//JANUARY/FEBRUARY 2014 /

Java™ magazine

By and for the Java community

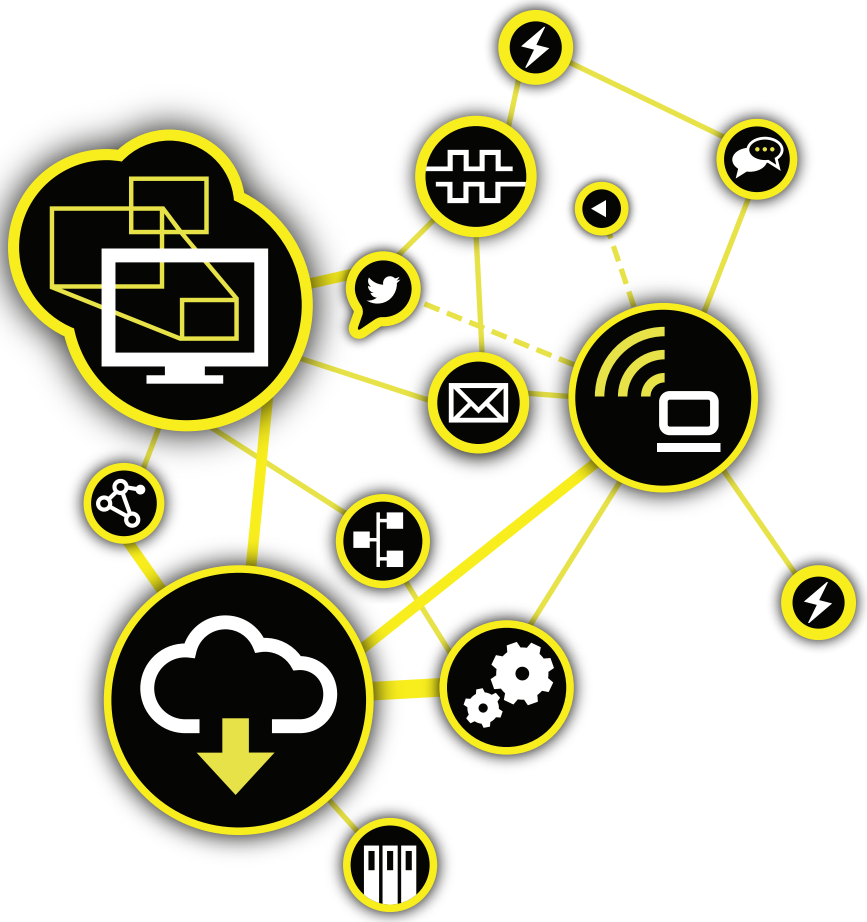
```
// Use lambda expressions integrated with core
// Collection libraries to process big data
// in parallel
int sum =
  txns.parallelStream()
       .filter(t -> t.getBuyer().getState()
             .equals("NY"))
       .mapToInt(t -> t.getPrice())
       .sum();
```

BIG DATA

How big data will change the way you code

ORACLE

//table of contents /



COVER ART BY I-HUA CHEN; PHOTOGRAPHY BY BOB ADLER

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, Arianna Pucherelli
Production Designers
Sheila Brennan, Kathy Cygnarowicz

PUBLISHING
Vice President
Jeff Spicer
Publisher
Jennifer Hamilton +1.650.506.3794
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

ADVERTISING SALES
President, Sprocket Media
Kyle Walkenhorst +1.323.340.8585
Western and Central US, LAD, and Canada, Sprocket Media
Tom Cometa +1.510.339.2403
Eastern US and EMEA/APAC, Sprocket Media
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 (US/Canada)
Oracle Services
+1.888.283.0591 (US)
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.

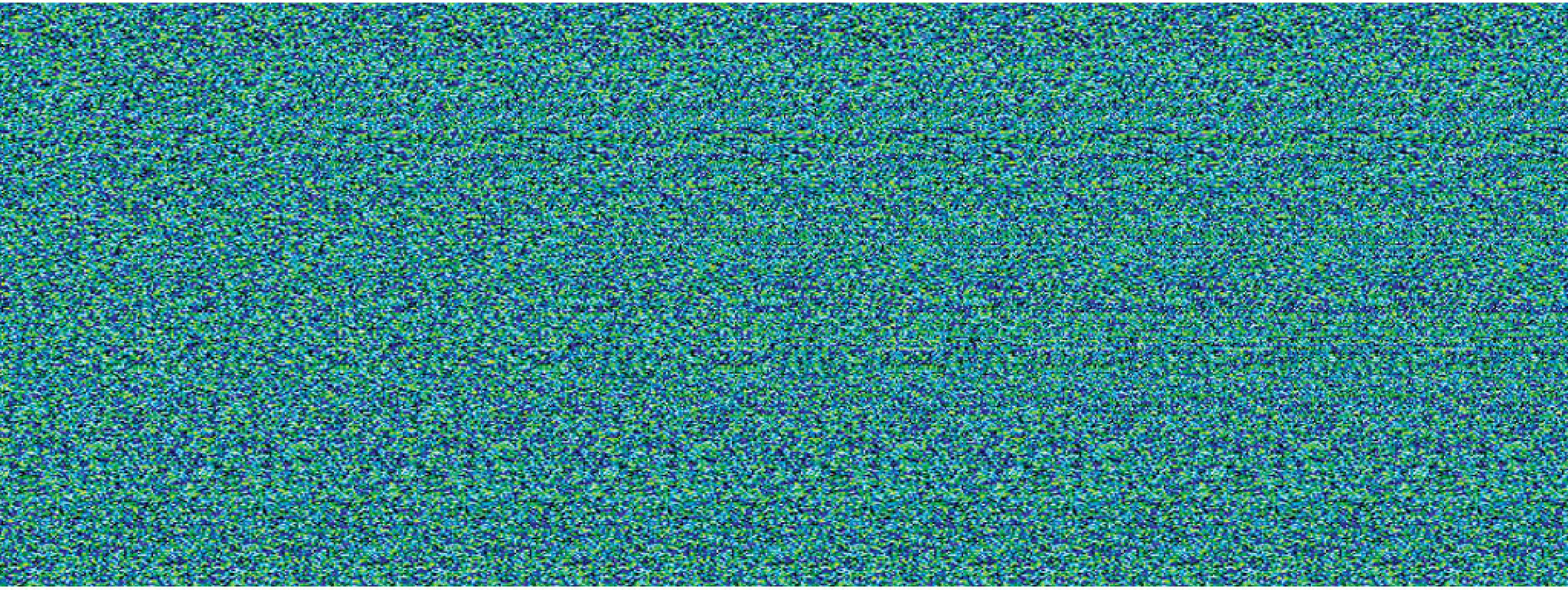
MAGAZINE CUSTOMER SERVICE
java@halldata.com **Phone** +1.847.763.9635

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 GTxcel

The answer is right in front of you



Java Image Enabling SDKs that Help You See the Big Picture

At first glance it may seem difficult, but it's really quite simple. Atalasoft's JoltImage product is a proven SDK for image enabling your Java-based web applications, easily. Image enabling helps to add dimension to your data, so you can uncover insights such as correlations and causations hidden inside your 2-dimensional documents. Our SDK does the heavy lifting for you, saving time, money, and the headaches of figuring it out yourself. Backed by our highly knowledgeable & caffeinated support engineers, JoltImage will enable your success and make the big picture so much easier to see.

Atalasoft JoltImage

Click for tips on viewing the stereogram

Image enabling experts & bacon connoisseurs. Visit us online to see our full line of SDK products for .NET and Java

Atalasoft
A Kofax Company
www.atalasoft.com

Simply put, big data is a big deal. As the volume, velocity, and variety of big data continue to increase, so do the opportunities for creating new applications for big data. The good news is that Java developers are well positioned to take advantage of this opportunity because so many of the tools for big data are built on the Java Virtual Machine (JVM). We've dedicated most of the pages in this issue to big data—just look for our new theme icon to identify which articles tackle this topic.

Big Data and Java

In our Q&A with Oracle's Dan McClary, we explore why Java and big data are such a good fit, and what some of the implications of big data are for Java developers.

We also get hands-on with many of the available tools for big data. In "Big Data Processing with Java," Fabiane Nardon and Fernando Babadopulos help you determine whether you've got a big data problem and introduce Apache Hadoop and some of the other tools that you can use to produce faster and more-efficient applications. Tom White, author of *Hadoop: the Definitive Guide*, provides an introduction to Hadoop; Kim Ross explores Apache Cassandra; and Trisha Gee discusses the flexibility of MongoDB.

"Power to the People" shows Java and big data in action, with sizeable results: Opower customers have saved more than US$350 million dollars on their utility bills and reduced greenhouse gases at the same time.

Are you creating new, disruptive applications for big data? Let us know how big data is changing the way you work.

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

MAKE THE FUTURE JAVA

Java™

FIND YOUR JUG HERE

One of the most elevating things in the world is to build up a community where you can hang out with your geek friends, educate each other, create values, and give experience to your members.

Csaba Toth
Nashville, TN Java Users' Group (NJUG)

LEARN MORE

ORACLE®

DEVOXX: AN EXCITING TIME FOR JAVA

Clockwise from top left: a robotically controlled chess set; Brian Goetz talks lambdas; Stephen Chin (left) and Richard Bair play chess; Mark Reinhold discusses why it's an exciting time for Java.

PHOTOGRAPHS COURTESY OF DEVOXX

Devoxx, held in Antwerp, Belgium, November 11 to 15, featured 9 tracks, 200 speakers, and 3,500 attendees from more than 40 countries, and proved itself to be the premier European developer conference. At the keynote, Oracle's **Mark Reinhold** said that it's an exciting time for Java. "No technology can survive by standing still," he said. Java SE 8, scheduled to be released in March 2014, is a great step forward for Java. Oracle's **Brian Goetz** put Java SE 8 in context, reiterated the Java language design principles, and explained the impact lambdas will have on Java programmers. To provide context for Java SE 8, he admitted that it had taken a long time for the release to come to fruition. The community spent a long time discussing the concept and implementation, he said. "With a community of 9 million developers, it's not always easy to keep everyone happy," he said with a smile. "It's easy to put in a language feature if you are also ready to rip it out quickly," but Java doesn't work that way, he added. Finally, and most importantly, the team wanted to do lambda expressions the right way, he said—not as something that felt bolted on.

Goetz then reminded the audience of the key Java language design principles: reading code is more important than writing code; simplicity matters; one language, the same everywhere.

DataCrunchers' Nathan Bijnens (left) and Geert Van Landeghem on big data

The first principle is the most important one: you write a line of code once but read it many times. "The more your code reads like the problem statement, the more successful you will be," he said. Lambdas provide a way of writing more readable and understandable code, and they will improve developer productivity, he said.

