//MAY/JUNE 2013 /

Java™ magazine

By and for the Java community

JAVA EE 7: LICENSE TO CODE

Boost your productivity and create next-generation Web apps

ORACLE.COM/JAVAMAGAZINE

ORACLE

//table of contents /

COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Caroline Kvitka

Community Editors
Cassandra Clark, Sonya Barry, Yolande Poirier

Java in Action Editor
Michelle Kovac

Technology Editors
Janice Heiss, Tori Wieldt

Contributing Writer
Kevin Farnham

Contributing Editors
Claire Breen, Blair Campbell, Karen Perkins

DESIGN

Senior Creative Director
Francisco G Delgadillo

Senior Design Director
Suemi Lam

Design Director
Richard Merchán

Contributing Designers
Jaime Ferrand, Nicholas Pavkovic

Production Designers
Sheila Brennan, Kathy Cygnarowicz

PUBLISHING

Vice President
Jeff Spicer

Publisher
Jennifer Hamilton +1.650.506.3794

Audience Development and Operations Director
Karin Kinnear +1.650.506.1985

ADVERTISING SALES

Associate Publisher
Kyle Walkenhorst +1.323.340.8585

Northwest and Central U.S.
Tom Cometa +1.510.339.2403

Southwest U.S. and LAD
Shaun Mehr +1.949.923.1660

Northeast U.S. and EMEA/APAC
Mark Makinney +1.805.709.4745

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (U.S./Canada)

Oracle Services
+1.888.283.0591 (U.S.)

Oracle Press Books
oraclepressbooks.com

ARTICLE SUBMISSION

If you are interested in submitting an article, please e-mail 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 e-mail address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly with a free subscription price by Oracle, 500 Oracle Parkway, MS OPL-3C, Redwood City, CA 94065-1600.

Digital Publishing by Texterity

You've been waiting, and now it's here. Java EE 7 has arrived, with lots of new tools designed to make you more productive and let you add new capabilities to your Web applications. If you're like me, when something new arrives at your doorstep, you want to open it up and start using it right away. And in this issue of *Java Magazine*, we help you do that with Java EE 7. So roll up your sleeves and dive in. (Look for our Java EE 7: License to Code icon for all of the Java EE 7–related content in this issue.)

First, Anil Gaur, vice president of software development at Oracle, brings you up to speed on what's new and notable in Java EE 7—including HTML5 support, WebSocket, JAX-RS, JSON Processing, batch processing, concurrency, and more—in our interview. Then Max Bonbhel explores the new APIs in Java EE 7. This is a big subject, so Max will continue with this topic in the coming issues. Next, Johan Vos digs into the productivity enhancements in the new release and shows you how Java EE 7 simplifies the default development and configuration. And finally, Eduardo Moranchel and Edgar Martinez, Java curriculum developers at Oracle, tackle the new features in Java EE 7 that allow you to create next-generation internet applications—WebSocket, HTML5, and JSON.

We also have stories of Java EE in action—from bringing sports fans up-to-the-instant information at ESPN to solving complex banking challenges in Brazil.

Do you have a story to tell about how you use Java or how your Java user group is thriving? Want to share your coding wizardry in our back-page Fix This column? Have other feedback? Send it our way.

Enjoy Java EE 7!

Caroline Kvitka, Editor in Chief BIO

//send us your feedback /

We'll review all suggestions for future improvements. Depending on volume, some messages may not get a direct reply.

PHOTOGRAPH BY BOB ADLER

What's in Your Code?

Palamida's CEO Mark Tolliver discusses how companies can ensure their software is compliant and secure.

Q: Why do developers use open-source software?

A: Software developers are constantly faced with the choice of taking the time to write code themselves, or searching the Internet for code that does what they're looking for. For ten years, that is what's been going on under the radar in the world of software development. They are under the gun, they've got tight schedules, and they decide it is more productive to find an element of code already out there on an open-source site, download it, put it in the code that they are developing for their company, and move on. And so what we find in today's world is that well over half of the code commercial companies ship is code that they didn't write. Our typical customer either has no knowledge of the open-source code that they're using—even though they're using hundreds of pieces—or they know just one percent of it.

Q: What are the issues around this?

A: There are three issues: number one is intellectual property compliance and copyright infringement—respecting the rights of the authors as expressed in the open-source license that they have chosen to govern the use of their software. Number two is maintaining visibility and traceability so that issues such as security vulnerabilities can be identified and fixed. Number three is traceability for compliance with export control regulations.

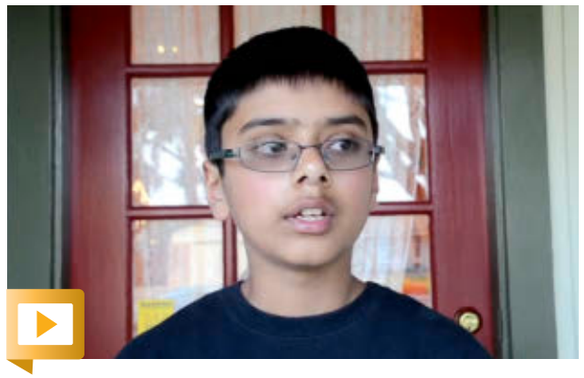
Mark Tolliver, CEO, Palamida

Q: How can Palamida help?

A: Palamida's special-purpose search engine allows companies to scan and discover what makes up their code. We will tell you what components and versions you're using and then supply information such as license, known security issues, and whether or not the open-source components contain cryptography, which would make it subject to export controls. Palamida offers two solutions to match your business needs: a software product licensed as a subscription for customers who prefer to scan code and report on results on their own, and a professional services product in which one of our experienced professionals scans the code and delivers a complete analysis to the customer.

Q: Why choose Palamida?

A: As a seven-year-old company, we have a proven track record in this area. The average duration of time our professionals have been with the company is over five years, so you know you're getting the best talent out there. Also, our search engine tracks over 800,000 different open-source components. With our patented search techniques, we search for and index the world's open-source software continuously, and every day we add somewhere between three to five gigabytes of new material to our library. We are confident that we can deliver superior search results and give our customers the information they need to effectively manage and secure their use of open source and other third-party content.



Aditya Gupta shares his thoughts on the workshop.

TEACH JAVA WITH MINECRAFT

Left: Young Minecraft aficionados listen to instructions; right: Arun Gupta introduces Java programming using car and fruit analogies.

PHOTOGRAPHS BY FRED WORLEY AND MENKA GUPTA

On March 16, 2013, 10 kids gathered at the home of Oracle Java Evangelist Arun Gupta to learn to program in Java using Minecraft. This wildly popular game among elementary and middle schoolers allows players to build constructions of textured cubes in a 3-D world. The game is written in Java and can be extended by writing "mods," or modifications. After spending a recent school break writing a few mods with his son, Gupta decided to invite some other young Minecraft enthusiasts to a coding workshop. The attendees ranged in age from 10 to 14, and most had no programming experience, let alone Java experience. The group did, however, possess a lot of Minecraft experience. Gupta leveraged their passion for the game and introduced them to Java programming. Using easily understandable concepts about cars and fruit, he introduced the core concepts of class, property, method, interface, and exception. Asked for a one-word description of building and running their first Hello World application using NetBeans, workshop attendees responded: "Fun," "Easy," "Quick," "Awesome," "Short," and "Intuitive." By the end of the session, the group had worked out an entire framework for making a mod. The mod built in the workshop added a new server-side command and printed a trivial message. But the joy on attendees' faces was priceless. Want to host your own Minecraft workshop? Get the details.

JAVA USER GROUP PROFILE

iJUG

The iJUG Association of Java User Groups, an umbrella group for Java user groups (JUGs) in Germany, Switzerland, and Austria, was formed in November 2011 and has been growing ever since. iJUG cofounder **Markus Eisele** spoke with *Java Magazine*.

Java Magazine: Who were the original iJUG founders?

Eisele: iJUG was founded by German Oracle User Group (DOAG), Java User Group Deutschland, Java User Group Munich, Java User Group Erlangen-Nürnberg, Java User Group Stuttgart, JUG Saxony Dresden, and JUG Köln.

Java Magazine: How many smaller, local JUGs coordinate their efforts with iJUG?

Eisele: Twelve JUGs are members.

Java Magazine: How many developers in total are part of iJUG?

Eisele: The 12 associated JUGs represent around 20,000 members and prospects.

Java Magazine: What are the benefits for the smaller JUGs?

Eisele: iJUG's focus is on fostering communication among the different members, and different activities help to achieve this. First of all, *Java Aktuell*, a Java-centered, German print magazine and corresponding monthly digital newsletter, is sent to the members. We also cooperate with the biggest IT publishing house in Germany, heise online, to get news items out. Going through iJUG allows individuals to join the JCP [Java Community Process] without having to sign a personal JSPA [Java Specification Participation Agreement].

Java Magazine: What else would you like to say about iJUG, or about JUGs and community?

Eisele: The biggest benefit iJUG is able to deliver is independence for individual members. Working locally within the existing organization is augmented by the umbrella with the power of roughly 9,000 active individuals. In line with the saying "Think globally, act locally," iJUG can funnel and communicate the feedback from its individual members effectively and help grow the local JUGs by setting them in the spotlight.

German-speaking developers can find out more at the iJUG Website.

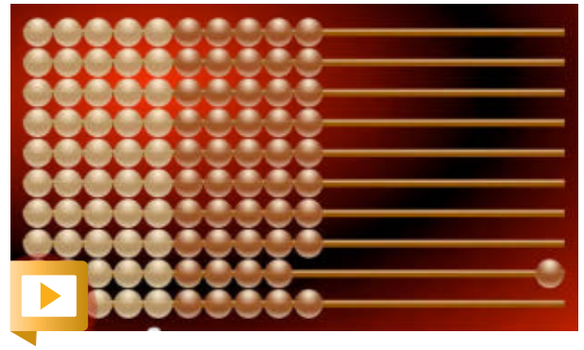
Dolphin Bridges Java Enterprise and Java Desktop

Dolphin, a free open source library from Canoo Engineering that protects business applications from the uncertainties of changing visualization technologies, was released in April 2013.

Dolphin, which is available for JavaFX and Swing, strictly separates business logic from visualization; has pluggable remoting, which keeps business logic on the server; and offers easy migration between client technologies.

In addition, visualization code resides on the client (View), domain models and controllers remain on the server, presentation models are shared between client and server, and a server may "push" commands to the client by means of a long poll.

See Dolphin in action.



The overflow in the JavaFX abacus

OpenJDK Governing Board Elects At-Large Members

OpenJDK The OpenJDK Governing Board, which oversees the structure and operation of the OpenJDK community, has elected two at-large members: **Andrew Haley** and **Doug Lea**. Both were previously serving as at-large members and began new one-year terms on April 1.

Haley, of Red Hat, wrote in his candidate statement that his goal is to "stand up for freedom and steer the governing board toward helping people who crank out code to get their job done."

Lea, of SUNY Oswego, wrote that he hoped "to continue my role as an advocate for continuing improvements in OpenJDK processes and mechanisms, especially as they impact the academic, research, and individual contributor communities."

JAVA CHAMPION PROFILE

ANTONIO GONCALVES

Antonio Goncalves *is a senior software architect, author, cofounder of the Paris Java User Group and Devoxx France, and an active member of the Java Community Process (JCP), where he is an Expert Group member for many Java EE JSRs. He became a Java Champion in June 2009.*

Java Magazine: Where did you grow up?

Goncalves: In Paris, but I'm Portuguese, so I've spent my life between two countries, two cultures, two languages.

Java Magazine: When did you first become interested in computers and programming?

Goncalves: At school when I was 12. We had Thomson TO7 and MO5 computers (Motorola 6809 processor–based) with some BASIC courses.

Java Magazine: What was your first computer and programming language?

Goncalves: Commodore 64 with BASIC and assembly language. It was a time when *Commodore Magazine* had dozens of pages with code to develop games. So I literally spent weeks writing code, saving it onto a tape, and running and debugging it. At the time, GOTO was a fantastic keyword.

Java Magazine: What was your first professional programming job?

Goncalves: Writing batches in COBOL. I wish I had kept on programming games with my Commodore 64, but COBOL was the language to know if you had to work and pay your bills. Then C++, Smalltalk, some Visual Basic, and Java in 1998.

Java Magazine: What do you enjoy for relaxation?

Goncalves: I studied art history for two years, and I live in Paris. So I use and abuse all of its cultural resources. I'm always at jazz concerts and art exhibitions.

Java Magazine: What happens on your typical day off?

Goncalves: I write code, blog, or do geeky stuff. My daughter is eight years old, so we go to concerts and exhibitions and take cycling day trips.

Java Magazine: What side effects of your career do you enjoy the most?

Goncalves: I have a very isolated job. Physically meeting people is very refreshing.

Java Magazine: Has being a Java Champion changed anything in your daily life?

Goncalves: It helps in meeting smart people.

Java Magazine: Java EE 7 is coming out soon. What is most significant about this new release?

Goncalves: Despite the updates in JMS [Java Message Service] 2.0, JTA [Java Transaction API] 1.2, and JAX-RS 2.0, the significant move about this release is the integration with CDI [Contexts and Dependency Injection]. More and more Java EE 7 specifications are starting to embrace CDI, and this will bring cohesion to the platform.

Java Magazine: What are you looking forward to?

Goncalves: To be surprised. My career has changed a lot. These changes happened thanks to what I aimed for, but also due to unpredictable surprises. As John Lennon wrote, "Life is what happens to you while you're busy making other plans."

Find more at AntonioGoncalves.org or on twitter.

DEVOXX UK AND DEVOXX FRANCE

Left: the expo floor at Devoxx France; right: the Devoxx UK team

A pair of Devoxx conferences took place March 26–29 in London, England, and Paris, France. These developer conferences were all about Java, with topics including Java-related technologies and platforms, methodologies, cloud, infrastructure, and the future of Java, to name a few. There was a packed schedule of presentations by international and local speakers, three-hour-long hands-on labs, birds-of-a-feather discussions, and plenty of space to code in the Hackergarden. Altogether, the two conferences attracted 2,000 developers from around Europe.

Oracle was a European Partner for Devoxx. Oracle technical staff presented sessions about the new features in upcoming Java EE 7, Java SE 8, the JavaScript engine Nashorn, JavaFX, Raspberry PI, the Java Community Process, Adopt-a-JSR, and more. The upcoming Java EE 7 generated a lot of interest, especially sessions about WebSocket and standardized batch processing.

Organized by the London Java Community, the first Devoxx UK was full of humor, and matched the atmosphere of creativity and passion that developers share for their work. **Kevlin Henney**'s keynote, titled "The Programmer," presented useful insights into who programmers are and tips to increase programmer productivity. Watch the replay on Parleys. **Martijn Verburg**, cofounder of Devoxx UK, said the conference provided "quality content that will help developers in their day jobs and in their careers and support them in the passion they have for computing as a whole."

For its second anniversary, Devoxx France added a CTO track, a day on DevOps methodology, coding days, and competitions. Programatoo, a programming workshop for 6- to 14-year-olds, ran for a day. According to **Nicolas Martignole**, cofounder of Devoxx France, Programatoo was Devoxx' best investment for the future.

Golo Language for JVM Demoed

Golo, a simple, dynamic weakly typed open source language for the Java Virtual Machine (JVM), was unveiled at Devoxx France 2013. Developers can pick up this language, which favors the explicit over the implicit, in a matter of hours, not days. The language was developed by the DynaMid research project members of the CITI Laboratory at INSA-Lyon.

Built from day 1 with invokedynamic, and currently in beta, Golo takes advantage of the latest advances of the JVM. It is also a showcase on how to build a language runtime with invokedynamic.

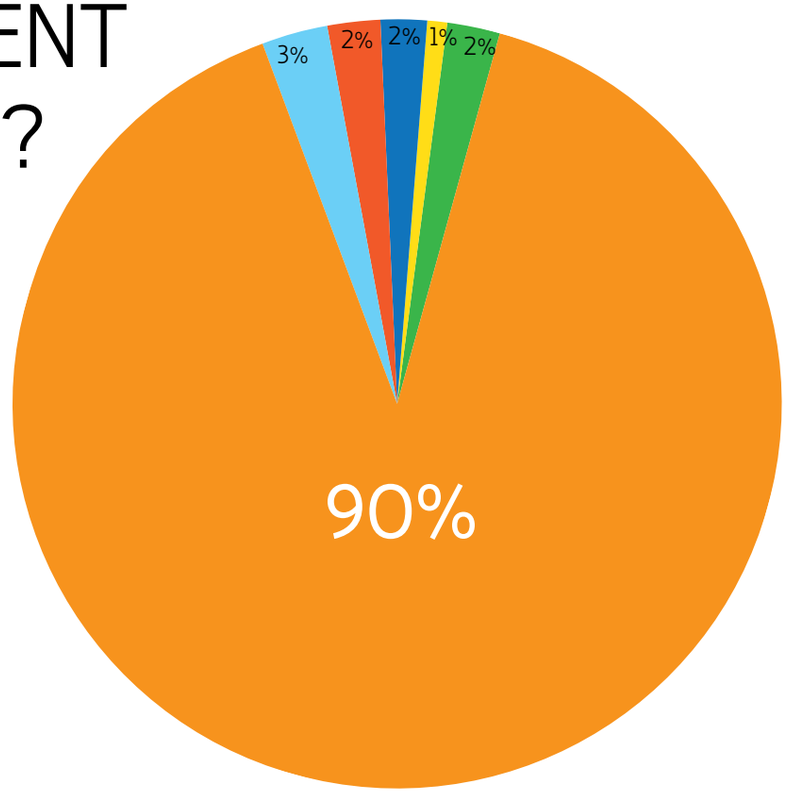
Golo founder **Julien Ponge** says there were two reasons why he decided to create a new language, despite so many languages being available for the JVM.

"I am working in a small team whose focus is around dynamic middleware and applications. As such, I am developing my research expertise on everything that sits between a dynamic application and its runtime environment," he explains. "Another good justification is that the only way to develop expertise in an area is simply to make all the possible mistakes you can."

JAVA.NET POLL

WHAT'S THE MOST IMPORTANT ENHANCEMENT IN JAVA EE 7?

In a recent Java.net poll, the Java developer community overwhelmingly selected HTML5 support as being the most important enhancement in Java EE 7. A total of 1,631 votes were cast during the two-week survey. The final voting shares were



90% HTML5 support, including WebSocket (JSR 356) and JSON Processing (JSR 353)

3% JavaServer Faces 2.2 (JSR 344)

2% JAX-RS 2.0 (the Java API for RESTful Web Services, JSR 339)

2% Expression Language 3.0 (JSR 341)

1% Java Message Service 2.0 (JSR 343)

2% Something else

Mobile World Congress: Redefining Mobile

Mobile World Congress used to just be about mobile phones and the industry around mobile networks. Judging from this year's conference, held February 25–28 in Barcelona, Spain, it's clear that mobile has redefined itself and is about sensors everywhere: cars, scooters, buildings, people, and more. The machine-to-machine or Internet of Things revolution is here, with mobile phones as just one of the many components that create an intelligent, connected world. As the mobile industry moves its focus from voice to data, developers now have the entire world as a potential for apps.

All this opportunity requires that decisions be made. Today's developers have a wide range of choices for what device to use and how to control it. The device and the platform you choose are key components for a successful implementation. Java enables devices to be intelligent, scalable, and supportable. Want to update a device remotely? Done. Want it to be headless? Done. Want a remote sensor on your grandmother that calls the hospital if she falls, and also lets the paramedics open the door to her house? Done.

A Java-powered remote sensor can save the life of an elderly person who falls.

See how Java makes it easy to get started in mHealth.

JUGS ADOPT A JSR FOR JAVA EE 7

The final push to release Java EE 7 has been given a boost by Java user groups (JUGs) from around the world. Java enterprise developers have created the Adopt-a-JSR for Java EE 7 initiative specifically for JSRs related to Java EE 7.

To participate, a JUG first selects one or more Java EE 7 JSRs that the members would like to support. Then, the JUG must join the Adopt-a-JSR program, review the specifications for the selected JSRs, and contact the specification lead or Expert Group. JSR support tasks available to JUGs include building and testing Reference Implementation (RI) builds, reporting bugs and other issues, helping moderate mailing lists, and helping build the Technology Compatibility Kit.

At press time, 19 JUGs across five continents were participating in the Adopt-a-JSR for Java EE 7 initiative: BeJUG, Campinas JUG, CeJUG, JUG Chennai, Cologne JUG, Congo JUG, FasoJUG, Houston JUG, Hyderabad JUG, Indonesia JUG, Jozi JUG, London Java Community, Madrid JUG, Mbale JUG, Morocco JUG, Peru JUG, Silicon Valley JUG, SouJava, and Toronto JUG.

EVENTS

JavaOne Shanghai *JULY 22–25*
SHANGHAI, CHINA

JavaOne Shanghai returns with four content-packed days of keynotes, technical sessions, hands-on labs, and demos.

Topics include constructing modern applications using Java SE; creating rich and immersive client-side solutions using JavaFX; building mobile and embedded Java applications targeting next-generation smart devices; composing sophisticated Java EE–based enterprise solutions; and building and deploying business applications in the cloud securely and seamlessly.

PHOTOGRAPH BY GETTY IMAGES

JUNE

No Fluff Just Stuff (NFJS)

JUNE 7–9, COLUMBUS, OHIO
JUNE 13–14, SANTA CLARA, CALIFORNIA
JUNE 21–22, SALT LAKE CITY, UTAH
JUNE 28–30, AUSTIN, TEXAS

No Fluff Just Stuff focuses on the latest technologies and best practices emerging in the enterprise software development space. The speakers are authors, consultants, open source developers, and recognized industry experts. Topics include enterprise Java, agility, Hibernate, Spring, Groovy/Grails, Ajax, JavaServer Faces, and more.

JULY

FISL

JULY 3–6
PORTO ALEGRE, BRAZIL

The International Free Software Forum (FISL) includes lectures, workshops, demos, and presentations about free software.

Java Forum Stuttgart 2013

JULY 4
STUTTGART, GERMANY

Java Forum, organized by Java User Group Stuttgart, includes presentations, workshops, demos, and birds-of-a-feather sessions about Java.

The Developer's Conference (TDC)

JULY 10–14
SÃO PAULO, BRAZIL

One of Brazil's largest conferences for developers, IT professionals, and students, TDC offers 21 tracks. Java-focused content includes introductory sessions in the Java University track, advanced and intermediate sessions in the Java EE track, and mobile and embedded sessions in the new mobile and embedded tracks. Other tracks include SOA, business process management, testing, architecture, and cloud computing.

ÜberConf 2013

JULY 16–19
DENVER, COLORADO

Part of the No Fluff Just Stuff Software Symposium Series, ÜberConf explores the ever-evolving ecosystem of the Java platform. This event offers 160 technically focused sessions including 25-plus hands-on workshops centered around architecture, cloud, security, enterprise Java, languages on the Java Virtual Machine, build/test, mobility, and agility.

OSCON

JULY 22–26
PORTLAND, OREGON

Open source pioneers, experts, and innovators have gathered at OSCON over the past 15 years. With a dozen tracks and hundreds of sessions, the conference offers tutorials, keynotes, and a wealth of information on open source languages and platforms. *Get a 20 percent discount on registration with the code JAVAMAG.*

Akure 2013 Inspires Youth

More than 1,000 students attended the Akure 2013 conference February 11–16 in Akure, Nigeria. This conference was the 15th annual national convention organized by the Nigerian Association of Computer Science Students (NACOSS), which spans more than 125 educational institutions and boasts 200,000 members. The conference theme, "Youth Relevance to Sustainable National Development through Information Technology," was inspired by NACOSS' passion to address the leadership challenges faced by young and upcoming leaders in Nigeria. Subthemes included leadership, the power of information and communications technology (ICT), ICT trends, and the importance of industry-academia cooperation.

Oracle Country Manager **Layo Ajay** gave a session on leadership and entrepreneurship, and Oracle Java Evangelist **Simon Ritter** followed with a "Welcome to the Petabyte Age" session that highlighted the growth in data and the "device to data center" approach of Java.

With a GDP growing at a rate of 7 percent and the country being called one of the "Next 11," Nigeria presents an opportunity for software startups.

JAVA BOOKS

CORE JAVA VOLUME I – FUNDAMENTALS, NINTH EDITION
By Cay S. Horstmann and Gary Cornell
InformIT (November 2012)
Designed for serious programmers, this no-nonsense tutorial illuminates key Java language and library features with thoroughly tested code examples. As in previous editions, all code is easy to understand, reflects modern best practices, and is specifically designed to help jump-start your projects. *Volume I* quickly brings you up to speed on Java SE 7 core language enhancements, including the diamond operator, improved resource handling, and catching of multiple exceptions.

BEGINNING JAVA EE, THIRD EDITION
By Antonio Goncalves
Apress (May 2013)
This step-by-step and easy-to-follow book describes many of the Java EE specifications and Reference Implementations, and shows them in action using practical examples. This book uses the latest version of GlassFish to deploy and administer the code examples.

Written by an expert member of the Java EE specification request and review board in the Java Community Process, *Beginning Java EE, Third Edition* offers an expert's perspective on enterprise Java technologies.

LEARNING JAVA, FOURTH EDITION
By Patrick Niemeyer and Daniel Leuck
O'Reilly (June 2013)
Java is the preferred language for many of today's leading-edge technologies—everything from smartphones and game consoles to robots, massive enterprise systems, and supercomputers. If you're new to Java, the fourth edition of this best-selling guide provides an example-driven introduction to the latest language features and APIs in Java 6 and Java 7. Advanced Java developers will be able to take a deep dive into areas such as concurrency and Java Virtual Machine enhancements. You'll also learn new ways to manage resources and exceptions in your applications.

3 Billion
Devices Run Java

Computers, Printers, Routers, BlackBerry Smartphones, Cell Phones, Kindle E-Readers, Parking Meters, Vehicle Diagnostic Systems, On-Board Computer Systems, Smart Grid Meters, Lottery Systems, Airplane Systems, ATMs, Government IDs, Public Transportation Passes, Credit Cards, VoIP Phones, Livescribe Smartpens, MRIs, CT Scanners, Robots, Home Security Systems, TVs, Cable Boxes, PlayStation Consoles, Blu-ray Disc Players…

Java™ | #1 Development Platform

ORACLE®

oracle.com/goto/java

Mark Little, vice president of engineering at Red Hat, overlooks a clean room cooling and air filtration system in the Newcastle University building where the Red Hat offices are located in northeast England.

JCP Executive Series

A Conversation with Mark Little

Red Hat's vice president of engineering discusses Java EE 7 and the JCP.

BY STEVE MELOAN

Continuing our series of interviews with distinguished members of the Executive Committee of the Java Community Process (JCP), we turn to Mark Little, vice president of engineering at Red Hat. Initially known for its Enterprise Linux OS, Red Hat acquired JBoss in 2006 and added enterprise middleware to its roster of technology offerings. Red Hat participated heavily in the development of both the Java EE 6 specification and the just-released Java EE 7 specification.

PHOTOGRAPHY BY JOHN BLYTHE

LIttle confers with a colleague.

Java Magazine: Red Hat has been very involved in the evolution of Java EE 7. Can you comment on leading the specifications for CDI [Contexts and Dependency Injection] 1.1, and Bean Validation, and participating in the development of a number of other JSRs?

Little: We led the CDI and Bean Validation updates, because we were already leading those in Java EE 6. And as you indicated, we've been very active on a number of other JSRs—including JTA [Java Transaction API], JCA [Java EE Connector Architecture], and JMS [Java Message Service] updates.

We've tried to be actively involved, in one way or another, with all of the JSRs that have been updated in Java EE 7. Even if we're not leading them, it's still important that we bring the perspective of Red Hat customers and our wider open source community to how Java EE is being adopted.

Java Magazine: Can you comment on the importance of these JSRs within Java EE 7, and your experiences during their refinement and evolution?

Little: We think that CDI is probably the most important addition to the whole Java EE architecture that we've seen in a number of years. And I don't just say that because Red Hat led the effort, because it was really a group effort. I say it because we were hearing from many vendors that it was hard at times to develop applications using the standard Java EE stack. CDI has tried to greatly simplify that. So annotations are a great addition to the language, and we're now seeing them being used in lots of different areas.

Since Java EE 6 was released, we've seen the momentum around it building. We're seeing a lot more people who didn't consider Java EE 6 on their radar screen but who are now really taking a look at it as a means of simplifying the development of enterprise applications. And they mention CDI time and time again. So it was pretty obvious to us that we wanted to lead the update to CDI in Java EE 7, because there were some things that we couldn't do in the version that went into Java EE 6. And there was also feedback that we'd gotten from users when Java EE 6 was finalized that we wanted to take into account.

In terms of the process, we've done pretty much the same this time around as we did with Java EE 6. All of our processes are open—so we have an open mailing list, all the participants see what everybody else is talking about, and we have open issue tracking. The drafts go through a very wide revision process.

Java Magazine: How has the Web Profile of Java EE 6 affected Red Hat technologies, and how will that further change for Java EE 7—in terms of new APIs like WebSocket and JSON-P?

Little: The growth and complexity of the Java EE stack is really not unique. If you look at CORBA and DCE and other standards, you see a similar phenomenon. Java EE 6 recognized that fact and introduced the concept of a Web Profile, which is essentially a stripped-down version of the full profile.

In terms of the impact that it had on Red Hat, we'd been delivering our own version of profiles to our communities for some time. There was no standard that we could present, but we could offer the ability to streamline the stack. If a customer didn't want Web services, for instance, we could provide that.

So I think the Web Profile was a really good thing to introduce as a standard. And the feedback that we've gotten has been extremely positive. We see a lot of people who might not have considered Java EE 6, who are now looking at the Web Profile, and then eventually upgrading to the full profile because some of the things that they want aren't in the Web Profile. So I think it's a really good way to on-board more users, and from a company perspective, to gain new customers.

We saw a lot of new APIs in Java EE 6, including JAX-RS and CDI, which have received a lot of attention for the benefits they've brought to developers. With

Java EE 7, we're seeing the APIs extend yet again with additions such as JSON-P and WebSocket, both of which are vital for keeping the enterprise Java stack at the forefront of waves such as cloud and mobile.

Java Magazine: A variety of cloud-related features originally intended for Java EE 7 have been deferred to Java EE 8. Can you comment on this change, how Java EE is used in cloud environments today, and how the technology will evolve in post–Java EE 7 releases?

Little: From discussions with our communities and our customers, what they wanted was the ability to offload applications from their own infrastructure onto somebody else's, but without having to reimplement. And the obvious way to do that is to ensure that the platform you've been running with your own hardware is on the cloud. From very early on, we've been working to ensure that our implementations will run on an infrastructure as a service, and therefore form a platform as a service.

When Java EE 6 came along, it actually offered us an easier route to do that, because with the profiles introduced in Java EE 6, it became easier for us to offer a standard Web Profile and a standard full-profile platform to customers who want to run their applications in the cloud. We announced our own platform as a service back in 2011, which was initially based on a pre-release of our Java EE 6–compliant application server. And then we released EAP6, which is our full implementation, where we made the announcement of OpenShift, our platform-as-a-service offering.

BIG SPEC

"We think that CDI is probably the most important addition to the whole Java EE architecture that we've seen in a number of years."

When we were originally working on Java EE 7, there were quite a lot of new JSRs that were going to be focused on making Java EE more cloud-aware. But as I said, Java EE 6 is pretty darn good in the cloud today. There are certain areas where it can be improved—in terms of modularity and multitenancy. But I agree with the deferment move on these features, so that we could get Java EE 7 out on schedule. Those features will be in the next release, and therefore Java EE 8 will be even better for evolving clouds.

Java Magazine: How will the JBoss Developer Studio IDE reflect/utilize the new offerings found in Java EE 7?

Little: We try to keep JDBS at the vanguard of getting things in front of the actual developers, so we can determine pretty quickly where the problems are. If a change in CDI isn't really right, for example, then we'll get a lot of feedback from developers through JDBS. So it's important to us that we get these features into the IDE as quickly as we possibly can, so we can get people to kick the tires.

Little says that open source communities "drive everything we do at Red Hat."

Java Magazine: Java EE 7 has pruned a number of older features (JSR 77, JSR 88, JSR 93, JSR 101, and so on). How will this be addressed in Red Hat's Java EE 7 offerings?

Little: This isn't the first time that JSRs have been pruned. What we tend to do, and what I expect we will do with these JSRs, is if they're no longer in the Java EE 7 spec, then we will remove them from our Java EE 7–compliant implementation.

But we've got the Java EE 6 implementation—AS7 [JBoss Application Server 7] is the community version, and EAP6 [JBoss Enterprise Application 6] is the product version. And EAP6 is supported for seven years. So custom-

INNOVATION'S ROLE

"One of the things that we're trying to do in the Executive Committee is to encourage innovation in upstream, open source communities, around things that we might want to see in Java EE 8 and Java EE 9."

ers who really need those features can have them in a supported way for another five or six years. For JSRs that become optional, we will typically support them in one way or another. We won't remove them until they're officially removed from the Java EE stack.

Java Magazine: In the big-picture sense of JCP involvement, how do you resolve and reconcile diverse interests when working with other member companies?

Little: Everyone understands that Java and Java EE are very important to a huge part of our industry. So we have a duty to work together in its evolution. It's really no different than working in OASIS or W3C or any of the other standards groups. Issues may come up, and you won't always get everything that you want, but you have to compromise.

Java Magazine: What are the issues and balances between preserving current standards and promoting future innovation?

Little: A big danger is standardizing too early—because all of the good standards are based on experience. If you standardize too early, you end up with APIs or data formats that are not fit for the purpose, and then nobody uses them.

But it's a trade-off—if you wait too long to standardize, then you can end up with vendor lock-in, or with multiple different ways of doing the same thing. One of the things that we're trying to do in the Executive Committee is to encourage innovation in upstream, open source communities, around things that we might want to see in Java EE 8 and Java EE 9.

Java Magazine: How do you think the JCP can better serve the Java community, and how would you suggest promoting greater participation?

Little: The work that the various Java user groups have been doing, like Adopt-a-JSR, is great. Everything we do at Red Hat is driven by open source communities—we get a lot of feedback on things that we're going to propose in the various JSRs long before we might actually include it in a document. So we try and bring our communities into the Executive Committee in that way.

Java Magazine: At last year's JavaOne, you and Cameron Purdy gave a session exploring non-Java languages running on the JVM [Java Virtual Machine]—how a Ruby developer, for example, can leverage the Java EE stack, utilizing features such as JMS or EJBs [Enterprise JavaBeans]. Can you discuss this trend, in terms of an expanding Java community/ecosystem?

