify the type E to be Object.
b. Modify line n1 to:
  public class Pair<? extends Clothing> {
c. Modify line n1 to:
  public class Pair<E super Clothing> {
d. Modify line n1 to:
  public class Pair<E extends Clothing> {
e. Modify the variable declarations on line n2 to:
  private Clothing left, right;
```

Question 3. In the following lambda, assume that the generic arguments are appropriately defined:

```
s->s.length()
```

Given that the length() method returns an int, to which two standard functional interfaces can the lambda expression be assigned? Choose two.

```
a. Function
```

b. IntFunction

c. IntSupplier

d. ToIntFunction

e. UnaryOperator



Question 1. The correct answers are Options A and D. This question revolves around the meaning of **private** accessibility. It's common to hear casual descriptions of the meaning of **private** in words along the lines of "**private** stuff is accessible only from inside the same class." However, this is inaccurate in a critical way, and this question hinges on that.

In essence, any private element is visible anywhere inside the enclosing top-level curly braces that surround it. For simple situations, the two descriptions might be the same, but in the case of nested or inner classes, the private elements of inner and outer classes are all visible from both classes. Section 6.6.1 of the Java Language Specification notes that "...declared private,...access is permitted if and only if it occurs within the body of the top level class...that encloses the declaration...."

The syntax Outer.this.x is the correct way of providing an explicit access to the field x in the enclosing instance. Of course, it's horrible style to have used the same name, x, in both inner and outer classes like this. If the variables were not both named x, they wouldn't collide, and there would be no need for this disambiguating syntax. However, this exam isn't about testing whether you know the things that everybody knows; it's about proving that you're better than average, and this syntax sometimes crops up, although more often in the maintenance of older code that has taken some abuse over some time. Section 15.8.4 outlines this form.

Given this, Options A and D are correct. Options B and E, which assert that the **private** fields are essentially accessible only inside their own defining classes, are incorrect.

The syntax suggested by Option C is also inappropriate. While super.x is a generally valid form, it would provide disambiguated access to a variable x in a parent class, not







an enclosing class (section 15.11.2). Given this, Option C is also incorrect.

Therefore, Options A and D are the only correct answers.

Question 2. The correct answer is Option D. There are two problems with the Pair class as it stands. The first is that the generic type variable E is not declared. This would require, at a minimum, that the class declaration on line n1 be modified to look like this:

public class Pair<E> {

However, if that is the only change made, the underlying type of E will be taken as Object; that is, there will be no restrictions on what types this Pair can be made to contain, other than that the left and right elements must be the same type. Because of this, the attempts to call the getSize and getColor methods in the isMatched method will fail.

For this class to work properly, we need to ensure that whatever E is in any given use of the class, it's something that implements the interface Clothing. That way, the basic type of the variables left and right will be Clothing, and the calls to the methods getSize and getColor will compile successfully.

To make this happen, the syntax of the class declaration is as shown in option D. (The syntax is formally called out in section 8.1.2 of the Java Language Specification.)

Section 8.1.2 also makes clear that type bounds of this kind are declared using extends, not using super. The use of super in angle brackets relates to lower-bounded wildcards and is not relevant here. Consequently, option C is incorrect.

Option B also tries to use a syntax that is incorrect in this context, and is more closely related to that of a bounded wildcard. Because line n1 must declare the generic type variable, it must be declared by name E and not by using the <? ...> format that is used by bounded wildcards.

Consequently, option B is also incorrect.

Option A is incorrect for two reasons. First, it fails to ful-fill the requirement that the Pair class be generic. However, more severely perhaps, it still would not permit the class to compile, because the attempt to invoke the getSize and getColor methods in the method isMatched would fail.

Option E also fails to compile. Although it would be possible to create this class in a nongeneric format, making the variables of type Clothing, this would also require that the constructor arguments be Clothing to allow the code to compile. Because that additional change is not mentioned in option E, the option fails and must be rejected. Further, even if the supporting changes were also made, the class would not be generic, as called for in the question.

Question 3. The correct answers are options A and D. Using lambda expressions, we give the compiler what amounts to "parts of a class definition" and ask the compiler to fill in the blanks from the context. Given a lambda without context, it's possible to view the lambda in different ways, and the blanks might be filled in very differently.

In this example, we know that the return type of the lambda is int, because that's the type of the length expression. So whatever we hope to make from this must return an int or, via autoboxing, an Integer.

Any of the listed functional interfaces could satisfy this requirement, so to move forward, we must consider what we know about the argument types of the lambda.

One thing we know is that there is a single argument, called s. That means that the IntSupplier is not compatible with this lambda expression, because IntSupplier (like all Supplier interfaces) takes zero arguments. So, we can rule out option C.

We also know that the argument type is some kind of object, not a primitive. We know this because we are able to invoke this mysterious length() method on the argument s. (Note







//fix this /

that it's tempting to assume that this argument is a String, but we neither know that nor need to know that.) Let's consider the remaining options in light of this, and assume that the type of the argument s is defined by some imaginary interface HasLength.

The argument and return types of a function are defined independently, so a Function<HasLength, Integer> would match our lambda successfully, and option A is a correct answer.

The argument type of an IntFunction is a primitive int; therefore, option B can be ruled out.

We already eliminated option C, so we'll ignore it now.

The ToIntFunction takes an argument of a generic type and returns a primitive int, so ToIntFunction<HasLength> fits our lambda perfectly, too. It's worth noting that this is in a sense the "best fit" because it avoids the autoboxing of the return value, but the question doesn't get into such value judgments. But clearly option D is correct, too.

The argument type and return type for a UnaryOperator are constrained to be the same. We already know the return is int, so the only way that UnaryOperator might possibly match our lambda is as UnaryOperator<Integer>. That would work only if the Integer object has a length() method. Of course, it doesn't, so option E cannot work either. </article>

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.









//contact us /



Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers.

Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open

source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone

+1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to me, **Andrew Binstock**, at <u>javamag_us@oracle.com</u>.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.