The Internet of Things was another keynote topic. With the new ARM support for Java SE and Java ME, Java can run on embedded devices from large to small. **Stephen Chin** and **Richard Bair** showed Java in action on several devices, including the DukePad, a Duke robot, and a robotically controlled chess set.

Oracle's Angela Caicedo shows off a Duke robot.

Globalcode Founder Vinicius Senger describes his embedded panel.

IoT HACK FEST AT DEVOXX

The Internet of Things (IoT) Hack Fest at Devoxx gave conference attendees a chance to build embedded applications using different boards and the Leap Motion controller.

Globalcode Founder **Vinicius Senger** and Java Champion **Yara Senger** introduced embedded Java on the Raspberry Pi and an embedded panel of their own design connecting Arduino, Raspberry Pi, Gemalto, and Beaglebone black boards with temperature, distance, heart-rate, alcohol-in-breath, and sound sensors, plus a camera.

During the two-day event, developers hacked the embedded panel and created applications using the sensors and relays with embedded Java, Pi4J, JavaFX, Fuzzy Logic, Things API, Apache Camel, Mosquito, and more. The final applications controlled lights, distance sensors, and animated LCD displays, to name a few.

ZeroTurnaround Senior Developer **Geert Bevin** presented the Leap Motion Controller, which detects hand and finger movements to interact with games and other software. He used it to move a robot through a maze. Bevin also built an application using a Leap Motion Controller and the Raspberry Pi with an Arduino Bridge. He controlled a multicolored LED strip with hand gestures.

Claude Falguière (right) demonstrates a Java-based instructional game for kids.

Geert Bevin describes how he created a "rainbow" of colored lights.

Java SE News

Oracle has created two new resources, the Java RIA Security Checklist and the Java Security Resource Center, to help you prepare for the next Java SE update, Java SE 7 update 51 (scheduled for January 2014). This release changes the deployment requirements for Java applets and Java Web Start applications with two new requirements: use of the Permissions manifest attribute and valid code signatures.

These changes will not affect developers of back-end or client applications; the scope is limited only to Java applets and Java Web Start applications.

The changes scheduled for Java SE 7 update 51 mean that the default security slider will require code signatures and the Permissions manifest attribute. The Java RIA Security Checklist provides best practices to help development teams track work necessary to accommodate user prompts.

The Java Security Resource Center aggregates security-related information for the Java community based on your role: developer, system administrator, home user, or security professional.

Note: To ensure that end users' systems are secure when using Java-based content, Oracle *strongly* recommends that you always upgrade to the most recent release. You can remove old versions of Java either during upgrades or by using the Java Uninstall Tool on Java.com.

JAVA CHAMPION PROFILE
ROMAN ELIZAROV

***Roman Elizarov** is a Java developer living in St. Petersburg, Russia. He works on financial software for brokerages and financial exchanges at Devexperts. Elizarov has contributed many Java bug fixes and has been active in JUG.ru. He became a Java Champion in July 2006.*

Java Magazine: Where did you grow up?

Elizarov: I was born and raised in St. Petersburg, Russia.

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

Elizarov: At around age 12, I read a magazine article on how to build your own primitive computer. That captured my imagination.

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

Elizarov: I did not own a computer until much later in life. I had tinkered with different computers and different languages. My first long-term language was Pascal on Yamaha computers in school, quickly superseded by Borland Turbo Pascal on IBM PS/2 computers.

Java Magazine: What was your first professional programming job?

Elizarov: I had a summer job writing a document management system using MS FoxPro for my father.

Java Magazine: What do you enjoy for fun and relaxation?

Elizarov: I read a lot.

Java Magazine: What happens on your typical day off from work?

Elizarov: I spend most of my free time with my wife and kids.

Java Magazine: Has being a Java Champion changed anything for you with respect to your daily life?

Elizarov: I fell in love with Java as soon as I learned it around the year 2000. I've been acting as a Java Champion since that time. The title is just recognition of this fact.

Java Magazine: What are you looking forward to in the coming years from Java?

Elizarov: I'm looking forward to Java SE 8, which brings closures and simplifies collection manipulation, radically shifting Java coding style. I'm looking forward to modularity in the future updates of the Java platform, which is long overdue given the platform's size.

Elizarov posts announcements of all his articles and conference talks, as well as some other programming news and information, on his Facebook page. You can also follow him on Twitter @relizarov.

FEATURED JAVA USER GROUP

GUADALAJARA JAVA USER GROUP

Left: Guadalajara Java User Group members; right: a speaker at a recent JUG Guadalajara meeting

The Guadalajara Java User Group (JUG) was founded in Mexico in March 2011 by a group of individuals who wanted to share their knowledge and passion for Java with the local developer community. JUG Guadalajara co-organizer **Eduardo Moranchel** reports that **Ulises Pulido**, **Roberto Carrera**, **Gilberto Alvarado**, **Raul Herrera**, and **Daniel Valencia** held the first meeting, with the theme "Java for non-Java Developers," in a modest conference room. After that first meeting, "slowly but surely people began to join the group," Moranchel says.

The early growth happened largely through word of mouth: People who were attending the JUG's early meetings talked to their friends who were involved in or interested in Java technology, and convinced them to come to a meeting. The opportunity to meet people who shared their interests and learn new aspects of Java technology led to more and more active JUG participants over time.

Moranchel says that JUG Guadalajara had more than 100 active members by the time the group decided to become a formal JUG in early 2013. Today, the group's Facebook page has 247 fans, the @Java_Gdl Twitter account has 300 followers, and their Meetup group has 162 active members.

Why start a JUG in Guadalajara? Moranchel notes that Guadalajara has "great potential to become a technological hub in Mexico; great talents are growing and being fostered by top technology companies that are active sponsors for the JUG."

The group holds monthly meetings that focus on a wide variety of Java technologies, with pizza and drinks provided at most events. Although the focus is on Java technologies topics, the discussions go beyond Java to helping members become better developers in general and addressing career planning. The meetings are usually held in conference rooms provided by sponsoring companies and coworking spaces.

In addition to the regularly scheduled meetings, JUG Guadalajara hosts a major event each year. The latest was the "Guadalajara Java Day" conferences and workshops in July 2013, in which 100 Java developers participated.

The JUG also recently initiated participation in the Adopt-a-JSR initiative. Through consensus, the JUG will soon begin working on the CDI, JAX-RS, and Date and Time JSRs.

PHOTOGRAPH BY GETTY IMAGES

EVENTS

Embedded World *FEBRUARY 25–27*
NUREMBERG, GERMANY

The Embedded World Conference brings together developers, designers, and scientists concerned with the development, purchasing, procurement, and application of embedded technologies. The event focuses on innovations in embedded systems and development and features more than 270 lectures and workshops delivered by forward-thinking speakers from the embedded community and 900 exhibitors.

FOSDEM
FEBRUARY 1–2
BRUSSELS, BELGIUM
FOSDEM is a free conference that brings together more than 5,000 developers from the open source community.

Jfokus
FEBRUARY 3–5
STOCKHOLM, SWEDEN
Jfokus is the largest annual conference for Java developers in Sweden. Topics include Java SE, Java EE, front-end and web development, mobile, continuous delivery and DevOps, the Internet of Things, cloud and big data, future and trends, alternative languages, and agile development.

HADOOP INNOVATION SUMMIT
FEBRUARY 19–20
SAN DIEGO, CALIFORNIA
This two-day event offers scientists, engineers, and architects a deep dive in Hadoop innovation with keynotes, hands-on labs, workshops, and networking opportunities.

DEVNEXUS
FEBRUARY 24–25
ATLANTA, GEORGIA
Organized by the Atlanta Java User Group for Java professionals, this event offers seven tracks and more than 50 sessions. Topics include data and integration, Java/Java EE/Spring, HTML5 and JavaScript, alternative languages, cloud, agile development, tools, and mobile.

MOBILE WORLD CONGRESS
FEBRUARY 24–27
BARCELONA, SPAIN
This industry-leading event focuses on in-depth analysis of present and future trends in the mobile industry. Highlights include keynotes and panel discussions, application development subconferences, a product and technology exhibition with 1,700 exhibitors, and an awards program.

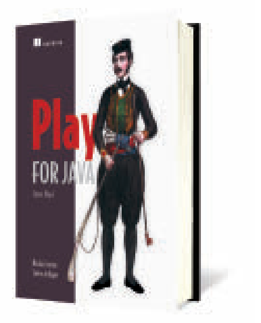
QCon London
MARCH 3–4 (Training)
MARCH 5–7 (Conference)
LONDON, ENGLAND
QCon London is the eighth annual London enterprise software development conference designed for developers, team leads, architects, and project managers. Topics include Java, HTML5, mobile, agile, and architecture.

ECLIPSECON 2014
MARCH 17–20
SAN FRANCISCO, CALIFORNIA
This conference brings together the Eclipse community to share best practices and innovation in software development with the Eclipse integrated development environment (IDE).

NightHacking Radio

The Java Spotlight Podcast is relaunching with a new name, a new host, and a new format. Look for the launch of NightHacking Radio with Oracle's **Stephen Chin** in early 2014. Follow @_nighthacking and visit the NightHacking website to learn more.

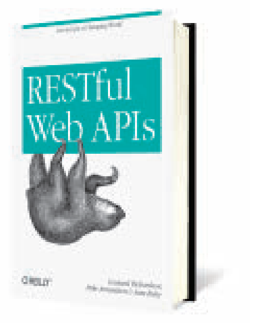
JAVA BOOKS



PLAY FOR JAVA
By Nicholas Leroux and Sietse de Kaper
Manning Publications (February 2014)

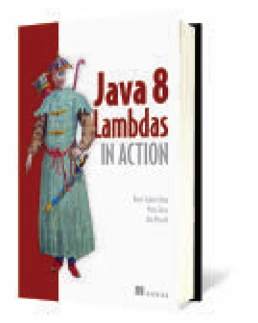
Play for Java shows you how to build Java-based web applications using the Play 2 framework. It introduces Play through a comprehensive overview example. Then, it looks at each facet of a typical Play application, both by exploring simple code snippets and by adding to a larger running example. It also contrasts Play and Java EE patterns and shows how a stateless web application can fit seamlessly into an enterprise environment.

Get 42 percent off until March 1, 2014, with code lerouxja.



RESTFUL WEB APIS
By Leonard Richardson, Mike Amundsen, and Sam Ruby
O'Reilly (September 2013)

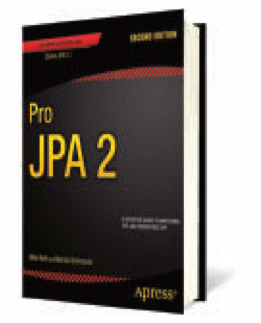
This guide shows you what it takes to design usable REST APIs that evolve over time. By focusing on solutions that cross a variety of domains, the authors explore how to create powerful and secure applications using the tools designed for the web. The book covers the concepts behind REST and strategies for creating hypermedia-based APIs, and helps you put everything together with a step-by-step guide to designing a RESTful web API.