Little: Over the last several years, we've seen a growing adoption of languages like JRuby and Clojure and Scala that run on the JVM. And we're developing our own language, called Ceylon. What we've been trying to do with these communities is encourage developers who might have needs for high-performance messaging, or transactions, or databases not to reinvent the wheel when there's already a perfectly good set of solutions in terms of the Java EE stack.

So we've tried to find out what they expect from a given API, say for messaging, write something with the wider developer community, and then layer our stack, or at least part of our stack, underneath that. So Ruby developers are defining the API, and we're providing a binding of that API to our JMS implementation.

The last thing we want with these new language communities is for them to reinvent the same bugs that we had 20 years ago. It's far better for them to use what we've already got and improve upon it. And we have some very successful projects—like the Ruby project, called TorqueBox. Ultimately, the developers don't even need to know that they're running on top of an application server, or that they're using a Java EE stack beneath the covers. `</article>`

Steve Meloan is a former C/UNIX software developer who has covered the Web and the internet for such publications as *Wired*, *Rolling Stone*, *Playboy*, *SF Weekly*, and the *San Francisco Examiner*. He recently published a science-adventure novel, *The Shroud*, and regularly contributes to *The Huffington Post*.

LEARN MORE

- Java Community Process
- Red Hat

Manny Pelarinos (left), senior director of engineering, and Sean Comerford, director of engineering, ESPN Technology, toss a ball around at the office.

PHOTOGRAPHY BY STAN GODLEWSKI/GETTY IMAGES

SLAM DUNK

Java takes **ESPN.com** to the winner's circle.

BY DAVID BAUM

SNAPSHOT

ESPN, INC.

espn.go.com

ESPN.com is the official Website of ESPN and a division of ESPN, Inc.

Headquarters:
Bristol, Connecticut

Industry:
Media, news, and publishing

Revenue:
US$8.2 billion

Employees:
6,500 worldwide

Java technologies used:
Java SE 1.6 and 1.7, Java EE 6

Pelarinos and Comerford in the pantheon of athletes at ESPN headquarters

It was a quintessential Kobe Bryant moment, and one that Los Angeles Lakers fans will never forget. During the final seconds of an ESPN night-time game against the Miami Heat, the Lakers were down by two as they took possession of the ball for a final play. Ron Artest quickly passed the ball to Bryant, but Dwyane Wade's aggressive defense wouldn't give him the opportunity for a shot. As the clock ticked down to almost zero, Bryant faked right, dodged left, and finally leaped backward to launch a long three-point jump shot over Wade's outstretched arms. The ball found the glass and banked into the hoop, beating the buzzer to send the Los Angeles Lakers to their eighth straight victory over the Miami Heat. The stadium erupted, and Lakers fans everywhere let out a collective cry of victory. Thanks to the work of engineers at ESPN Technology, millions of people had an up-to-the-instant account of the action through free mobile apps on their smartphones, laptops, and tablets.

As the leading provider of sports on the internet, ESPN.com provides fans with late-breaking news, game statistics, schedules, player updates, Fantasy games, and video through WatchESPN, in addition to instantaneous scores from live events like this one. National Basketball Association (NBA) games are among the thousands of events that ESPN covers. Millions of sports enthusiasts tune into the site to follow football, baseball, hockey, soccer, NASCAR racing, and many other types of matches, races, tournaments, and games. During major sporting extravaganzas such as the FIFA World Cup soccer championship, ESPN handles more than 100,000 hits per second. With help from Java, ESPN's consumer-facing Websites have been architected to handle this crushing load.

"We build and enhance systems that must scale to meet the needs of millions and millions of fans," says Manny Pelarinos, senior director of engineering at ESPN Technology, a division charged with creating, designing, investigating, implementing, and maintaining technologies used at ESPN. "One of our most important tasks from an engineering standpoint has been building our systems so that they perform extraordinarily well and scale horizontally. Java has always been in our DNA. Right from the get-go, we built the site using Java. In fact, ESPN has been using Java since before the servlet specification was even created."

INSTANT ACCESS
Engineers at ESPN Technology gave millions of people an up-to-the-instant account of the quintessential Kobe Bryant moment through free mobile apps.

Click over to ESPN.com to check out the action. You'll find scoreboards for your favorite teams, game highlights, and profiles of all the players. You can tune into current news, watch replays from recent sporting events,

ON JAVA EE 7

"With the introduction of Java EE 7 and its ease-of-use improvements in JMS, we will have an opportunity to simplify our code going forward and bring new developers up to speed more quickly on this critical area of our architecture."

and learn about everything from poker tournaments to tennis matches.

What appears to be a few simple Web pages is actually the front end of a massive digital infrastructure comprised of dozens of databases and applications spread across hundreds of servers. Pelarinos and his team use servlets to dynamically generate Web pages that can be interleaved with static Web markup content. The resulting pages are compiled and executed on ESPN's server farm via a Java Virtual Machine runtime environment.

ESTABLISHED POSITION

These fundamental Java technologies have been the foundation of ESPN's popular Website since its launch in 1997, along with a content management system that hosts important information such as sports headlines, news stories, and configuration data about how to lay out the content on the site. This nimble Java infrastructure delivers current news along with results, game highlights, and stats to ESPN's mobile experiences and Websites. It breaks down the information by sport, team, player, and columnist, and infuses it with personalized content based on consumer preferences.

"The content management system is the lifeblood of our Website," says Pelarinos. "All the textual data that comes from our editors is housed in that platform."

"We have deals with the NBA to consume data right from their systems," Pelarinos continues. "The data takes the form of a feed that gets put into a Java Message Service [JMS] queue. We have message-driven beans that marshal these feeds into our Java domain object model."

The data is persisted into another ESPN data store optimized for digital consumption using Java Persistence API (JPA), a framework for managing relational data in Java SE and Java EE applications. Moments later, it updates all the caches in the ESPN application tier to send updates to millions of eager sports fans. All the entities are updated in near real time. Another system polls a specialized API once or twice every second and pushes updates to fans.

"Fans get score updates almost immediately, and their rendition of the ticking clock represents the actual game clock in real time," says Pelarinos. "The updates happen so fast that it gives you the impression that you are right there at the stadium."

TECH MVP

To achieve this level of speed, scalability, and reliability, ESPN.com relies on a caching architecture and JPA.

"Our JPA object model allows us to push data to Web applications or mobile services powering apps," explains Sean Comerford, director of engineering at ESPN Technology.

Another important part of the architecture is JMS, a message-oriented middleware API for sending and receiving information between ESPN.com's many servers and applications. JMS controls how Java applications create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous. These are essential attributes for a site that depends on constant communication among a constantly expanding user base. Subscribers can click on updates to discover additional information about the event in question or drill into ESPN's extensive databases of content. They can sort sports information by city if they want to read local news or find out about upcoming events, games, and matches in their area.

"We're heavily utilizing Java messaging," states Comerford. "Over the last five years, we've adopted JMS as a standard interface for asynchronous messaging. It works reliably and solves many of the challenges around guaranteeing a message is processed once and only once."

Another major benefit of JMS is the publish/subscribe model, which permits ESPN to share information with internal systems and add new systems without changes to the publisher. Content about sports, teams, and players, along with news stories and updates, flows through ESPN's JMS hub and is consumed by a myriad of consumers who can take action on that content. In addition, that same JMS hub is responsible for replicating content in real time to other environments, such as development and QA.

"This approach is truly service-oriented, as it cleanly decouples the applications that publish content from those that consume it," Comerford adds. "In fact, with the introduction of Java EE 7 and its ease-of-use improvements in JMS, we will have an opportunity to simplify our code going forward and bring new developers up to speed more quickly on this critical area of our architecture."

JPA has enabled ESPN.com to achieve this same level of abstraction in the database tier. "The JPA code is easier to read, understand, and maintain," says Comerford. "Another benefit of this architecture is platform independence; we can fairly easily change our underlying data store if we need to."

FAST BREAK TO THE FUTURE

ESPN Technology is on the cusp of innovation as it continues to enhance one of the world's most viewed Websites.

"We're able to use the performance and reliability of Java to continue to power our extremely high-scale systems," says Pelarinos. "Digital media is on the leading edge to begin with, and here at ESPN.com we are always experimenting with new ideas and technologies. It keeps our work continually fresh and interesting."

"At the end of the day, our goal is to serve sports fans," Pelarinos concludes. "We continue to invest in new technologies to ensure that this happens." `</article>`

Based in Santa Barbara, California, **David Baum** writes about innovative businesses, emerging technologies, and compelling lifestyles.

Banking on Java

A new system built on Java EE brings stability and transparency to the complex world of trading financial credit assignments in Brazil.

BY PHILIP J. GILL

From left: Cláudio Silveira, Einar Saukas, and Fabiano Marques outside TIVIT headquarters in São Paulo, Brazil

PHOTOGRAPHY BY PAULO FRIDMAN

Marques, Saukas, and Silveira collaborate in the TIVIT offices.

SNAPSHOT

TIVIT
tivit.com.br

Headquarters:
São Paulo, Brazil

Industry:
IT services

Employees:
28,000

Revenue:
R$1.5 billion
(2012 estimate)

Java technology used:
Java EE 6

Brazil has one of the world's largest and most dynamic economies—but in 2010 the country's decade-long economic success story was threatened not by the global economic slowdown, but by a surprising discovery made by the country's central bank, the Banco Central do Brasil. Some Brazilian banks were reselling credit assignments—essentially loan contracts arising from consumer, financing, and leasing transactions—more than once. Such double-dealing, the central bank realized, had created a R$4.2 billion hole in the Brazilian banking system and the potential for a nationwide financial crisis.

"Apart from being obviously illegal, the actions of these banks created instability in the Brazilian financial system," says Einar Saukas, a principal architect at Java specialty firm Summa Technologies do Brasil and a consultant to TIVIT, an IT services company, both based in São Paulo. "The banks lost confidence in their ability to buy those credits, and to compensate for that, they raised interest rates to cover the increased risk," explains Saukas.

In turn, Saukas adds, "this decreased business turnover, slowed overall economic growth, and raised interest rates for millions of consumers and businesses."

To quell the uncertainty and return stability to the Brazilian economy, the central bank—which performs the same functions in Brazil as the US Federal Reserve—set out to ensure that such double-dealing could never again threaten the Brazilian banking system and economy.

Enter the Central de Cessão de Crédito (C3)—in English, Central Credit Assignment. C3 is a scalable, high-performance, distributed transaction system that now monitors the integrity of all credit assignment transactions. Written in Java, C3 was implemented in less than a year and quickly returned stability to the Brazilian banking system, says Saukas, who adds that it "reduced risk, restored confidence, lowered interest rates, and helped the economy to grow again." C3 also won one of the first Duke's Choice Awards Latin America—a special honor for Saukas, who has worked on two other Duke's Choice Award–winning projects.

For C3, the central bank contracted TIVIT, which chose to build the system on Java EE. "The Java EE platform was a natural choice for C3," explains Cláudio Silveira, TIVIT's operations director for applications. "In Brazil, Java has become a standard for banking applications."

CONSUMER CULTURE

Worldwide demand for Brazil's natural resources, agricultural products, manufactured goods, and business services has resulted in annual GDP growth exceeding 5 percent in most years since 1999 and has transformed South America's largest country into the world's sixth-largest economy as of 2011, surpassing the UK.

That growth also lifted 31 million Brazilians into the middle class, which now comprises nearly half the country's 193 million citizens. As in other thriving economies, growth of the middle class has been fueled by and has led to a boom in demand for consumer loans for cars, mortgages, and leases. Today about a third of all Brazilians have some kind of consumer debt.

To keep their coffers flush with cash for new loans, Brazilian banks—like banks in the US and elsewhere—resell

their loan contracts to other banks and financial institutions as long-term investments. "When a person goes to the bank to get a loan to buy a new car, for instance, they sign a contract to get the loan," explains Saukas, who consulted with TIVIT on the development of C3. "Since these institutions specialize in making loans, they need more money fast to make new loans to other people, so they resell these contracts to other banks."

In its first stage, C3—which operates under the supervision of the central bank—was designed to monitor all credit assignment operations between participating banks and to record whenever one bank sold a credit to another financial institution. "That negotiation has to be registered with C3, which detects instantly if someone is trying to sell the same credit to two different banks," says Saukas.

In the second stage, installed in 2012, additional functions were added that allow C3 to negotiate and control the actual sale through the central bank's payment system, known as CETIP. "Through C3, the buyer and seller negotiate the sale of those credits and the buyer is able to confirm all the information from the other side," says Saukas. "The system controls the ownership, and payment for the credits is actually controlled by CETIP and the central bank. C3 transfers the ownership from one bank to another, so the system is now fully automated."

MODERN ICON

Brazil's federal capital, Brasília, is considered an iconic masterpiece of modern architecture.

PIPELINE ARCHITECTURE

TIVIT's choice of Java EE, the Java platform for scalable, high-performance, transaction-oriented applications that are also highly secure and reliable, was a straightforward one. In addition to its widespread popularity in Brazilian banking circles, Java EE was the only platform that integrates the comprehensive set of APIs needed for a system as complex as C3; these include the Java Message Service (JMS) and Java Open Transaction Manager (JOTM).

In addition, Java EE has widespread support from the open source community; open source technologies and development tools used to develop C3 include XA distributed transaction protocol, the Spring Framework, JBoss Serialization, Apache Commons, dom4j, XStream, Hudson, Sonar, Nexus, JUnit, Mockito, Maven, and Eclipse.

The platform was also chosen because TIVIT had developed an earlier system with similar high-volume requirements using Java EE: the Débito Direto Autorizado (DDA), or Authorized Direct Debit, a real-time electronic billing and payment system. DDA, also written in Java, exchanges payment data between participating Brazilian banks and allows Brazilians to both receive and pay bills electronically. The system began operating in October 2009 and currently has more than 8.6 million registered bank customers and more than 729 million registered bills, processing 10 percent of the payment volume in Brazil.

The C3 system was developed and put into production in less than a year, because TIVIT was able to reuse DDA's

Marques readies a presentation while Saukas and Silveira discuss projects.

Java in Brazil

Besides being one of the world's largest economies, Brazil also boasts one of the world's largest Java communities—it's home to more than 142,000 Java developers, according to Java.net.

In fact, Brazil is home to the world's largest and second-largest Java user groups (JUGs), each with more than 40,000 members. The largest is SouJava, based in São Paulo, the country's largest city and an international business and financial center. The second largest is DFJUG, based in Brasília, the modernist federal capital district in the country's central highlands.

Einar Saukas, a principal architect at São Paulo–based Summa Technologies do Brasil, is a founder of SouJava and has worked on three Duke's Choice Award–winning projects, including the Central de Cessão de Crédito (C3) distributed transaction system. Saukas explains that Java's popularity is due in part to the Brazilian government's encouragement. "A few years back, the Brazilian government realized it had millions of documents in a proprietary format, Microsoft Word," Saukas says. "The government did not want to risk vendor lock-in, so officials encouraged the country's technology sector to embrace open standards such as Java."

Einar Saukas

In addition to C3, Saukas has helped develop two other major Java applications that have won Duke's Choice Awards—the Unified Health System for the Brazilian federal government in 2003 and the City of São Paulo Integrated Patient Scheduling System in 2005.

underlying software infrastructure. "Using the same software infrastructure lower layer as DDA reduced the development effort considerably," says Fabiano Marques, TIVIT's chief Java architect. "Because of this, the entire C3 project, from initial concept to production, could be done in the same year. Most of the development for the initial phase of C3 was executed in under two months."

That software infrastructure layer includes a unique "pipeline" architecture specifically designed to accommodate the high volume of transactions and large amounts of data; this architecture connects to a back-end DB2 database and a hierarchical storage management system.

BIG MONEY

Brazil was the world's sixth-largest economy in 2011, surpassing the United Kingdom.

"To allow the flexibility needed by the complex Brazilian banking system, the C3 system is structured as transactional Enterprise JavaBeans [EJBs] connected through JMS queues," explains Saukas. "Requests are processed by a sequence of EJBs working as a 'pipeline,' in which each bean reads from an internal JMS queue, performs a certain task or tasks, and forwards the results to the next bean. The EJBs use JOTM to guarantee transactional consistency and integrity between JMS and DB2 operations through XA."

"Integration through JMS queues," Saukas continues, "also ensures automated load balance and fault tolerance—since multiple bean instances, from different servers, process requests from each JMS queue as they become available, without the need for a centralized distribution logic."

To date, C3 has managed more than 700 million credit installments worth more than R$127 billion. More importantly, C3 has helped return stability and certainty to the Brazilian financial system. "The Brazilian National Banks Federation and the central bank both credit the system for decreasing operational risks in credit assignments and allowing financial institutions to lower interest rates on loans, financing, and leasing operations," says Saukas, noting that this has had a definite positive impact on ordinary Brazilians as well. "The lower rates directly benefit the 61 million Brazilian citizens who currently have consumer loans." `</article>`

Philip J. Gill is a San Diego, California–based freelance writer and editor who has followed Java technology for 20 years.

MAX BONBHEL

BIO

Part 1

Three Hundred Sixty–Degree Exploration of Java EE 7

Learn how to use the new and important APIs added to Java EE 7.

This article is the first part of a three-part series, and it will demonstrate how to use Java EE 7 improvements and newer Web standards such as HTML5, WebSocket, and JSON processing to build modern enterprise applications.

In this part, we will see in detail how to use NetBeans IDE 7.3 and GlassFish 4 to build a complete three-tier end-to-end application using the following technologies, which are built into Java EE 7:

- Java Persistence API (JPA) 2.1 (JSR 338)
- JavaServer Faces (JSF) 2.2 (JSR 344)
- Contexts and Dependency Injection (CDI) 1.1 (JSR 346)

Note: The complete source code for the application designed in this article can be downloaded here.

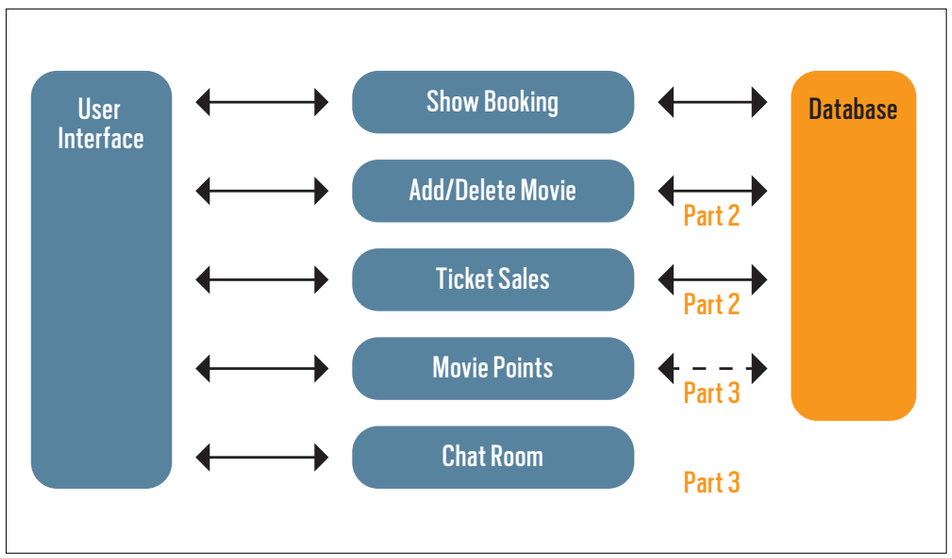


Figure 1

PHOTOGRAPH BY ALLEN MCINNIS/GETTY IMAGES

What Is Java EE?

Java Platform, Enterprise Edition (Java EE) is a dedicated Java platform for developing and running enterprise applications. Java EE provides all you need to create large-scale, multi-tiered, scalable, reliable, and secure network applications for business.

Prerequisites

Enough explanation is provided in this guide to get you started with the sample application; however, this article is not a comprehensive Java EE tutorial. Readers are expected to know about basic Java EE concepts, such as Enterprise JavaBeans (EJB), JPA, and CDI. *The Java EE 6 Tutorial* is a good place to learn all these concepts.

Download and install the following software, which was used to develop the application described in this article:

- JDK 7
- NetBeans IDE 7.3 or higher "All" or "Java EE" version
- GlassFish 4.0 b80

Note: This article was tested using the latest version of NetBeans IDE (version 7.3, at the time this article was written).

It is based on work created and shared by Arun Gupta, Java evangelist at Oracle.

Create a Real-Life Enterprise Application with NetBeans IDE 7.3

In this practical section, we will build a real-life Java EE 7 enterprise application step by step so that you can learn quickly the basics about the new and important APIs

FLOW	DESCRIPTION
USER INTERFACE	WRITTEN ENTIRELY IN JSF
▪ SHOW BOOKING ▪ SHOW TIME FOR THE SELECTED MOVIE AND ALL MOVIES ▪ CONFIRM THE SELECTION	USES LIGHTWEIGHT EJB BEANS TO COMMUNICATE WITH THE DATABASE USING JPA

Table 1

improved in Java EE 7, such as JPA 2.1, JSF 2.2, and CDI 1.1.

What we are going to do is create a typical three-tier Java EE 7 Web application that allows customers to view the show times for a movie showing at a seven-theater Cineplex and then to make reservations.

Figure 1 shows the features that will be added and explained in this article and the upcoming Part 2 and Part 3.

Table 1 details the components designed in this article and the selected technology used in each component's implementation.

Specifically, we will perform the following tasks:

- Install and configure GlassFish and NetBeans IDE 7.3.
- Load a sample Maven application by following the instructions and guidance.
- Review the configured datasource and generate database artifacts, such as tables.
- Create the Web pages that allow a user to book a particular movie show in a theater.
- Add and update existing code in order to demonstrate the usage of different technology stacks in the Java EE 7 platform.
- Run and test the sample application.

Install and Configure NetBeans IDE 7.3 and GlassFish 4

Let's establish our development environment by downloading and installing the latest version of NetBeans IDE and GlassFish.

1. Download and install the latest version of NetBeans IDE:
   a. Download the All or Java EE version of NetBeans IDE 7.3 or higher.
   b. Follow the installation instructions here.
2. Download the GlassFish server and configure it in NetBeans IDE. To be sure we use GlassFish as our application server, we are going to specify it in NetBeans IDE:
   a. Download GlassFish 4.0.
   b. Unzip the compressed file in the location where you want the glassfish4 folder to be placed.
   c. Open NetBeans IDE and click the **Services** tab.
   d. Right-click **Servers** and then choose **Add Server**, as shown in **Figure 2**.
   e. Select **GlassFish Server 3+** in the **Add Server Instance** wizard, as shown in **Figure 3**.
   f. Type GlassFish4.0b80 in the **Name** field, and click **Next**.
   g. Click **Browse**, browse to the location where you unzipped the GlassFish files, and select the glassfish4 folder.
   h. Click **Finish**.

Bravo! Now your environment is ready for fun.

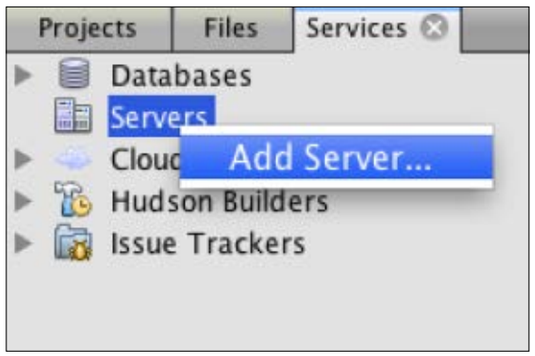


Figure 2

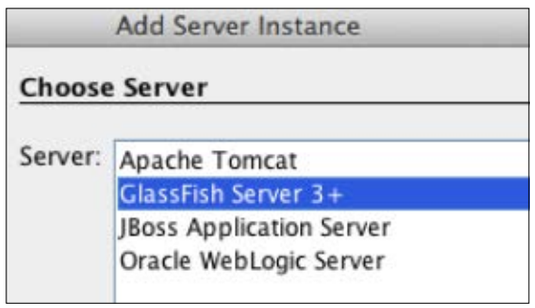


Figure 3

Walk Through the Sample Application

In this section, we will download the sample application. Then we will walk through the application

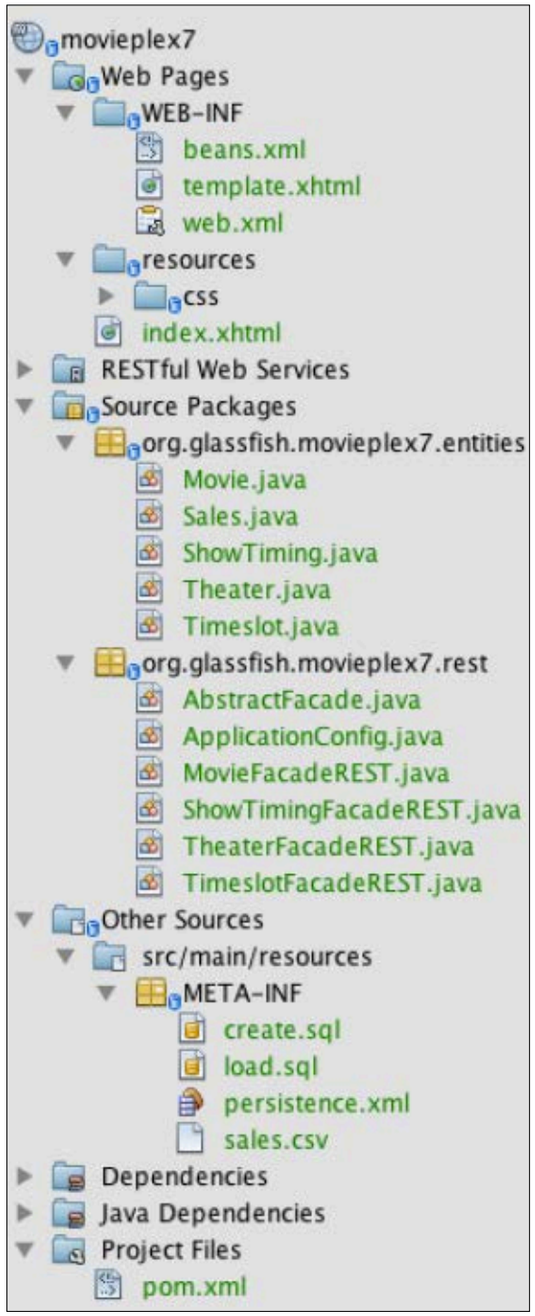


Figure 4

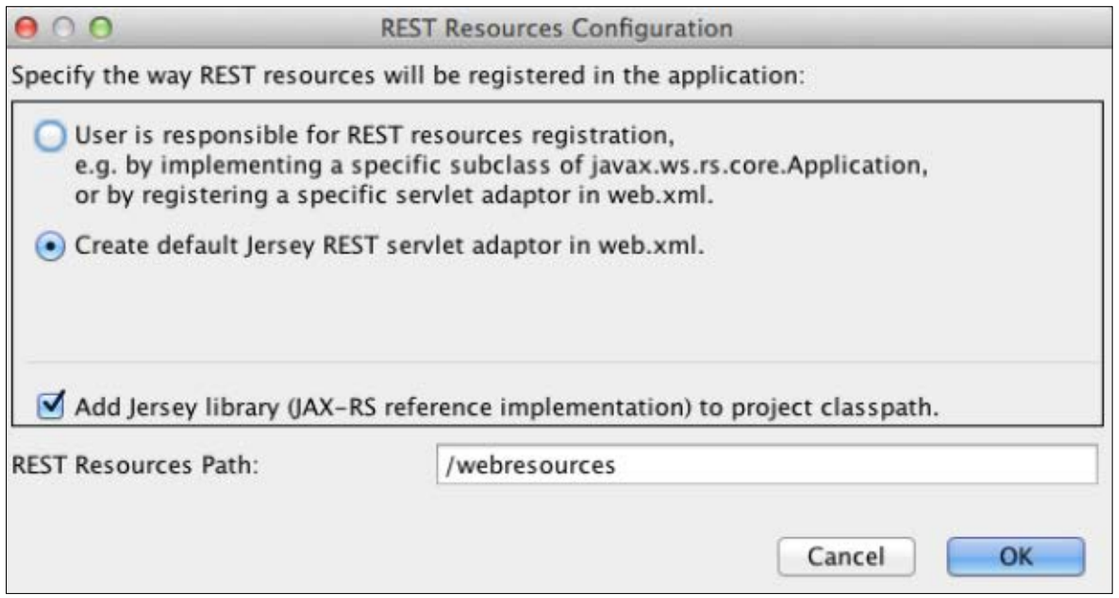


Figure 5

so we get an understanding of the application architecture.

1. Download the initial NetBeans project:
   a. Download the sample application here.
   b. Unzip the compressed file in the location where you want the movieplex7 folder to be placed.
   c. Open NetBeans IDE and from the **File** menu, choose **Open Project**.
   d. Select the unzipped directory, and click **Open Project**. The project structure is shown in **Figure 4**.
   e. Opening the project will prompt you to create a configuration file to configure the base URI of the REST resources bundled in the application, as shown in **Figure 5**. The application already contains a source file that provides the needed configuration, so click **Cancel** to dismiss this dialog box.
2. Take a look at the Maven configuration:
   a. Open the **Project Files** node of the **movieplex7** project and double-click the **pom.xml** file.
   b. In the pom.xml file, you can see that the javaee-api is specified as a project dependency, as shown in **Figure 6**. This will ensure that Java EE 7 APIs are retrieved from Maven. Notice that a particular version number is specified and this must be used with the downloaded GlassFish 4.0 build.

```
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0-b80</version>
  </dependency>
</dependencies>
```

Figure 6

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="movieplex7PU" transaction-type="JTA">
    <!--
<jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    -->
    <properties>
      <property
        name="javax.persistence.schema-generation.database.action"
        value="drop-and-create"/>
      <property
        name="javax.persistence.schema-generation.create-source"
        value="script"/>
      <property
        name="javax.persistence.schema-generation.create-script-source"
        value="META-INF/create.sql"/>
      <property
        name="javax.persistence.sql-load-script-source"
        value="META-INF/load.sql"/>
      <property
        name="eclipselink.deploy-on-startup"
        value="true"/>
      <property
        name="eclipselink.logging.exceptions"
        value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Figure 7

Review the Datasource and Generate the Database Schema

In this section, we will see in detail how the components of the application are using JPA to communicate with the database.

1. Configure the datasource:
   a. Open the **Other Sources** node of the **movieplex7** project and then expand the **src/main/resources** node.
   b. Open the **META-INF** node,

and double-click **persistence.xml**. By default, NetBeans opens the file in Design View.

c. Click the **Source** tab to view the XML source file, which is shown in **Figure 7**.

Notice that <jta-data-source> is commented out; that is, no datasource element is specified. This element identifies the JDBC resource to connect to in the runtime environment of the underlying application server.

The Java EE 7 platform defines a new default datasource that must be provided by the runtime. This preconfigured datasource is accessible under the JNDI name java:comp/DefaultDataSource.

The JPA 2.1 specification says that if neither the jta-data-source element nor the non-jta-data-source element is specified, the deployer must specify a Java Transaction API (JTA) datasource or the default JTA datasource must be provided by the container.

d. For GlassFish 4, the default datasource is bound to the JDBC resource jdbc/__default. Clicking back and forth between the **Design** view and the **Source** view might prompt the warning shown in **Figure 8**. This will get resolved when we run the application. Click **OK** to dismiss the dialog box.

2. Generate the database schema. The scripts to generate the schema are located in the META-INF directory. Because the location of these scripts is specified as a URL, the scripts can be loaded from outside the WAR file as well.

a. Open the **Other Sources** node of the **movieplex7** project and then expand the **src/main/resources** node.

b. Open the **META-INF** node and execute the create .sql script to generate the database schema shown in **Figure 9**.

Figure 8

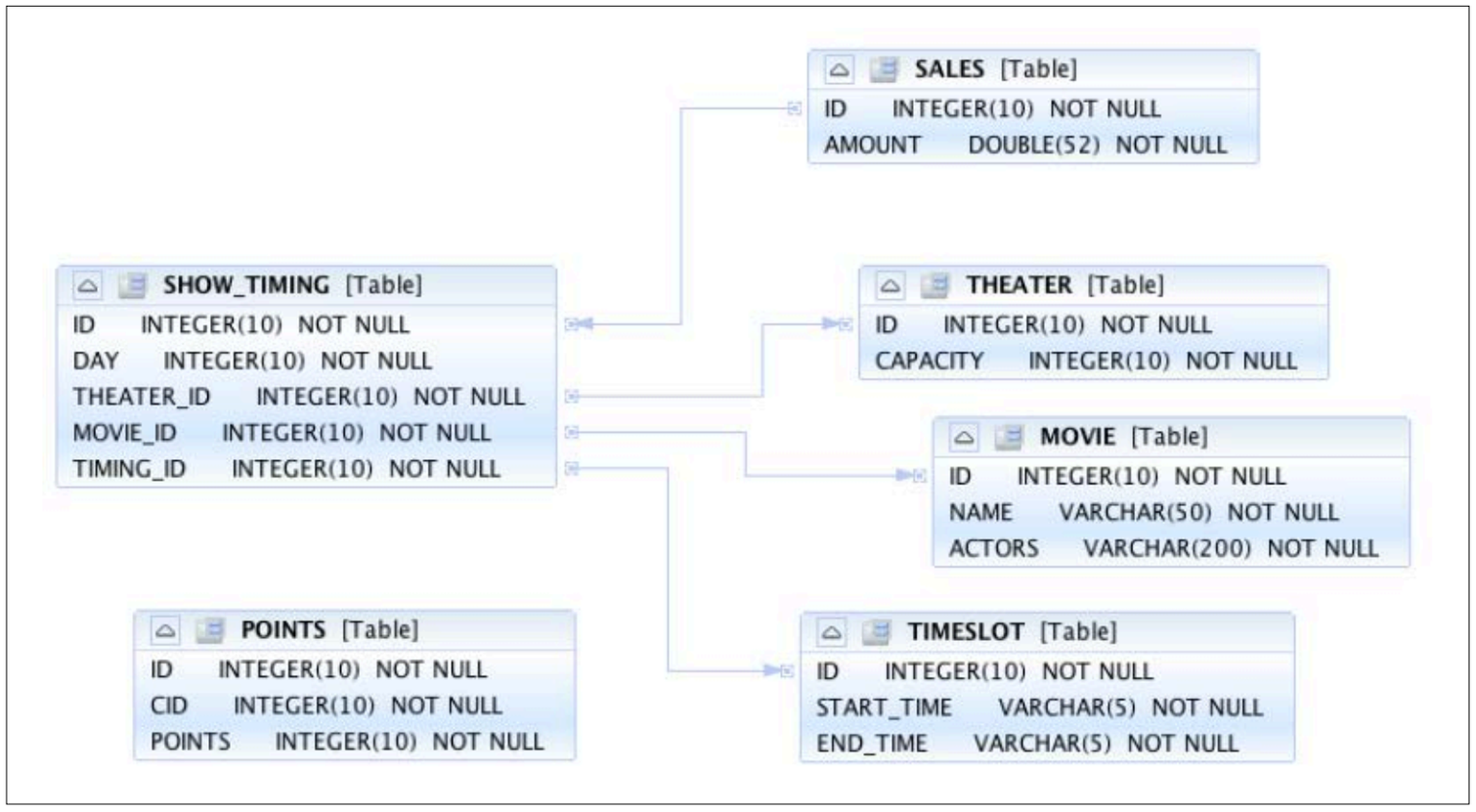


Figure 9

JPA 2.1 introduces a new range of javax.persistence.schema-generation.* properties that can be used to generate database artifacts such as tables, indexes, and constraints in a database schema. This feature allows your JPA domain object model to be directly generated in a database. The generated schema might need to be tuned for the production environment. The properties, their meanings, and possible values are explained in **Table 2**.

Create the Web Pages to Allow a User to Book a Movie

In this section, we will create Web pages that allow a user to book a particular movie show in a theater. We are going to use a new feature introduced with version 2.2 of JSF.

JSF 2.2 introduces Faces Flow, which provides an encapsulation of related views/pages with application-defined entry and exit points. Faces Flow borrows core concepts from Oracle ADF task flows, Spring Web Flow, and Apache MyFaces CODI.

Our application will build a flow that allows the user to make a movie reservation. The flow will contain four pages whose purpose is the following:

- Booking page: Displays the list of movies
- Show Times page: Displays the list of available show times for the selected movie
- Confirmation page: Allows a user to create a reservation for a selected movie, show time, and theater
- Print page (Reservation Confirmed): Shows the reservation information

We will also use the WEB-INF/template.xhtml file to define the template of the Web page with a header, a left navigation bar, and a main content section. The file is located in the WEB-INF directory so that the template is accessible only from the pages that are bundled with the application. If the template were bundled with the rest of the pages, it would be accessible outside the application and, thus, other external pages could use it as well.

PROPERTY	MEANING	VALUES
javax.persistence.schema-generation.database.action	SPECIFIES THE ACTION TO BE TAKEN BY THE PERSISTENCE PROVIDER WITH REGARD TO THE DATABASE ARTIFACTS	none, create, drop-and-create, drop
javax.persistence.schema-generation.create-source	SPECIFIES WHETHER THE CREATION OF DATABASE ARTIFACTS IS TO OCCUR ON THE BASIS OF THE OBJECT/RELATIONAL MAPPING METADATA, A DATA DEFINITION LANGUAGE (DDL) SCRIPT, OR A COMBINATION OF THE TWO	metadata, script, metadata-then-script, script-then-metadata

Table 2

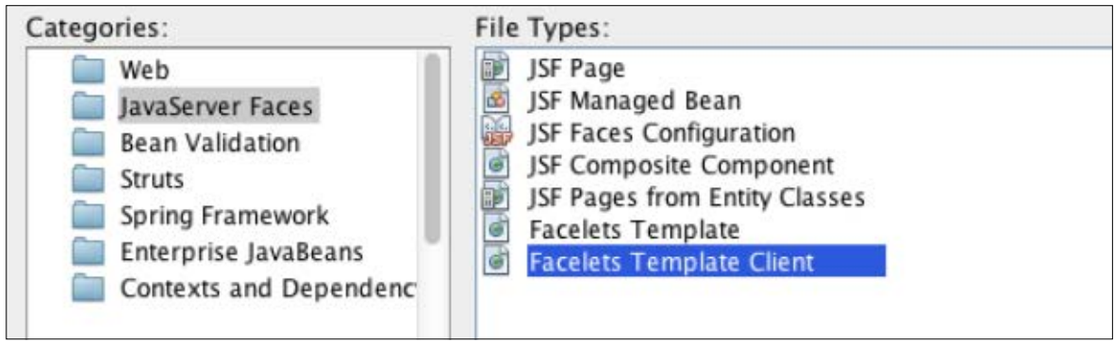


Figure 10

1. Create the Booking page:
   a. In NetBeans IDE, right-click the **Web Pages** node of the **movieplex7** project and select **New** and then **Folder**.
   b. Type booking in the **Folder Name** field, and click **Finish**.
   c. Right-click the newly created folder, and select **New** and then **Other**.
   d. In the dialog box, select **JavaServer Faces** in the **Categories** section and **Facelets Template Client** in the **File Types** section, as shown in **Figure 10**. Then click **Next**.
   e. Type booking in the **File Name** field.
   f. Click **Browse** next to **Template** and then select the **template.xhtml** file under the **WEB-INF** node.
   g. Click **Finish**.
2. Edit and modify the booking.xhtml file:
   a. Change the <ui:define> section, so it looks like that shown in **Listing 1**.

The code in **Listing 1** builds an HTML form that displays the list of movies as radio buttons. The chosen movie is bound to #{booking.movieId}, which will be defined as a flow-scoped bean. The value of the action attribute on commandButton refers to the next view in the flow, which, in our case, is showtimes.xhtml in the same directory.

   b. Click the hint (denoted with a yellow lightbulb) and click the suggestion to add a namespace prefix. Do the same for the c: and f: prefix, as shown in **Figure 11**.

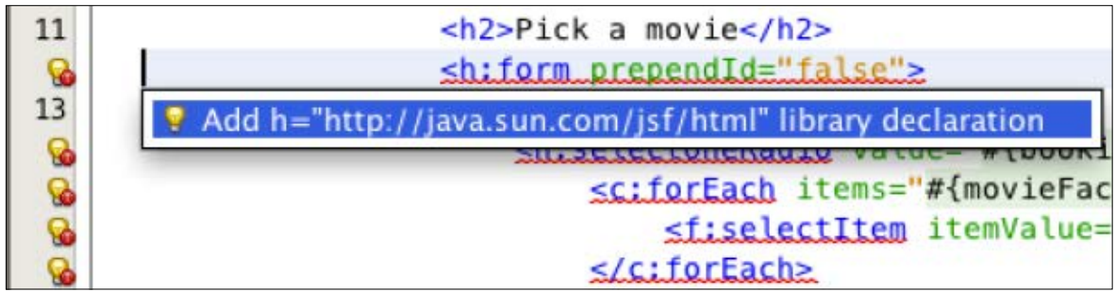


Figure 11

3. Create a booking Java bean and define it as a flow:
   a. Right-click the **Source Packages** node of the **movieplex7** project and select **New** and then **Java Class**.
   b. Type Booking in the **Name** field and enter org.glassfish.movieplex7.booking in the **Package Name** field.
   c. Add the @Named class-level annotation to make the class Expression Language (EL)–injectable.
   d. Add @FlowScoped("booking") to define the scope of bean as the flow. The bean is automatically activated or passivated as the flow is entered or exited.

      Add the following property in the Booking.java file: int movieId.
   e. Generate getters/setters by right-clicking anywhere in the edited source code.
   f. Select **Insert Code**, then select **Getter and Setter**, and select the field.
   g. Add the public String getMovieName() method, as shown in **Listing 2**. This method will return the movie name based upon the selected movie.
   h. Inject EntityManager in this class by adding the following code:

```
@PersistenceContext
EntityManager em;
```

      Alternatively, a movie ID and name can be passed from the selected radio button and parsed in the backing bean. This will prevent an extra trip to the database.
4. Create the Show Times page:
   a. Repeat Steps 1 and 2 to change the content of the <ui:define> section, as shown in **Listing 3**

      The code in **Listing 3** builds an HTML form that displays the chosen movie's name and all the show times. #{timeslotFacadeREST.all} returns the list of all the movies and iterates over them using a c:forEach loop. The ID and start time of the selected show are bound to #{booking.startTime}. The (value="Back") command button allows the user to go back to the previous page, and the (value="Confirm") command button takes the user to the next view in the flow, which is confirm.xhtml, in our case.

      Typically, a user will expect to see only the show times for the selected movie, but all the show times are shown here. This allows us to demonstrate going back and forth within a flow if an incorrect show time for a movie is chosen.
5. Modify the Booking.java file:
   a. Add the properties and methods that will parse the values received from the form, as shown in **Listing 4**.
   b. Add another method that will find the first theater available for the chosen movie and show time, as shown in **Listing 5**.
6. Create the Confirmation page:
   a. Repeat Steps 1 and 2 to change the content of the <ui:define> section, as shown in **Listing 6**.

LISTING 1 LISTING 2 / LISTING 3 / LISTING 4

```
<ui:define name="content">
  <h2>Pick a movie</h2>
  <h:form prependId="false">

   <h:selectOneRadio value="#{booking.movieId}"
layout="pageDirection" required="true">
    <c:forEach items="#{movieFacadeREST.all}" var="m">
     <f:selectItem itemValue="#{m.id}" itemLabel="#{m.name}"/>
    </c:forEach>
   </h:selectOneRadio>

   <h:commandButton id="shows" value="Pick a time"
action="showtimes" />}"/>
  </h:form>
</ui:define>
```

Download all listings in this issue as text

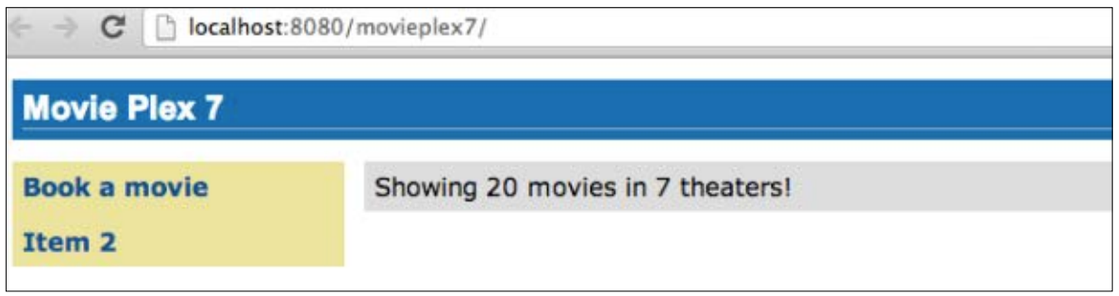


Figure 12

The code in **Listing 6** displays the selected movie, the show time, and the theater, if available. The reservation can proceed if all three are available. print.xhtml, identified by the action of the commandButton with the Book value, is the last page, which shows the confirmed reservation.

actionListener can be added to the commandButton to invoke the business logic for making the reservation.

7. Create the Print page (Reservation Confirmed):

a. Repeat Steps 1 and 2 to change the content of the <ui:define> section, as shown in **Listing 7**.

The code in **Listing 7** displays the movie name, the show times, and the selected theater. The commandButton is an exit point from the flow and takes the user back to the home page of the application.

8. Configure the flow:
Since all the pages (booking.xhtml, showtimes.xhtml, confirm.xhtml, and print.xhtml) are in the same directory, the runtime needs to be informed that the views in this directory are to be treated as view nodes in a flow.

a. Right-click the **Web Pages/booking** folder, and select **New** and then **Other**.
b. Select **XML** and then **XML Document**.
c. Type booking-flow in the **Name** field and click **Finish**.
d. Modify the generated XML file booking-flow.xml so it looks like **Listing 8**.

9. Invoke the flow in the template.xhtml file. To do this, edit the WEB-INF/template.xhtml file and change the commandLink from the code shown in **Listing 9** to the code shown in **Listing 10**.

commandLink renders an HTML anchor tag that behaves like a form submit button. The action attribute points to the directory where all views for the flow are stored. This directory already contains booking-flow.xml, which defines the flow of the pages.

LISTING 5 LISTING 6 LISTING 7 LISTING 8 LISTING 9 LISTING 10

```
public String getTheater() {
 // for a movie and show
 try {
  // Always return the first theater
  List<ShowTiming> list =
     em.createNamedQuery
("ShowTiming.findByMovieAndTimingId",
      ShowTiming.class)
    .setParameter("movieId", movieId)
    .setParameter("timingId", startTimeId)
    .getResultList();
  if (list.isEmpty())
   return "none";
  return list
   .get(0)
   .getTheaterId()
   .getId().toString();
 } catch (NoResultException e) {
  return "none";
 }
}
```

Download all listings in this issue as text

Test Our Sample Application

Now it's time to test our movie reservation application. We will invoke all the pages we created.

1. Run the project by right-clicking the **movieplex7** node and then selecting **Run**. The browser shows the updated output, as shown in **Figure 12**.
2. Click **Book a movie** to see the Booking page, as shown in **Figure 13**.

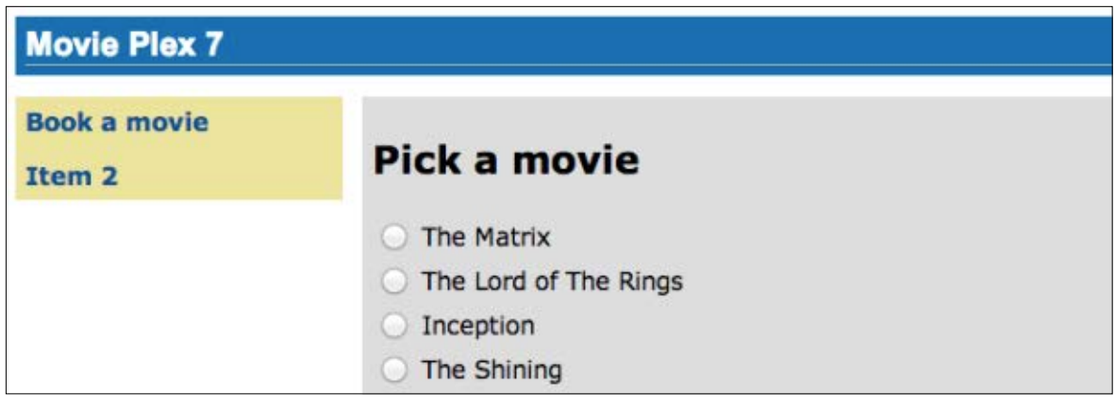


Figure 13

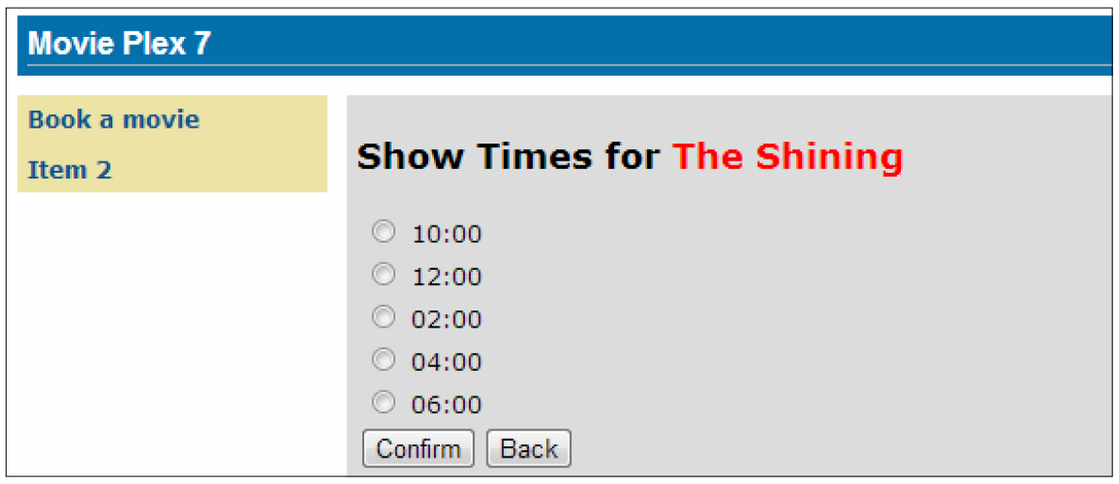


Figure 14

3. Select a movie, for example, *The Shining*, and click **Pick a time** to see the Show Times page, which is shown in **Figure 14**.
4. Pick a time, for example, 04:00, and click **Confirm** to see the Confirmation page, as shown in **Figure 15**.
5. Click **Book** to confirm the reservation and see the Reservation Confirmed page, which is shown in **Figure 16**. Bravo!

Conclusion

Java EE 7 includes a number of interesting new features. In this article, we created a three-tier Web application using the following technology, which is part of Java EE 7:

- Java Persistence API 2.1 (JSR 338)
- JavaServer Faces 2.2 (JSR 344)
- Contexts and Dependency Injection 1.1 (JSR 346)

Our sample movieplex7 Web application was designed so that each of the new APIs matches a specific need, for example, JavaServer Faces was used to implement the user interface and Enterprise JavaBeans and the Java Persistence API were used to implement the "show booking" functionality.

In the next articles in this series, we will improve the movieplex7 Web application by adding three new functionalities—Add/Delete Movie, ticket sales, movie points, and chat room—using important components and APIs provided by Java EE 7, such as Batch Applications for the Java Platform 1.0 (JSR 352), Java API for JSON Processing 1.0 (JSR 353), Java API for WebSocket 1.0 (JSR 356), Java Transaction API 1.2 (JSR 907), and JAX-RS 2.0 (JSR 339). `</article>`

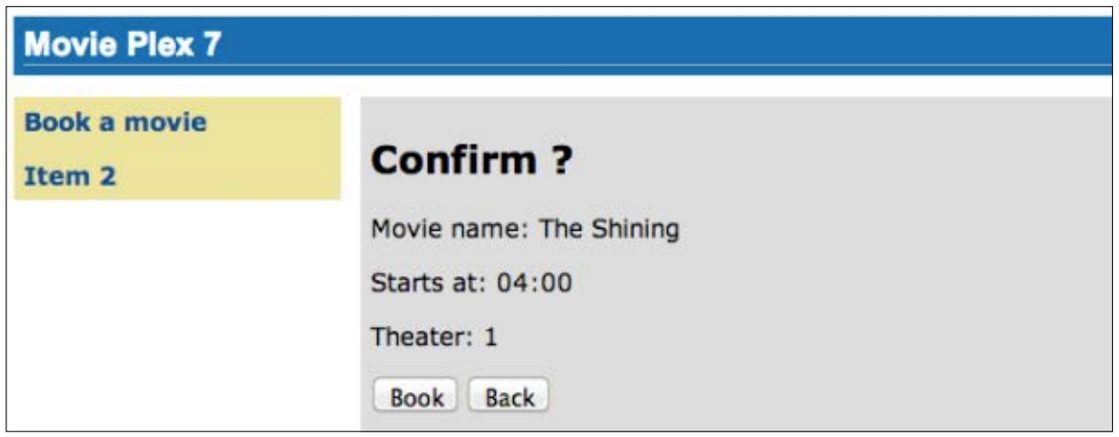


Figure 15

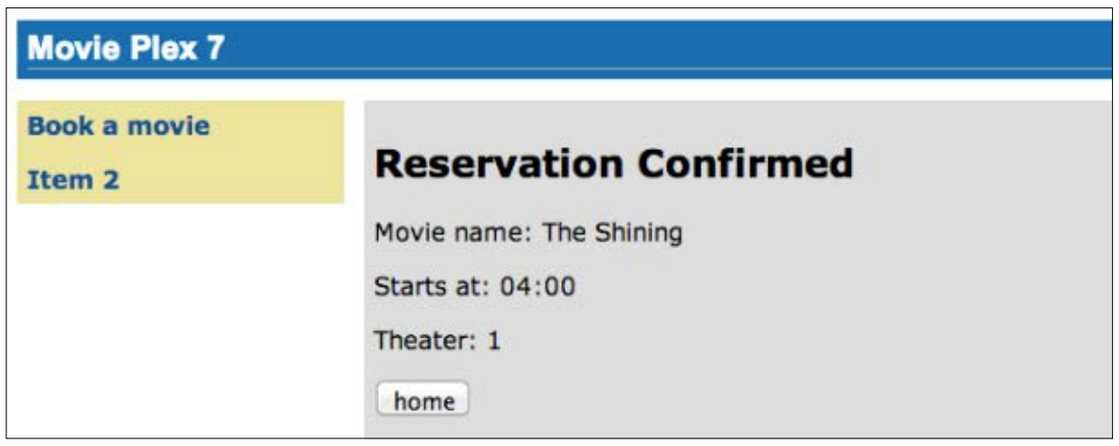


Figure 16

LEARN MORE

- GlassFish resources
- "Java EE 7—Introduction"
- "Java EE 7 Hands-on Lab"
- Java EE Platform specification

Part 2

The Advanced Java Compiler API

Build a full graphical code analysis tool that can understand Java code at the statement level.

JOSH MARINACCI

BIO

Part 1 of this series introduced the Java Compiler API and showed how to perform basic code parsing. Here, in Part 2, we will use what we've learned to build a full graphical code analysis tool that goes beyond the public API and can actually understand Java code at the statement level.

Because the compiler APIs are some of the most underdocumented parts of the JDK, what I will demonstrate here might not work in all circumstances. We will only use public APIs, but we will stress them to the limit.

We will build a full code analysis tool that will display a parsed codebase in a three-paned view. Each pane has a list: one for packages, one for classes, and one for methods and constructors. Using this browser, you can navigate through the entire source tree.

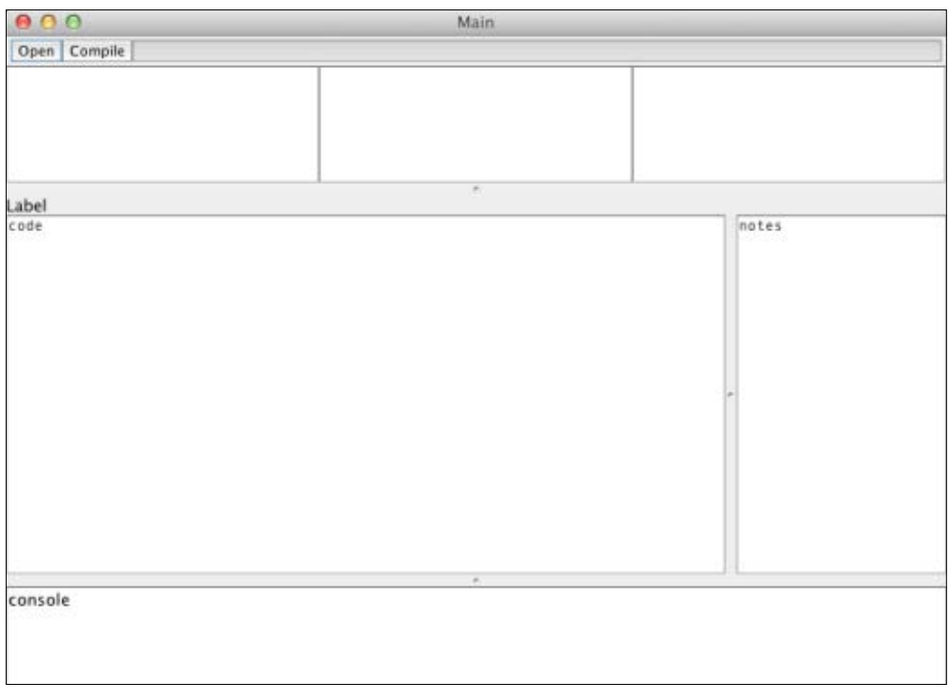


Figure 1

When a package is selected, the class list is updated to show only the classes in that package. The method list behaves the same way for the selected class. When you select a method, the source of that method is displayed in the main text area below the three panes. To the right is a smaller text area for notes. On the bottom is a console area to show status and errors. The finished GUI looks like **Figure 1**.

Note: Because this article is about the Java Compiler API, not Swing, I will skip the code for the GUI. It is standard GridBagLayout Swing code that I built with the IntelliJ IDEA GUI Builder. If you want to see the full source code, download this file.

The Code Model

This is a big project, so let's start small. Everything in the tool works on an abstraction of the source codebase, so we need to build a *core model*. The actual tree structures provided by the Java Compiler API contain all the information we need, but they are not in a convenient package, so we will build our own wrapper classes.

First is the MethodModel, which represents everything we know about a single method. A method has a name, the body (the actual statements inside the method), and a list of parameter variables and their types. The definition looks like **Listing 1**. I've also included two longs for start and end as well as a reference to the CompilationUnitTree. I will explain these later.

Next is the *class model*, which represents a single class in the codebase. It has

PHOTOGRAPH BY
CHRIS PIETSCH/GETTY IMAGES

a class name and two collections of MethodModel representing the methods within this class. I am using both a list and a method map because I need the methods to be ordered (which the list provides) and also addressable by name (which the map provides).

Next is the ClassModel definition (see **Listing 2**). There are also accessor and utility methods for getting a particular method by name or index and for sorting the methods. These accessors are not shown. See the full source for details.

The methodMap also ensures that the methods are unique. Java allows two methods to have the same name, so we can't use just the name as the key for the hash map. We need something unique. To fix this, MethodModel.getUniqueName() will return a unique key based on the method name as well as the names of the parameters, which must be unique.

The PackageModel and CodebaseModel classes look very similar to the ClassModel; they are just higher levels (see **Listing 3**). PackageModel represents a single package. The CodebaseModel represents the entire codebase full of packages. Again, utility and accessor methods are not shown.

TO THE CORE

Everything in the tool works on an **abstraction of the source** codebase, so we need to build a core model.

Full Code Processing

In the previous article, we used an annotation processor to analyze the codebase. We will again use an annotation processor, but this time we need to add an extra API. The Annotation Processing API goes down only to the method level. To go inside the methods, we need an optional (but supported) API called the Java Compiler Tree API. This API provides a tree structure that represents every language construct in Java, including if statements, for loops, and try blocks. It also has utility classes to assist with parsing tasks.

To start, we will build a new processor called a FullCodeProcessor, as shown in **Listing 4**. As before, it is annotated as a processor for all types. I have overridden the init function to create an instance of the Trees class. This class will convert objects between the standard javax.lang.model API and the com.sun.source.tree API.

As before, we implement the process method. This time, we will create a TreePathScanner instead of an ElementVisitor6. A TreePathScanner is similar to the other visitor classes, but it works with the Java Compiler Tree API. It also keeps track of where we are in the tree, so at any time, we can get the path back to the root. The code in **Listing 5** starts the processing.

For each root element (pack-

LISTING 1 LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5

```
public class MethodModel implements Comparable<MethodModel> {
  public String name = "";
  public String body = "";
  public CompilationUnitTree compUnit;
  public long start;
  public long end;
  private List<String> varTypes = new ArrayList<String>();
  private List<String> varNames = new ArrayList<String>();
```

Download all listings in this issue as text

age), we add a new package model to the codebase and then get the matching tree path. Then we call the scanner on the tree path. Once scanning is complete, we sort the classes in each package and the methods in each class. Our implementation of TreePathScanner (CodeScanner) does the real scanning work (see **Listing 6**). First, it overrides visitClass to keep track of which class we are inside as well as add to the codebase model.

Next, it overrides visitMethod to create a model for each method (see **Listing 7**). Inside, it loops over all the method's parameters to add a name and type for each. The type could be either a primitive or an identifier. An identifier is a reference to some real class, whereas a primitive is just one of the standard Java primitives (int, float, boolean, and so on). For each case, it calculates a type string.

visitMethod also calculates the source position to pull out the method body. Let's back up for a second. A codebase is composed of CompilationUnit instances, which usually map to the actual .java files on disk. Each .java file is just a big array of characters. The source positions are the offsets into this character array where a particular method starts and ends. By knowing these offsets, we can pull out the actual source code that defines each method. The Java Compiler Tree API defines a SourcePositions class for getting these values, which are obtained from Trees. The code in **Listing 8** gets the current tree path, finds the start and end offsets, and then saves the method body into the model.

Hooking Up the GUI

Now the FullCodeProcessor is complete. It is run in the callback method for the Open button. The code in **Listing 9** looks very similar to what we did last time, but it has a few very important differences. Let's walk through it.

First, it creates a ConsoleHandler, which will report any errors. We will look at the code for ConsoleHandler in a moment. Then, it gets the default compiler and file manager. Notice that the ConsoleHandler is passed into the getStandardFileManager method. This will let the file manager report any file errors to the handler. Next, it sets the output directory for compiled classes to /tmp and it sets the source path to a project I already have on my computer. (You would change this to the path of the source code you want to analyze.)

To actually load the code, we need a list of all the source files using filemanager.list(). The kinds variable contains a Set of JavaFileObject.Kind attributes. This tells the file manager which kinds of files to list.

Finally, we create the actual compiler task. Notice that this time, the task is cast to a JavacTask. JavacTask is a subclass of CompilerTask provided by the Java Compiler Tree API. It does everything the CompilationTask does, plus a few extra things. Rather than invoking the compiler as a whole, you can separately call the parse, analyze, and generate phases of the compiler, as shown in **Listing 10**. This task is specific to Oracle's implementation of javac, so it will not work with other compilers, such as Jikes.

Now that we have a CompilerTask, we can set the processor, add the ConsoleHandler to capture errors, and then parse and ana-

LISTING 6 LISTING 7 LISTING 8 LISTING 9 LISTING 10

```
private class CodeScanner extends TreePathScanner<Void, Void> {
  public Void visitClass(ClassTree classTree, Void aVoid) {
    String clsName = classTree.getSimpleName().toString();
    currrentClass = new ClassModel();
    currrentClass.name = clsName;
    currentPackage.add(currrentClass);
    return super.visitClass(classTree, aVoid);
  }
```

Download all listings in this issue as text

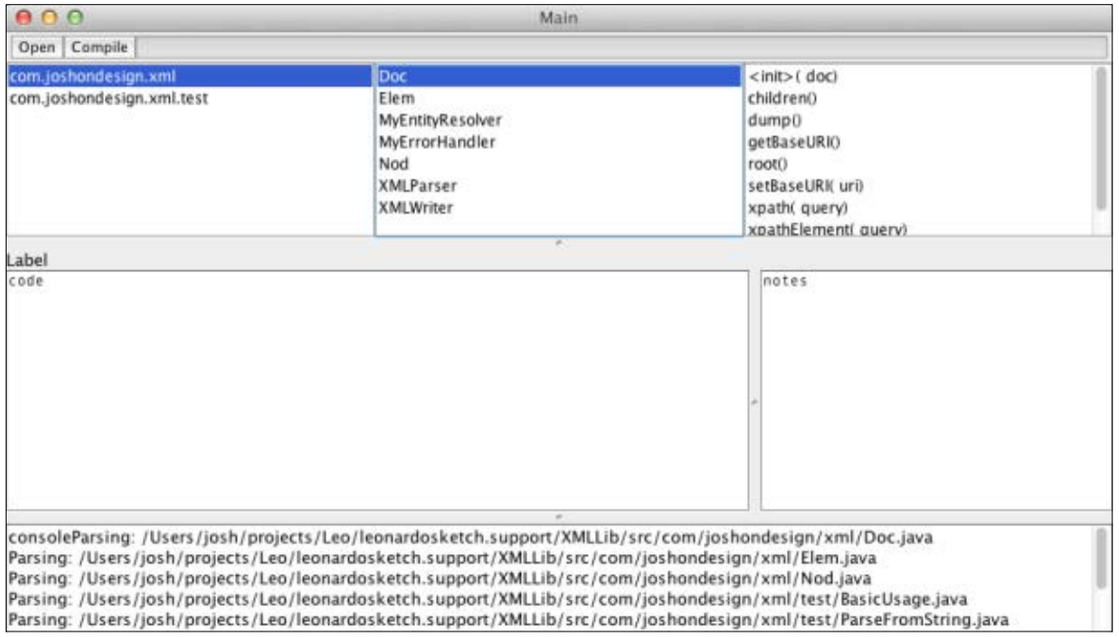


Figure 2

lyze the code. Once the codebase is fully processed, we call a Swing Runnable to update the GUI.

Notice in **Listing 10** that we call parse before analyze. The parse() method just goes through all the Java code looking at the structure. It will check for basic syntax errors. Then analyze() performs a second compiler pass to see whether the code actually makes sense. In this phase, it will determine the types of variables and class references. To understand the difference, consider this piece of code:

```
String foo = new String("foo");
Ensign bar = new Ensign("bar");
```

After the parsing phase, we would know that this code contains two assignment statements resulting in two variables named foo and bar. We know that the type of foo is String. We also know that the type of bar is a class called Ensign, but we don't know anything else about it. We don't know where Ensign is defined, what methods it has, or if it even exists. Only when the analyze phase is complete will we know this information. Actually, this is true for the String type as well, because the developer could have overridden String with a different import statement. We don't know anything about *any* types until the analysis has been done.

If we run the project at this point, we can load up a codebase and browse by package, class, and method. The GUI will look like

LISTING 11

```
private class CaretMoveHandler implements CaretListener {
  public void caretUpdate(CaretEvent caretEvent) {
    if(currentMethod == null) return;
    int dot = caretEvent.getDot(); //cursor position
in characters

    CurrentMethodScanner scanner = new CurrentMethodScanner(
            dot,
            currentMethod,
            trees,
            Main.this);
    //scan every AST
    for(CompilationUnitTree ast : ASTs) {
      scanner.scan(ast,(Void)null);
    }
    locationLabel.setText(scanner.getDeepestPath());
    if(scanner.getInvokedMethod() != null) {
      notesArea.setText(scanner.getInvokedMethod().body);
    } else {
      notesArea.setText("");
    }
  }
}
```

Download all listings in this issue as text

Figure 2. Any parsing errors will be shown in the bottom console area.

Converting Text into Trees

Now that we can browse the code, let's do something more useful. When we move the cursor around the code, it would be nice if the GUI could show us the source for the method under the cursor. To do this, we need to first convert from the position in the text window to the element in the tree that represents that same part of code. Then, if the cursor is in a method call, we must match the method call with the real method and show the source in the notes area on the right.

The code in **Listing 11** defines an event handler that is called whenever the cursor moves in the main

editing text area. It will be called upon mouse clicks and arrow keystrokes.

The caretEvent has a property called dot. This is the position of the cursor in characters from the start of the text area. The event handler creates a new scanner called CurrentMethodScanner, which will look for the element that matches dot. Once the scanner has finished going through the entire codebase (abstract syntax trees), it will set the location label to the deepest path that the scanner found. This will let you know where you are inside the codebase. If you are inside a method invocation, the body of that method is shown in the notes area.