JAVA 8 LAMBDAS IN ACTION
By Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft
Manning Publications (Summer 2014; early access now available.)

Java 8 Lambdas in Action is a guide to Java 8 lambdas and functional programming in Java. It begins with a practical introduction to the structure and benefits of lambda expressions in real-world Java code. The book then introduces the Stream API and shows how it can make collections-related code radically easier to understand and maintain. Along the way, you'll discover new functional-programming-oriented design patterns with Java 8 for code reuse and readability.

Get 43 percent off until March 1, 2014, with code urmajm.



PRO JPA 2, SECOND EDITION
By Mike Keith and Merrick Schincariol
Apress (October 2013)

Pro JPA 2, Second Edition introduces, explains, and demonstrates how to use the new Java Persistence API (JPA) 2.1 from the perspective of co-Spec Lead Mike Keith. The book provides both theoretical and practical coverage of JPA usage for both beginning and advanced developers. The authors take a hands-on approach, based on their experience and expertise, by giving examples to illustrate each concept of the API and showing how it is used in practice. The examples use a common model from an overriding sample application.

Opower's Rick McPhee (right) meets with Thermostat team members Tom Darci (bottom left), Thirisangu Thiyagarajanodipsa, and Mari Miyachi.

POWER TO THE PEOPLE

Opower's Java and open source software infrastructure analyzes big data and helps residential consumers save big dollars and reduce greenhouse gases.

BY PHILIP J. GILL

PHOTOGRAPHY BY BOB ADLER

Big Data and Java

Turning off lights, lowering the thermostat or air conditioning, and replacing lightbulbs and older home appliances with newer, high-efficiency models are all common ways residential energy consumers cut consumption, lower their utility bills, and reduce greenhouse gases.

Thanks to Opower, an Arlington, Virginia–based software and services provider, some 88 million people in 22 million homes in eight countries in the Americas, Europe, Asia, and Australia have something else: monthly or bimonthly home energy consumption reports that detail not just how much energy they use but also how their energy use compares to that of their neighbors.

Arriving five to seven days after their regular utility bills, these reports are intended to foster a "keeping up with the Joneses" feeling in residential energy customers, but in reverse—encouraging them to consume less rather than more, explains Rick McPhee, Opower's senior vice president of engineering. "Our

Shane Arney, senior software engineer on the Thermostat team, talks with McPhee about his experience as a new Opower employee.

founder believed that many residential energy consumers were undereducated about not just how they use energy but also how much energy they use and how their usage compares to people similar to them," explains McPhee.

Energy companies have been supplying customers with usage data—usually year-over-year or month-over-month comparisons—for some time, but Opower is "the first to leverage the idea of normative comparisons to encourage consumers to use less energy," McPhee notes.

Normative comparisons are typically used in clinical trials and behavioral psychology to evaluate the effectiveness of a particular treatment or therapy against an established norm. Opower has adapted that concept to consumer behavior in the residential energy market—using a group of 100 neighbors as the established norm against which a consumer is compared.

"We look at not just a household's basic energy consumption but also at consumption by HVAC systems, base loads, consumption during particular seasons, and other things that may help consumers tease out insights into how they use energy and how they may reduce their consumption," McPhee says. "We give people the information they need to save energy now."

That information and other Opower services, delivered by a Java and open source software infrastructure, have helped reduce energy consumption by 3 terawatt hours. That's enough to power for one year a city of 600,000 people—Las Vegas, Nevada; Baltimore, Maryland; or Bristol, England, for example.

It has also translated into a US$350 million savings on consumers' utility bills and has prevented—or *abated*, in energy conservation lingo—the release of more than 4.6 billion pounds of carbon dioxide (CO2) and other greenhouse gases into the atmosphere.

MANDATED EFFICIENCY

The widespread availability of renewable, sustainable non-carbon-based

SNAPSHOT

Opower
opower.com

Headquarters:
Arlington, Virginia

Industry:
Software

Employees:
400-plus

Java technologies used:
JDK 6, JDK 7

McPhee confers with Tyler Savage, engineering specialist at Opower, while Carl Gottlieb, director of product management, works away.

BIG SAVINGS

With Opower, consumers have saved more than US$350 million on their utility bills.

and alternative energy sources is the long-term solution to the environmental effects of greenhouse gases. In the short term, increased energy efficiency is the best solution available today. McPhee notes that there are 33 US states and several foreign countries, including the UK, that mandate strict energy efficiency.

Those mandates created the business opportunity for Opower. Founded in 2007, the company today provides energy consumption reports and other services to more than 90 electricity and gas utilities around the world, including 8 of the 10 largest in the US.

The company's Opower 4 software platform uses a service-oriented architecture built on Java and open source products and delivered as a software as a service (SaaS) to its client utilities and their customers. "The only piece of Opower software that resides on clients' computers is the data export software that sends customer meter data to our systems," notes McPhee.

The company, which operates multiple data centers, receives that customer data on a regular basis, depending on that client's need. "For most customers it's every 15 minutes, but for some customers it's as little as once a year," says McPhee. "A significant amount of the data Opower collects, processes, and leverages is time-series consumption data from residential meters for electricity and gas."

"To that we add a lot of other data so that we are able to group households using criteria that lead to a reasonable expectation of similar energy requirements," he continues. "We use census, parcel, and permit data. We also use home size, occupancy, age, heating type, weather patterns, and other factors to group households to compare their usage data to each other accurately."

ANALYZING BIG DATA

To produce the home energy reports, all the data is combined, processed, and analyzed using Apache Hadoop, an open source framework for running big data applications on large clusters of commodity hardware servers. Opower's core database, for example, holds 3.5 terabytes of customer meter data and related information.

Hadoop is specially designed for handling sequential I/O rather than the type of random access, frequent updates, and high transaction rates that relational databases are known for. "The Hadoop platform is highly optimized for large amounts of time-series data and analytic processing," says McPhee. "Hadoop has allowed us to scale our batch processing from a couple of utilities with monthly reads to hundreds of utilities with smart meters that send user reads every 15 minutes."

Another key piece of Opower's software infrastructure is Apache HBase, an open source nonrelational distributed database that works as a layer on top of Hadoop. Because Hadoop files tend to be very large and sequential, McPhee explains, it's difficult to access smaller pieces of data, such as information on a particular meter at a particular consumer's house. "You usually have to look at the whole file," he says, "so with HBase we've built a time-series database on top of Hadoop that allows us to do both efficient, scalable batch processing and also access information on a particular meter or insight on a meter with low latency as well."

In addition, the Opower open source software stack includes Apache Hive, a

McPhee catches up on e-mail between meetings.

Trying *Not* to Keep Up with the Joneses

Opower got its start in 2007 when the Sacramento Municipal Utilities District (SMUD), a publicly owned electricity utility in Sacramento, California, agreed to a pilot program in which 35,000 of its residential customers would receive Opower's home energy consumption reports. The reports compare a residential customer's energy usage to that of a group of 100 neighbors, in an effort to encourage them to reduce energy consumption, save money, and reduce their carbon footprints.

Since that time, the company has expanded its software-as-a-service (SaaS) offerings to include a social app that encourages customers to compare and compete to reduce energy with their neighbors and a web portal for accessing consumption information, retail and marketing services, home energy controls, and more.

One of the more intriguing offerings is its demand response program, which sends out a series of voice, text, and e-mail message alerts during peak demand for energy. For instance, Baltimore Gas and Electric (BGE) of Baltimore, Maryland—one of Opower's largest clients—uses this service in the summertime to reduce the need for electricity to power air conditioners. "Baltimore has a lot of humid summers," says Rick McPhee, senior vice president of engineering for Arlington, Virginia–based Opower. "Twenty-four hours before a particular day or time of day, when BGE management feels the company will have a hard time meeting demands, it will use the Opower platform to send messages to millions of its consumers. These text messages, 'robocalls,' and e-mails ask them to please reduce their energy usage during these peak demand times."

"After the event," he continues, "we send them an e-mail comparing how they performed during the event and comparing their performance to their neighbors."

But it doesn't stop there; Opower's program provides continuous reinforcement of the conservation ethic. "With regular frequency during the summer," says McPhee, "we provide a piece of paper in the mail that gives those consumers an update on how much energy they have saved, and how much they could save if they continue to conserve. We continue to encourage them to save."

McPhee gets an update from Senior Operations Engineer Unai Basterretxea as Operations Engineer Lisa Danz rides by.

data warehousing infrastructure built on top of Hadoop for providing data summarization, query, and analysis; Hibernate, an object-relational mapping and persistence framework for Java; the Spring framework for Java application development; the Linux operating system; and the MySQL relational database.

The core technology of the Opower platform, however, remains Java; the company is currently transitioning from JDK 6 to JDK 7. "About 80 percent of our software and *all* of our customer-facing applications are built in Java," says McPhee.

In the beginning, he says, other options, including Ruby and Python, were considered, but Java quickly won out. "In my opinion, Java provides the best balance between stability and consistency in developing and deploying software with flexibility and accessibility," says McPhee. "And that is important when you're working on open source projects with engineering teams all over the globe."

"Java provides enough strongly typed constructs, strict interfaces, and strict ways of doing things that there is enough consistency to make sharing easy," he says, "yet it is still very accessible and very configurable. I think that's why the open source community is largely focused on Java as well."

"Open source is a great way to recruit," he adds. "It's a great way to keep engineers involved in whatever you are doing so they are constantly learning best practices. And it's a great way not to have to build or buy software all the time." **</article>**

MORE ON TOPIC:

Big Data and Java

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

PHOTOGRAPH BY ADAM LUBROTH/GETTY IMAGES

Java in Business

Banking on Java

Global bank BBVA manages big data volumes with Java.

BY KEVIN FARNHAM

The IT Corporate Investment Banking (CIB) group at global bank BBVA manages very large volumes of data and has a particular need for low-latency performance. Java Magazine *talked with the IT CIB Architecture team about applying Java technologies to accomplish this.*

Java Magazine: Why Java?

IT CIB: Java has proven to perform well in low-latency and large-data-volume scenarios; Java runs on our Windows, Linux, and Oracle Solaris systems; mature software lifecycle tools are available; there are reliable open source utility and technology integration libraries; and the expert Java developer community is large and global.

Java Magazine: What role do Java tools play in the IT CIB development process?

IT CIB: Maven and Jenkins are vital when you have distributed teams. Artifacts that are guaranteed to have the necessary version are generated automatically. Unit tests are executed continuously, allowing early detection of errors. We've integrated tests into our continuous build process that track the impact of code changes.