Now, let's look into the method scanner (see **Listing 12**). It is a subclass of TreePathScanner. Last time, we overrode one of the visit* methods. This time, we will just override scan(), which is called for all elements in the tree. For each element in the tree, we will check whether the element contains the cursor position we are looking for. If it does, we will save the location. The real magic happens in containsPosition().

containsPosition gets the current tree path and the current compilation unit. The unit will tell us the actual source file we are looking at. If it is the right source file, then the code gets the start and ending positions of the current element. The position of an element means the offset (in bytes) of that element within the actual source file. These positions are nested.

For example, in the code String bar = new String("bar"), the constructor invocation new String("bar") is inside the assignment statement, which contains the entire line. This statement might itself be inside a block, which could be inside a method and inside a class.

Positions always nest strictly. If the start and end position of the element contain the cursor position, then we can say the cursor is currently inside this element. Notice that we have to add the start position of the current method. This is because the compilation unit counts from the start of the text file, whereas our text area counts only from the start of the method. Adding the method offset accounts for this.

The saveLocation() method saves the path to the current cursor location (see **Listing 13**). If the element is a method invocation or a member select, then it calls findMemberSelect() to find the real method. The findMemberSelect() method first looks at a member select tree to find the real method it calls. First, it needs to know

LISTING 12 LISTING 13 / LISTING 14

```
public class CurrentMethodScanner extends TreePathScanner<
Void,Void> {
  private boolean containsPosition(Tree tree, long pos) {
    TreePath pth = getCurrentPath();
    if(pth == null) return false;
    CompilationUnitTree unit = pth.getCompilationUnit();
    if(unit.getSourceFile().equals(
currentMethod.compUnit.getSourceFile())) {
      SourcePositions sp = trees.getSourcePositions();
      long realpos = pos + currentMethod.start;
      long start = sp.getStartPosition(unit,tree);
      long end =   sp.getEndPosition(unit,tree);
      if(realpos >= start && realpos < end) {
        return true;
      }
    }
    return false;
  }
```

Download all listings in this issue as text

the real class, so we need to dig into findClassModel(), as shown in **Listing 14**.

Here, the code gets really tricky, so we need some background. The JDK uses two separate models. First, we have a tree API that represents the source code parsed into a tree. Then, we have the Element API, which represents the actual classes and methods along with their relationships (who calls whom, who extends whom, and so on). These two APIs are separate, but we can convert from one to another using the Trees utility class, which has a method called getTypeMirror() that will convert a tree object into a type mirror. From there, we can get the real element.

First, the code checks whether the kind of tree is an IDENTIFIER and returns null if it is not. Then, it uses Trees to get the type mirror for the identifier. It rejects types that are not *declared types* (they almost always are). From there, it can get the underlying TypeElement.

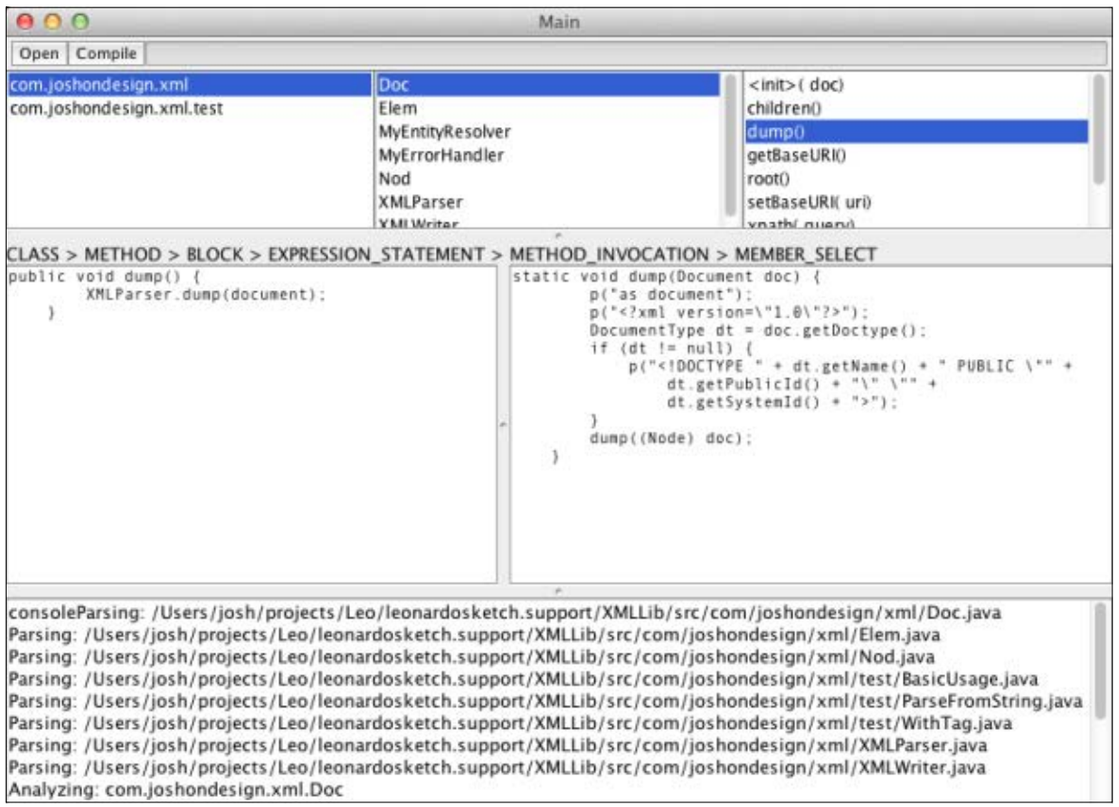


Figure 3

With the element in hand, we can now fetch our class model using the class and package name (code not shown). The code just paws through our class model map. Now we can return to the findMemberSelect method shown in **Listing 15**.

Now that we have the ClassModel, we can get the method element using the same process. We use trees.getTypeMirror() to go from the tree path to a mirror, then we get the element, and then we look up the method model from the ClassModel.

If we run the code now, we will see that the method location is updated as we navigate through the source. Whenever the cursor is over a method invocation, the target method body will be shown in the notes area.

Notice that some method invocations don't show source. Instead, the notes area is empty. This is because the compiler couldn't find that method. The compiler can analyze only the source to which it has access. If your code calls other libraries that the compiler doesn't have the source for, the method body can't be shown. This is especially true of code that calls Java runtime environment (JRE) methods.

For example, if you run the tool on code that calls Document.getChildNodes(), the compiler will never find the body of getChildNodes() because that is part of the org.w3c.dom.Document class, not my XMLLib. The only way to fix this would be to add the source to the JRE and to the source path when you set up the compiler.

LISTING 15 **LISTING 16**

```
private void findMemberSelect(
MemberSelectTree memberSelectTree) {
  ClassModel clazz = findClassModel(memberSelectTree);
  if(clazz != null) {
    TreePath curr = getCurrentPath();
    TreePath pth =
TreePath.getPath(curr.getCompilationUnit(),memberSelectTree);
    TypeMirror mirror = trees.getTypeMirror(pth);
    if(mirror.getKind() != TypeKind.EXECUTABLE) return;
    ExecutableType ex = (ExecutableType) mirror;

    String methName =
calculateMethodName(ex,memberSelectTree.getIdentifier());
    MethodModel methodModel = clazz.getMethod(methName);
    if(methodModel != null) {
      invokedMethod = methodModel;
    }
  }
}
```

Download all listings in this issue as text

Finishing Touches and Next Steps

For the finishing touches, we need to add a few more things. Remember the ConsoleHandler class? **Listing 16** shows the source.

It implements both DiagnosticListener and TaskListener so that it can print messages and event information for both the JavacTask and the file manager. All the messages and events are added to the console text area at the bottom of the screen.

I've also added a Compile button, which will simply call JavacTask.generate() to generate real class files from the parsed codebase. The class files will go into whatever was set as the CLASS_OUTPUT location.

Figure 3 shows what the final application looks like.

From here, there is a lot more we could do. First, some of the code is not as efficient as it could be. The CurrentMethodScanner scans the entire codebase. It could be improved to scan only the compilation unit of the method you are currently viewing. It also handles only some tree cases. It can look up methods but not constructors (which are technically different than methods in the Java language).

We could also add some simple syntax highlighting, because our scanners can now go down to the statement level. If you really want to get fancy, you could make the editor live, meaning each time you edit code, the compiler generates and runs new class files. Now that we have the ability to process full source code, there are endless possibilities.

COOL TOOL

We will build a full code analysis tool that will display a parsed codebase in a three-paned view. Using this browser, you can navigate through the entire source tree.

Conclusion

This two-part article introduced you to many parts of the Java Compiler API, including annotation processing, the language model, the compiler tree, and the javac internal API. We typically think of only dynamic languages as targets for code manipulation, but with the Java Compiler API, we can perform many of these same dynamic tricks with plain old, superfast, mature Java code. `</article>`

LEARN MORE

- "The Advanced Java Compiler API: Part 1"

JULIEN PONGE

BIO

Part 2

Demystifying invokedynamic

Learn how to master the details of invokedynamic.

Part 1 of this series served as a gentle introduction to invokedynamic. It showed how to obtain *method handles*, bind them to *call sites* at runtime, and then emit Java Virtual Machine (JVM) bytecode with actual invokedynamic instructions. Part 1 introduced you to invokedynamic, but the finer details of the java.lang.invoke API remain to be explored.

In particular, the method handle combinator chains that we built in Part 1 remained fairly linear. Indeed, we had a method handle to a target method, and we sometimes performed arity and type conversions to adapt a call site and a target of different types, for example, adapting a call site of type (Object, Object)Object to a method handle of type (int, int[])int.

This article shows that more-elaborate logic can be put into method handle chains. We will see that values can be filtered, conditional branching can be done, and exceptions can be caught. A very important point is that you can create a lot of programming logic by combining method handles. In effect, correctly used, the MethodHandle API is versatile enough to abstract us away from ever explicitly having to use bytecode. This is very powerful. We will present each of the corresponding combinators through simple examples and explain how they could be used in the context of real dynamic language implementations.

Dancing with the Types

The need to do type conversions quickly arises when implementing a language on the JVM, especially for Java interoperability purposes. Fortunately, doing conversions is often a simple matter of using filtering combinators.

Suppose that we have the RichInt class shown in **Listing 1**. It wraps a primitive int type and adds a square method. A full implementation of an "improved integer" in a language would, of course, provide more than the square method, but this implementation shows the basic idea.

We can obtain a method handle to square and invoke it as long as we are passing an instance of RichInt as the receiver, as shown in **Listing 2**. Wouldn't it be convenient to pass int values instead, invoke square on them, and get back an int value?

This kind of conversion can be performed using the filterArguments and filterReturnValue combinators of the MethodHandles class. filterArguments takes a position in the arguments list, followed by a variable number of method handles. Each method handle filters an argument starting from that position. filterReturnValue works similarly, albeit with a single method handle.

We perform the method handle invocation using invokeExact, which requires that the arguments and return value types exactly match those of the method handle. This is not always possible (that is, known at compile time), in which cases invoke and invokeWithArguments can be used instead. The difference is that those methods need to perform type conversions, such as boxing and unboxing primitive types. It is preferable to use invokeExact whenever possible, because it has better performance.

Creating a new instance of RichInt does the conversion from an int to a RichInt. Calling the intValue virtual method of RichInt does the opposite. The code in **Listing 3** illustrates how it is done. By

PHOTOGRAPH BY
MATT BOSTOCK/GETTY IMAGES

doing so, we can treat int like a RichInt, call its square method, and return an int.

Branching with guardWithTest

Dynamic language implementers frequently need to map if/then/else conditional branching at a call site, for instance, to dispatch a method invocation to different targets based on the receiver type or depending on some argument value. The previous article showed only linear combinator chains, but fortunately, it doesn't have to be that way.

The MethodHandles class has a method called guardWithTest that is used to create a combinator with conditional branching. To do so, it takes three method handles as its input:

- A predicate method handle that implements the guard condition and returns a Boolean
- A method handle that is executed when the guard returns true
- A method handle that is executed when the guard returns false

While the guard method handle consumes the arguments that are being passed on the combinator chain, it might not consume all of them. For instance, it might evaluate its condition based only on the first parameter. The true and false branch method handles need to process all arguments, either directly or through further combinator-based adaptation.

Let's build an example in a class called Guards. We want to create a chain of method handle combinators for a call site with an (Object) String signature, where we pass an object and obtain a string representation for it. We expect specialized representations for Integer and String arguments along with a general-purpose representation for any other type. To do that, we provide three static methods (see **Listing 4**).

We will take advantage of guardWithTest to create a branching structure so that the target is any of these three methods depending on the argument class. Because there are three cases, we need two guard methods, which are shown in **Listing 5**.

Note that we suppose that the value won't be null. We could have used the instanceof operator, but it might be slow with deep type hierarchies, so comparing references against getClass() (and maybe null, too) is the best bet in all cases where an instanceof check can be avoided. Our test method is shown in **Listing 6**.

We first obtain a lookup object, and then we get method handles for the target and guard methods. Because the target methods have different signatures, we need to

LISTING 1 LISTING 2 LISTING 3 LISTING 4 LISTING 5 LISTING 6

```
public class RichInt {

 private final int value;

 public RichInt(int value) {
  this.value = value;
 }

 public int intValue() {
  return value;
 }

 public RichInt square() {
  return new RichInt(value * value);
 }

 @Override
 public String toString() {
  return "RichInt{" + "value=" + value + "}";
 }
}
```

Download all listings in this issue as text

adapt each to the call site type, which is (Object)String, using the asType combinator. This works because it is obviously possible to deal with an Integer or a String as an Object.

Remember that method handle chains are type-checked when built: failing to adapt the signatures with asType would raise an exception at runtime. Last but not least, never forget that boxing and unboxing primitive types are costly operations that, as of yet, cannot always be eliminated by the Java HotSpot VM.

Finally, we build a branching method handle using a first guard WithTest. If the object type is an Integer, the intDisplay branch is taken; otherwise, the code falls back to a second guardWithTest construct that looks for a String. It takes the stringDisplay branch or falls back to the general-purpose objectDisplay.

Running this example yields the following console output:

```
Integer = 1
String = foo
Object = true
```

In real dynamic languages, guardWithTest is often used to build inline caches with self-modifying mutable or volatile call sites. This idea comes from the days of Smalltalk. Briefly, the idea is to cache the target method handle when doing method invocations, as long as the receiver objects encountered at a call site remain of the same type. Doing this yields dramatically better performance than performing a method handle lookup for each invocation.

Inline caches can also be used in adaptive runtimes, such as JVMs. The just-in-time (JIT) compiler generates native code outside of the interpreter that makes speculative assumptions about the code being executed, such as a receiver type being of a certain type or locks never being used in a multithreaded context. Because such assumptions might be invalidated, generated native-code blocks (the "fast" path) have guards to fall back to previously optimized code or the interpreter (the "slow" path).

Showing an example of an inline cache in this article would be too lengthy, and many details of an inline cache's implementation depend on the target language's runtime specificities. Instead, you are encouraged to look at the example by Rémi Forax in the JSR 292 Cookbook project and then experiment with an adaptation for your own language's runtime.

LISTING 7

```
static class Runner extends Thread {

 final MethodHandle runHandle;
 final MethodHandle waitHandle;

 Runner(MethodHandle runHandle, MethodHandle waitHandle) {
  super();
  this.runHandle = runHandle;
  this.waitHandle = waitHandle;
 }

 @Override
 public void run() {
  while (true) {
   try {
    System.out.println((String) runHandle.invokeExact());
    waitHandle.invokeExact();
   } catch (Throwable e) {
    throw new RuntimeException(e);
   }
  }
 }
};
```

Download all listings in this issue as text

Switch Points

The SwitchPoint class is another useful addition to the language implementer toolbox when it comes to branching, especially when a target should be executed as long as "some condition" is met, forever reverting to a fallback target as long as the condition no longer holds.

A typical case would be a dynamic language in which methods in object instances can be replaced at runtime. A switch point holds for as long as the method is stable, and then it gets invalidated once it is replaced. It is equivalent to a guardWithTest-based construction, albeit sometimes more straightforward to use when many call sites depend on the same condition not changing.

Listing 7 is an example of a runner thread that takes two method

handles. The code is an infinite loop in which runHandle is a method handle of type ()String. It yields a string value whenever it is called. waitHandle is of type ()void and is supposed to put the thread to sleep for some arbitrary time.

Our test method will exercise the thread in two phases:

- runHandle will return "a" every 500 milliseconds for about 5 seconds.
- Then runHandle will return "b" every 1,200 milliseconds afterward.

To do that, we first define the static methods in **Listing 8**, which will later be used as method handle targets. Our test method is shown in **Listing 9**.

As usual, we first construct method handles to our target methods. We then instantiate two switch points: one for the runner target and one for the waiting target.

A SwitchPoint gives us a method handle by invoking the guardWithTest method. It is very similar to directly using a guardWithTest combinator, as we previously did, except that it takes only two method handles. The first will be executed as long as the switch point is valid; the second is executed as soon as the switch point is no longer valid.

No explicit guard code is executed each time the SwitchPoint is called. The same semantics could be achieved using guardWithTest with a guard that evaluates against some reference object or expression.

We can then start the thread, passing it the method handles for the switch point targets. The thread is then put to sleep for five seconds, after which we invalidate the two switch points using the invalidateAll static method of the SwitchPoint class. At this point, "a" gets printed every 1,200 milliseconds until the application is terminated.

Invalidating a set of switch points causes all of them to switch their target branches. This is immediately reflected across concurrent threads with volatile semantics. While this is a potentially costly operation, it can be more effective to use a SwitchPoint in combination with a MutableCallSite than to rely on a VolatileCallSite just to ensure that changes get propagated as soon as the invalidation happens. Again, the choice is best made based on performance analysis against the invalidation rate of your call sites.

The choice of a SwitchPoint versus a guardWithTest construct also depends on how invalidation is done. In situations in which the invalidation depends only on the receiver type or arguments, a guardWithTest is a good choice. Invalidation is a local call-time decision and applies for, say, inline cache constructs.

Now, consider the case of a dynamic language in which methods can be modified not just on class definitions but also on objects. A good example is the Ruby programming language, in which you can define a class as follows:

```
class Foo
  def bar
    "bar"
  end
end

foo = Foo.new
puts foo.bar # prints bar
```

LISTING 8 LISTING 9

```
static String a() {
  return "a";
}

static String b() {
  return "b";
}

static void slow() {
  try {
    Thread.sleep(1200);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

static void fast() {
  try {
    Thread.sleep(500);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

Download all listings in this issue as text

This calls method bar on object foo, which is an instance of class Foo. Ruby also allows you to redefine bar on the foo instance:

```
def foo.bar
  "[bar]"
end
puts foo.bar # prints [bar]
```

In such cases, there are many call sites that can depend on a given method to remain valid, such as calls to bar in the previous example. Invalidation happens when the instance method is changed. This could be done with guardWithTest and an instance-wide condition as a guard, but SwitchPoint is much better. Indeed, all call sites can be invalidated at once under volatile semantics and, more interestingly, the overhead of a SwitchPoint is virtually zero for as long as it does not need to be invalidated.

Not So Exceptional Exceptions

Method handle targets can throw exceptions, just like any Java method. Because call sites are dynamically bound at runtime, it is likely that you will find yourself in a situation in which the target raises an exception, but the bytecode surrounding the invoke dynamic instructions makes no assumption about that exception to be thrown.

The MethodHandles class provides a combinator called catchException that can mimic a try/catch construction at the method-handle level. It takes a method handle that is the regular target, an exception class, and a method handle to be invoked as a catch block equivalent. Similarly, there is a throwException combinator in the MethodHandles class that can be used to throw exceptions at a call site.

Our example is inspired by the *safe navigation operator* (?.) found in the Groovy programming language, which can be used to avoid the infamous NullPointerException when chaining method calls. Indeed, if a method in the middle of the chain returns null, the next call raises a NullPointerException. Placed as a method invocation operator, the safe navigation operator avoids this issue and returns null instead:

```
def mrbeanPhone =
person?.friends["mrbean"]?.phone
```

If friends is null, or if there is no "mrbean" entry, mrbeanPhone is null. The same code with just "." could raise a NullPointerException in such cases.

Let's build runtime support for a similar construction in which we return a value when the regular target throws an exception. Consider the target method shown in **Listing 10**.

LISTING 10 LISTING 11 / LISTING 12

```
static String stringAndHashCode(String prefix, Object obj) {
  return new StringBuilder(prefix)
        .append(obj.toString())
        .append(" #")
        .append(obj.hashCode())
        .toString();
}
```

Download all listings in this issue as text

Obviously, passing null as the obj parameter will raise NullPointerException. In such cases, we would like to return a different representation using the target method handle shown in **Listing 11**, which works with any call site signature as long as the arguments are being wrapped into an array. We can leverage catchException as shown in **Listing 12**.

The method handle to stringAndHashCode is straightforward. The one to npeHandler requires two adaptations using intermediate combinators: one to collect all arguments into an array of two arguments (asCollector) and then a call site type adaptation. Once this is done, catchException yields to us a method handle that properly routes a method call dispatch to stringAndHashCode,

unless a NullPointerException is raised. Executing the program gives the following output:

```
>>> 666 #666
>>> Hey! #2245797
[null] -> [>>> , null]
[null] -> [null, null]
```

The interesting point is that exception handling and throwing can be done using combinators. This does not require the corresponding bytecode-level support, which is, incidentally, more complicated, because you need to delimit a try lock and specify an exception type and a jump to a catch block. Multiple catch clauses require such constructs to be nested in bytecode. By contrast, a combinator-based solution comes up with simple bytecode and a composition of functions.

HANDLING LOGIC
Elaborate logic can be put into method handle chains; for example, values can be filtered, conditional branching can be done, and exceptions can be caught.

The question of performance is worth raising. Why not use a guardWithTest to check for a null value instead of waiting for an exception to be thrown? As usual, your mileage varies depending on how often you expect exceptions to be raised. As their name suggests, exceptions should be rare. In such cases, you can gamble on a fast dispatch to the target method handle and a slower dispatch in the rare cases in which an exception happens to be thrown. Always benchmark your solutions.

Conclusion

This article concludes a series on invokedynamic. We covered portions of the java.lang.invoke APIs that allow more-elaborate logic to be made out of method handles. You should now be ready to explore more of invokedynamic by yourself. This is a fairly low-level API, but it comes with everything you need for supporting late binding while not confusing the JVM when it comes to adaptive optimizations.

Last but not least, higher-level libraries, such as the Dynalink linker, can greatly help by providing some common runtime plumbing logic. Indeed, developing runtime support for a dynamic language using the java.lang.invoke API is anything but trivial. Dynalink provides ready-to-use building blocks such as ready-made and efficient guarded invocations, type conversions, overloaded method resolution, variable arguments handling, and more. Dynalink also offers a metaobject protocol library that can greatly facilitate the interoperability between Java and Dynalink-powered JVM languages such as Oracle's Nashorn.

Will you contribute to the invokedynamic support of an existing dynamic language for the JVM? Will you design your own language? Will you make fun hacks with invokedynamic? March on—the rest is history.

Acknowledgements. *The author would like to thank Marcus Lagergren for his thoughtful and positive feedback during the review process.* **</article>**

LEARN MORE

- John R. Rose, "Bytecodes Meet Combinators: invokedynamic on the JVM"
- "JooFlux: Hijacking Java 7 InvokeDynamic to Support Live Code Modifications" by Julien Ponge and Frédéric Le Mouël
- "JSR 292 Cookbook," a collection of common invokedynamic patterns by Rémi Forax
- Dynalink: Dynamic Linker Framework on the JVM by Attila Szegedi

BEN EVANS AND
MARTIJN VERBURG

BIO

PHOTOGRAPHS BY
JOHN BLYTHE AND BOB ADLER

Part 3

Advanced Operations Using Java 8 Lambda Expressions

Lazy evaluation and parallel operations enable optimizations that were not available before.

In this article, we will build on the discussion started in Part 1 and Part 2 of this series about lambda expressions. If you are not familiar with the basic syntax of Java 8 lambda expressions or the streams approach, you should read those articles first to familiarize yourself with those concepts.

Note: It is important to realize that lambda expressions are still a moving target. This means that the code examples from earlier builds of lambdas might not always work with more-recent beta builds. In particular, the code in this article uses the latest code from the OpenJDK Project Lambda. This represents an evolution of the API, which is ahead of the Oracle Early Access releases but which is likely to be much closer to the API that ultimately ships with Java 8.

The Limits of Collections

Java collections have served the Java language extremely well. However, they are based upon the idea that all the elements of a collection exist and are represented somewhere in memory. This means that they are not capable of representing more-general data, such as infinite sets.

Consider, for example, the set of all prime numbers. This cannot be modeled as `Set<Integer>` because we don't know what all the prime numbers are, and we certainly don't have enough heap space to represent them all. In Java 7 and earlier versions, the only way to model the set would be to work with an `Iterator` representing the set instead.

It is possible to construct a view of data that works primarily with iterators and relegates the underlying collections to a supporting role. However, this requires discipline and is not an immediately obvious approach to working with the Java collections. With Java 7 and earlier versions, if you wanted to use this type of approach, you would typically depend upon an external library that provided better support for this functionality.

The upcoming release of Java 8 addresses this use case by introducing the `Stream` interface as an abstraction that is better suited to dealing with more-general data structures than basic, finite collections. This means that a `Stream` can be thought of as more general than an `Iterator` or a `Collection`.

However, a `Stream` is not really a data structure; instead, it's an abstraction for handling data.

What does this distinction mean? It means a `Stream` does not manage the storage for elements nor does it provide a way to access individual elements directly from the `Stream`.

Lazy and Eager Evaluation

Let's dig a little deeper into the consequences of modeling an infinite sequence of numbers. Some consequences include the following:

- We can't materialize the whole stream to a `Collection`, so methods such as `collect()` won't be possible.
- We must operate by pulling the elements out of the `Stream`.

- We need a bit of code that returns the next element as we need it.

This approach also means that the values of expressions are not computed until they are needed. It might not seem so, but this simple statement actually represents a huge step for Java.

Before Java 8, the value of an expression was always computed as soon as it was bound to a variable or passed into a function. This is called *eager evaluation* and it is, of course, the default behavior for expression evaluation in most mainstream programming languages.

With Java 8, a new programming paradigm is being introduced for Java: Stream uses *lazy evaluation*. This is an extremely powerful new feature, and it takes a bit of getting used to.

Let's look at a map/filter pipeline, such as the one we discussed in the last article (see **Listing 1**).

The stream() method returns a Stream object. The map() and filter() methods (like almost all the operations on Stream) are lazy. At the other end of the pipeline, we have a collect() operation, which materializes the entire Stream back into a Collection. This method is eager, so the complete pipeline looks like **Listing 2**.

This means that, apart from the materialization back into the Collection, the platform has complete control over how much of the Stream to evaluate, which opens the door to a range of optimizations that are not available in Java 7 and earlier versions.

Lazy evaluation requires more care from the programmer, but this burden largely falls upon the library writer. The aim of lambda expressions in Java 8 is to simplify life for the ordinary programmer, even if that requires extra complexity in the platform.

PARALLEL SUPPORT

The lazy evaluation approach for streams allows the lambda expression framework to provide support for parallel operations.

Handling Primitives

One important aspect of the Stream API that we have glossed over until now is how to handle primitive types. Java generics do not allow a primitive type to be used as a type parameter, so we cannot write Stream<int>.

However, the eagle-eyed reader might have noticed that in the last article, we had a bit of code like the code in **Listing 3** for finding the average age of some otters.

LISTING 1 LISTING 2 / LISTING 3

```
stream()  filter()  map()  collect()
Collection -> Stream -> Stream -> Stream -> Collection
```

Download all listings in this issue as text

This code actually uses primitive types over most of the pipeline, so let's unpack this a bit and see how the primitive types are used in code like this.

First off, don't be confused by the cast to double, which is just to ensure that Java does a proper average instead of performing integer division.

The argument to map() is a lambda expression that takes in an Otter and returns an int. If we could write the code using Java generics, the lambda expression would be converted to an object that implements Function<Otter, int>. However, because Java generics do not allow this, we need to encode the fact that the return type is int in a different way—by putting it in the name of the type.

So the type that is actually inferred is ToIntFunction<Otter>. This type is known as a *primitive specialization* of the function type, and it is used to avoid boxing and unboxing between int and Integer. This saves unnecessary generation of objects and allows us to use function types that are specific to the primitive type that's being used.

Let's break down the average calculation a little more. To get the age of each otter, we use this expression:

```
ots.stream()
  .map(o -> o.getAge())
```

Let's look at the definition of the map() method that is being called:

```
IntStream
map(ToIntFunction<? super T> f);
```

From this we can see that we are using the special function type ToIntFunction, and we're also using a specialized form of Stream to represent the stream of ints.

After this, we pass to reduce(), which is defined as follows:

```
IntStream
map(ToIntFunction<? super T> f);
```

This, too, is a specialized form that also operates purely on ints and takes a two-argument lambda (both arguments are ints) to perform the reduction.

reduce() is a collecting operation (and, therefore, eager), so the pipeline is evaluated at that point and returns a single value, which is then cast to a double and turned into the overall average.

If you missed all of this detail about primitives last issue, don't worry. One of the good things about type inferencing is that some of these differences can be hidden from the developer most of the time.

We've talked about why the Stream<T> abstraction is potentially more general than Java 7 collections, so let's turn to one of the most important use cases: parallelism.

Parallel Operations

One of the primary goals for Project Lambda was to upgrade Java support for collections to allow efficient use of multicore processors.

for loops are serial. In Java 7 and earlier versions, all operations on collections are serial. This means that no matter how large the collection that is being operated on is, only one core will be used to execute the operation. As data sets get larger, this is not appropriate.

The lazy evaluation approach for streams allows the lambda expression framework to provide support for parallel operations.

The primary assumption in the Stream API is that creating a Stream (whether from a Collection or by some other means) should be cheap, but there might be operations in the pipeline that are expensive. This assumption allows us to characterize a parallel pipeline like this:

```
s.stream()
 .parallel()
 .streamOps()
 .collect();
```

The single method parallel() converts a Stream that was previously serial so it is parallel. This capability allows ordinary developers to rely on parallel() as the entry point to parallelism, and it places the burden of providing parallel support on the library writer.

THE POWER OF LAZY

With Java 8, a new programming paradigm is being introduced for Java: **Stream uses lazy evaluation.** This is an extremely powerful new feature, and it takes a bit of getting used to.

To finish, let's look under the hood to see how the platform implements parallelism by relying on a new concept called a Spliterator.

Spliterators

In order to parallelize, we start with a single Stream and end up with multiple streams (which might need to be recombined at the end). This requires an abstraction that can both iterate and split itself into multiple parts. This abstraction is represented by a new interface type called java.util.Spliterator, which has two methods—tryAdvance() and trySplit()—that represent the iterative and splitting natures of the Spliterator.

The task of writing a library that can automatically parallelize a Stream is not straightforward, so any special properties an individual Stream has that can help should be exploited. In the Spliterator interface, this is handled by declaring any useful characteristics as a set of flags. The following are some of the most common flags:

- DISTINCT indicates that all elements coming from the source are different from each other.
- SORTED indicates that the elements are produced in a sorted order.
- SIZED indicates that there are a known and finite number of elements in the stream.
- NONNULL indicates that no elements coming from the source are null
- IMMUTABLE indicates that the sequence of elements cannot be modified after the source has been created.

The standard library has support for spliterators at a number of levels.

One of the simplest ways to use a Spliterator is to start with an Iterator and use a factory method in the java.util.Spliterators class to produce a Spliterator. From this, a Stream can be easily constructed from one of the factory methods in java.util.stream.Streams.

Spliterators, like the rest of the Stream API, also contain primitive specializations of the classes to support using primitive types in places where this would not be allowed by the rules of Java generics.

Conclusion

Java 8 introduces the Stream API, which enables developers to work

with datasources that are more general than basic, finite collections. For example, a Stream can represent an infinite set.

The Stream API is lazily evaluated and makes heavy use of lambda expressions, which means the platform has complete control over how much of a Stream to evaluate. Lazy evaluation and the lambda expression framework open the door to a range of optimizations that were not available in Java 7 and earlier versions, for example, support for parallel operations.

Automatic parallelism is achieved through the use of the parallel() method, which converts a Stream that was previously serial so it is parallel. Under the hood, a new abstraction called a Spliterator is responsible for implementing the parallelization functionality. Spliterators, like the rest of the Stream API, contain primitive specializations of classes to support using primitive types in places where this would not be allowed by the rules of Java generics.

In this series of three articles, we have explored the lambda expressions feature coming in Java 8. We looked at the syntax, and discussed automatic conversion, type inference, and the upgrades to the collections libraries (including default methods). We talked about the new Stream abstraction, lazy evaluation, and primitive specialization. Finally, we introduced automatic parallelism and the mechanism known as Spliterators, which provide much of this functionality.

It seems certain that with the release of Java 8, lambda expressions and their associated upgrades will become a very valuable part of the Java developer's toolbox. Their introduction breathes new life into the platform's capabilities, and will have a major impact on almost every Java codebase and project. `</article>`

LEARN MORE

- For help with lambda expressions or to join our global hack days, join the Adopt OpenJDK project.

DIBY MALAKAR

BIO

What's Driving the Cloud for Developers?

A cloud primer for Java developers.

With every passing year, we draw closer and closer to achieving the vision of the ubiquitously connected global village. The number of internet users today exceeds 2 billion worldwide, with global internet penetration estimated to grow from 34.3 percent at the end of 2012 to 45 percent of the world's projected population by the end of 2016. For the world of software development, this offers unique opportunities to lower development costs and improve productivity and throughput by utilizing geographically dispersed teams and processes.

Data production is expected to be 44 times greater in 2020 than it was in 2009, reaching a worldwide total of 7.9 zettabytes (or eight trillion gigabytes) by 2015. Developers are under pressure to use that data to create more-compelling applications. Also, companies are searching for solutions that will provide unprecedented capacity, scalability, and speed to keep pace with that explosive growth.

There's another trend affecting developers: mobility is untethering individuals from the need to be at any particular place in order to be productive. In 2011, 420 million smartphones and 66 million tablets were sold. In 2016, those numbers are projected to balloon to 1.2 billion smartphones and tablets being sold. Project teams are global, sharing code and collaborating from anywhere in the world as seamlessly as possible. They need infrastructure that allows them to do that.

Social platform adoption is such an integral part of our lives that 90 percent of all internet users now have an account on at least one social service, and there are more than 13 million business pages on Facebook alone. As social behaviors continue to increase, modern development environments must keep pace and offer similar levels of seamless and frequent interaction.

Due to the need for cutting-edge infrastructures, enterprises are facing pressure to modernize to achieve the flexibility, scalability, and responsiveness that will help them remain competitive. But companies can rarely afford to rip out and replace their entire infrastructure or retrain their staffs. Adopting a standards-based cloud infrastructure enables companies to modernize at a gradual pace.

As a result of these trends, application needs are growing exponentially. In addition, the pace and competitiveness of development put enormous pressure on productivity, throughput, and quality. But development resources continue to grow as linearly as ever, resulting in a growing "development gap" (see **Figure 1**). It's the ability to fill in this development gap that has made the cloud such a valuable force in the developer world.

According to Gartner, the public cloud services market is forecast to grow 18.5 percent in 2013 to total US$131 billion worldwide, up from US$111 billion in 2012. Market dynamics vary substantially when considering cloud services' market

size and market growth across the different regions of the world. According to Gartner, the emerging markets in Asia Pacific, Latin America, Eastern Europe, the Middle East, and North Africa show the highest growth rates while representing the smallest overall markets. China is the exception, being both a large and growing market. The mature markets of North America, Western Europe, Japan, and the mature Asia Pacific countries constitute the larger, but slower-growth, markets.

Fundamental Models of Cloud Computing

Cloud computing providers offer their services according to three fundamental models: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS), where IaaS is the most basic model and each higher model abstracts from the details of the lower models. The distinctions are as follows.

IaaS. In the most basic cloud-service model, providers of IaaS offer computers—physical or (more often) virtual machines—and other resources. To deploy their applications, cloud users install operating system images and their application software on the cloud infrastructure. In this model, the cloud user patches and

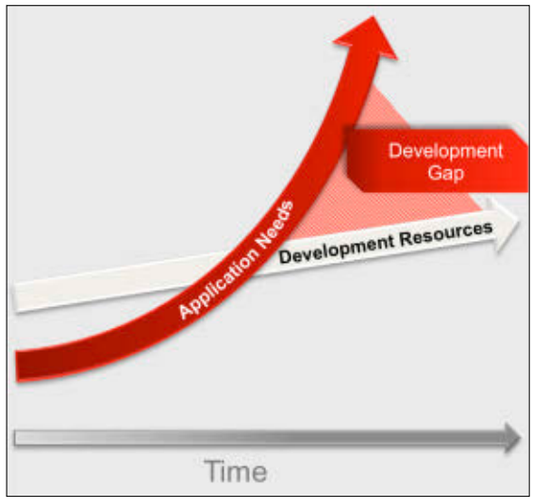


Figure 1

maintains the operating systems and the application software.

PaaS. In the PaaS model, cloud providers deliver a computing platform that typically includes an operating system, a programming language execution environment, a database, and a Web server. Application developers can develop and run their software solutions on such a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers.

SaaS. In the SaaS model, cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients. The cloud users do not manage the cloud infrastructure or the platform on which the applications are running. This eliminates the need to install and run the applications on the cloud

SPOTLIGHT: Oracle Java Cloud Service

Oracle Java Cloud Service provides an enterprise-grade platform to develop and deploy business applications in the cloud. It uses Oracle WebLogic Server in the cloud and offers choice, an open environment, security, and ease of use. Oracle Java Cloud Service supports Java standards and offers flexibility, portability, and deployment options.

Choice. Developers have a choice when it comes to the IDE, application development and deployment, service integration, and monitoring and management services. Starting with the IDE, Oracle Java Cloud Service supports various platforms—from Eclipse and Oracle JDeveloper to NetBeans. At the click of a button, you can deploy your applications onto Oracle Java Cloud Service. If your development cycle involves team development, you can easily deploy your applications from your key development environment onto Oracle Java Cloud Service or other standards-based environments. There are also choices for service integration, including Web services, SOAP, and REST APIs. For managing and monitoring services, there is a command-line interface with plug-ins for Maven and Ant.

Open. Create Java EE applications within the Oracle cloud and deploy them in standard WAR (Web Application Archive) or EAR (Enterprise Archive) format. Beyond the standard Java EE specifications, Oracle Java Cloud Service supports the deployment of applications using Oracle WebLogic Server–specific extensions as well as Oracle Application Development Framework constructs.

Secure. Applications are secured at multiple layers, because the Oracle Java Cloud Service instance itself is isolated. The service includes dedicated virtual machines (VMs) and schemas that sit on top of Oracle Exalogic Elastic Cloud. The VMs then map to a corresponding database schema on an Oracle Exadata Database Machine. The security is not limited to applications deployed to Oracle Java Cloud Service. If you're using another service in Oracle Cloud—such as Oracle Fusion Applications—and building custom extensions in Oracle Java Cloud Service, the identity information and user profiles can be securely populated from Oracle Fusion Applications onto Oracle Java Cloud Service using the Identity Service in Oracle Cloud's common infrastructure.

Easy to use. The entire service lifecycle—from subscription to service use, to monitoring and management—can be done through a single, easy-to-use Web-based interface.

Oracle also offers a comprehensive set of platform services to build rich applications, including Oracle Database Cloud Service, Oracle Developer Cloud Service, Oracle Storage Cloud Service, and Oracle Messaging Cloud Service.

user's own computers, simplifying maintenance and support.

IaaS cloud solutions offer the ability to deploy compute, network, and storage resources with minimal effort. While those abilities are definitely very beneficial, from a developer's perspective, the advantage of the cloud does not stop there. In fact, greater value lies beyond the provisioning of the infrastructure layer where developers devote more than 80 percent of their time and effort. This is where PaaS provides tremendous benefits for application development, testing, deployment, and management.

With on-demand infrastructure, automated provisioning, flexible development and runtime tools, and management automation, IaaS and PaaS cloud services are poised to have a profound impact on the developer experience.

MIND THE GAP

Application needs are growing exponentially, but development resources continue to grow as linearly as ever, resulting in a growing "development gap."

What to Look for in a Cloud Offering

When evaluating cloud options for developers, you should consider standards, choice, breadth, depth, self-service capability, and data integrity and portability.

Support for standards. Are the cloud services built on standards, such as SQL, Java, and HTML5? Building on standards has a lot of benefits. One of the most immediate benefits is that you can run your applications in the cloud more transparently, which means you can port an application to your local on-premises environment without having to alter any code. Conversely, if you're running an on-premises application built on open standards and would like to move it to the cloud, you will be further along in the process to make that change.

Choice. Your vendor's cloud offerings should span public and private clouds and, if applicable, its product portfolio. Does your vendor provide comprehensive products for cloud providers to build and manage private clouds, and managed cloud services to help customers fully manage and/or host their clouds? Private clouds enable organizations to have complete control and visibility over security, regulatory compliance, service levels, and functionality.

Breadth. One of the big advantages of cloud computing is that it breaks down silos across your organization. Your cloud solution should help you avoid the fragmentation of data and business processes that occurs when using multiple, siloed public clouds. If your vendor offers multiple types of cloud services, this gives you maximum flexibility. The platform services must enable the developer community to build and deploy applications. The application services should enable business users to start using packaged functionality right away. For example, are packaged applications—such as enterprise resource planning, customer relationship management, and human capital management—available on a subscription basis? Are social services integrated with the applications or tacked on?

Depth. Look for services that are not only rich in functionality but, more importantly, integrated to work together seamlessly. The services should run on proven infrastructure, offering you the reliability and performance of an enterprise-grade solution.

Self-service capabilities. Can you develop, deploy, and manage your applications or platform with minimal involvement of the vendor or your IT staff?

Data integrity and portability. Does your vendor commingle your data with other customers' data? Surprisingly, this is pretty common in a public cloud environment. You should consider asking whether the vendor can provide data isolation. Also, how portable is your data? Don't let your vendor hold your data hostage. Make sure it is easy and straightforward to get all your data out of the cloud in a format you can use.

Conclusion

Developers can leverage the cloud to address the "development gap" that results from the increasing pressure to create and deploy applications faster, lower the cost of innovation, and dramatically reduce risk. A good cloud vendor will provide you with a comprehensive suite of technologies and applications that are based on open standards and integrated to work together seamlessly. `</article>`

LEARN MORE

- Oracle Java Cloud Service tutorials
- "Deploying an Application to Oracle Java Cloud Service" tutorial
- "Securing a Web Application on Oracle Java Cloud Service" tutorial
- Oracle Java Cloud Service 30-day free trial

CloudBees

Java™ Infrastructure Like Electricity

Plug into CloudBees and get your Java applications running in the cloud

Egraphs wows sports fans with cloud-based Scala application
Egraphs, a company that connects sports fans and stars, used the CloudBees PaaS to meet an aggressive deadline for its innovative web application. Developers used CloudBees to accelerate deployment of their Scala- and Play framework-based application to the cloud. Read the full story.

CloudBees offers the first Java PaaS that supports the entire application lifecycle, from development through to production.

No more hassles to get IT resources. No more ongoing software upgrades. No more infrastructure maintenance. PaaS eliminates these headaches from the development process, enabling you to regain control of your time, productivity and, most importantly, your application.

>> **Develop and deploy applications more quickly** – Egraphs completed an initial deployment in an hour.
"We threw our WAR at the CloudBees platform, and it just worked."
— Erem Boto, Co-founder, Egraphs

>> **Scale instantly** – Egraphs counts on CloudBees to automatically scale in response to spikes in traffic.
"…our application needed to handle the load when the league sent out announcements to people on its ten-million-member email list."
— Erem Boto, Co-founder, Egraphs

"At Egraphs, we think of the CloudBees platform much like electricity or water — we turn it on and it works."
— William Chan, Co-founder, Egraphs

Egraphs
www.egraphs.com

Try CloudBees PaaS for FREE!

Want to learn more?
Download *CloudBees Advantages: A Guide for Java Developers*

JAVA EE 7 ARRIVES

Anil Gaur on the evolution of Java EE

BY STEVE MELOAN

As the industry standard for enterprise Java computing, Java Platform, Enterprise Edition, has constantly evolved to keep pace with the emerging usages, patterns, frameworks, and technologies of the enterprise space. Java EE 7 is now here, with essential support for modern-era Web applications—including HTML5, WebSocket, Java API for RESTful Web Services (JAX-RS), JSON Processing (JSON-P), and more.

Java Magazine *sat down with Anil Gaur, vice president of software development at Oracle, to explore the new features and technologies of the just-released Java EE 7*

Anil Gaur (center), vice president of software development at Oracle, chats with Java EE 7 Platform Specification Leads Bill Shannon and Linda DeMichiel.

PHOTOGRAPHY BY PHIL SALTONSTALL

Gaur discusses Java EE 7 capabilities with Arun Gupta (dark shirt), Java evangelist at Oracle, and Shreedhar Ganapathy, engineering manager at Oracle.

platform and examine what it means to enterprise developers and customers.

Java Magazine: What's new and notable in Java EE 7?

Gaur: The Java EE platform is at a critical point right now, in that we have to rapidly move forward to keep pace with emerging Java usages and patterns. These changes are being greatly driven by modern Web applications. Over the past few years, many other frameworks have evolved, and we also wanted to align the platform with those changes.

So the main theme of Java EE 7 is productivity and HTML5 support. The HTML5 enhancement comes from WebSocket, Java Servlet 3.1, the NIO [non-blocking I/O] feature, Server-Sent Events [SSE], JAX-RS, and JSON-P.

In addition, several enhancements have been made throughout the platform to improve productivity, because that's another pain point, and it directly reflects on the cost of development and maintaining applications. With Java EE 7, developers will have to write less boilerplate code, and the platform will also offer richer functionality through various new specifications, such as batch processing and concurrency.

What's more, the JMS [Java Message Service] spec has gone through major enhancements in JMS 2.0, where we've introduced some of the concepts already available in the Servlet and EJB [Enterprise JavaBeans] specs. And the Expression Language specification has also been enhanced in version 3.0, which will benefit many other technologies within the platform.

The various specs that have gone through changes are JPA 2.1 [Java Persistence API], Java Servlet 3.1, EJB 3.2, Bean Validation 1.1, and CDI [Contexts and Dependency Injection for the Java EE platform], which was added in Java EE 6 and was one of the most popular and widely used specs there.

Java Magazine: How will Java EE 7 provide developer simplification while also offering richer application possibilities?

Gaur: We've particularly focused on simplification in terms of introducing the concept of profiles. We've heard many times that the platform has been growing into a monolithic bundle that was sometimes difficult for very specific, targeted usages. And we took that very seriously. So we invested heavily in the Java EE 6 platform, where we introduced the Web Profile, a runtime stack of standard APIs specifically targeted to Web application development. And that profile has allowed us to put in place a basic framework that we can use toward defining future profiles. Developer simplification also occurs by providing more defaults and by using CDI, which reduces the use of XML coding.

We've seen great demand for the new specs that we've added in Java EE 7. People were already using technologies such as WebSocket and JSON, but prior to Java EE 7, there were no standard APIs. Now, developers no longer need to package their own JSON libraries in the applications. The

support is in the base platform. And they can invoke WebSocket endpoints directly from Java clients. So that also makes for significant development simplification, while helping to minimize the size of the applications that are deployed.

Batch processing is another addition to the platform, which has been regularly requested by customers. This provides a standardized programming model to implement batch applications. The JSR defines the domain area, batch job, application executor, and job manager. And in addition to the support in Java EE, some of the features will also be included in Java SE. But Java EE is a superset of Java SE, so it inherently provides a richer feature set. JTA [Java Transaction API], for example, is only found in Java EE.

We've also added concurrency to the platform, which had similarly been requested by developers. This allows application-level work to be performed concurrently, using different threads of execution. Prior to this release, various application servers had their own proprietary implementations, but that reduced application portability. Their inclusion in Java EE 7 provides a standard API available to all EE vendors, customers, and developers.

And, as I indicated earlier, the JMS 2.0 spec has been revised and simplified in a variety of ways that had been requested by the community. JMS 2.0 will make greater use of annotations, as did the EJB and Servlet specs in prior versions of the platform. JMS connection factory will also leverage Dependency Injection [DI]; in addition, the API is being cleaned up so that interim objects are not required, and then there are AutoCloseable resources. These changes should be very attractive to developers and allow them to start using JMS more extensively.

RESPONDING TO FEEDBACK

We've heard many times that the platform has been growing into a monolithic bundle that was sometimes difficult for very specific, targeted usages. And we took that very seriously.

So, yes, we've added more APIs to the platform, but most of the additions have been based on community requests and what was needed to solve the issues we've seen for developing enterprise applications.

Java Magazine: What's been pruned from the release, what does that achieve, and how does it potentially affect application backward compatibility?

Gaur: While we've added many new features to Java EE 7, we have also made certain APIs optional. So the platform has grown in terms of functionality, but there has also been some cleanup of older features—which will not only help reduce the platform footprint, but free developers from having to deal with obsolete specifications.

This process began in Java EE 6, where we marked several technologies as optional, and targeted them for eventual pruning—Java EE Management [JSR 77]; Application Deployment [JSR 88]; JAXR, for interfacing with UDDI registries [JSR 93]; JAX-RPC, for XML-based RPC [JSR 101]; and EJB 2.x Container Managed Persistence, which is effectively replaced by the Java Persistence API [JSR 317].

Since these were marked for pruning in the Java EE 6 release, it makes it possible for us to remove them in this release. But that's optional, of course. It's up to each vendor to make that decision for the release. If they're seeing continued demand, if their customer base is still using these specs, they can keep them in the platform. But we will remove them in the next release [Java EE 8]. And I expect most vendors will follow that same strategy.

Java Magazine: What does the Web Profile stack for Java EE 7 bring to the release and to Web application developers?

Gaur: The Web Profile was introduced in Java EE 6 and became very popular, because it offered a subset of the full platform, providing standard APIs that are specifically targeted to meet the

Gaur and Java EE 7 team leads share a lighthearted moment.

needs of the majority of Web applications. Not every developer needs all the Java EE APIs, so this offers a reduced runtime memory footprint, simplified installation, and enhanced responsiveness.

So as with Java EE 6, the Java EE 7 release includes a Web Profile. But there are new specs being added—WebSocket and JSON-P. And the JAX-RS 2.0 enhancement, the client APIs, is another notable addition. These specifications are, of course, not restricted to just the Web Profile.

But more and more developers are using JavaScript and HTML5 in their applications, and WebSocket and JSON-P are two specs very well aligned with that programming model. So that makes it very attractive and suitable for next-generation Web application development.

Java Magazine: Can you discuss the decision to defer some cloud-related features originally intended for Java EE 7 to Java EE 8 and the features involved?

Gaur: This was a tough decision, but I think we made the right call. And we consulted with our community members, Expert Groups, and vendors before making the final decision.

This decision occurred partly due to lack of maturity in the cloud space—specifically, in provisioning, multitenancy, and elasticity. We came to realize that fact when we were already well into the implementation. Ultimately, we wanted to avoid adding APIs or functionalities that were either not fully realized or not ideally suited to the intended goal.

So we chose a more conservative approach—to make sure that we reach our goal in a way that can maximally benefit the industry and developers. After consulting with community members, and conducting a survey, we decided to shift the focus of Java EE 7 toward HTML5 and productivity. This shift in scope allowed us to better retain our focus on enhancements, simplification, and usability—and to still deliver the release on schedule. In the end, we didn't want to hold up features that were ready to go for something that's not fully realized.

Java Magazine: What specific JSRs were deferred?

Gaur: There were no specific JSRs, as such. The cloud requirements were cross-cutting, impacting many different specs. So no actual specs were dropped in the process. But features *within* several specs, including the Java EE 7 platform spec, were affected.

Java Magazine: In the interim period, what facilities exist for running Java EE applications in the cloud?

Gaur: If you look at many of the vendors out there, in most cases, their cloud platforms are built on Java, which is great news for us. And Java EE 6 is already compatible with those environments. So the container-based model that we have in Java EE, along with security isolation, makes it very suitable for writing PaaS [platform as a service]–enabled applications. And JAX-RS, the API for RESTful Web Services, is also well tailored for cloud applications. So given all this, it's still a good platform for building cloud solutions.

Java Magazine: How has the Java community been involved in the evolution and refinement of Java EE 7, particularly with the recent cloud-related decisions?

Gaur: We have been developing the Java EE Reference Implementation as an open source project since Java EE 5, under Project GlassFish. And with Java EE 7, we decided to go even further, such that all the Java EE specs are now developed with participation from a broader community in addition to the Expert Groups and Java EE vendors. So over the last two or three years, the spec development process has

become more and more transparent. And that's not by accident. Oracle and the community had a strong desire to engage more community members—and earlier in the process—so that the community could review and provide feedback as we developed the specifications.

We filed JSR 348 to define the rules by which a JSR needs to be run—in a transparent manner. All the Oracle-led JSRs are now following that process. The Expert Groups have open e-mail aliases for technical discussions, the interim drafts of specifications are visible on the project page, and we use JIRA, which contains a list of all the features that we target for each of the JSRs. So everyone who is interested can now file a JIRA request if they want certain features to be added or if they have feedback on features that are already planned. Each JSR has a dashboard for JIRA requests.

This has allowed a much broader community to provide their ongoing feedback. And, as a result, we can deliver a superior technology at a faster pace. In addition, we engaged 19 Java user groups [JUGs] who participated in a program called Adopt-a-JSR, in which developers can volunteer their time to review what's being proposed and how it might impact technologies in the platform. JUGs have also developed several small applications to try out what's being proposed. The feedback provided by these JUGs helped iron out issues long before the platform was finalized.

Gaur talks with team members Rajiv Mordani (left) and Nazrul Islam at Oracle's Santa Clara, California, offices.

Java Magazine: What features are now planned for Java EE 8, and what are the target dates?

Gaur: After the release of Java EE 7, Oracle and other vendors will begin working on Java EE 8. The focus of the release will be a standards-based cloud programming model—deferred from Java EE 7—in which we will add support for multi-tenancy and elasticity.

You can also expect to see modularity based on Jigsaw in Java EE 8. There is also strong interest in some sort of NoSQL support in the Java EE platform, so we will evaluate that as well. But in the meantime, the platform will continue to evolve. It's not as if, after Java EE 7, the only time you'll get to see something new is in Java EE 8. We'd like to have new JSRs, such as JCache and JSON-B [JSON Binding], finalized well before Java EE 8. So there will be incremental progress in the platform, where our customers will be able to see and experiment with those new APIs before Java EE 8 is ultimately released.

In terms of a formal release date for Java EE 8, the community will help define the scope and the features of the release, and that will ultimately determine the dates. The Expert Groups will play a major role in coming up with a final feature set and the release dates.

Java Magazine: At JavaOne 2012, Hasan Rizvi pointed out that the ongoing success of Java is based on a triad of technology innovation, community participation, and Oracle's stewardship. Do you have any thoughts on further enhancing the partnership between

Oracle and the Java community?

Gaur: I'm pleased to say that the relationship with the Java community has strengthened quite significantly under Oracle's stewardship. And that has been demonstrated in several different ways. The community is very actively engaged in defining the next-generation platform, and I expect that to continue. And we will continue to run the Expert Groups in an open fashion, which will enable more people to participate without requiring them to join the JCP or adhere to any heavyweight process. We will also continue to develop Reference Implementations as open source, and that will help us gather feedback quickly and, ultimately, develop a platform that is appealing to both enterprise customers and developers.

As evidence of this open approach, the Java EE 6 release was adopted more quickly than any previous release. I attribute that to the fact that we've been engaging the community at earlier stages, and I expect that we'll continue on that path with similar positive results.

Java Magazine: How can developers access Java EE 7?

LISTENING TO THE COMMUNITY

We've added more APIs to the platform, but most of the additions have been based on community requests and what was needed to solve the issues we've seen for developing enterprise applications.

Gaur: The Java EE SDK is our most popular bundle, which developers can use to learn the new technologies in the platform. That is now available on Oracle Technology Network. The SDK bundle consists of GlassFish Server Open Source Edition 4.0, which is also the Reference Implementation for Java EE 7. It contains a comprehensive tutorial that covers the new APIs, and there are many sample programs that will help developers understand each of the specifications and get up to speed faster.

I should also touch upon IDE support. The NetBeans IDE will be the first IDE to support the Java EE 7 APIs. We worked very closely with the NetBeans folks as we developed the platform, so that enables us to have a much tighter integration between the NetBeans IDE and GlassFish and a superior out-of-the-box experience. And I expect Eclipse and IntelliJ will be next in line with support for the Java EE 7 platform.

Java Magazine: What can the GlassFish community expect with GlassFish 4.0, which is being released in conjunction with Java EE 7?

Gaur: As with Java EE 5 and Java EE 6, the Reference Implementation of Java EE 7 is derived from Project GlassFish. Being the Reference Implementation, GlassFish is always up to date with the latest Java EE specifications. So when we ship the Java EE 7 platform, the community will have GlassFish Server Open Source Edition 4.0. It will offer support for all the Java EE 7 APIs, but the clustering and high availability features are there for evaluation purposes only. Those production deployment–centric features will be made available in the subsequent release, 4.1.

GlassFish 4.0 will be available for download in two different flavors. The first distribution contains support for all the APIs in Java EE 7, while the second contains a subset of APIs defined by the Java EE 7 Web Profile specification. So developers will have a choice—they can start with the SDK bundle that contains the open source bits of GlassFish 4.0, or they can download the GlassFish bits separately from Java.net. `</article>`

Steve Meloan is a former C/UNIX software developer who has covered the Web and the internet for such publications as *Wired*, *Rolling Stone*, *Playboy*, *SF Weekly*, and the *San Francisco Examiner*. He recently published a science-adventure novel, *The Shroud*, and regularly contributes to *The Huffington Post*.

LEARN MORE

- Download Java EE 7
- GlassFish 4.0
- NetBeans IDE

JOHAN VOS

BIO

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

Java EE 7

Boosting Developer Productivity with Java EE 7

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

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

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

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

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

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

The Java EE 5 requirements for simplicity are twofold:

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

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

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

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

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

MAKING IT SIMPLER

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

PHOTOGRAPH BY TON HENDRIKS

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

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

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

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

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

Batch Processing

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

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

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

LISTING 1 LISTING 2

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

Download all listings in this issue as text

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

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

JSON

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

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

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

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

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

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

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

LISTING 3 LISTING 4 / LISTING 5

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

Download all listings in this issue as text

WebSocket

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

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

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

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

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

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

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

Concurrency

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

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

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

EVER EVOLVING

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

Other Upgrades in Java EE 7

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

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

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

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

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

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

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

LISTING 6 LISTING 7 / LISTING 8

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

Download all listings in this issue as text

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

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

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

Conclusion

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

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

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

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

LEARN MORE

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

EDUARDO MORANCHEL AND EDGAR MARTINEZ

BIO

Java EE 7

Embracing HTML5

Explore new features for creating next-generation internet applications.

Java EE 7 supports new and evolving standard technologies such as WebSocket, HTML5, and JSON, providing an API for each one.

This article first presents some background on WebSocket and then offers a Web application to demonstrate the new platform features.

Note: The source code for the example described in this article can be downloaded as a Maven project here.

WebSocket Background

Modern Web applications have grown in interactivity, demanding greater communication between the browser and the Web server. At first, constant communication was achieved by servlets with long polling in the browser, but problems in the server appeared because the threading model of the Web container demanded that every request be handled by a thread until the response is written. Then, asynchronous servlets came with Java EE 6 providing a more efficient approach that allowed requests to be processed asynchronously.

The Polling, long pooling, and Comet/Ajax approaches simulate a full-duplex communication between the server and the browser. WebSocket is a full-duplex and bidirectional API, which not only involves the server but the Web browser, too.

WebSocket changes the way the Web server reacts to client requests: instead of closing the connection, it sends a 101 status and leaves the connection open, expecting messages to be written on the stream and to be able to write messages, as well. The client also expects messages to be written and is also able to write messages at its end of the stream. This allows both ends to read and write to the stream and communicate in real time.

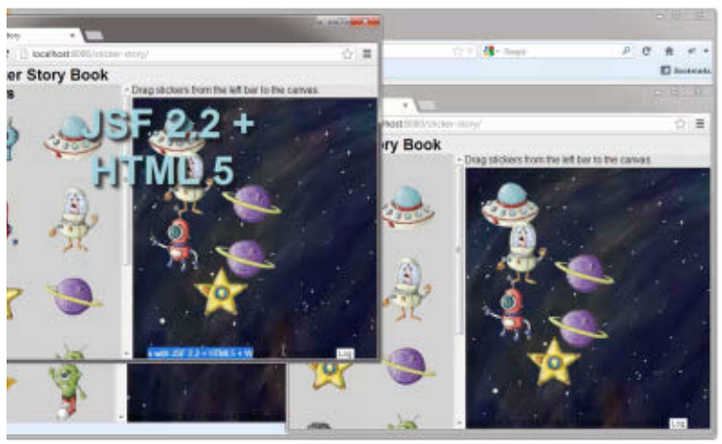
Before Java EE 7, implementing a WebSocket was Web container–dependent: you could write a WebSocket for Tomcat, but if you wanted to migrate to Jetty, you had to reimplement it for the new server.

Note: As with every new technology, WebSocket is being adopted by different Web browsers at different times. You can check this page for more information about browser compatibility.

Overview of the Sticker Story Application

Let's start with a simple example to see how the things we have discussed so far work.

This article showcases a sticker story Web application that children can use to create a story collaboratively by dragging stickers into a book or canvas. At the instant a sticker is placed into a child's



This video provides a demo of the sticker story application.

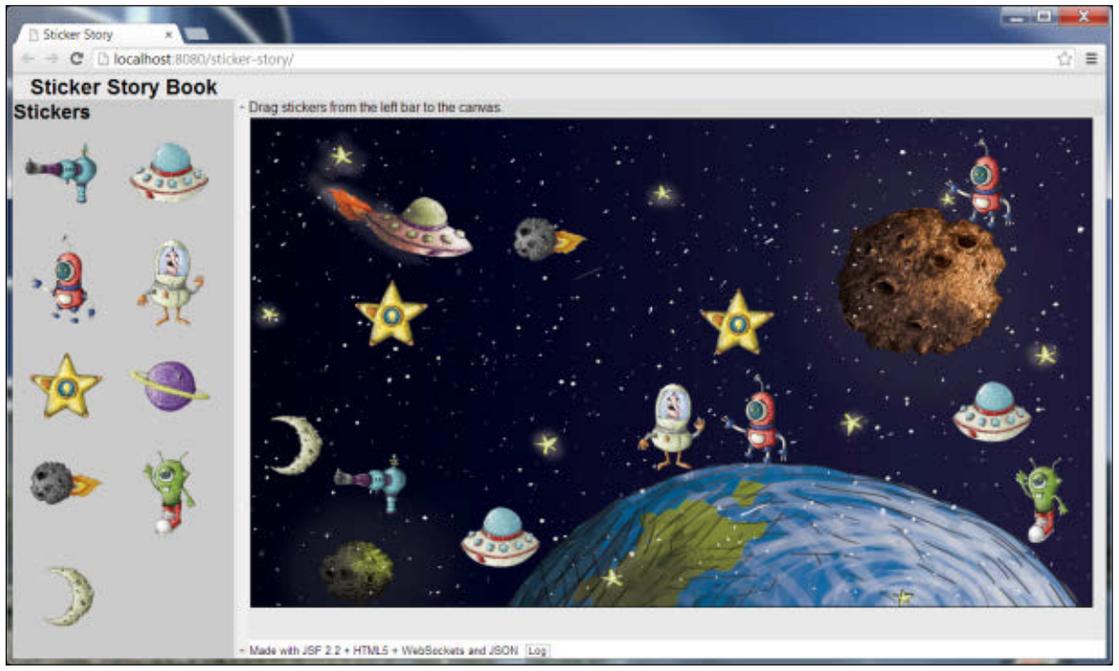


Figure 1

book, the sticker is drawn within other children's books. Watch the video to see the application in action.

Figure 1 shows the home page for the sticker story Web application. When you drag a sticker from the sidebar on the left to the canvas and drop it, the sticker is rendered into all the Web browsers that are open at that time, as shown in the video.

Obtaining and Running the Sticker Story Application

1. Download GlassFish v4 promoted build 79 (glassfish-4.0-b79.zip). GlassFish can be obtained here as a zip archive.
2. Unzip GlassFish into a directory of your choice.
3. Download and install NetBeans. The version used in this article is NetBeans 7.3.
4. Download the sticker story Maven project link to sticker-story.zip.
5. Expand the project into a directory of your choice.
6. Start NetBeans, and on the **Services** tab, right-click the **Servers** node, and then click **Add Server**.
7. From the Server list, select **GlassFish Server 3+**, and then click **Next**.
8. In the **Server Location** field, enter the path to the directory where you unzipped GlassFish v4, and then click **Finish**.
9. Select **File** -> **Open Project**.
10. From the Open Project dialog box, open the sticker story Maven project.
11. Right-click the project and select **Run**. Select the recently installed GlassFish server (GlassFish Server 3+) when prompted.

You can now point your Web browser to http://localhost:8080/sticker-story to see a page similar to that of **Figure 1**. Make sure your browser is compatible with WebSocket by checking this page.

Application Contents

The application contains the following files and Java classes:

- index.html is the home page and contains JavaServer Faces (JSF) 2.2 and HTML5 code.
- The org.sticker.jsf.StickerSheet class is the managed bean used by the home page to display all the available stickers.
- story-page.js contains the JavaScript code for the WebSocket client and the drag-and-drop functionality.
- The org.sticker.websocket.Sticker class is the object being sent and received by the WebSocket.
- The org.sticker.websocket.StickerEncoder and org.sticker.websocket.StickerDecoder classes transform the data sent by a WebSocket into objects.
- The org.sticker.websocket.StoryWebSocket class is the WebSocket handler. It also stores the stickers in the current story book.

Combining HTML5 with the JSF 2.2–Friendly HTML Markup

Just as Ed Burns, spec lead of JSF 2.2 (JSR 344), mentioned in his blog, JSF 2.2 allows page authors to have complete control of the HTML rendering by using the Friendly Markup feature of JSF 2.2. It also facilitates the separation of markup and business logic.

Let's see an example. The view of the sticker story application is provided using HTML5 and JSF. **Listing 1** shows the content of the index.xhtml file, which is mostly HTML5; JSF is used to render the sticker images in the sidebar of the browser using the <h:graphicImage> tag.

- JSF 2.2 lets you pass through HTML attributes. There are two ways to do this: with the namespace for the pass-through attributes, which is http://java.sun.com/jsf/passthrough, or by using the child TagHandler f:passThroughAttribute (or f:passThroughAttributes for multiple attributes). In our example, we use the former.

In **Listing 1**, notice the <h:graphicImage>tag. There are three attributes (draggable, ondragstart, and data-sticker) in our graphicImage tag that are not part of the attributes for graphicImage but that we need in order to make our stickers work. Therefore, we use the prefix p: so that JSF pass-through adds these three attributes. **Note:** Because graphicImage is a JSF markup, we have used the namespace http://java.sun.com/jsf/passthrough. If you want to pass through elements into a non-JSF-aware markup, use the namespace http://java.sun.com/jsf. Look for some examples in **Listing 1**, such as head or script tags.

Listing 2 shows the managed bean, which provides the names of the stickers. The @Named annotation exposes the bean to JSF, and later the bean method is called using the #{stickerSheet.allStickers} expression as the value in the JSF ui:repeat tag in the index.xhtml file shown in **Listing 1**.

DID YOU KNOW?

WebSocket is a full-duplex and bidirectional API, which not only involves the server but the Web browser, too.

Writing Drag-and-Drop Code

In HTML5, drag-and-drop functionality is simplified and requires specific events that are handled using JavaScript code that is divided into two parts: the dragged-object events and the receiving-object events.

The dragged object (that is, the sticker) must define the ondragstart event (see **Listing 1**) and handle it using JavaScript. As we discussed earlier, in JSF 2.2, you can use the pass-through parameters to add this event. The JavaScript code in **Listing 3** specifies what will be executed when the drag starts.

In drag-and-drop interactions, you might have to use an intermediate object to store the data being transferred between components. In JavaScript, you do this by storing the data inside the drag event.dataTransfer property. Because this property allows only strings to be stored, the easiest way to store data is using JSON.

The JavaScript code for the drag event creates the sticker representation, gets from the screen the coordinates of the location to which the sticker was dragged, and—using another HTML5 feature—gets the sticker image filename from the tag's data-sticker

LISTING 1 LISTING 2 / LISTING 3

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:jsf="http://java.sun.com/jsf"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p=" http://java.sun.com/jsf/passthrough">
 <head jsf:id="head">
  <title>Sticker Story</title>
  <script jsf:target="body" jsf:name="story-page.js"/>
  <link jsf:name="styles.css" rel="stylesheet" type="text/css" />
 </head>
 <body jsf:id="body">
  <header>
   <h1>Sticker Story Book</h1>
</header>
  <nav>
  Drag stickers from the left bar to the canvas.
 </nav>
 <aside>
  <h2>Stickers</h2>
  <div id="stickerContainer">
   <ui:repeat var="imgName"
      value="#{stickerSheet.allStickers}">
    <h:graphicImage library="stickers" name="#{imgName}"
          style="float:left" p:draggable="true"
          p:ondragstart="drag(event)" p:data-
          sticker="#{imgName}" />
   </ui:repeat>
  </div>
 </aside>
 ...
 </body>
</html>
```

Download all listings in this issue as text

attribute. In HTML5, you can use custom attributes to store application-specific data by adding the data- prefix to the attribute; in JSF 2.2, you use the pass-through attributes to add such attributes.

For the drop action, the drop target object (that is, the receiving end) must define two event handlers.

One is the ondragover event handler, which must handle the drag-over (hover action) of the mouse to determine whether a drag is valid. To allow drops with further validation, the code just calls ev.preventDefault(), since the default behavior is to deny drops.

The second event handler is ondrop, which handles the drop event when the mouse button is released while something is being dragged. The JavaScript code that handles this event (see **Listing 4**) begins by preventing the default behavior (denying the drop) and then gets the data from dataTransfer. Because we are transferring text, we have to convert it to an object using the JSON.parse method, which is a standard JavaScript method. Next, we convert the dragged object to the actual object that we are going to send to the server. The drop method in **Listing 4** also calculates the coordinate for the position of the upper left corner of the sticker on the canvas and adds the name of the sticker to the object. The object then is converted to a JSON string and finally sent using the WebSocket.send method.

A WebSocket Client with JavaScript

Using WebSocket, the sticker story application sends data to the server when a drag-and-drop action occurs. So how is a WebSocket connection established? Let's see.

After the page loads, the initialize method is called (see **Listing 3**), and the first thing this method does is draw the background for the application using the HTML5 canvas. Next, it opens the WebSocket connection by using the statement socket = new WebSocket("ws://localhost:8080/sticker-story/story/notifications") to establish and open the communication. Note that it uses the WebSocket protocol instead of HTTP and the path is absolute.

Note: If you are running the Web server in a different port, you will need to change this line to use your server's port.

The next line in **Listing 3** defines what the client will call when the server pushes a message to the browser.

Now, take a look at the onSocketMessage method in **Listing 4**. This method will handle

LISTING 4

```
function drop(ev) {
 ev.preventDefault();
 var bounds = document.getElementById("board")
.getBoundingClientRect();
 var draggedText = ev.dataTransfer.getData("text");
 var draggedSticker = JSON.parse(draggedText);
 var stickerToSend = {
  action: "add",
  x: ev.clientX - draggedSticker.offsetX - bounds.left,
  y: ev.clientY - draggedSticker.offsetY - bounds.top,
  sticker: draggedSticker.sticker
 };
 socket.send(JSON.stringify(stickerToSend));
 log("Sending Object " + JSON.stringify(stickerToSend));
}

// Web socket on message received:
function onSocketMessage(event) {
 if (event.data) {
  var receivedSticker = JSON.parse(event.data);
  log("Received Object: " + JSON.stringify(receivedSticker));
  if (receivedSticker.action === "add") {
   var imageObj = new Image();
   imageObj.onload = function() {
    var canvas = document.getElementById("board");
    var context = canvas.getContext("2d");
    context.drawImage(imageObj,
receivedSticker.x, receivedSticker.y);
   };
   imageObj.src =
"resources/stickers/" + receivedSticker.sticker;
  }
 }
```

Download all listings in this issue as text

WebSocket messages from the server side.

The sticker story application sends JSON objects as text using the WebSocket connection. Therefore, when a message is received, it is transformed to a Sticker object and the onSocketMessage method does the following:

- Transforms the data
- Gets the image for the sticker
- Draws the image on the canvas using the coordinates from the sticker

And because the connection is open at this time, sending a message to the server is done using the send method in the WebSocket JavaScript object. It's just that simple.

WebSocket for Real-Time Communication

Creating a WebSocket in Java EE 7 is very easy. You just need to create a couple of annotations and your WebSocket will be configured.

In the sticker story application, the class configured as a WebSocket is the StoryWebSocket class (see **Listings 5a** and **5b**).

Notice the @ServerEndpoint annotation at the top of the class declaration. Just as with servlets, you have to define a mapping. For a WebSocket, the value of the @ServerEndpoint defines the mapping of the WebSocket. This means that you can declare a WebSocket just by using the following and the server will wait for connections at the /websocketMapping URL to connect to the WebSocket:

```
@ServerEndpoint(
"/websocketMapping")
```

For now, skip the encoders and decoders attributes and head over to the methods of the class. To define the methods that will handle the WebSocket events, you use annotations as well.

The @OnOpen, @OnClose, and @OnError annotations are related to the lifecycle of the WebSocket connection. You can annotate a method with any of them and the method will be called when a connection opens or closes or when there is an error in the connection (an exception was not handled).

Our application uses the @OnOpen annotation to push the stickers to the connected user. This way, a newly connected user receives all the stickers previously posted on the canvas by other users.

The @OnMessage annotation is the core of the WebSocket implementation. The annotated method will be called whenever the client sends a message; in the appli-

LISTING 5a LISTING 5b

```
package org.sticker.websocket;

import java.io.IOException;
import java.util.*;
import java.util.logging.*;
import javax.websocket.EncodeException;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;

@ServerEndpoint(
      value = "/story/notifications",
      encoders = {StickerEncoder.class},
      decoders = {StickerDecoder.class})
public class StoryWebSocket {

 private static final List<Sticker> stickers =
Collections.synchronizedList(new LinkedList<Sticker>());

@OnOpen
 public void onOpen(Session session) {
  synchronized (stickers) {
   for (Sticker sticker : stickers) {
    try {
     session.getBasicRemote().sendObject(sticker);
    } catch (IOException | EncodeException ex) {
     Logger.getLogger(StoryWebSocket.class.getName()).log
(Level.SEVERE, null, ex);
    }
   }
  }
 }
```

Download all listings in this issue as text

cation, the client sends stickers and then the WebSocket stores the added sticker and pushes the sticker to all the connected peers.

Notice that the @onMessage method receives a Sticker object (see **Listing 6**) and a Sticker object is written to the other peers using the following method:

```
session.getBasicRemote()
.sendObject()
```

We discussed the fact that WebSocket sends and receives text. So, why do the methods receive a Sticker object, and why are we writing Sticker objects to the sessions? The answer lies in the @ServerEndpoint declaration, specifically in the encoders and decoders attributes.

A WebSocket may use a decoder to transform the text message into an object and then handle it in the matching @OnMessage method, and whenever an object is written to the session, a WebSocket may use an encoder to convert the object to text and send it to the client.

URL path parameters. You can set URL path parameters by adding them to the @ServerEndpoint mapping. Do so by enclosing the parameter name inside brackets:

```
/pathToWebSocket/
{parameter1}
```

To use it, add a string parameter to any annotated method and add the @PathParam("parameterName") annotation to the method parameter. In the example in **Listings 5a** and **5b**, you would add the @PathParam("parameter1") string parameter. You can add as many parameters as you need.

And while we are at it, for the methods annotated using any WebSocket–related annotation, you can receive the WebSocket session as a parameter. In the application, we add it as a parameter, but it is not needed and it can be omitted completely.

The @OnMessage annotation applies to a method that can have an extra parameter. This parameter is the message being received and can take the form of any object decoded using the declared decoders in the @ServerEndpoint annotation or String, if no decoders are specified.

EASY AS 1, 2, 3

Creating a WebSocket in Java EE 7 is very easy. You just need to create a couple of annotations and your WebSocket will be configured.

LISTING 6

```
package org.sticker.websocket;

public class Sticker {
 private int x;
 private int y;
 private String image;

 public Sticker() {
 }
 public int getX() {
  return x;
 }

 public void setX(int x) {
  this.x = x;
 }

 public int getY() {
  return y;
 }

 public void setY(int y) {
  this.y = y;
 }
 public String getImage() {
  return image;
 }
 public void setImage(String image) {
  this.image = image;
 }
}
```

Download all listings in this issue as text

Using the New JSON API with WebSocket

The new Java API for JSON Processing (JSR 353) is part of Java EE 7, and we are using it in our example.

The sticker story application communicates with the client and the WebSocket server by using a JSON string representation of objects. The WebSocket uses decoders and encoders to transparently convert the text sent to the WebSocket into a Sticker object and to convert Sticker objects sent to the client into text.

In the StickerDecoder and StickerEncoder classes (**Listing 7** and **Listing 8**), the application uses the JSON API for the following:

- Reading from the socket stream to get objects
- Transforming objects into JSON string representation and writing them to the socket

You can create appropriate decoders depending on the format in which you want to read the data from the WebSocket. Similarly, you can create encoders depending on how you want to write the data to the socket.

As shown in **Listing 7**, a decoder must implement any interface inside the parent Decoder interface. There are several types of decoders and encoders, and they vary in the parameters that the decode and encode methods take.

The StickerDecoder class in **Listing 7** uses the Decoder.TextStream interface, which allows you to read the data from the socket using a Reader. To convert a string into a JsonObject you use a JsonReader, which conveniently takes a Reader to be constructed.

Note: The JsonReader object should not be created; instead, use the JsonProvider class to create readers and writers.

Listing 8 shows the StickerEncoder class that writes the object to the WebSocket stream.

Conclusion

WebSocket technology, JSF's HTML5 pass-through parameters, and JSON object serialization are powerful additions to Java EE 7 that greatly simplify the development of applications that require constant communication between the browser and the Web server.

In this article, we've built a Web application to show how to combine HTML5 with the JSF 2.2–friendly HTML markup. We also wrote a WebSocket client and drag-and-drop functionality for the application with JavaScript. Lastly, we built a WebSocket server by using the new Java APIs for WebSocket and JSON processing. The example shown in this article barely touches on their potential.

LISTING 7 LISTING 8

```
package org.sticker.websocket;

import java.io.IOException;
import java.io.Reader;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.spi.JsonProvider;
import javax.websocket.DecodeException;
import javax.websocket.Decoder;

public class StickerDecoder implements
Decoder.TextStream<Sticker> {

 @Override
 public Sticker decode(Reader reader) throws DecodeException,
IOException {
  JsonProvider provider = JsonProvider.provider();
  JsonReader jsonReader = provider.createReader(reader);
  JsonObject jsonSticker = jsonReader.readObject();
  Sticker sticker = new Sticker();
  sticker.setX(jsonSticker.getInt("x"));
  sticker.setY(jsonSticker.getInt("y"));
  sticker.setImage(jsonSticker.getString("sticker"));
  return sticker;

 }
}
```

Download all listings in this issue as text

Acknowledgements. *The authors would like to thank Marco Velasco, a young Mexican artist, for the illustration of the sticker story application.* `</article>`

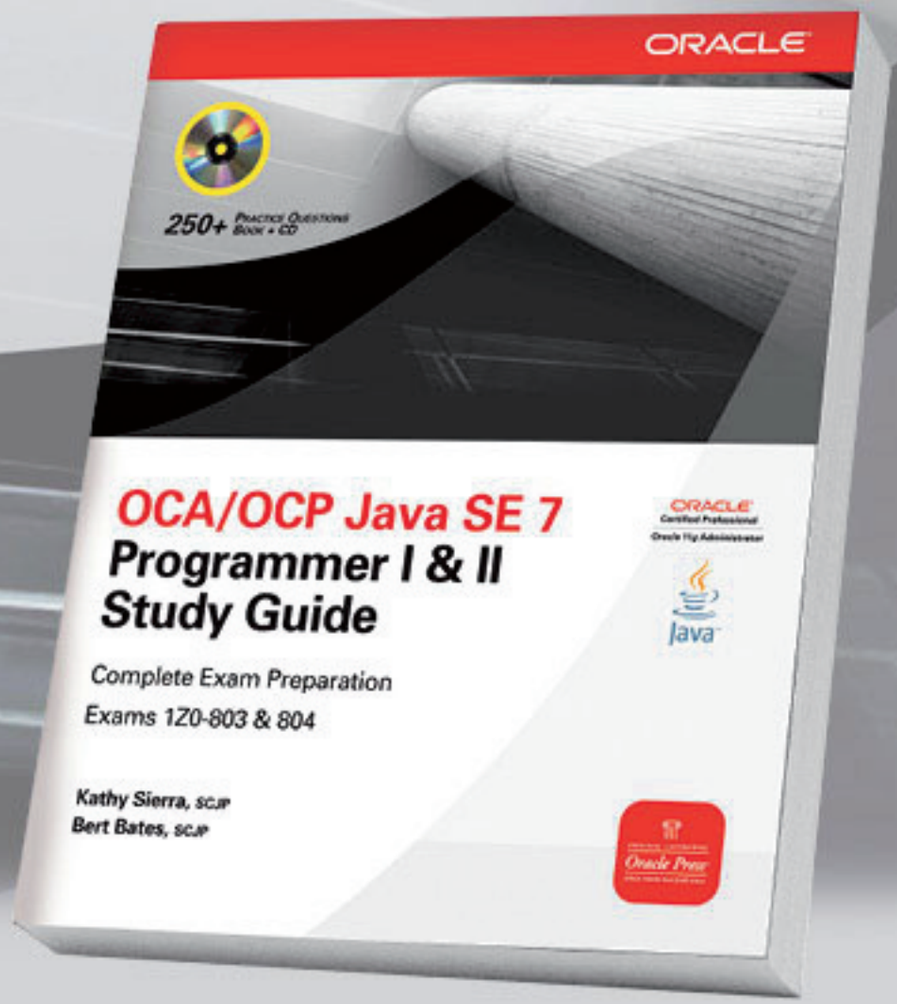
LEARN MORE

- JSF 2.2 JSR
- Java API for WebSocket JSR
- Java API for JSON Processing JSR
- Arun Gupta's video about Java EE 7 and the WebSocket API

Oracle Press™

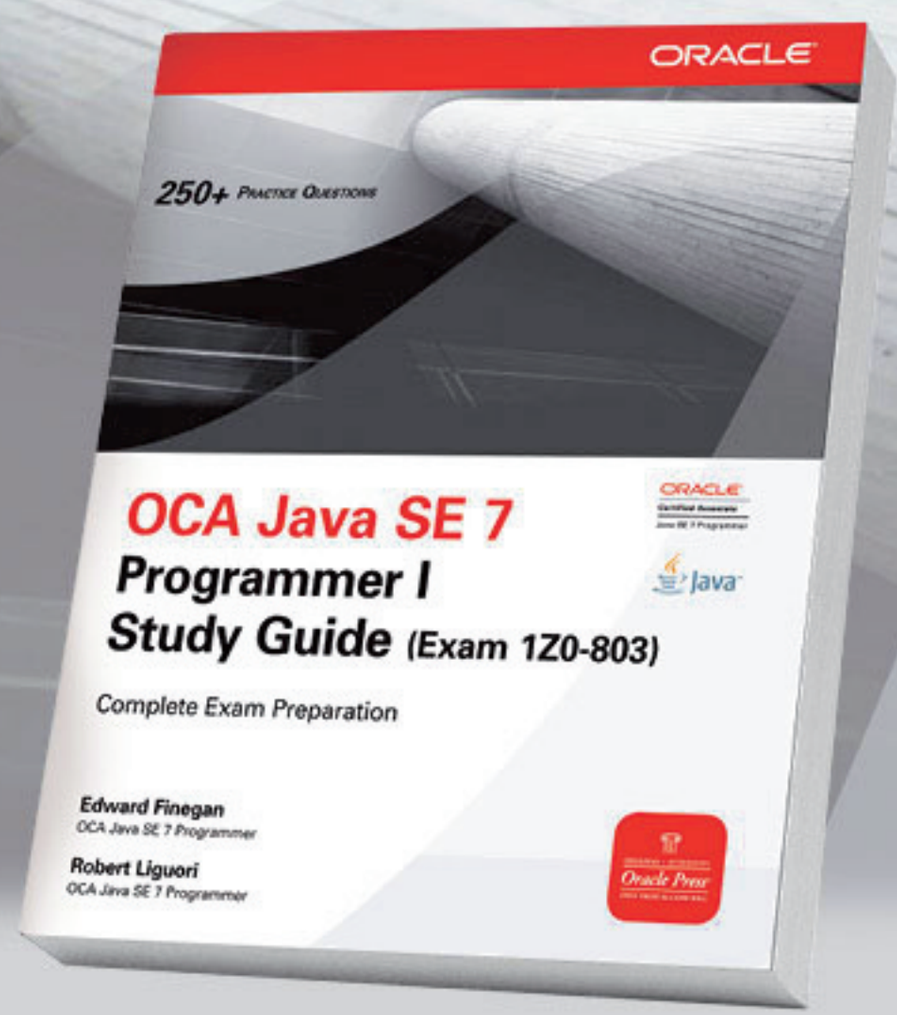
Your Destination for Java Expertise

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



OCA/OCP Java SE 7 Programmer I & II Study Guide
Kathy Sierra, Bert Bates

Written by the co-developers of the original SCJP exam, this book offers complete coverage of all objectives for exams 1Z0-803 and 1Z0-804, challenging practice exam questions, and electronic practice exams.



OCA Java SE 7 Programmer I Study Guide
Edward Finegan, Robert Liguori

Prepare for Oracle Certified Associate exam 1Z0-803 with in-depth coverage of all exam objectives, challenging practice exam questions, and electronic practice exams.

Available in print and e-book formats

Visit us online at OraclePressBooks.com • Follow us and Like us OraclePress

ADAM BIEN

BIO

*Listen to author **Adam Bien** discuss the outcomes of using JavaFX Scene Builder.*

Part 1

Integrating JavaFX Scene Builder into Enterprise Applications

Pragmatically integrate JavaFX Scene Builder output into multiview enterprise applications.

What-You-See-Is-What-You-Get (WYSIWYG) Java editors have promised rapid development and higher productivity since the mid-'90s. Usually the UI construction was based on code generation and protected regions. The developer had to blindly rely on the correctness of the generated code—any modeling mistakes often led to corrupted UIs and some frustration. Without a clear strategy for the separation of the generated code, WYSIWYG-driven development quickly becomes unmaintainable and heavily dependent on a proprietary Java editor.

JavaFX 2 comes with Scene Builder—a tool for the visual construction of user interfaces. This article covers a pragmatic integration of Scene Builder's output into multiview *enterprise applications* using a create, read, update, and delete (CRUD) example.

The sample CRUD application—AirHacks Control (shown in **Figure 1**)—is intended to enable a user to quickly maintain information about attendees who are attending a multiday event. Depending on their registration day, attendees appear in a particular tab. Existing attendees can be maintained in the table as well as in the input area. The data is persisted in a local Apache Derby database, but it could be moved to the server easily without affecting the overall design.

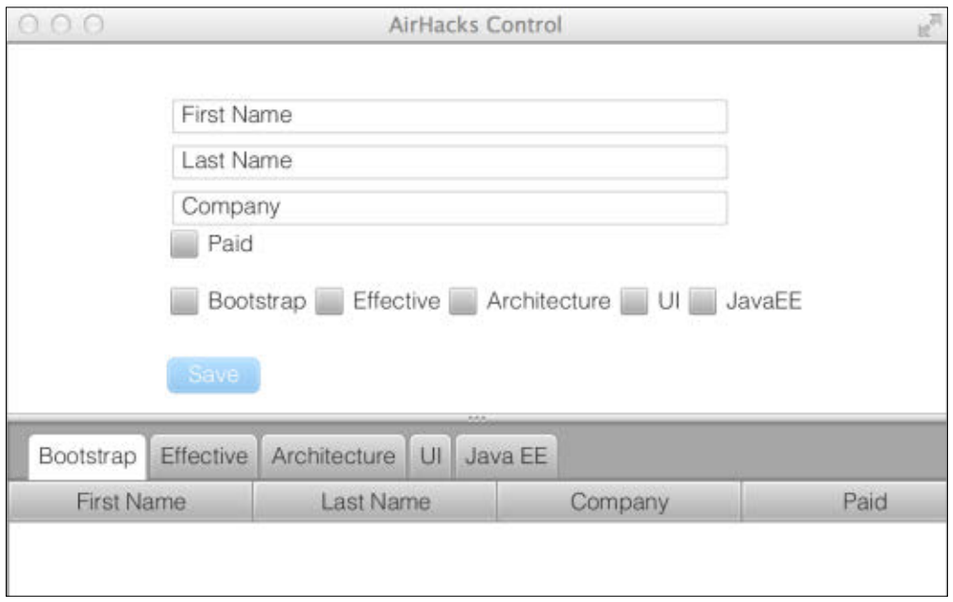


Figure 1

PHOTOGRAPH BY THOMAS EINBERGER/ GETTY IMAGES

Note: The source code for the AirHacks Control application can be downloaded here.

What Does *Enterprise* Mean?

Although the boundary between consumer and enterprise applications has begun to blur, enterprise applications are still driven by a different set of requirements and processes. A successful consumer application tends to focus on a single problem and solves that problem as elegantly as possible. Branding, pixel-perfect positioning, and even nice-looking dock icons significantly contribute to the commercial success of a consumer application.

Enterprise applications do not have to be ugly—however, slickness is not their primary goal. Paramount

are long-term maintainability and views (screens) that directly reflect a use case or user story.

Also, the entire business logic of an enterprise application sits on the server exposed via REST, SOAP, IIOP, or other proprietary protocols. Consumer applications tend to contain more business logic and communicate less frequently with the back end, which is often represented by different cloud providers.

The UI of a successful consumer application is highly influenced by designers and usability experts, whereas the UI of an enterprise application is usually a direct reflection of requests for enhancement (RFEs) from the domain experts and users.

This article focuses on enterprise applications, which typically require larger development teams and, therefore, a clearer and more-obvious structure.

MAKE A SCENE

JavaFX 2 comes with Scene Builder— a tool for the visual construction of user interfaces.

Scene Builder's XML Roots

The Don't Repeat Yourself (DRY) principle made XML disappear from pragmatic Java EE applications. Java EE became extremely productive by eliminating the repetition and maintaining the required metadata with annotations directly on the affected classes. To get started with Java EE, you need to know only a few annotations.

Scene Builder is highly based on XML. The UI definition is maintained in an XML dialect called FXML. Although it uses XML, there is no repetition. In fact, the generated FXML is the "single truth" and so it is still DRY. Also, the main difference between an average Java EE application and a typical UI is the amount of configuration and metadata required. Because fewer conventions are available for a typical UI, even a simple view requires the configuration of colors, layout, positioning, and size, so it vastly exceeds the configuration of any Java EE application.

An FXML file is actually only a configuration of the javafx.fxml.FXMLLoader factory. The FXMLLoader consumes an FXML file and creates the root node (javafx.scene.Parent) with some interesting extensions for Dependency Injection (DI) frameworks, which are covered later.

Instead of creating a Parent manually, you can use the FXMLLoader for this purpose. The entire configuration of the FXMLLoader is provided by Scene Builder with

LISTING 1

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.net.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<AnchorPane id="AnchorPane" prefHeight="243.0" prefWidth="600.0"
styleClass="mainFxmlClass" xmlns:fx="http://javafx.com/fxml"
fx:controller="...">
 <children>
  <VBox id="VBox" fx:id="inputBox" alignment="CENTER"
layoutX="100.0" layoutY="33.0" prefWidth="343.0" spacing="5.0">
   <children>
    <TextField fx:id="firstName" prefWidth="200.0"
promptText="First Name" />
   <CheckBox fx:id="paid" mnemonicParsing="false"
prefWidth="103.0" text="Paid" />
   </children>
  </VBox>
  <Button fx:id="saveButton" defaultButton="true"
disable="true" layoutX="98.0" layoutY="192.0" onAction="#save"
prefHeight="22" prefWidth="57.0" text="Save" />
  <HBox id="HBox" alignment="CENTER" layoutX="100.0"
layoutY="149.0" spacing="5.0">
   <children>
   <!-- .... -->
   </children>
  </HBox>
 </children>
 <stylesheets>
  <URL value="@attendeeinput.css" />
 </stylesheets>
</AnchorPane>
```

Download all listings in this issue as text

the FXML file. Nodes in JavaFX are organized hierarchically; FXML exactly mirrors the structure of the view and it is even readable, as you can see in **Listing 1**.

FXML is not only readable, but complete. You could even generate Java code from an FXML file.

The Challenge and Inversion of Control

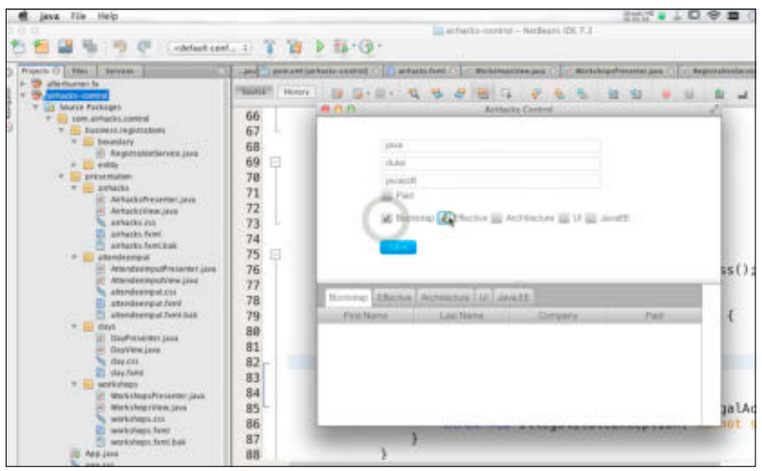
With Scene Builder, we get an instantiated tip of a node tree entirely constructed from FXML. Usually, FXML does not contain any business logic. Script event handlers are optional and not "mainstream." But, what is the link between logic implemented with Java and FXML?

FXML comes with a simple but great idea: any POJO (with a default constructor) can be loaded and created by the FXMLLoader and then bound declaratively to UI events, as shown in **Listing 2**. Methods of the loaded class can be associated with UI events within Scene Builder, as shown in **Listing 3**.

The name of the method has to correspond with the value of the attribute in the FXML file. All UI components can be optionally injected into the POJO by denoting them with the @FXML annotation, as shown in **Listing 4**.

Also, in this case, the name of the injected component has to correspond with the value of the fx:id attribute in the FXML file.

WYSIWYG builders generated complete view code, which was supposed to be integrated with the remaining presentation logic. Scene Builder follows the Inversion of Control approach and instantiates the presentation logic from the generated XML file. In addition, the bindings to the manually created code are generated, which makes Scene Builder almost transparent.



Adam Bien walks you through the AirHacks Control application.

LISTING 2 LISTING 3 / LISTING 4

```
<AnchorPane (...) fx:controller="com.(...).AttendeeInputPresenter">
```

Download all listings in this issue as text

With Scene Builder, the manually created code is not dependent on generated artifacts; rather it is the reverse. FXML contains bindings to the manually created code. The relation is based on naming conventions.

Convention over Configuration

The remaining difference between consumer and enterprise applications is the number of developers and roles involved in the process. Consumer applications are sometimes built with surprisingly few resources; the head count for an average enterprise application can be orders of magnitude higher.

Having more developers requires more structure and convention. Easy and self-explanatory rules lead to higher productivity and maintainability. In optimal cases, there is a recurring recipe for the creation of each view or form.

Let's introduce a convention for our AirHacks Control application: a view with the name AttendeeInput resides in a package with the same name (attendeeinput) and contains the following artifacts:

- AttendeeInputPresenter: The manually created presenter class loaded by FXMLLoader. The presenter contains all the presentation logic and is independent from the generated artifacts.
- AttendeeInputView: A class wrapping the FXMLLoader for convenience. AttendeeInputView inherits from FXMLView, but it is not dependent on any generated artifact.
- attendeeinput.css: An (initially) empty stylesheet for maintaining the look and feel.
- attendeeinput.fxml: The configuration generated by Scene Builder, which contains bind-

ings to the manually created AttendeeInputPresenter.

Creating subclass javafx.scene.Parent with FXMLLoader from an FXML descriptor is simple but repetitive. The class FXMLView not only encapsulates the view construction, but it also implements the aforementioned conventions (see **Listings 5a** and **5b**).

FXMLView provides access to the Parent and the presenter POJO, and it also loads all the required files.

If you can live with the convention, a typical view can be left empty, as shown in **Listing 6**. The subclass is necessary—all the mentioned conventions are derived from the class name, and the Cascading Style Sheets and FXML files are loaded from the package.

Now the view can be instantiated and the presenter obtained using three lines of code (see **Listing 7**). The AttendeeInputView user is fully decoupled from the FXML, the CSS, and Scene Builder.

A Bit More Juice with Guice

So far, no logic has been implemented in the view, and the FXML-instantiated POJO should contain only presentation logic. No matter what kind of enterprise services need to be accessed by the POJO (aka the presenter), the service wrapper—or even the actual business logic implementation—needs to be instantiated, initialized, and passed to the presenter.

A singleton is the obvious choice as a service wrapper but comes with its own shortcoming. A singleton instantiated once is hard to "uninitialize," and so it is difficult to test. Unit tests usually rely on a clearly defined initial state.

In our CRUD example, the JavaFX application directly accesses the Java Persistence API (JPA) 2 EntityManager, which needs to be instantiated and cached between invocations. The entire CRUD functionality is implemented within a single class—the RegistrationService (see **Listing 8**, which increases usability through an EntityManager wrapper).

The RegistrationService is implemented as a simple POJO, not a singleton. During unit tests, the invocation of the method RegistrationService#init can be omitted and both the EntityManager and EntityManagerFactory mocked out for test purposes. However, it would be even nicer to mock out the entire RegistrationService and test the UI with a mock back end. With mockable business logic, you could even prerecord the expected behavior and omit the entire back-end conversation.

Fortunately, JavaFX 2 introduced a callback for DI frameworks. The

LISTING 5a | LISTING 5b | LISTING 6 | LISTING 7 | LISTING 8

```
public abstract class FXMLView {
  public static final String DEFAULT_ENDING = "view";
  protected FXMLLoader loader;

  public FXMLView() {
    this.init(getClass(), getFXMLName());
  }

  private void init(Class clazz, String conventionalName) {
    final URL resource = clazz.getResource(conventionalName);
    this.loader = new FXMLLoader(resource);
    this.loader.setControllerFactory(new Callback<Class<?>,
Object>() {
      @Override
      public Object call(Class<?> p) {
        return InjectorProvider.get().getInstance(p);
      }
    });
    try {
      loader.load();
    } catch (Exception ex) {
      throw new IllegalStateException(
"Cannot load " + conventionalName, ex);
    }
  }
  public Parent getView() {
    Parent parent = this.loader.getRoot();
    addCSSIfAvailable(parent);
    return parent;
  }
```

Download all listings in this issue as text

FXMLLoader passes a class, and the DI frameworks are expected to return an instance and perform the injection magic. The creation of views and presenters is performed by FXMLLoader and encapsulated within the abstract FXMLView. FXMLView is also the natural place for the FXMLLoader instrumentation. **Listing 9** shows enabling DI with the ControllerFactory.

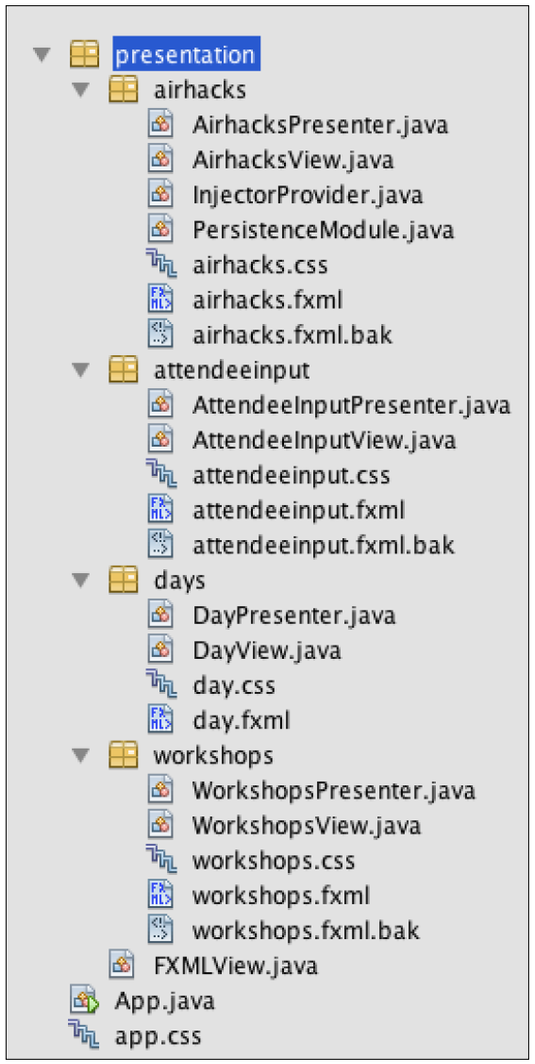


Figure 2

The InjectorProvider utility shown in **Listing 10** is a lean wrapper around the Guice Injector.

Guice DI injection is implemented in a type-safe manner with classes implementing the com.google.inject.Module interface. The PersistenceModule class shown in **Listing 11** cares about the instantiation and lifecycle of the RegistrationService.

The PersistenceModule instantiates and initializes the RegistrationService. Because the RegistrationService.class is bound to the created instance, RegistrationService effectively becomes a singleton without sacrificing any testability.

Now the RegistrationService can be injected with the standard @javax.inject.Inject into any presenter and it becomes available in the Initializable#initialize method. **Listing 12** shows the injection of the business logic.

Guice was chosen only because at the moment it is the simplest and leanest—but still type-safe—implementation of the JSR 330 DI standard. Guice implementation was encapsulated within the InjectorProvider and can be easily replaced with any other DI framework, such as Weld or OpenWebBeans, without affecting the existing presenter implementation. A DI framework replacement would only affect the InjectorProvider utility and make the PersistenceModule superfluous.

LISTING 9 LISTING 10 / LISTING 11 / LISTING 12