Java Magazine: How does the team use Java to achieve the needed scalability?

IT CIB: For projects with complex business logic, large data volumes, or a large number of requests where latency isn't critical, we use OSGi, scaling services vertically and horizontally.

Java Magazine: How do you meet your low-latency requirements?

IT CIB: We developed an in-house Java framework to manage threads, concurrency, high availability, and messaging through a low-latency bus. Our messaging layer avoids excessive object creation and employs lock-free methods where possible.

Java Magazine: How do you cope with an overload of simultaneous events?

IT CIB: First, we do rigorous testing to understand how the application behaves under the best and worst load scenarios. We also implement mechanisms that detect when applications are starting to suffer from an incoming message storm.

Java Magazine: Any closing thoughts?

IT CIB: Using cloud services for a global big data platform needs to be balanced with local access paths for the components that require low latency. `</article>`

BBVA IT Corporate Investment Banking Architecture team members John Kenneth Henriksson (far left), Monica Aguado Sanchez, and Javier Soler Luján walk along the Paseo de la Castellana in Madrid, Spain.

Big Data and Java

FAST FACTS

- BBVA operates franchises in Spain, Mexico, South America, and the United States.
- BBVA Wallet is an iOS and Android app for customers in Spain that manages bank card transactions.
- BBVA ranks 13th on the World's Best Multinational Workplaces list published by Great Place to Work.

Steve Winkler, a technology strategist at SAP, adds to a sticky note wall that is part of the design thinking process at SAP.

JCP Executive Series

SAP Tames Big Data with Java Technology and the JCP

SAP's Steve Winkler discusses key issues for enterprise computing and his varied roles within the Java community. **BY STEVE MELOAN**

PHOTOGRAPHY BY PHIL SALTONSTALL

Big Data and Java

Steve Winkler is a technology strategist focused on open standards and open source in SAP's development organization. He has more than 18 years of Java programming experience, including the design and development of an enterprise-class Java-based XML Messaging System that enables the com-

Winkler chats with colleagues over coffee in the break room.

munication of SAP systems with non-SAP systems. Since 2004 he has been an active participant in numerous community-driven standards bodies, including the Java Community Process (JCP), where he has served as a member of the JCP Executive Committee since 2011.

Java Magazine: When did you join the JCP, what motivated your decision, and what have been some of your major activities?

Winkler: I personally joined the JCP in 2004, but SAP had been involved for quite some time. My involvement stemmed from being a software architect at SAP headquarters in Germany, designing and building SAP's Adapter Framework, which allows SAP systems to communicate with non-SAP systems. While reviewing the Java Connector Architecture specification, in connection with this project, I had a few ideas for improvement, which eventually led to joining the JCP to make those contributions. Since then, I've worked on a number of specifications related to Enterprise JavaBeans (EJB) and web services. And for the past two years I've served on the Executive Committee on behalf of SAP. In parallel, SAP has been involved in a broad range of related work including the Java Persistence API [JPA], JavaServer Faces [JSF], Java Management Extensions [JMX], Concurrency Utilities, WebSocket, and so on.

Java Magazine: Given your background with RFID, and SAP's focus on the business internet, can you share some insights into the era of big data, and the role of the JCP?

Winkler: RFID is a good segue into the big data topic. The proliferation of RFID and M2M [machine-to-machine] communication generates huge amounts of valuable data. The quantity of data being captured by business systems each year is just exploding. The challenge is to process this data quickly and efficiently so that our customers can leverage it to make both strategic and operational decisions.

SAP has a product called the SAP HANA Cloud Platform that allows Java developers to implement scalable in-memory applications quickly and inexpensively to meet their big data needs.

Java EE 6 Web Profile is the default programming model for the SAP HANA Cloud Platform. So JSRs that advance the Java EE platform to support big data will be important to our activities. Right now I think we have the optimal level of standardization in the big data area—vendors can differentiate themselves and experiment, but everyone is still moving in the same direction.

The topics of big data and the cloud are relatively new for both developers and industry. We need more experience and understanding before standardization can occur. It will happen, but gradually.

Java Magazine: Let's talk about data security within B2B software, and JCP activities surrounding this issue.

Winkler: Data security is of paramount importance to SAP, given the highly sensitive business content that our systems store. We are constantly evaluating our environments for security insights that we can pass back to the community. We've been involved in JSR 351, the Java Identity API. With the advent of social networking sites, and increased use of the internet in conducting business, the need for Java developers to safeguard the disclosure of network identity is crucial. We've also contributed to the OpenJDK on this topic. Security is obviously very important for B2B systems, or any platform technology that enables B2B, and security clearly benefits from the community working together.

Java Magazine: Are there any aspects of the JCP that you would like to modify?

Winkler: All of the current platform JSRs, and a majority of other JSRs, have been submitted and led by Oracle. That presents a problem for the future of the community. Businesses today are focusing more than ever on listening; social media is driving innovation based on dialogue with customers. Facilitating outside-in innovation to stay competitive is more important now than it has ever been. Taking the Java platform to the next level will require strategies to encourage users to contribute to and lead JSRs that will shape the future of Java and make it a better product.

Java Magazine: Is there any interaction between the JCP and the SAP Community Network [SCN]?

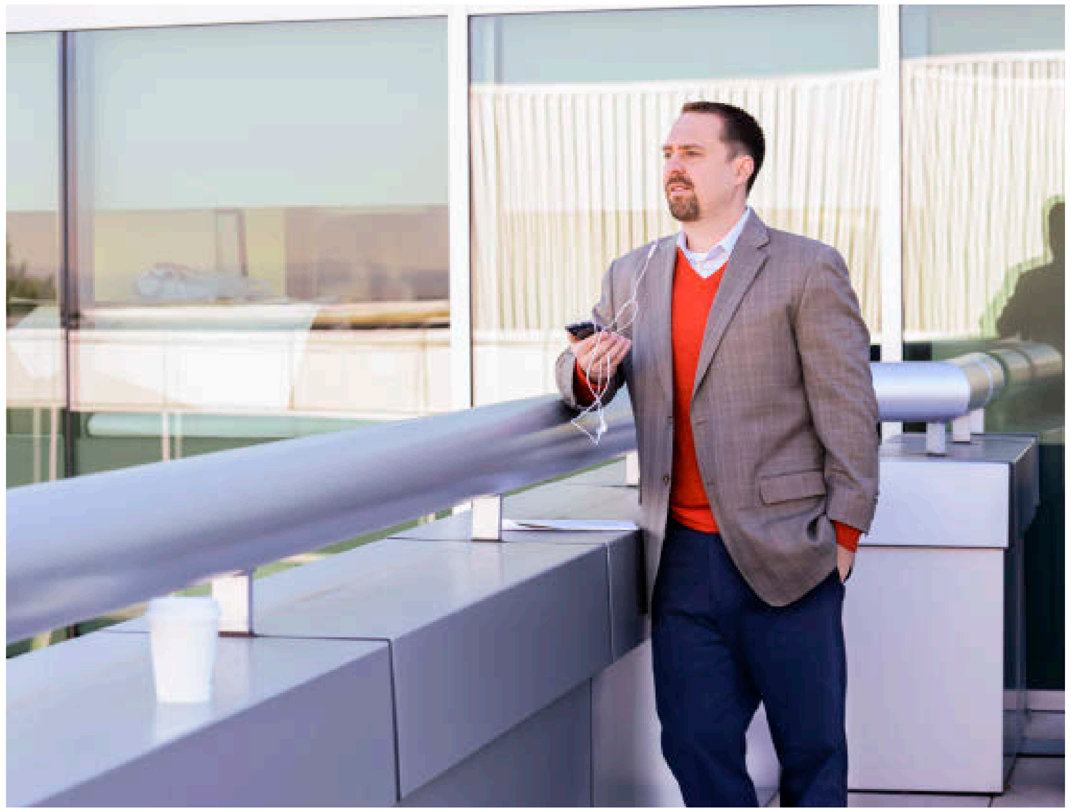
Winkler: SCN is a community for SAP-focused developers. We have a fairly active Java forum on SCN where developers—mostly our customers and partners—can come together to ask and have their questions answered by either SAP employees or their peers. We've received quite a few Java EE–related questions over the years, and we make that knowledgebase available as a resource for future developers.

This is our way to interact with customers and build expertise, so it does influence the capabilities we bring to the JCP and JSR processes.

Java Magazine: As a board member of the Eclipse Foundation, does SAP have any interactions with the JCP?

Winkler: Absolutely. Eclipse is a community for individuals and organizations to collaborate on commercially friendly open source software. We have executive and board seats on both the JCP and Eclipse, so SAP is very involved. In fact, in terms of Eclipse active committers (those with write access to source repositories), we are third, only behind IBM and Oracle.

We're also very happy to see alignment between Eclipse and the JCP through our colleague Mike Milinkovich. He is a JCP Executive Committee member and director of the Eclipse Foundation. In these roles, he helps to ensure a certain level of coordination between the two bodies. He's also able to bring his Eclipse experience to the table as an advisor to the process of revising the Java Specification



Winkler makes a call and gets some fresh air outside his office in Palo Alto, California.

Winkler meets with Big Data Analytics team members Mohammad Shami (left) and Peter Danao.

Participation Agreement [JSPA; JSR 358]. This is a hugely important task facing the JCP right now. Mike can, and does, offer his expertise about how the Eclipse community has been able to manage some of the tricky legal issues involved in this process. In addition, he has valuable insights about increasing participation. From SAP's point of view, alignment between the JCP and Eclipse is a win-win for everyone.

Java Magazine: Could you share some of your experiences working on JSRs from the point of view of both an Expert Group member and an Executive Committee member?

Winkler: When I first joined SAP as a software architect, I read specifications while writing code, as more of a casual observer. Then I became directly involved, and started participating in the JSR process. That provided a new perspective on how specifications are actually written. And now, as an Executive Committee member, I serve as one of the stewards of the community, so I need a much better understanding of how the granular details affect every type of participant.

As a casual observer reading specifications, there's no reason to have an awareness of the legal framework of the JCP. But as an Expert Group participant, you must sign a legal document to join. At that point, your awareness is focused on details that affect you and/or your organization regarding participation in one specific JSR. As an Executive Committee member, on the other hand, you need to understand the details of the legal framework in and out, up and down, and know how the wording of each clause—even the subtle change of a word or two—can have a huge impact on certain types of participants in the process. Executive Committee members have to be able to put themselves in the shoes of a variety of different community members and understand their perspectives.

Java Magazine: Any closing thoughts?