```
this.loader.setControllerFactory(new Callback<Class<?>, Object>() {
        @Override
        public Object call(Class<?> p) {
          return InjectorProvider.get().getInstance(p);
        }
      });
```

Download all listings in this issue as text

Single Page, but Many Views

In rich clients, the flow is implemented by substituting smaller view portions (for example, tabs), rather than by using a coarser page flow as in JavaServer Faces (JSF). A single view is represented as a package. Navigation between views needs to be implemented by a superordinate presenter. Also, the application frame needs to be provided by a main view and it is created with Scene Builder providing the skeleton.

There is no difference to the structure already described, except the naming (see **Figure 2**). The package of the main view is named after the application, not after the view's responsibility. Hence, the application view's main responsibility is controlling the navigation between views and provisioning common functionality, such as saving and loading the application folder, which may contain additional supporting classes.

In fact, the InjectorProvider as well as the PersistenceModule classes are located within the main application package. The application package ("airhacks," in our case) resides at the same level as the view-specific packages inside the presentation package.

The AirhacksPresenter creates and coordinates the subordinate

views and adds them to the skeleton layout, as shown in **Listings 13a** and **13b**.

The WorkshopsView encapsulates and manages the tabs and makes the entire lower area replaceable. More precisely, the subordinates views are managed by the WorkshopsPresenter initialized by the FXMLLoader encapsulated within the WorkshopsView.

Neither javafx.scene.control.TabPane nor the javafx.scene.control.Tab instances are injected into the WorkshopsPresenter. Only the javafx.scene.control.Tab content, the javafx.scene.control.AnchorPane, is directly injected into the WorkshopsPresenter. See **Listings 14a** and **14b**.

Scene Builder allows the injection of the leaf views (AnchorPane instead of TabPane, as shown in **Figure 3**), which makes the code significantly leaner.

Our CRUD use case can be solved without any reference to the javafx.scene.control.TabPane class. If we had to react to tab alternation, the TabPane could be also injected. Only the fx:id needs to be set within Scene Builder and a field with the same name and type as TabPane needs to be declared in the presenter.

An Emerging View-Presenter-View (VPV) Pattern

A single utility class, FXMLView, allows us to fully decouple from the FXML file and, thus, from any artifacts generated by Scene Builder. With that, a recurring pattern emerges: the developer instantiates an FXMLView subclass, which initiates the instantiation of the corresponding presenter. With the implementation of the javafx.util.Callback passed to the method FXMLLoader#setControllerFactory, the initialization process is delegated to a JSR 330 implementation, which, in our example, is Google's Guice. Guice not only instantiates the presenter, but it also performs the injection of preconfigured components and cares about the scoping.

For our use case, it is essential to have only a single instance of the RegistrationService available. With the integration of a DI framework, any services or service wrappers (such as the Business Delegate pattern) can be transparently injected to

THE XML FACTOR

Scene Builder is highly based on XML. The UI definition is maintained in an XML dialect called FXML, and although it uses XML, there is no repetition.

LISTING 13a / LISTING 13b / LISTING 14a / LISTING 14b

```
public class AirhacksPresenter implements Initializable {

  @FXML
  AnchorPane input;
  @FXML
  AnchorPane overview;
  AttendeeInputPresenter attendeeInputPresenter;
  WorkshopsPresenter workshopsPresenter;
  @Inject
  RegistrationService rs;

  @Override
  public void initialize(URL url, ResourceBundle rb) {
    AttendeeInputView inputView = new AttendeeInputView();
    this.attendeeInputPresenter =
(AttendeeInputPresenter) inputView.getPresenter();
    this.attendeeInputPresenter.newAttendeeProperty(
).addListener(new ChangeListener<Attendee>() {
      @Override
      public void changed(ObservableValue<? extends
Attendee> ov, Attendee old, Attendee newAttendee) {
        if (newAttendee != null) {
          workshopsPresenter.newAttendeeProperty(
).set(newAttendee);
        }
      }
    });
    WorkshopsView workshopsView = new WorkshopsView();
    this.workshopsPresenter =
(WorkshopsPresenter) workshopsView.getPresenter();
    this.attendeeInputPresenter.selectedAttendeeProperty(
).bind(workshopsPresenter.selectedAttendeeProperty());
    input.getChildren().add(inputView.getView());
    overview.getChildren().add(workshopsView.getView());
  }
```

Download all listings in this issue as text

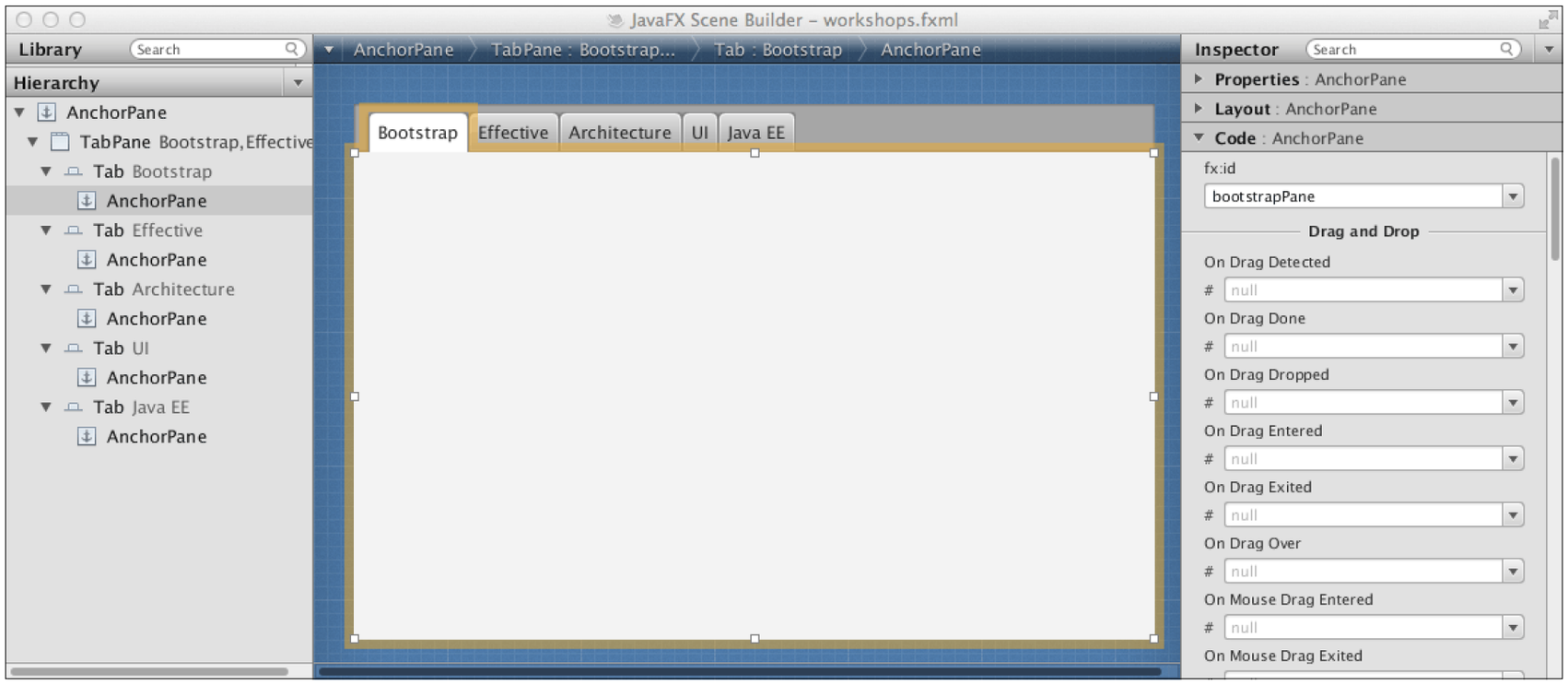


Figure 3

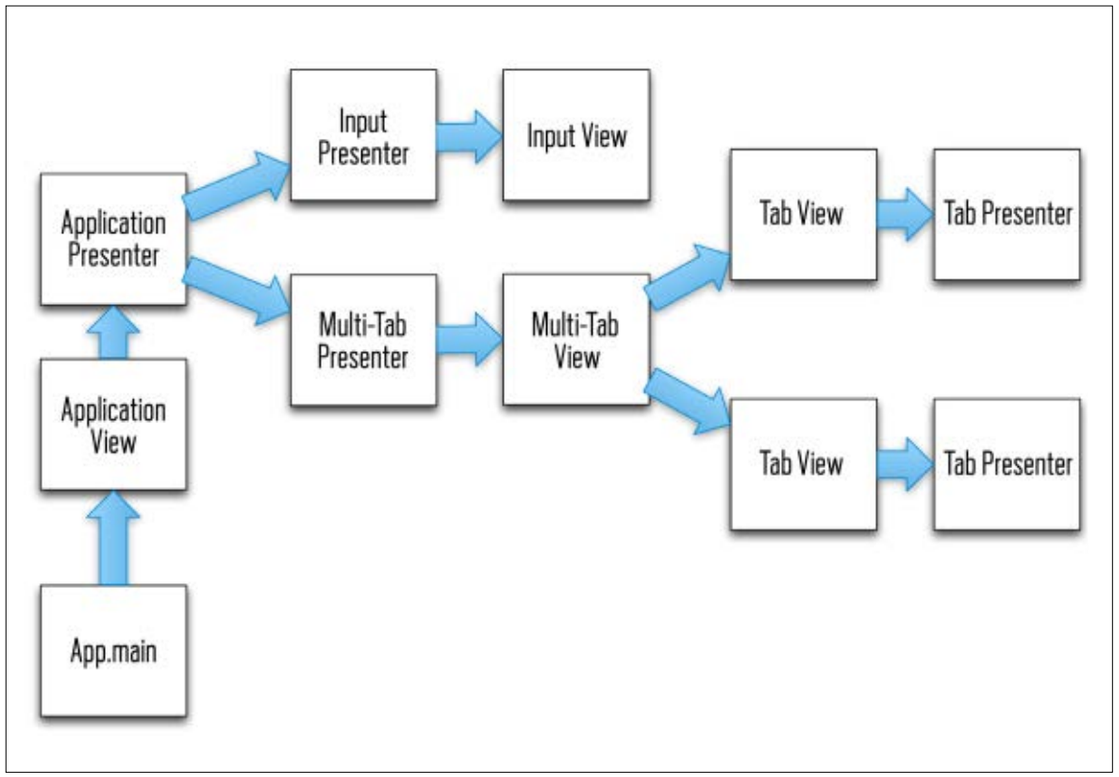


Figure 4

presenters and instantiated by the DI framework.

The presenter is only loosely coupled to the FXML framework or Scene Builder. Only optional @FXML and @Inject annotations are marking the injection points, so there is no other dependency. An ordinary non-FMXL presenter also instantiates views. See **Figure 4**.

All in all, the VPV patterns allow us to seamlessly include generated Scene Builder code (there is no significant difference between XML and Java code) into a manually crafted application. With VPV, the component hierarchy was factored out from the view into a configurable factory (FXMLLoader) instrumented with an FXML file. At any time, you could replace the FXML file with a manually coded layout and also manually instantiate the presenter.

The Model-View-Presenter (MVP) Pattern

The classic MVP pattern was retired by Martin Fowler in 2006. More precisely, the MVP pattern was split into the Passive View, Supervising Controller, and Presentation Model patterns.

Although Fowler's patterns took data binding into account, there is no mention of the integration of WYSIWYG editors. JavaFX enables DI with WYSIWYG by inverting the relations between generated code and manually created code. The VPV pattern introduced here is in the purest form, and it is adequate for the Passive View pattern, which is defined by Fowler as follows:

> *"A screen and components with all application specific behavior extracted into a controller so that the widgets have their state controlled entirely by controller."*

However, you could also bind the view components directly to domain objects or the model, and then this approach would be similar to the intentions defined in the

Supervising Controller pattern, which is defined by Fowler as follows:

> *"Factor the UI into a view and controller where the view handles simple mapping to the underlying model and the controller handles input response and complex view logic."*

Strictly speaking, the separation between the presenter and JavaFX specifics is not very strict. The presenter is still dependent upon JavaFX-specific events and controls. With the introduction of an additional class, all clean presentation logic could be factored out into a JavaFX-independent POJO. The presenter would transform all low-level JavaFX events into higher-level application events.

In this case, the presenter would also have to expose all injected JavaFX widgets as Java primitives. The "pure" POJO presenter would not be coupled to any UI framework and could potentially be reused across frameworks such as Swing, JSF 2, Wicket, or Vaadin. Because the JavaFX-dependent presenter, as well as the back-end services, could be easily mocked out, such a plain POJO presenter would also increase testability even further. This pattern is available under the name Presentation Model, which is defined by Fowler as follows:

> *"Represent the state and behavior of the presentation independently of the GUI controls used in the interface."*

However, there are too many presenters in the description. This is also the reason why the Presentation Model pattern is often called the Model-View-ViewModel (MVVM) pattern: the View (subclass of the FXMLView) contains specific GUI controls and defines the appearance; the ViewModel (presenter) represents the state and behavior of the presentation; and the Model is the domain object (Attendee JPA entity). The MVVM pattern also introduces a controller (the super-presenter, for example, AirhacksPresenter), which implements the workflow and mediates between the views.

On the other side, an additional class would drastically decrease developer productivity and introduce code duplication. Views with complex presentation logic could benefit from a pure presenter, but it is not a good idea to maintain an additional class for simplistic views.

IN A BIND

With Scene Builder, the manually created code is not dependent on generated artifacts; rather it is the reverse. **FXML contains bindings to the manually created code.** The relation is based on naming conventions.

Conclusion

The inverted relation between the generated and manually crafted code introduces interesting opportunities for rapid application development. The combination with Convention over Configuration boosts productivity and maintainability at the same time. Scene Builder can be used without any negative impact on code clarity and so maintainability.

The next article in this series will focus more closely on the interactions between the views and presenters by combining JavaFX data binding with Scene Builder, FXML, and the Supervising Controller pattern. `</article>`

LEARN MORE

- Scene Builder
- Source code for AirHacks Control application
- The ideas extracted from AirHacks Control as "framework"
- afterburner.fx

JOSH JUNEAU

BIO

*Listen to author **Josh Juneau** introduce the topic of integrating Jython with Java.*

Part 2

Jython 101—A Refreshing Look at a Mature Alternative

Learn how to integrate Jython with Java.

Jython is an implementation of the Python language for the Java Virtual Machine (JVM). As the name implies, it allows developers to use the Python language syntax and the standard library, plus apply coding patterns that are used with CPython, the canonical implementation of Python written in the C language. Jython is also an old-timer when it comes to dynamic languages for the JVM. As such, it is a great option for developing solutions for the JVM in a productive and dynamic manner.

This is the second article in a series on Jython. In the first article, we took an introductory look at using Jython and some of the benefits that Jython can add to a developer toolset. In this article, we will discuss one of the most beneficial features of Jython: its ability to integrate with Java.

Using Java from Jython

As developers for the JVM, we have numerous libraries at our disposal that can be used to help build robust applications. That said, Jython developers can access an increased toolset by combining the productivity of the Python programming language with the ability to use existing Java libraries.

IT'S DYNAMIC
Jython is a **great option** for developing solutions for the JVM in a productive and dynamic way.

Any library that is part of the JVM can also be used with Jython, which means that Jython developers have both Python and Java libraries available. That is, pure Python libraries (not those implemented in C++) are available for use via Jython.

Many third-party Java libraries, or libraries written in other JVM languages, can also be used via Jython. Simply adding third-party libraries to the CLASSPATH makes the libraries available. By default, nearly all libraries that are available on the JVM are inherently available for use by Jython.

To utilize Java code from within Jython, it must first be imported into the namespace. A standard Python import statement can be used to import any Java class. It is important to note that if a name conflict occurs, the most recent import overrides any other modules or Java classes by the same name. Therefore, it sometimes is necessary to use the import as syntax when working with imports. Not only does this import syntax provide an alias for the code that is imported, but it can also be a matter of convenience if you use a shorter name.

Let's begin by looking at a couple of examples using the Java Logging API. To begin working with it, simply import the necessary Java classes. Java classes are imported using the standard Python import statement, as follows:

```
from java.pkg import JavaClass
```

In **Listing 1**, a log file is created and messages are written to it to demonstrate the use of a standard Python import statement.

The example in **Listing 2** changes the first example just a bit and utilizes the import as syntax, rather than the standard import, as follows:

```
From java.pkg import MyCls as A
```

As mentioned previously, it is also possible to include external or third-party Java libraries in Jython applications. In order to do so, ensure that the JAR file containing the Java library is contained within your CLASSPATH, and then import as you would normally. To add a JAR file to the CLASSPATH, use one of the following techniques:

For a UNIX-based OS, use this:

```
export CLASSPATH=$CLASSPATH:/
jar
```

For Microsoft Windows, use this:

```
set CLASSPATH=C:\.
jar%CLASSPATH%
```

To demonstrate the use of external or third-party Java code, a JAR file named JythonArticleExamples.jar has been provided with the source code for this article. A class named JavaExample accepts two int values as input and then returns either the sum or product of those two numbers, depending on the method that is called.

The code for the JavaExample class can be seen in **Listing 3**. To use the example class from Jython, add the JAR file containing the class to the CLASSPATH, and then import, as demonstrated in **Listing 4**.

When a Java class is brought into use from Jython, type coercion occurs on any Java types that are present. Java types are coerced into the Python types that equate most closely to their native types. To see a list of the Java types and what Python types they are coerced into, see Table 10-1 in *The Definitive Guide to Jython*.

In Java, interfaces play a key role in object-oriented development. Jython developers can use interfaces as well, and Jython classes can also subclass Java classes. To demonstrate this concept, the Java interface in **Listing 5** (Pool) has been written to define a type of swimming pool, and the Jython class in **Listing 6** implements the Pool interface, as you can see from the following class declaration:

```
class VolumeCalculator(Pool):
```

The ability for Jython classes to implement Java interfaces is key to using Jython modules from Java, as you will see next.

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
>>> from java.util.logging import FileHandler
>>> from java.util.logging import Logger
>>> handler = FileHandler("/java_dev/my_log.log")
>>> log = Logger.getLogger("app_log")
>>> log.addHandler(handler)
>>> log.info("This is an informational message")
Nov 22, 2012 00:00:00 AM org.python.core.PyReflectedFunction
_call_
INFO: This is an informational message
>>> log.warning("This is a warning message")
Nov 22, 2012 00:00:00 AM org.python.core.PyReflectedFunction
_call_
WARNING: This is a warning message
>>> log.severe("This is a severe message, take caution!")
Nov 22, 2012 00:00:00 AM org.python.core.PyReflectedFunction
_call_
SEVERE: This is a severe message, take caution!
>>> handler.close()
```

Download all listings in this issue as text

Using Jython from Java

There are many use cases for using Jython in a Java application. For example, Jython has a less-verbose syntax than that of Java, and in some cases, code can be more productive or more efficiently implemented in Jython than in Java. Whatever the case, there are a couple of different ways in which Jython can be used from Java code. In most cases, the use of a technique referred to as *object factories* is the most efficient way to work with Jython from Java (as of version 2.5.x or prior). If you are using Jython 2.7.x or later, there are some even more-efficient ways of working with Jython via Java.

Object Factories—Manual Object Coercion

The concept behind an object factory is to call upon a proxy that performs the conversion of Jython into compiled Java source. The code conversion involves performing a series of steps to prepare a specified Jython module and then coding an object factory class that is responsible for making the conversion. The most time-consuming task is the implementation of the object factory for each Jython module that you wish to use, but there are ways to avoid coding a factory manually, which we will look at later.

It is important to understand the concept behind the object factory, so this section shows how to code one manually, although most developers should not choose to use this option.

Note: To work with Jython from within a Java application, the Jython-required dependencies must reside within the CLASSPATH. The easiest way to ensure that all dependencies are in the CLASSPATH is to package the standalone jython.jar with your application. To learn more about using the CLASSPATH with Jython, refer to Appendix B of *The Definitive Guide to Jython*.

To create a Jython module that can be coerced, you must write a Java interface that includes the definitions for any methods that you wish to invoke within the Jython module. The Jython module must then implement the Java interface, and once the module has been converted into Java, the interface is called upon in order to make the method calls.

Let's look at an example of this technique by making some modifications to the volume calculator program that we've previously demonstrated. In **Listing 7**, a Java interface named org.javamagazine.interfaces.PoolCalculator contains a number of method and variable declarations. A Jython module

LISTING 7 LISTING 8

```
public interface PoolCalculator {
    public static final double circularCoefficient = 5.9;
    public static final double rectangularCoefficient = 7.5;

    public double calculateVolume();
    public void setDepth(double shallowDepth, double deepDepth);
    public void setLength(double length);
    public void setWidth(double width);
    public void setDiameter(double diameter);
}
```

Download all listings in this issue as text

named JyVolumeCalculator.py (**Listing 8**) implements this Java interface, and the module is responsible for performing all the volume calculation tasks. Note that in this example, setter methods are specifically implemented to allow for error handling when using the calculator.

The final (and most important) piece of the puzzle is the object factory implementation. The object factory is responsible for coercing the Jython module into a Java class via the use of a Jython utility known as the PythonInterpreter. The PythonInterpreter comes bundled with Jython, and it is an entry point into the Jython runtime that can be used for embedding Jython, thereby compiling Jython into Java. First, obtain a new instance of the PythonInterpreter, as follows:

```
\\ Create PythonInterpreter
PythonInterpreter interpreter =
 new PythonInterpreter();
```

Before the PythonInterpreter can be used to compile a Jython module, it must obtain a reference to the module that we wish to use and store it in a local PyObject wrapper for the module. This can be done in the constructor for the object factory class. The interpreter can execute the Jython code responsible for importing the module for use. After the module has been imported, a call to the interpreter.get() method can be made, passing the string-based name of the module that is being interpreted. The interpreter.get() method will return the wrapped PyObject, as shown below.

```
\\ Obtain reference to module
String j="JyVolumeCalculator";
interpreter.exec(
 "from JyVolumeCalculator " +
 "import JyVolumeCalculator");
\\ Return the PyObject
poolCalcClazz = interpreter.get(j);
```

All of this code can be placed into the constructor so that it is executed when the object factory is initialized. The object factory must also contain a method that is responsible for performing the actual coercion of the PyObject into a Java class, and this creation method is called against the object factory when using the Jython code from Java.

In our example, the method that is named create() makes a call to the _call_ special method of PyObject, which returns a new PyObject. If any parameters need to be passed to the underlying Jython module, they can be passed into the _call_ method as Python objects.

For instance, if you want to pass a parameter with a type of Java String, wrap it in a PyString object before passing to _call_. For an entire listing of Python objects that can be used for passing parameters, see the Jython documentation.

Once a true PyObject has been generated by the _call_ method, it can be coerced into Java by calling its _tojava_ special method, and passing the Java interface that the Jython module implements as a parameter. The _tojava_ method returns a Java object that is of the same type as the Java interface. This resulting Java class is then returned to the caller of the create() method and can be used to work with the Jython object. The complete listing for an object factory is shown in **Listing 9**.

The excerpt from **Listing 10** below demonstrates how to perform the method calls from the Java code. As you can see, the object factory implementation is hidden from the developer who is making the calls against the Jython module.

```
// Create the factory
PoolCalculator volumeCalc =
  factory.create();
// Invoke the Jython methods
volumeCalc.setLength(32);
volumeCalc.setWidth(16);
volumeCalc.setDepth(3, 8);
// Call to perform calculation
volumeCalc.calculateVolume();
```

Behind the scenes, an object factory compiles the Jython code into Java. Since the Jython module implements a Java interface, the methods and variables are accessible to code against. Such an implementation would be impossible if there were no Java interface, because the actual Jython function names are mangled at compile time.

Object Factories Without All the Coding

As mentioned previously, you can use Jython from Java without all the object factory coding. A project titled PlyJy focuses on abstracting away the worries of object factory coding, allowing the developer more time to work with application code rather than internal details.

It is possible to implement an object factory that can be used with any Jython module, rather than creating a one-to-one design. In this case, the object factory is not hardcoded to accommodate specified object parameters, but instead it is implemented generically so that any Jython module will work with it.

Listing 11 shows the implementation of a loosely coupled object

factory such as the one described. Note that the factory is a singleton, so it is instantiated only once per application. A method named createObject() accepts the Java interface that is implemented by the Jython module to be coerced, along with the string-based name of the module. The createObject() method is responsible for creating the PyObject that is used to coerce the Jython module into Java using the call _tojava_.

Using a loosely coupled object factory, such as those available for use from the PlyJy project, is very similar to using a one-to-one object factory, such as the one described in the previous section.

Any Jython module that is to be used from within Java must implement a Java interface, which includes definitions for those methods and variables that will be used from the Java code. Because a loosely coupled factory contains no specific customizations for any Jython module, you must define the setter methods for setting any variable values within the Java interface so that the setters are exposed for use.

As such, instantiation of a loosely coupled factory contains no arguments, and all values must be explicitly set by calling the setter methods. The createObject() method accepts the Java interface that is implemented by the Jython module, along with the string-based name of the module itself. **Listing 11** demonstrates this technique.

Yet another technique that makes use of the object factory strategy involves working with the org.python.core.PySystemState class rather than the PythonInterpreter. It is normally a fairly time-consuming task to instantiate a PythonInterpreter, and if an application requires several new instances of a PythonInterpreter, it makes sense to reduce the overhead.

It is possible to get around working with PythonInterpreter if PySystemState is used to obtain a reference to the importer. For brevity, we will not go into the full details for using the PySystemState object factory. Instead, **Listing 12** demonstrates how to use the technique. If you are interested in learning more about it, refer to *The Definitive Guide to Jython*.

Using JSR 223 to Embed Jython Scripts

There are a couple of different ways to embed Jython in Java. You can either make use of a PythonInterpreter to invoke Jython statements and expressions or use a Jython engine along with JSR 223. We introduced the

LISTING 9 LISTING 10 / LISTING 11 / LISTING 12

Note: The following listing has been excerpted for space. The full code listing is available by downloading the code listings for this issue.

```
import org.javamagazine.interfaces.PoolCalculator;
import org.python.core.PyObject;
import org.python.util.PythonInterpreter;

public class VolumeCalculatorFactory {

  private PyObject poolCalcClazz;


  public VolumeCalculatorFactory() {
    PythonInterpreter interpreter = new PythonInterpreter();
    interpreter.exec(
"from JyVolumeCalculator import JyVolumeCalculator");
    poolCalcClazz = interpreter.get("JyVolumeCalculator");
  }


  public PoolCalculator create () {
    \\ The _call_ function also accepts other Python
    \\ objects as parameters to be passed to the underlying
    \\ module, if needed
    PyObject poolCalcObj = poolCalcClazz._call_();
    return (PoolCalculator)poolCalcObj._tojava_(
PoolCalculator.class);
  }

}
```

Download all listings in this issue as text

PythonInterpreter class previously when discussing the use of object factories.

The PythonInterpreter does as its name implies—it accepts Jython code and interprets it into Java. An instance of a PythonInterpreter can be used to execute multiple lines of Jython code by calling its exec() method, thereby embedding the Jython in Java.

The following lines of code demonstrate how to execute Jython code using PythonInterpreter within Java:

```
PythonInterpreter i =
  new PythonInterpreter();
i.exec("print 'hello'");
```

Listing 13 demonstrates how to use the PythonInterpreter to perform imports and how to use a multiline string to contain your Jython code.

JSR 223 was introduced into the Java language to bring the use of scripting languages into Java. This enabled developers to embed lines of code that were written in different JVM languages, such as Jython, into Java programs. For instance, to execute a line of Jython code from within a Java application, you can include the JSR 223 engine for Jython in your CLASSPATH, and then use lines such as those in **Listing 14** to invoke the engine and execute the code.

Since the release of Jython 2.5.1, the JSR 223 Jython engine is packaged along with Jython, so simply adding Jython.jar to your CLASSPATH makes the engine available for use by your application.

Using JSR 223 is much like using PythonInterpreter, although there is a bit more tweaking that must be done in order to make it work. First, create an instance of the ScriptEngine by calling the ScriptEngineManager.getEngineByName() method and passing the string "python".

Once an engine instance is obtained, Jython code can be sent to the engine by calling the eval() method, much like calling the exec() method of the PythonInterpreter. The one additional worry with this technique is that a ScriptException must be either caught or thrown by the block. However, in some cases, this makes code easier to debug if issues arise. **Listing 14** demonstrates a complete example using the JSR 223 ScriptEngine.

Jython 2.7 and Beyond

More-current implementations of Jython include even more ways to integrate with Java. For example, in Jython 2.7+, it is possible to generate proxy classes (compiled Java classes) by annotating Python objects using annotations via a third-party API named Clamp.

The Clamp project leverages the use of the org.python.compiler.ProxyMaker class to generate proxy classes based upon method signatures that are specified using Clamp annotations. For instance, to coerce a specific Java method signature from a Python

LISTING 13 LISTING 14

```
import org.python.util.PythonInterpreter;

public class PythonInterpreterExample {

   public static void main(String[] args){
     PythonInterpreter interpreter = new PythonInterpreter();

     System.out.println("ArrayList Using Java");
     interpreter.exec("from java.util import ArrayList");
     interpreter.exec("arr = ArrayList()");
     interpreter.exec("arr.add('one')");
     interpreter.exec("arr.add(2)");
     interpreter.exec("for a in arr:"
         + "    print a");

     System.out.println("Jython Standard Input");
     String pyStandardInput =
         "import sys \n"
         + "lang = raw_input('What is your favorite
development language?')\n"
         + "print 'The BEST Development Language is
%s' % lang";

     interpreter.exec(pyStandardInput);



   }

}
```

Download all listings in this issue as text

method by specifying return type and arguments, decorate the Python method with @clamp.method(return_type, arg_types=[], name=None). In this case, return_type refers to the method return type, arg_types is an array of argument types that the method accepts, and name accepts an alternative method name.

Listing 15 shows a version of the volume calculator Python class that is annotated accordingly for use with Clamp. If you would like to have the proxy class generated by simply instantiating the Python class, you could put the code listed in **Listing 16** directly into the Python class.

To generate the proxy class from the decorated Python class, you can call ProxyCompiler from Java as follows:

```
ProxyCompiler.compile(
"src/VolumeCalculatorClamp.py",
"src/VolumeCalculatorClamp");
```

Other techniques for Jython/Java integration that *might* become part of Jython 2.7 and beyond include controlling generated proxies, exposing Java types as Python types for better performance, gradual typing, and blame tracking. To learn more about these and other advanced Java and Jython integration techniques, refer to the PyCon 2013 presentation by Jython experts Jim Baker and Shashank Bharadwaj titled "Integrating Jython with Java."

INTERCHANGE
Any Java class that is part of the standard library **can be imported into Jython** for use.

Conclusion

There are a variety of ways to use Java from Jython and vice versa. Interoperability between the two languages has been a key to the success of Jython. Any Java class that is part of the standard library or is included in the CLASSPATH can be imported into Jython for use. Using Jython from Java is a bit more complex but still offers useful flexibility.

This article demonstrated how to use Jython from Java through object factories and through embedding Jython with both the PythonInterpreter and the JSR 223 ScriptEngine. For more information regarding integration between the two languages, refer to *The Definitive Guide to Jython*.

Acknowledgements. *The author would like to give special thanks to the following Jython/Java integration experts: Jim Baker, Jython core developer and expert Python developer; Frank Wierzbicki, Jython project lead and expert Python developer; and Darjus Loktevic, Clamp project lead.* `</article>`

LISTING 15 LISTING 16

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
from org.javamagazine.interfaces import Pool
from java.lang import (Object, Void, Long, Integer, Float)
import clamp

class VolumeCalculatorClamp(Pool):
. . .
. . .
@clamp.method(Void.TYPE,[Integer.TYPE])
  def set_diameter(self, d):
    self.diameter = d
    self.isRect = False

  @clamp.method(Void.TYPE, [Integer.TYPE, Integer.TYPE])
  def set_depth(self, shallow=0, deep=0):
    self.shallowDepth = shallow
    self.deepDepth = deep

  @clamp.method(Void.TYPE)
  def calculateVolume(self):
    volume = 0
    if self.isRect:
      return self.calc_rectangular()
    else:
      return self.calc_circular()
. . .
. . .
```

Download all listings in this issue as text

LEARN MORE

- Jython Website
- Jython Wiki
- Jython Monthly newsletter

Programmable Communications Network

Use open, standards-based APIs to build compelling communications services.

SEBASTIAN GRABOWSKI, JAROSŁAW LEGIERSKI, AND DOUGLAS TAIT

BIO

Telecommunications is evolving from simple phone calls to a new breed of hybrid services that include video; content from weather, traffic, or media; or anything accessible from the internet.

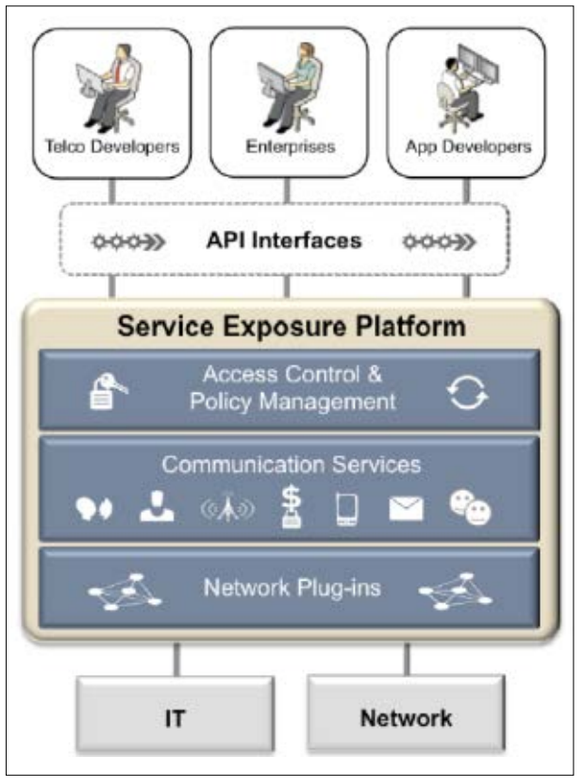


Figure 1

Next-generation wireless networks support these new services. Wireless networks have evolved to handle high volumes of voice, video, and data with all IP technologies. While combining wireless communications with internet services might seem like rocket science, the communications industry has adopted the internet model of exposing core network assets through open application programming interfaces (APIs).

Simple communication APIs include sending messages via short message service (SMS), seeing if someone is present on the network, and getting the location of a device. The communication APIs can get more complex when the applications include charging to a telephone account, authentication for others, multimedia messaging service (MMS), or multimedia conferencing.

This article discusses programming the communications network using open standard APIs to build interesting, compelling, and—with luck—revenue-generating communications services.

The Business of Making Connections

Programming the communications network not only involves APIs, but also service-exposure platforms that connect the internet to the communications network and provide a relationship with the communications service provider (CSP). A developer directs all API interaction to the API service-exposure platform (see **Figure 1**).

The API service-exposure platform protects the network from any harmful activity through access and policy controls. A CSP builds a relationship with developers through Websites, conference, newsletters, and so on, and the relationship is translated into the service-exposure platform through access and policy controls.

Telecommunications service providers are being challenged to stay relevant to their customers, compete with internet voice providers, and provide compelling applications that use their network. Because they are making less money from simple telephone calls, they look to build business by

exposing their core network assets and letting Java developers dream up the next cool application that works on their network.

Therefore, telecommunications service providers are publishing open APIs as standards-based interfaces into their business—the business of making connections. The new communications services are built using a programmable communications network.

TO THE CORE

To expose core network assets, a telephone communications provider will attach a service-exposure platform to its network.

Open Java APIs for Communications

Open APIs over the internet that address many industries have grown dramatically over the past five years. For example, for developing applications with the latest "news," there are open APIs defined at developer.nytimes.com and guardian.co.uk/open-platform. For open APIs into Facebook, go to developers.facebook.com, and for Twitter, go to dev.twitter.com/doc.

The key to API success for vendors is to clearly expose their greatest strength. For telephone service providers, their strength is communications and billing. Some telephone companies claim that exposing network APIs is difficult and risks commoditizing the core voice and messaging assets. They argue that there are difficulties in securely exposing call-control and messaging, which explains their delay in implementing APIs. However, some vendors have deployed an API service-exposure platform that provides access and policy protection and is not only secure and robust, but also enables an important source of revenue for them. Several telephone companies have recently opened up their networks to Java developers. A few examples include Verizon, AT&T, and Orange Labs.

CSPs are unique in their lack of adoption of APIs for service innovation, which puts them at a disadvantage in harnessing ecosystems to deliver value to their customers. The first API implementation for CSPs was based on CORBA, a protocol that was not user friendly for developers and was complicated to implement.

The next-generation telecommunications open APIs were offered as Web services, which were popular in the IT world because of their standards-based use of HTML and XML. There are many kinds of Web service APIs for telecommunications networks (fixed and mobile), which enable software developers to use the capabilities of an underlying network. The APIs are deliberately high-level abstractions and designed to be simple to use. An application developer can, for example, invoke a single Web service request to get the location of a mobile device or initiate a telephone call.

Standards-based APIs for communications have emerged from the ParlayX Web Services API and the Group Special Mobile Association (GSMA) OneAPI:

- ParlayX Web services are defined jointly by the European Telecommunications Standards Institute (ETSI), the Parlay Group, and the Third Generation Partnership Project (3GPP). The APIs are defined using Web Services Description Language (WSDL), and they conform to the Web Services Interoperability (WS-I) Basic Profile.
- OneAPI is a set of APIs supported by the GSMA that expose network capabilities over the internet. The APIs are REST interfaces that can be used to access the capabilities of mobile networks—such as messaging, authentication, and payments—worldwide.

Service-Exposure Platform

To expose core network assets, a telephone communications provider will attach a service-exposure platform to its network (see **Figure 2**). From a software architecture point of view, the service-exposure platform is located between the operator network and the internet.

In **Figure 2**, the southbound (network-facing) interfaces of a service-exposure platform are connected to the operator network and integrated with telecommunications functionalities such as SMS, MMS, location services, presence services, charging services, and so on. Northbound (application-facing) interfaces connect to the internet with standard SOAP and REST interfaces.

The service-exposure platform is also connected with telecommunications operations support systems (OSS)—which are responsible for maintenance, inventory, provisioning, and fault management—as well as with the business support systems (BSS) that handle rating, charging, billing, and payments.

An open, standards-based service-exposure platform is designed to interact with external programs through the API, provide policy enforcement, and leverage the unique network assets with third-party developers and content partners. Many telephone companies have successfully deployed a service-exposure platform in their networks to create an open, standards-based service-exposure layer.

As telephone companies gradually evolve and extend their service domain to better use the IT and Web developer communities, the service-exposure platform maximizes their legacy, fixed-network, and mobile-network investments, such as those for location services, payments, and messaging. It also future-proofs investments that are made for exposing services through IP networks.

Java Applications

Orange Labs created a society of developers called Open Middleware 2.0 Community, which is oriented for telecommunications APIs. With the cooperation of several universities, Orange Labs pursued master's and bachelor's students to develop innovative communications applications. More than 300 Java developers developed 30 services in Java with programmable network APIs. The following are a few interesting applications developed by this community:

- Emergency Button: This is a simple e-health application for locating a person who is in a dangerous situation. The main idea of this application is to establish one-button communication between an elderly person or someone in need of monitoring due to a health condition and a guardian, who needs to be informed about that person's condition. The application uses APIs to receive Unstructured Supplementary Service Data (USSD) messages, receive terminal location information, and send SMS messages.
- Integration Unified Communications with Telco 2.0: This practical integration of an enterprise Unified Communications (UC) system with a telecommunications service provider's service delivery platform (SDP) enables sending SMS and USSD messages and geolocalizing a user's mobile terminal directly from the user interface of the UC system.
- BusStop: This application supports public transportation in agglomerations. Using this application, a user can ask for the location of the closest bus stops and receive information via SMS about routes and timetables
- CourierAssistant: This application improves transportation planning in a courier company. A call-center employee takes an order from a client using a Voice over IP (VoIP) phone. The application uses terminal-location API searches to find couriers that are near the client. The nearest courier is informed through SMS and USSD channels about the package that needs to be picked up.

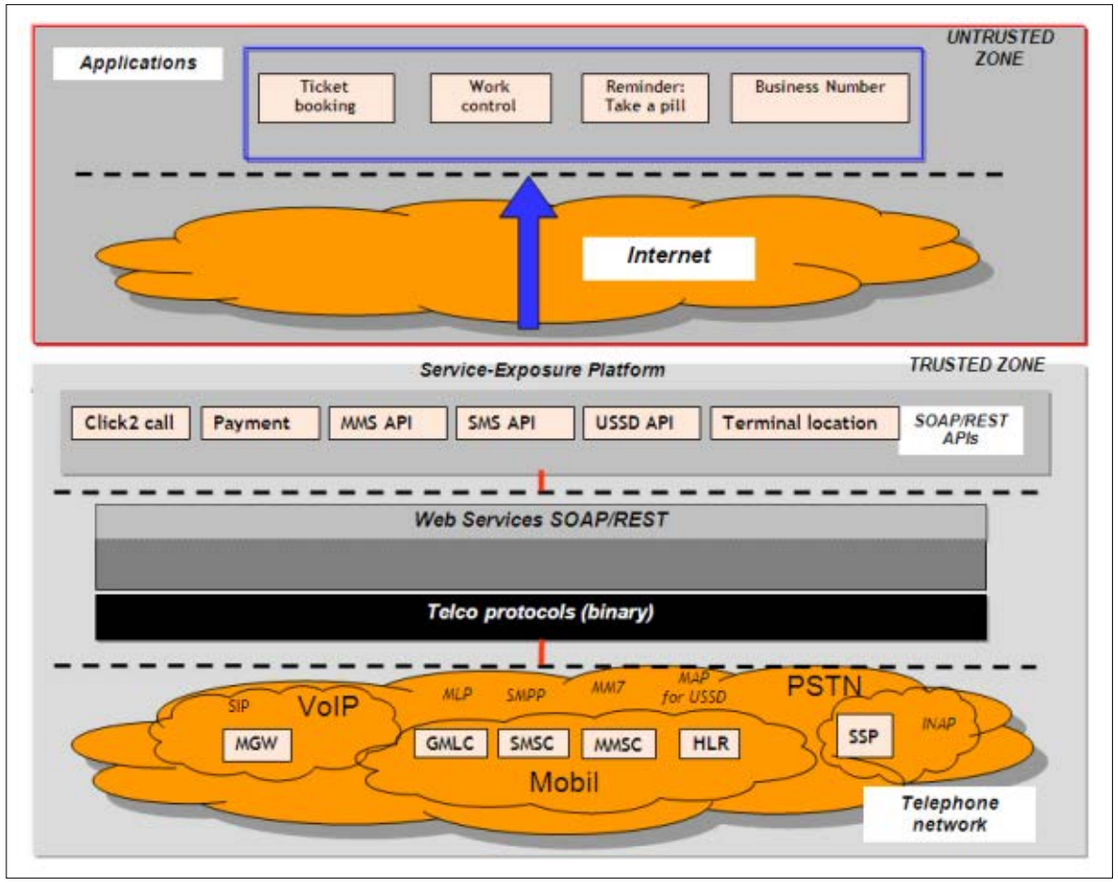


Figure 2

The following sections provide examples, including a ParlayX SMS implementation, a OneAPI payment gateway implementation, a OneAPI authorization application, and a OneAPI quality of service (QoS) API.

Developing a ParlayX SMS Application

The advantage of the ParlayX APIs is that the Java developer can invoke a network function, such as sending an SMS message, without knowing all the minutiae of

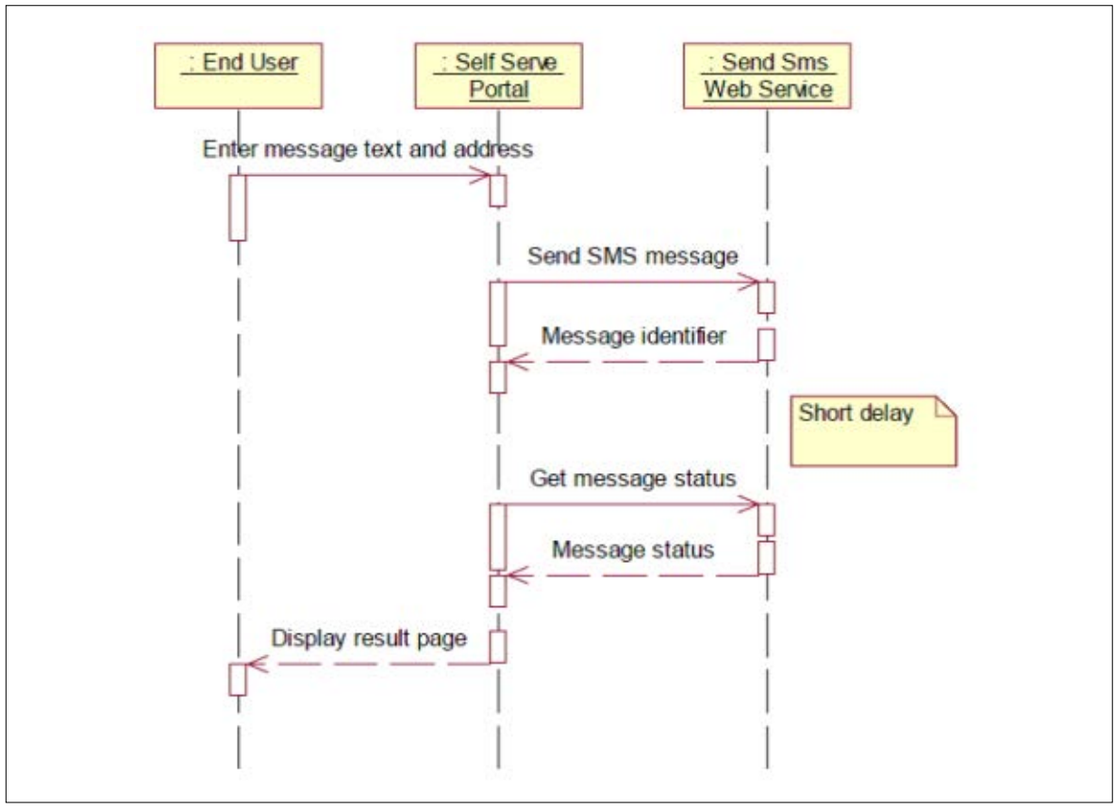


Figure 3

the network. ParlayX provides the primitives to handle communications in a simple way. Sending SMS messages from Web portals is a common capability offered by service providers.

The sequence shown in **Figure 3** illustrates a self-care portal providing SMS service via the ParlayX SendSMS API.

The endpoint for the SendSms interface is http://<host>:<port>/parlayx21/sms/SendSms. The application would include the code shown in **Listing 1**.

The SMS messages are retrieved using a registrationIdentifier that was used when the notification was registered using a provisioning step in the Services Gatekeeper. **Listing 2** shows sample code for getting messages received by the Services Gatekeeper.

OneAPI Payment API

The OneAPI Payment API provides an opportunity to improve the customer experience by allowing purchasing without the need to collect credit card numbers, financial information, user logins, or registration information through a third-party Website. Applications use the API to charge an amount to an end user's account, refund amounts to that account, query charge amount status, and list the amounts charged for transactions. Applications can also reserve amounts, reserve additional amounts, charge against the reservation, and release a reservation. Below are two samples for using the OneAPI Payment API.

OneAPI application requests flow through a service-exposure platform via communication services. A communication service consists of a service type (such as Multimedia Messaging, Terminal Location, and so on), an application-facing interface (also called a *north interface*), and a network-facing interface (also called a *south interface*).

LISTING 1 LISTING 2

```
SendSms request = SendSms();
SimpleReference sr = new SimpleReference();
sr.setEndpoint(new URI(
"http://localhost:8111/SmsNotificationService/services/SmsNoti
fication?WSDL"));
sr.setCorrelator("cor188");
sr.setInterfaceName("InterfaceName");
ChargingInformation charging = new ChargingInformation();
charging.setAmount(new BigDecimal("1.1"));
charging.setCode("test");
charging.setCurrency("USD");
charging.setDescription("some charging info");
sendInf.setCharging(charging);
URI[] uri = new URI[1];
uri[0] = new URI("1234");
request.setAddresses(uri);
request.setCharging(charging);
request.setReceiptRequest(sr);
request.setMessage("we are testing sms!");
request.setSenderName("6001");
SendSmsResponse response = smport.sendSms(request);
String sendresult = response.getResult();
System.out.println("result: " + sendresult);
```

Download all listings in this issue as text

Sample Charge Amount operation. The Charge Amount operation charges an amount directly to an end user's application using the Diameter protocol. **Listing 3** shows sample Java code for invoking payment using the Charge Amount interface.

Sample Refund Amount operation. The Refund Amount operation refunds a currency amount directly to a subscriber's application using the Diameter protocol. **Listing 4** shows sample Java code for invoking the Payment Data interface.

OneAPI Authorization API

With the OneAPI Authorization API, also known as OAuth, the service-exposure platform secures resource access for third-party applications. OAuth authenticates subscribers and authorization of subscriber resources; exposes carrier authentication as a service to partner sites and applications; offers a delegated authentication mode to support third-party authentication; and provides an interface to manage access tokens. The following two samples illustrate how to use OAuth.

Sample OAuth2 Authorize operation. In **Listing 5**, Client1 builds up a request to endpoint /oauth2/authorize and a service-exposure platform redirects the client to the Authorize and Authenticate (AA) server .

Sample OAuth2 Token operation. In **Listing 6**, the token endpoint is used by the client to obtain an access token by presenting its authorization grant (that is, the authorization code) or refresh token. After authentication, the AA server redirects Client1 to OCSG to issue the code, and then Client1 builds up a request to OCSG endpoint /oauth2/a/token.

API DRIVERS

Some telephone companies have deployed an API service-exposure platform that provides access and policy protection and is not only secure and robust, but also enables an important source of revenue for them.

OneAPI QoS API

To allow customers to pay for quality, the service-exposure platform offers differentiated services to customers, such as increased bandwidth when watching a video, clear voice for important calls, and low latency for real-time applications such

LISTING 3 LISTING 4 / LISTING 5 / LISTING 6

```
OneAPIClient client = new OneAPIClient(
JAXBContextResolver.class,
"oma.xml.rest.payment._1.ObjectFactory",
"createAmountTransaction", AmountTransaction.class);
client.setUri("http://localhost:8001/oneapi/1/payment/tel%3A1234/
transactions/amount");
client.setAuthorization(getAuthorizationHeader());
AmountTransaction requestData = new AmountTransaction();
PaymentAmount paymentData = new PaymentAmount();
ChargingInformation chargingData = new ChargingInfomation();
chargingData.setAmount(100);
chargingData.setCurrency("USD");
chargingData.setDescription("Sample");
ChargingMetaData metaData = new ChargingMetaData();
metaData.setOnBehalfOf("Example Games Inc");
metaData.setPurchaseCategoryCode("Game");
metadata.setTaxAmount(0);
paymentData.setChargingInformation(chargingData);
paymentData.setChargingMetaData(metaData);
requestData.setTransactionOperationStatus(
TransactionOperationStatus.CHARGED);
requestData.setEndUserId("tel:1234");
requestData.setClientCorrelator("987654321");
requestData.setPaymentAmount(paymentData);
requestData.setReferenceCode("Rfc4006");
try{
 ret = client.post("", requestData);
} catch (final OneAPIClientException e) {
   .....
}
```

Download all listings in this issue as text

as games. A service-exposure platform allows an entry point for applications to influence QoS and enable management of the relationship among multiple applications, subscribers, and network resources.

The following two samples are for using the QoS API.

Sample ApplyQoS operation. The ApplyQoS API is used by an application to request that either a temporary or a default QoS feature be set up on the end-user connection. **Listing 7** shows sample code.

Sample DeleteQoS operation. The DeleteQoS API is used by an application to release a temporary QoS feature that is currently active on the end-user connection. **Listing 8** shows sample code.

DID YOU KNOW?

An open, standards-based service-exposure platform is designed to interact with external programs through the API, provide policy enforcement, and leverage the unique network assets with third-party developers and content partners.

Conclusion

APIs are now a central strategy for businesses to expand and future-proof their core strengths. APIs are simply an evolution of the Web. They make it easy to work with partners, enable open innovation, and create a positive feedback loop around customer experience and revenue.

CSPs are still looking for new open APIs and services to better monetize their networks. Developers benefit from more open APIs because only they provide the possibility to create interesting and profitable applications. Telephone companies can no longer afford to delay; they need an API strategy across their entire business—not just for third parties, but also for innovation internally and with existing partners—in order to remain relevant to their customers. **</article>**

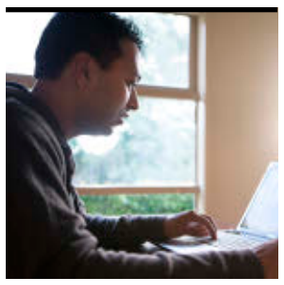
LEARN MORE

- ParlayX Web services
- OneAPI

LISTING 7 LISTING 8

```
ApplyQoSFeatureResponse response = null;
OneAPIClient client = new OneAPIClient(
JAXBContextResolver.class,
"oracle.ocsg.rest.qos.v1_0.ObjectFactory",
"createQoSFeatureProperties", QoSFeatureProperties.class,
ApplyQoSFeatureResponse.class);
QoSFeatureProperties data = new QoSFeatureProperties();
data.setDuration(3600);//unit is seconds
data.setApplicationIdentifier("sampleAppId");
data.setServiceInfoStatus(
ServiceInfoStatus.FINAL_SERVICE_INFORMATION)
data.setsubscriptionId("samepleSubId");
data.setFramedIPAddress(new byte[]{0x0a,0x98,0x47,0x96});
data.setCalledStationId("WAN_1");
MediaComponentDescription mcd = new MediaComponentDescription();
mcd.setMediaComponentNumber(0);
mcd.setMediaType(MediaType.VEDIO);
mcd.setMaxRequestedBandwidthUL(500);
mcd.setMaxRequestedBandwidthDL(1000);
mcd.setMinRequestedBandwidthUL(50);
mcd.setMinRequestedBandwidthDL(100);
mcd.setFlowStatus(FlowStatus.ENABLED);
data.getMediaComponentDescription().add(mcd);
String uri = new String(
"http://localhost:8001/ApplicationQoSService/tel%3A1234/qos");
client.setUri(uri);
try {
   response = (ApplyQoSFeatureResponse)client.post("", data);
} catch (OneAPIClientException e) {
 ......
 }
```

Download all listings in this issue as text



VIKRAM GOYAL

BIO

*Listen to author **Vikram Goyal** introduce the topic of building Instagram applications in Java.*

Part 1

Build Your Own Instagram Java ME Application

Create a Java ME application that captures, filters, and transforms images.

Did you hear about Instagram being sold to Facebook for a cool US$1 billion? And do you think that you can do better with your own Java ME–based application? Well, you can. At least, you can build your own application that mimics the behavior of the Instagram application. Selling it to Facebook . . . well, that's another kettle of fish.

In this article, which is Part 1 of a two-part series, I will cover image manipulation and transformation. In Part 2, I will show how to post the modified images to Facebook.

Note: The source code for the examples described in this article can be downloaded here.

Figure 1

PHOTOGRAPH BY JONATHAN WOOD/ GETTY IMAGES

What Is Instagram?

Instagram is a photo-sharing application that is available for iOS and Android devices. There is a plethora of similar applications for these devices, but Instagram stands out for its ease of use, the many filters that help give images a retro feel, and easy integration into the social sphere.

The Instagram application has two main parts: filters and social sharing. Both are key to the application's success. Of course, sharing is not limited to Facebook; Instagram works with the Instagram Website, Twitter, and several other Websites that support OAuth authentication.

Instagram also provides its own API, which developers can use to create their own applications on top of Instagram and then customize as they see fit.

Let's get started by creating our own photo-sharing application in Java ME.

Creating Our Own Instagram

Several elements are needed to develop a comparable application. We need to take an image using the Mobile Media API (JSR 135); manipulate it using the Image class RGB (red-green-blue) data; and then finally, make this data available as a byte array. And that's just the first part.

To keep it simple, I will show you how to write two filters: a "vintage" filter and a grayscale filter. But first, we need to understand the application flow and how image manipulation works.

PhotoShareMIDlet Application Flow

A simple MIDlet called PhotoShareMIDlet will be used to cycle through the different choices that we have: taking images, transforming them, and publishing them. This MIDlet will create the ImageCanvas class that is used to display the camera (the VideoControl) for taking images. Finally, the ImageTransform class will

be used to actually manipulate the images.

Figure 1 is a screenshot of the application running in an actual Nokia device showing all the available options.

Manipulating images is performed at the pixel level by modifying the ARGB values. *ARGB* stands for alpha, red, green, and blue, and it is the building block for image data. Let's understand this concept in detail before we proceed further.

THE PROCESS

Several elements are needed to develop an Instagram-like application: we need to take an image using the Mobile Media API, manipulate it using the Image class RGB data, and then make the data available as a byte array.

Understanding ARGB

In Java image representation, ARGB represents the pixel-level information about an image. *RGB* represents the red, green, and blue colors of the image, and *A* stands for the alpha level. The alpha level describes how transparent or opaque each pixel is. Thus, an alpha level of 0xff (1111 1111) is completely opaque, while an alpha level of 0x00 (0000 0000) is completely transparent.

It helps to think of a complete image broken down into individual pixels and then identify each pixel using its integer value. Thus, an image in Java is simply an array of integers, and each integer represents the ARGB value for a pixel.

Table 1 represents a single image pixel as a 32-bit integer:

- The first eight bits are what is called the alpha channel, representing the opacity of this particular image pixel. In this case, the image at this pixel is completely opaque.
- The next eight bits are the red channel. In this case, the red value of this pixel is 0101 0101 (0x55).
- The next eight bits are the green channel. In this case, the green value is 1010 1010 (0xAA).
- Finally, the last eight bits are the blue channel, with a blue value of 1111 1111 (0xFF).

With the image data available at a pixel level in Java ME as an array of integers, it becomes very easy to manipulate the data. Thus, in the example shown in **Table 1**, if we needed to convert the pixel to completely transparent (from completely opaque), we would need to manipulate the first eight bits to 0000 0000 (0x00).

1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1

Table 1

In Java ME, you get this data using the getRGB() method of the Image class. To do the reverse, that is, to draw an image once you have the RGB data, you use the createRGBImage() of the Image class.

Now that we know this basic theory of ARGB and how Java ME allows us to manipulate the data, let's see how we can use this to give our images an Instagram feel. But first, we need to capture the images.

Capturing Images

Capturing images is done by using the Mobile Media API, which is a fairly mature technology that is easy to use. (I even wrote the only book on it; see "Learn More" at the end of this article.)

To capture the image, I have created a separate class to handle the whole process: the ImageCanvas class. In its constructor, I initialize a VideoControl so that the user can see through the camera, as shown in **Listing 1**.

Once users have decided to take an image, they can press the FIRE button (the default button) to take a snapshot. The snapshot is stored in an imageArray and presented back to the calling MIDlet, as shown in **Listing 2**. Note that this is done within a separate thread and that the user will be asked for permission to take the image. Also, the camera at this stage must not be used by any other application.

The PhotoShareMIDlet now contains an image that can be shown to the user for further manipulation.

Implementing Image Filters

At this stage, the user has a choice of applying one of two filters that we discussed earlier: vintage or grayscale (see **Listing 3**). Of course, the user can apply these filters one after the other as well.

Image transformation is performed using the ImageTransform class; I have implemented two methods dealing with each manipulation separately.

The vintage filter. A vintage look is achieved by aging the image. Aging is traditionally achieved by changing the red and green channels.

I have done this by creating two generic methods. They are generic because they can be used to change any channel, not just the red or green channel. The

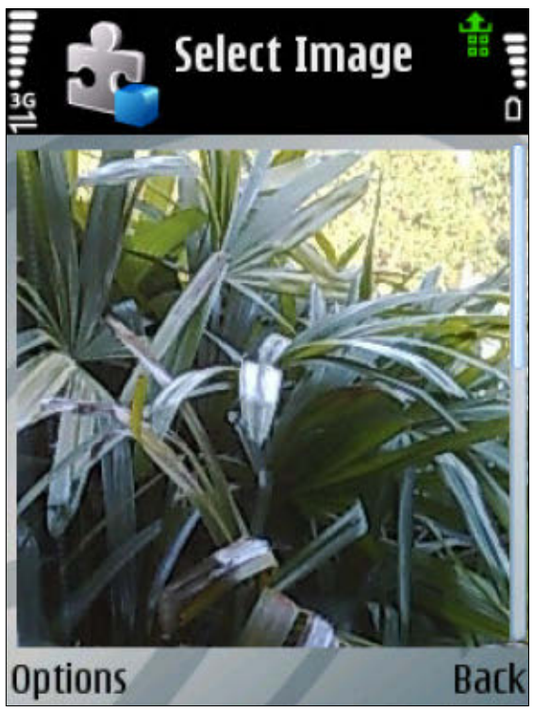


Figure 2

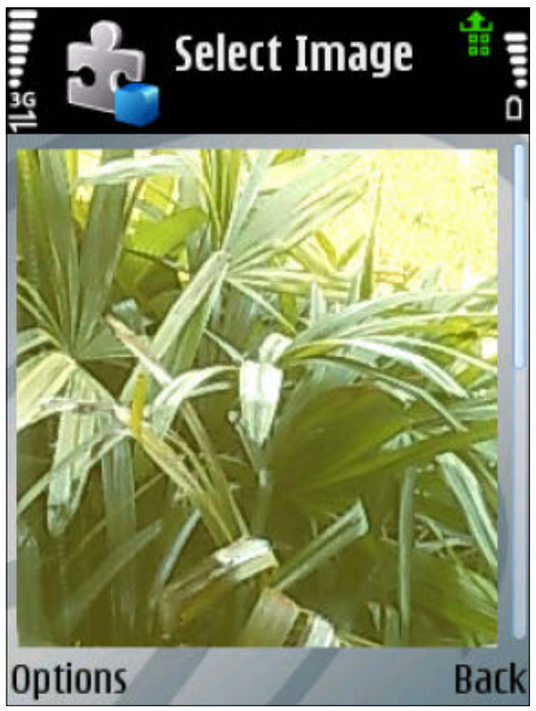


Figure 3

first method is the public method (see **Listing 4**). It accepts information about how much each channel needs to be changed, which is called the *delta value* of the change. The method accepts the original image and the delta value of each channel. Then it proceeds to create the delta arrays for the transformation of each channel.

To create the vintage effect, I have set the red and green delta each at 60. You should play around with these values (and the delta values of the alpha and the blue channels) to see the different effects you can create.

The method calls the internal doTransformARGB_Internal method with the values of the arrays. This internal method does the actual integer manipulation. For vintage effect, only the red and green channels are manipulated, but again, this internal method is also generic. Part of the code that manipulates the red channel is shown in **Listing 5**.

The manipulation of the alpha, green, and blue channels is similar (but not the same); that code is in the associated source code file. The code in **Listing 5** has the effect of changing the red channel with the delta array value that we prescribed while calling this code.

Figure 2 shows an original image, and **Figure 3** shows how

LISTING 1 LISTING 2 LISTING 3 LISTING 4 LISTING 5

```
public ImageCanvas(PhotoShareMIDlet midlet)
      throws Exception {

  this.midlet = midlet;

  videoControl =
       (VideoControl) midlet.getPlayer().getControl(
       "javax.microedition.media.control.VideoControl");

  if (videoControl == null) {
   throw new Exception("No Video Control for capturing!");
  }

  videoControl.initDisplayMode(
VideoControl.USE_DIRECT_VIDEO, this);

  try { // try and set to full screen
   videoControl.setDisplayFullScreen(true);
  } catch (MediaException me) {

   videoControl.setDisplayLocation(5, 5);
   try {
    videoControl.setDisplaySize(
getWidth() - 10, getHeight() - 10);
   } catch (Exception e) {
   }

   repaint();
  }
  videoControl.setVisible(true);
 }
```

Download all listings in this issue as text

Figure 4

the image looks after it has been filtered using the code in **Listing 5**. These images were taken on an actual Nokia device.

The grayscale filter. Grayscale transformation means removing all color from the image and reducing the image simply to black and white. This is achieved by using two methods (see **Listing 6**). The first iterates over all the pixels within the image, while the second removes the color and opacity.

In a very rough way, the code in **Listing 6** achieves grayscale by creating an average of all the individual color components while leaving the opacity as is.

As before, **Figure 2** shows the original image, and **Figure 4** shows the image after running the grayscale filter. These images were captured on an actual Nokia device.

Conclusion

Here in Part 1 of this two-part series, you learned how to create filters and manipulate images using advanced pixel manipulation of the Image class, which was demonstrated by developing two different filters. You learned to do this as part of creating an Instagram-like application.

In the next article in this series, you will learn how to share a manipulated image online using your Facebook identification. `</article>`

LEARN MORE

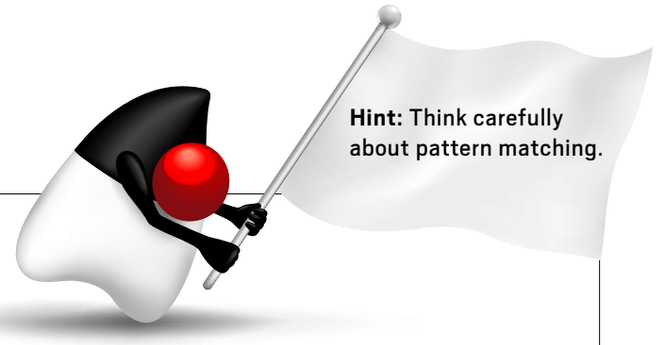
- Mobile Media API
- *Pro Java ME MMAPI: Mobile Media API for Java Micro Edition* (Apress, 2006)

LISTING 6

```
public static Image doGrayScaleTransform(Image original) {

 int w = original.getWidth(); // original width
 int h = original.getHeight(); // original height
 int[] buffer = new int[w * h]; // buffer for new image

 original.getRGB(buffer, 0, w, 0, 0, w, h); // as RGB values

 // now, for each individual pixel, apply the transformation
 for(int i = 0; i < buffer.length; i++) {
  buffer[i] = doGrayScaleTransform_Internal(buffer[i]);
 }

 return Image.createRGBImage(buffer, w, h, true); // return trans image
}

private static int doGrayScaleTransform_Internal(int bite) {

 // create an int array to hold the 4 levels of
 // transforms we are going to do
 int[] out = new int[4];

 out[0] = (int)((bite & 0xFF000000) >>> 24); // Opacity level
 out[1] = (int)((bite & 0x00FF0000) >>> 16); // Red level
 out[2] = (int)((bite & 0x0000FF00) >>> 8); // Green level
 out[3] = (int)(bite & 0x000000FF); // Blue level
 int new_bite = out[1]/3 + out[2]/3 + out[3]/3;
 out[1] = new_bite;
 out[2] = new_bite;
 out[3] = new_bite;
 return (out[0] << 24 | out[1] << 16 | out[2] << 8 | out[3]);
}
```

Download all listings in this issue as text

//fix this /



1 THE PROBLEM

Sometimes when loading data from a file, we have problems with parsing dates. A simple solution that comes to mind is to use SimpleDateFormat. However, is that a good solution?

2 THE CODE

Consider the following code fragment and its result.

```
        String value = "01-04-1989";
        String dateFormatString = "yyyy-MM-dd";
        SimpleDateFormat dateFormat = new SimpleDateFormat(dateFor
matString);
        try {
            System.out.println("Result: " +
                                dateFormat.format(dateFormat.
parse(value)));
        } catch (ParseException e) {
            System.out.println("Error date parsing: " + value + "
in format " +
                                dateFormatString);
            e.printStackTrace();

        }
```

3 WHAT'S THE FIX?

1) Error parsing: 01-04-1989 in format yyyy-MM-dd
2) 0006-09-10
3) 1989-04-01
4) 0104-19-89

In the March/April 2013 issue, Jason Hunter and Boris Shukhat gave us a code challenge around how the Java Persistence API (JPA) handles parameterized SQL and the result set. There were actually two bugs in the code challenge, and the correct answer is both #3 and #4.

For #3, when we set parameters in native SQL they are indexed starting from 1, not 0. This fact is discussed in the JDBC documentation but not in JPA. The programmer was not aware of the fact and used zero-based indexing as with Java collections.

For #4, the query.getResultList() call always returns a List. If the result set is empty, it returns an empty List. This fact is not documented in JPA either.

The correct code looks like this:

```
EntityManager em = ...;
String sql = "select department from emp where fname=?
and mname=? and lname=?";
Query query = em.createNativeQuery(sql);
String[] name=new String[3];
for (int i=1; i<4; i++) query.setParameter(i, name[i]);
List departments = query.getResultList();
if (departments.isEmpty())
      System.out.println("Not available");
else
      for(String department : departments)
            System.out.println(department);
```

This issue's challenge comes from Michal Lisiecki, a software development specialist at the Center for Information Technology in Poland.

GOT THE ANSWER?

Look for the answer in the next issue. Or submit your own code challenge!

ART BY I-HUA CHEN