Winkler: I previously mentioned JSR 358, targeted at a major revision of the JSPA. It tackles licensing and open source issues, transparency, the role of individual members, patent policy, IP flow, and so on. This is a delicate process, and we want to make absolutely sure that any changes create better conditions for the Java community and help to ensure the continuing viability of the Java platform. The current technology environment is very dynamic, very competitive. The JSPA revisions must be a top priority. </article>

MORE ON TOPIC:

Big Data and Java

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

- SAP Community Network: Java Development
- Introduction to SAP HANA Cloud Platform course
- SAP HANA Cloud Platform Developer Center

JOSH JUNEAU
BIO

Part 2

Three Hundred Sixty–Degree Exploration of Java EE 7

Leverage the latest features in the Java API for RESTful Web Services and the JSON Processing API.

This article is the second in a three-part series demonstrating how to use Java EE 7 improvements and newer web standards, such as HTML5, WebSocket, and JavaScript Object Notation (JSON) processing, to build modern enterprise applications.

In this second part, we will improve upon the movieplex7 application, which we started in the first article, by adding the ability to view, delete, and add movies within the application database.

To review, the movieplex7 application is a complete three-tier application that utilizes 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)
- Java API for Batch Processing (JSR 352)
- Java API for RESTful Web Services (JAX-RS; JSR 339)

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

Overview of the Demo Application

In Part 1, we learned how to download, install, and configure NetBeans 7.3.x and GlassFish 4, which will be used for building and deploying the application in this article as well.

Note: NetBeans 7.4 RC1 has been available for use since summer 2013; you can use that IDE to work with the application in this article, but some features will differ from those described here since this article uses version 7.3.1.

We configured the movieplex7 application within NetBeans, and generated the database schema within Apache Derby via schema generation configured within persistence.xml. We then built the JSF 2.2–based user interface for booking a movie, including binding to the back end by generating managed beans and making them injectable by adding the @Named CDI annotation. We also encapsulated a series of related views/pages with application-defined entry and exit points using the new Faces Flow.

In this article, we will utilize JAX-RS to perform the view and delete operations, and we will make use of the Java API for JSON Processing (JSON-P) for adding movies.

For the remainder of the article, please follow along using the Maven-based project that you created in NetBeans IDE for Part 1 of this series.

Creating the Web Pages and Logic for Viewing and Deleting Movies

The movieplex7 application utilizes the JAX-RS API for viewing and deleting movies. In this section, we will walk through the steps for adding views and business logic that allow movieplex7 users to view all movies, view movie details, and delete existing movies using JAX-RS.

The JAX-RS 2 specification adds the following new features to the API:

- Client API
- Asynchronous processing capability
- Filters, interceptors, and well-defined extension points

- Bean validation, which enables you to easily validate data by applying annotations to bean fields

Create a RESTful client. Let's begin with adding to the application a bean that will act as a client, invoking the REST endpoint. To do so, follow these steps:

1. Right-click **Source Packages**, and then select **New** and then **Java Class**. Name the class MovieClientBean, and set the package to org.glassfish.movieplex7.client. Lastly, click **Finish** to create the bean. Once the bean has been generated, it will open in the editor.
2. Add the @Named and @SessionScoped class-level annotations above the class definition.

   By adding @Named, we are enabling CDI for this class, which allows it to be injected into an Expression Language (EL) expression. The @SessionScoped annotation signifies that the bean is to be automatically activated and passivated with a session.
3. Make the class implement java.io.Serializable because it is in session scope.
4. Resolve imports, as needed. Any unresolved imports will show as errors within NetBeans (see **Figure 1**).

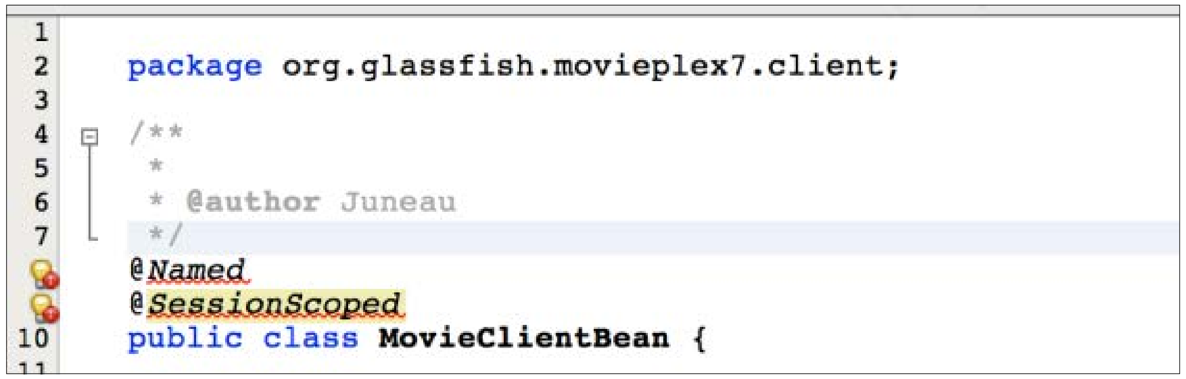
NetBeans makes it easy to resolve the imports by either clicking the yellow lightbulb or right-clicking within the editor and selecting the **Fix Imports** option (Shift + Cmd + I keys on Mac OS X).

Each of the annotations can resolve against a couple of different imports, and NetBeans chooses the imports that best suit the situation. Note that for @SessionScoped, both the javax.enterprise.context .SessionScoped and javax.faces .bean.SessionScoped classes may be imported. If you are using CDI, along with importing @Named, you should be sure to import @SessionScoped. And if you are working with JSF and @ManagedBean, you should import @SessionScoped.

It should also be noted that @ManagedBean is deprecated as of JSF 2.2, so @Named (CDI managed beans) should be used unless your application requires @ManagedBean for backward support.

Now that the class has been created, it is time to add code to implement the client.

1. First, declare Client and WebTarget class variables, and then create the lifecycle callback methods: init() and destroy(). Annotate the init() method with @PostConstruct and annotate the destroy() method with @PreDestroy.



```
package org.glassfish.movieplex7.client;

/**
 *
 * @author Juneau
 */
@Named
@SessionScoped
public class MovieClientBean {
```

Figure 1

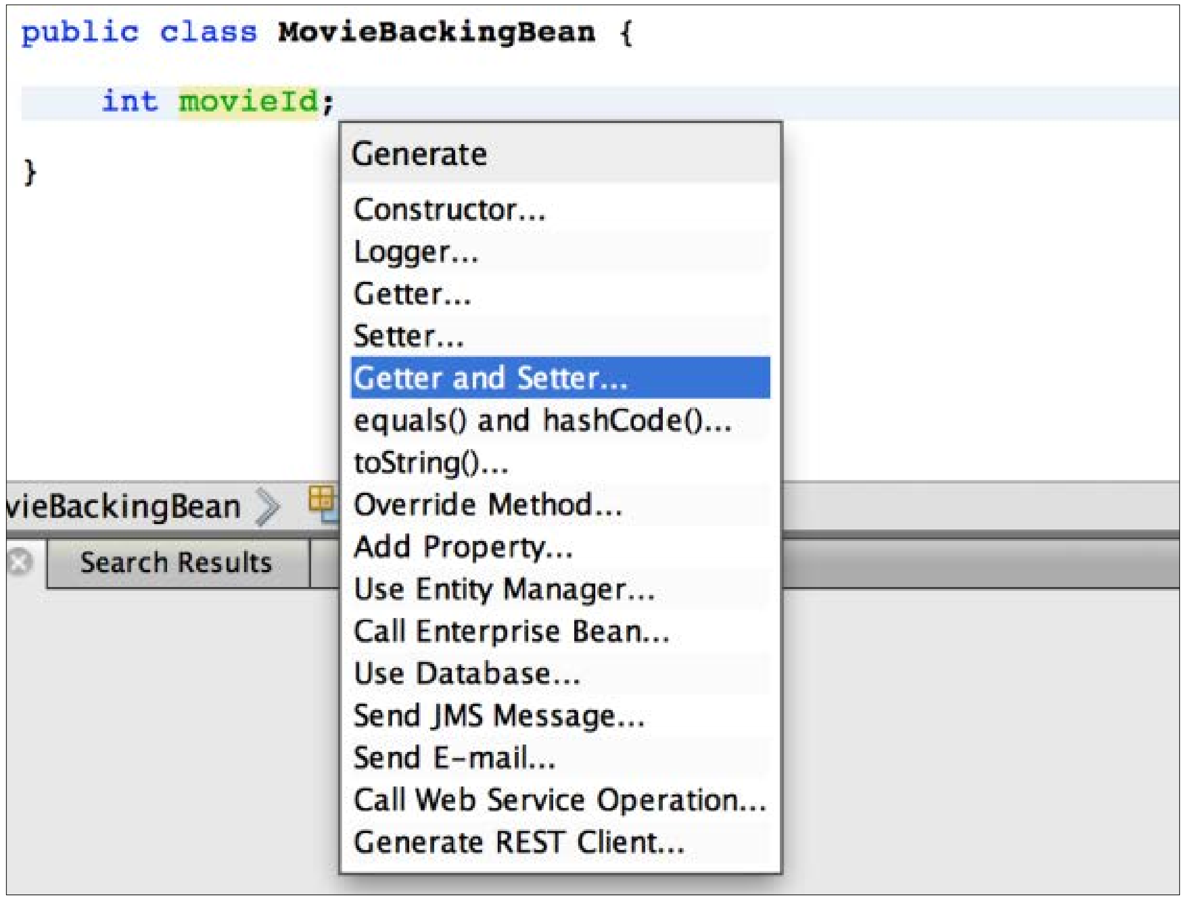
Annotating these methods will prompt the application server container to invoke the init() method before the bean is constructed and, similarly, to invoke the destroy() method before it is destroyed.

2. In the init() method, obtain a new client using ClientBuilder and, in turn, set the endpoint URI of the web service target by calling the client.target method.

   Take care when creating Client instances, because they are resource-intensive objects that are used to manage the client-side communication infrastructure. Be sure to create only the required number, and close them when you are finished. The end result should resemble the code in **Listing 1**.
3. Implement the method that will return the movies to the caller by adding the code shown in **Listing 2**. To return the movies, the target.request method is invoked, which in turn invokes the HTTP GET method, returning a value of type Movie[].
4. Lastly, resolve imports.

Create a view to invoke the client. Now that the client has been created, it is time to write the view that will invoke the client.

1. Within the NetBeans project, right-click **Web Pages**, select **New -> Folder**, and specify the name client for the folder. Click **Finish**.
2. Right-click the newly created client folder, and choose **New**, **Other**, **JavaServer Faces**, and **Faces Template Client**, and then click **Next >**. Name the new file movies.xhtml, click **Browse** (next to **Template**), expand **Web Pages** and **WEB-INF**, select **template.xhtml**, and click **Select File**. Click **Finish**. Remove the <ui:define>



```
public class MovieBackingBean {

    int movieId;

}
```

Figure 2

LISTING 1 / LISTING 2 / LISTING 3

```
Client client;
WebTarget target;

@PostConstruct
public void init() {
   client = ClientBuilder.newClient();
   target = client
       .target("http://localhost:8080/movieplex7/webresources/movie/");
}

@PreDestroy
public void destroy() {
   client.close();
}
```

Download all listings in this issue as text

sections with "top" and "left" as names, because these are inherited from the template.

3. Once the file is opened in the editor, replace the content within the <ui:define> area with the code fragment shown in **Listing 3**.

The code within the movies.xhtml view obtains all the movies by invoking the getMovies method within the MovieClientBean. It does so by binding the items attribute within the c:forEach element to the getMovies method, which returns an object of type Movie[] and iterates over each element.

Each movie within the array is then displayed as a separate item via the <f:selectItem> element, which utilizes the var attribute of the c:forEach as a handle to expose the itemValue and itemLabel values for each object within the Movie[]. Note that the view also contains a commandButton element that is labeled "Details", which contains a movie action. When clicked, this button will look for a view named movie.xhtml, because the action matches the name of the view to invoke.

After adding the code, use the NetBeans auto-importer to resolve the namespace prefix-to-URI resolution by clicking the yellow lightbulb on the left.

Next, create the backing bean for the view:

1. Right-click the org.glassfish.movieplex7.client package, select **New->Java Class**, specify MovieBackingBean for the name, and click **Finish**.
2. Add the following field to the class:

```
int movieId;
```

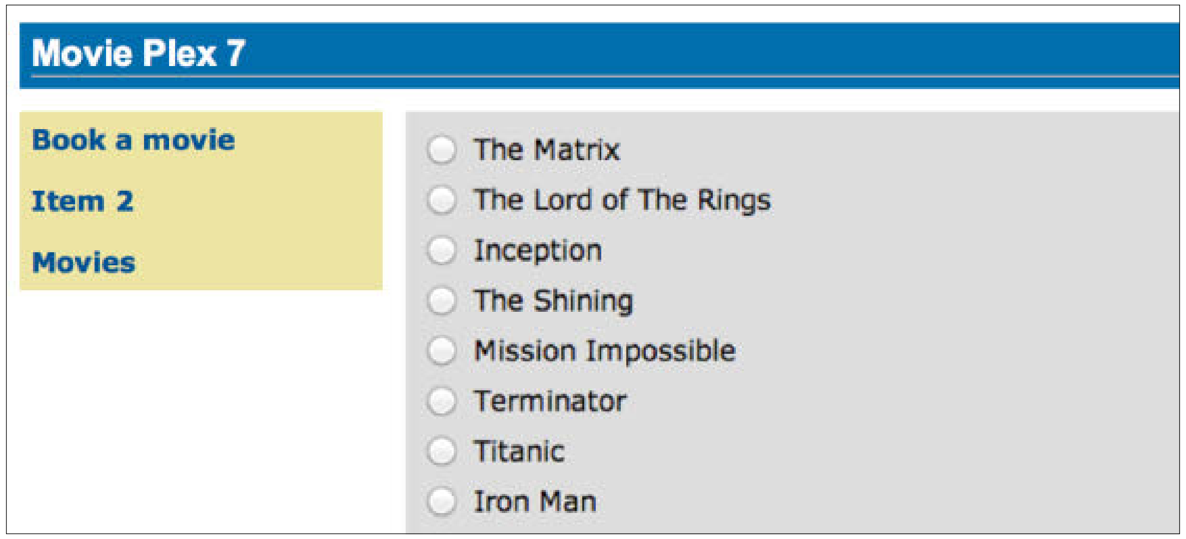
3. Create getters/setters for the new field by right-clicking the editor pane and selecting **Insert Code**, which will open a contextual menu. Select **Getter and Setter** within the menu, as shown in **Figure 2**. Select the checkbox next to the field when the Generate Getters and Setters window opens, and then click **Generate**.
4. Add @Named and @SessionScoped class-level annotations, along with implementing java.io.Serializable, and resolve imports.

Add menu-item navigation and movie details. Provide a way for the user to access the movie listing, and code the Details button for the user interface:

1. In template.xhtml, add the code shown in **Listing 4** to the <ui:insert> with the name="left" attribute within the <form> element.
   This will generate the menu item for navigation to the movies.xhtml view.
2. To enable the functionality for selecting a movie and clicking the Details button, provide the ability for the MovieClientBean class to access the MovieBackingBean by injecting MovieBackingBean into the class. To do this, add the following code to MovieClientBean:

```
@Inject
MovieBackingBean bean;
```

3. After adding the @Inject annotation, select the yellow lightbulb to import javax.inject.Inject.
4. Make use of the Client and WebTarget instances to obtain the currently selected Movie object. To do so, add a variable named movie, which represents the currently selected object, to the REST endpoint URI by placing {movie} at the end of the path. It can then be bound to a concrete value using the resolveTemplate method, passing "movie" as the binding string and bean.getMovieId() as the binding value, which is populated from the currently selected movie. To enable this functionality, add to MovieClientBean the code shown in **Listing 5**.
5. Create the movie.xhtml view in the client folder of your project. To do so, copy the code shown in **Listing 6** and paste it into the <ui:define> element of that view. Click the yellow lightbulb to resolve any namespace prefix-URI mappings.


Movie Plex 7

Book a movie
Item 2
Movies

- The Matrix
- The Lord of The Rings
- Inception
- The Shining
- Mission Impossible
- Terminator
- Titanic
- Iron Man

Figure 3

LISTING 4 / LISTING 5 / LISTING 6

```
<p/><h:outputLink value="${
facesContext.externalContext.requestContextPath
}/faces/client/movies.xhtml">Movies</h:outputLink>
```

Download all listings in this issue as text

Run the project. Run the project by right-clicking the project name and then selecting **Run** from the contextual menu (Fn + F6 keys on Mac OS X). Click the **Movies** option in the left navigation bar to show the output shown in **Figure 3**.

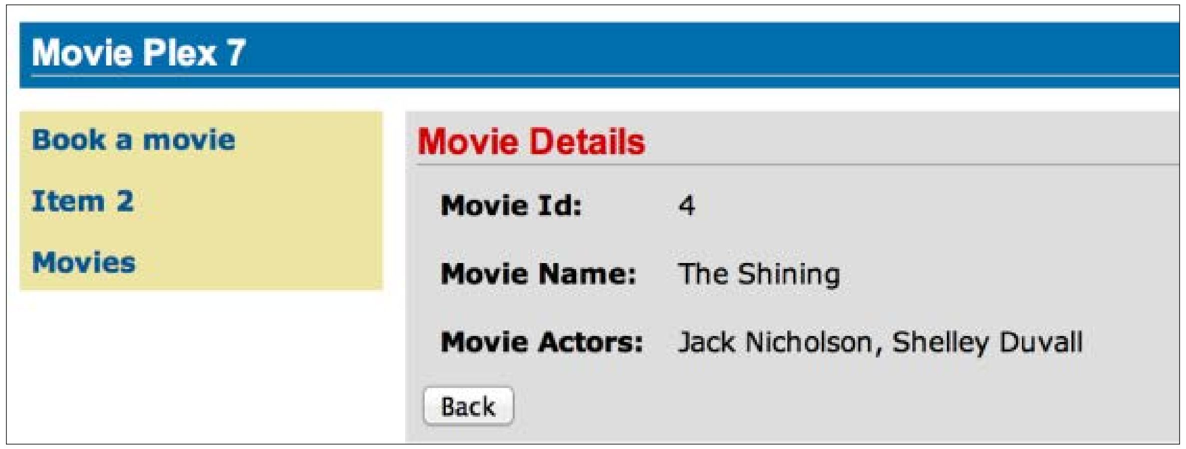
Every movie in the array is listed within the view, with a button next to each. The output is generated from a REST endpoint, as opposed to the traditional Enterprise JavaBeans (EJB)/JPA–backed endpoint.

Choose a movie by selecting the button next to it and clicking on the **Details** button. Once clicked, the details of the movie will be shown (see **Figure 4**). Click the **Back** button to return to the list of movies.

Providing Delete Capability

Enhance the application by providing the ability to delete a movie.

1. Use a commandButton to invoke a method within the backing bean named deleteMovie(), which will delete the selected movie. To do so, add


Movie Plex 7

Movie Details

Movie Id: 4

Movie Name: The Shining

Movie Actors: Jack Nicholson, Shelley Duvall

Figure 4

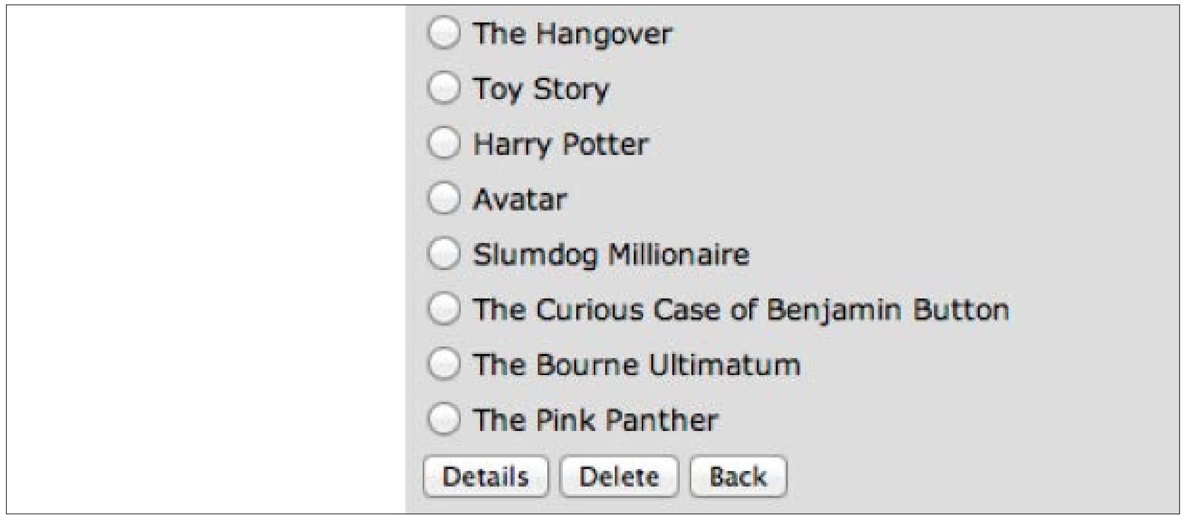

- The Hangover
- Toy Story
- Harry Potter
- Avatar
- Slumdog Millionaire
- The Curious Case of Benjamin Button
- The Bourne Ultimatum
- The Pink Panther

Figure 5

LISTING 7 LISTING 8

```
<h:commandButton
  value="Delete"
  action="movies"
  actionListener="#{movieClientBean.deleteMovie()}"/>
```

Download all listings in this issue as text

the code in **Listing 7** to the movies.xhtml view. The button should be added after the code for the Details button.

Note: To format your code nicely within the view, right-click within the editor and choose the **Format** option.

2. Add the deleteMovie() method to MovieClientBean by pasting into the Java class the code shown in **Listing 8**. The method is annotated with @Transactional, which is new in JTA 2.1 (released as part of Java EE 7).

   This annotation provides the ability to control the transactional boundaries on CDI managed beans.

3. Resolve imports and run the application.

Running the project now displays the Delete button at the bottom of the movie listing, as shown in **Figure 5**. If you select a movie and then you click the **Delete** button, the movie is removed from the database and the view is refreshed.

Providing the Ability to Add Movies

Now that we've enhanced the application by providing the ability to view and delete movies, it is time to provide the capability to add movies to the database. To do so, we will make use of the JSON Processing 1.0 (JSON-P) API, which provides a standard API for parsing and generating JSON for use by applications. We will also be using the JAX-RS API for reading and writing JSON.

Before delving into the application examples for adding a movie, let's briefly explore the functionality that the JSON-P and JAX-RS APIs bring to the table. For those who are unfamiliar, JSON is a data exchange format that is widely used via web services and other connected applications. JSON is composed of objects and arrays of data that can be used as a common format for working with data in internet applications. The JSON-P API provides a standard for parsing, transforming, and querying JSON data using the object model (tree-based) or the streaming model (event-based parser). The JSON-P API is composed of the following packages:

- javax.json: Contains interfaces, utility classes, and Java types for JSON objects
- javax.json.stream: Contains parser and generator interfaces for the streaming model

The JAX-RS API is the Java API for building RESTful web services, which are lightweight web services. In this example, we will use JAX-RS Entity Providers, which supply mapping services between internet data representations and their associated Java types. There are several predefined type mappings, including String, byte[], and many others. If an application needs to define its own mapping to custom types, that can be done using the MessageBodyReader and MessageBodyWriter interfaces.

Create the MessageBodyReader. The following steps can be used to provide the ability to add a new movie to the application. Please note that for production applications, add, delete, and update functionality should be made available only after proper authentication.

1. Right-click **Source Packages**, select **New** and **Java Package**, and then specify the package name as org.glassfish.movieplex7.json and click **Finish**.
2. Right-click the newly created package, and select **New** and then **Java Class**. Then specify the name as MovieReader and click **Finish**.
3. Add the following class-level annotations to the new MovieReader class:

```
@Provider
@Consumes
(MediaType.APPLICATION_
JSON)
```

The @Provider annotation allows the implementation to be discovered by the JAX-RS runtime during the provider scanning phase. It can be used for any class that is of interest to JAX-RS. The @Consumes annotation specifies the MIME media type of representations that were sent by the client and can be consumed by a resource. In this case, MediaType.APPLICATION_JSON indicates that a JSON resource will be consumed.

4. Resolve imports by clicking the yellow lightbulb (Shift + Cmd + I keys on Mac OS X) to import the following classes:

```
javax.ws.rs.core.MediaType
javax.ws.rs.Consumes
javax.ws.rs.ext.Provider
```

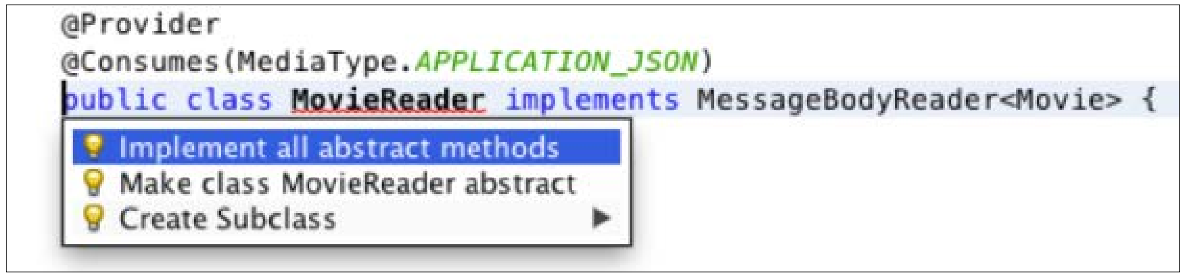
5. Modify the class signature so that it implements MessageBodyReader<Movie>, resolve imports, and then click the yellow lightbulb hint in the left column and select **Implement all abstract methods**, as shown in **Figure 6**.
6. Replace the generated isReadable() method with the code shown in **Listing 9**, which will determine whether the MessageBodyReader can produce an instance of a particular type.
7. Replace the generated readFrom() method with the code shown in **Listing 10**, and resolve imports, as needed.

This code first reads a type from the input stream named in, and then creates a JsonParser object from that stream by calling Json.createParser(in). Key values are then read from the parser, and they

LISTING 9 LISTING 10

```
@Override
  public boolean isReadable(Class<?> type, Type type1,
Annotation[] antns, MediaType mt) {
    return Movie.class.isAssignableFrom(type);
  }
```

Download all listings in this issue as text



```
@Provider
@Consumes(MediaType.APPLICATION_JSON)
public class MovieReader implements MessageBodyReader<Movie> {
```

Implement all abstract methods
Make class MovieReader abstract
Create Subclass

Figure 6

are used to populate a new Movie instance. The newly created Movie is then returned.

Create the MessageBodyWriter.

1. Right-click the newly created package, and select **New** and **Java Class**. Then specify the class name as MovieWriter and click **Finish**.
2. Add the following class-level annotations:

```
@Provider
@Produces
(MediaType
.APPLICATION_JSON)
```

@Produces is used to specify the MIME media type of the resource being produced. In this case, JSON will be produced.

3. Resolve imports by clicking the yellow lightbulb (Shift + Cmd + I keys on Mac OS X).

   Note: Be sure to import the javax.ws.rs.Produces class.
4. Modify the class definition so that it implements MessageBodyWriter<Movie> and resolve imports. Then click the yellow lightbulb hint in the left column and select **Implement all abstract methods**.
5. Overwrite the implementations for the isWriteable(), getSize(), and writeTo() methods with those shown in **Listing 11**. Resolve the imports.

The isWriteable() method determines whether the specified type can be written. getSize() returns the length in bytes of the serialized form of the object type, Movie. In JAX-RS 2.0, this method is deprecated, and all MessageBodyWriter implementations are advised to return -1. Lastly, the writeTo() method writes a given type to an HTTP message. A JsonGenerator is created by passing the OutputStream to Json.createGenerator, and the resulting JsonGenerator is then used to write the JSON data in a streaming manner.

Create the Add Movie form and backing bean components.

1. Create a new view within the client folder named addmovie.xhtml, and replace the content within <ui:define> with the code shown in **Listing 12**.

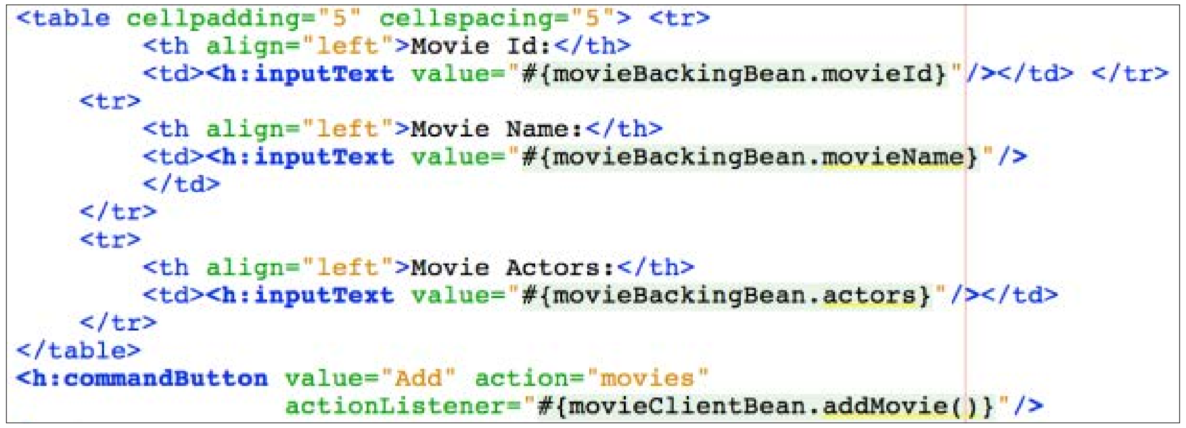
The code in addmovie.xhtml creates a form for input of the id, name, and actors of a movie. The values are bound to fields within the MovieBackingBean managed bean controller. The actionListener attribute of the commandButton is bound to an action method in the controller, which is responsible for adding the field values to the database. The action attribute is set to movies, which is the view to which control will be returned.

Note: Some of the field and method bindings within the view have yellow lines under-

LISTING 11 LISTING 12

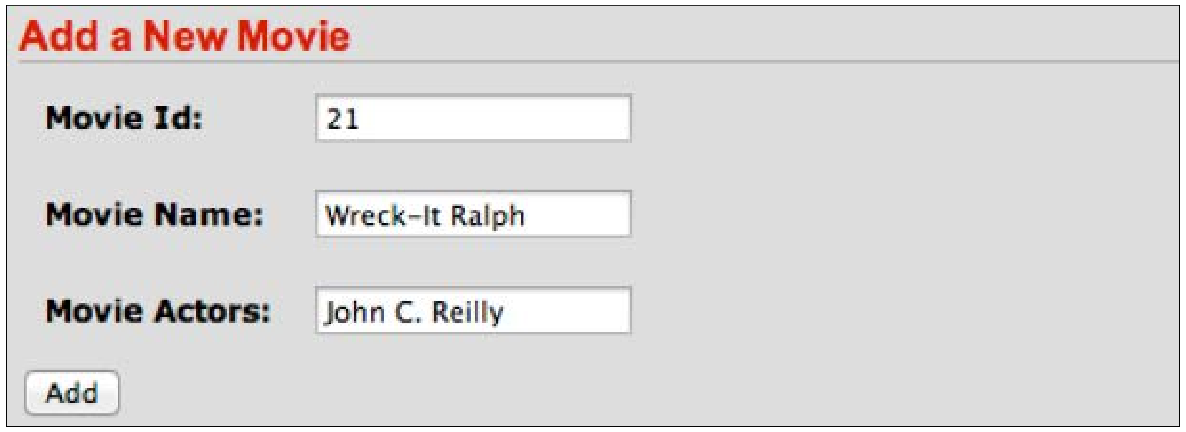
```
@Override
  public boolean isWriteable(Class<?> type, Type type1,
Annotation[] antns, MediaType mt) {
      return Movie.class.isAssignableFrom(type);
  }

  @Override
  public long getSize(Movie t, Class<?> type, Type type1,
Annotation[] antns, MediaType mt) {
      return -1;
  }

  @Override
  public void writeTo(Movie t, Class<?> type, Type type1,
Annotation[] antns, MediaType mt, MultivaluedMap<String, Object>

mm, OutputStream out) throws IOException, WebApplicationException {
      JsonGenerator gen = Json.createGenerator(out);

      gen.writeStartObject().write("id", t.getId())
           .write("name", t.getName()).write("actors", t.getActors()).writeEnd();
      gen.flush();
  }
```

Download all listings in this issue as text



```
<table cellpadding="5" cellspacing="5"> <tr>
        <th align="left">Movie Id:</th>
        <td><h:inputText value="#{movieBackingBean.movieId}"/></td> </tr>
    <tr>
        <th align="left">Movie Name:</th>
        <td><h:inputText value="#{movieBackingBean.movieName}"/>
        </td>
    </tr>
    <tr>
        <th align="left">Movie Actors:</th>
        <td><h:inputText value="#{movieBackingBean.actors}"/></td>
    </tr>
</table>
<h:commandButton value="Add" action="movies"
                 actionListener="#{movieClientBean.addMovie()}"/>
```

Figure 7



Add a New Movie

Movie Id:	21
Movie Name:	Wreck-It Ralph
Movie Actors:	John C. Reilly

Figure 8

neath them (see **Figure 7**). These lines signify that the fields and methods do not yet exist.

2. Open MovieBackingBean in the IDE, and add the following fields to the class. Then create the getters and setters:

```
String movieName;
String actors;
```

3. For navigation, add a commandButton to addmovie by adding to movies.xhtml the code shown in **Listing 13**.

4. Add the addMovie() method shown in **Listing 14** to MovieClientBean. Resolve imports.

   The addMovie method generates a new Movie instance, and then populates it using the values that are current in the backing bean. Once completed, the bean is sent to the REST endpoint by invoking the target.register() method to register a MovieWriter, which then provides conversion from the Movie plain old Java object (POJO) into JSON format.

5. Test the application by running it, selecting the **Movies** link in the left-hand menu, and then clicking the **New Movie** button at the bottom of the movie listing. Complete the "Add a New Movie" form by providing a value for the MovieID field, as shown in **Figure 8**. When you click **Add**, the new movie will be displayed within the movie listing.

Conclusion

In this article, we modified the movieplex7 application that was started in Part 1, providing the ability to view, add, and delete movies. The article covered the following technologies:

- JSF 2.2
- JAX-RS 2.0
- Java API for JSON-P 1.0

In the third and final article in this series, we will be adding the ability to process movie ticket sales, assign movie points, and chat with others. To add these capabilities, we will make use of Batch Applications for the Java Platform 1.0 (JSR 352), Java Message Service 2.0 (JSR 343), and Java API for WebSocket 1.0 (JSR 356), respectively. </article>

LISTING 13 LISTING 14

```
<h:commandButton value="New Movie" action="addmovie" />
```

Download all listings in this issue as text

LEARN MORE

• Part 1 of this series

```
// Use lambda expressions integrated with core
// Collection libraries to process big data
// in parallel
int sum =
  txns.parallelStream()
       .filter(t -> t.getBuyer().getState()
             .equals("NY"))
       .mapToInt(t -> t.getPrice())
       .sum();
```

BIG DATA FOR JAVA DEVELOPERS

Oracle's **Dan McClary** on why big data and Java were made for each other **BY TIMOTHY BENEKE**

ART BY I-HUA CHEN; PHOTOGRAPHY BY BOB ADLER

Dan McClary (right), principal product manager for big data at Oracle, and Jacco Draaijer, senior director of development for Oracle Big Data Appliance, meet with members of the Oracle big data team.

Big Data and Java

Big data is a term that has, in recent years, entered the mind of just about anyone with even a passing interest in information technology. The rise of cloud computing, the proliferation of social networks, and growing computing power have joined together to provide access to huge amounts of data, which, if we only knew what to do with it, would increase our insight, enhance our decision-making, and make our lives easier. Something big is coming, but it's not clear what—or how best to prepare for it. In 2012, Gartner estimated that 4.5 million new jobs worldwide would be generated globally by 2015 to support big data. Many IT experts worry that there will not be enough talent to fill these jobs.

To find out what big data means for Java developers, we sat down with Dan McClary, principal product manager for big data at Oracle. McClary holds a PhD in computer science from Arizona State University, where his work centered on adoptive optimization of mobile and ad hoc networks.

Java Magazine: Much of the focus on big data is how it will enable us to better predict the future. How should we think about it?

McClary: Big data is touted for its ability to make predictions, but in fact, it's about something more fundamental. It's important to realize that discovery is not so much about finding new landscapes, but about seeing what is in front of us with new eyes. Big data expands our ability to see not just in the predictive sense, but in terms of the information we can now examine. It's a set of tools that allows us to inspect more closely the problems with which we are entrusted. This may take the form of greater prediction, or may simply enable us to reimagine and reconceive existing processes. This will require a lot of hard work.

Java Magazine: It is easy to see why it is hyped, because it offers the possibility of a significant extension of human intelligence.

McClary: Yes, we will have more and more kinds of better information, but it is up to us to analyze that information. We would not have been able to process all the information coming out of CERN that led to the discovery of the Higgs boson particle, which won a Nobel Prize, were it not for recent IT developments. Physicists were able to confirm a theory that they could not confirm otherwise.

But we must realize that new eyes require more effort. We must ask new questions, or ask questions in a different way, and many systems and analytical skills will be necessary. A fundamental X factor is the creativity needed to ask new questions and to tell stories based upon greater amounts of data.

Java Magazine: How necessary is a knowledge of statistics?

McClary: It's not a prerequisite. If you are working on things that are specifically related to machine learning that are actually predictive, then you may need a heavy numerical or statistical background. But if you have orders of magnitude more data than in the past, you are often able to avoid some of the more complex techniques that were developed in previous decades to deal with limited amounts of data. So unless you're doing something that is fundamentally statistically oriented, there's not a huge need for the average developer to learn complicated analytical methods. But it certainly doesn't hurt because it does open up opportunities. What is more fundamental is the ability to ask the right question.

Java Magazine: How is Java equipped to handle the volume requirements of big data?

McClary: If you look at many of the frameworks that are central to the notion of big data, they're all Java Virtual Machine [JVM]–based, especially those that are designed to deal with volume. This is because the JVM is mature, flexible, and scalable enough as

McClary talks with Qi Pan, technical staff member of the Oracle big data team, about the Oracle Big Data Appliance configurator.

McClary and Draaijer discuss the maximum number of racks that can connect in a Hadoop cluster.

a systems programming environment to handle massive volumes of data and a variety of hardware.

The ability to plant JVMs on many different pieces of commodity hardware and have them operate well together is the only way to really get to volume. It would be far more difficult to use MPI [message passing interface] to gather commodity hardware and run C binaries on it, which might require different processors and engines. The volume of data we're talking about requires massive distributed systems, and the JVM is well suited to this in a way that other environments are not.

Java Magazine: How do some of the languages for the JVM fit in with big data?

McClary: Because the JVM's underlying frameworks and a lot of the common pieces of Hadoop and Cassandra end up being written in Java EE, it makes sense to use it because it's scalable, it's strongly typed, it has known behaviors, and there are people who understand how to both write code to take the greatest advantage of the JVM and, at the same time, tune the JVM for maximum performance.

The existing JVM-based scripting languages, such as Jython, JRuby, and Groovy, play an important role because you're able to take analysts and programmers familiar with scripting languages and immediately insert them into an environment where they can write scripts using the scripting languages they're comfortable with, and the scripts can scale by virtue of having the JVM handle them. You're able to democratize access to this data because of the variety of scripting languages. It also makes the development and integration of languages feasible. Languages such as Hive and Pig, which are JVM-based, were developed specifically to operate on Hadoop.

Java Magazine: How does Java handle the variety of datatypes that big data brings?

A BETTER VIEW

It's important to realize that discovery is not so much about finding new landscapes, but about seeing what is in front of us with new eyes.

McClary: InputFormat, which is an abstract class, is not talked about enough. People talk about big data and sometimes mention Schema-on-Read. This allows people interrogating this data to say what shape they want the data to take. InputFormat enables people to specify how they want to be able to read the bytes.

Because InputFormat implementers are well-defined Java classes, they become extremely portable. New systems can pick up these readers and have flexibility of access. We can ask lots of different questions in lots of different ways.

Java Magazine: What about velocity—*fast data*—and the JVM?

McClary: The notion of fast data is becoming increasingly important. Fast data involves processing fast-moving streams of data as quickly as possible. Tools such as Storm or Oracle Event Processing are both Java based. It's very easy to see the extension of Java to the fast data environment because we can deploy it as widely as we can in the volume case, and we have the performance characteristics necessary to process fast-moving streams of data.

Java Magazine: What is Oracle's vision of big data?

McClary: Oracle's core competency is data management—so of course

The Big Data Toolbox

If you're building applications for big data, there are many tools available for the job. Here's a quick overview, with pointers to more information in this issue.

Apache Hadoop: A framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The project includes the following modules:

Hadoop Common: The common utilities that support the other Hadoop modules.

Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data.

Hadoop YARN: A framework for job scheduling and cluster resource management.

Hadoop MapReduce: A YARN-based system for parallel processing of large data sets. Learn more about Hadoop and its modules on pages 34 and 40.

Other Hadoop-related projects at Apache include the following:

Ambari: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters.

Avro: A data serialization system. See more on page 40.

Cassandra: A scalable multimaster database with no single point of failure. See more on page 47.

Chukwa: A data collection system for managing large distributed systems.

Crunch: A framework for creating MapReduce pipelines. See more on pages 34 and 40.

HBase: A scalable, distributed database that supports structured data storage for large tables. See more on page 34.

Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying. See more on pages 34 and 40.

Mahout: A scalable machine learning and data mining library. See more on page 34.

Pig: A high-level data-flow language and execution framework for parallel computation. See more on pages 34 and 40.

ZooKeeper: A high-performance coordination service for distributed applications.

Other tools for big data include the following:

Cascading: A framework for creating MapReduce pipelines. See more on pages 34 and 40.

Esper: A component for complex event processing and event series analysis.

MongoDB: An open source NoSQL document database that provides high performance, high availability, and automatic scaling. See more on page 51.

Neo4j: A transactional property graph database for managing and querying highly connected data.

Storm: An open source, language-independent big data processing system intended for distributed real-time processing.

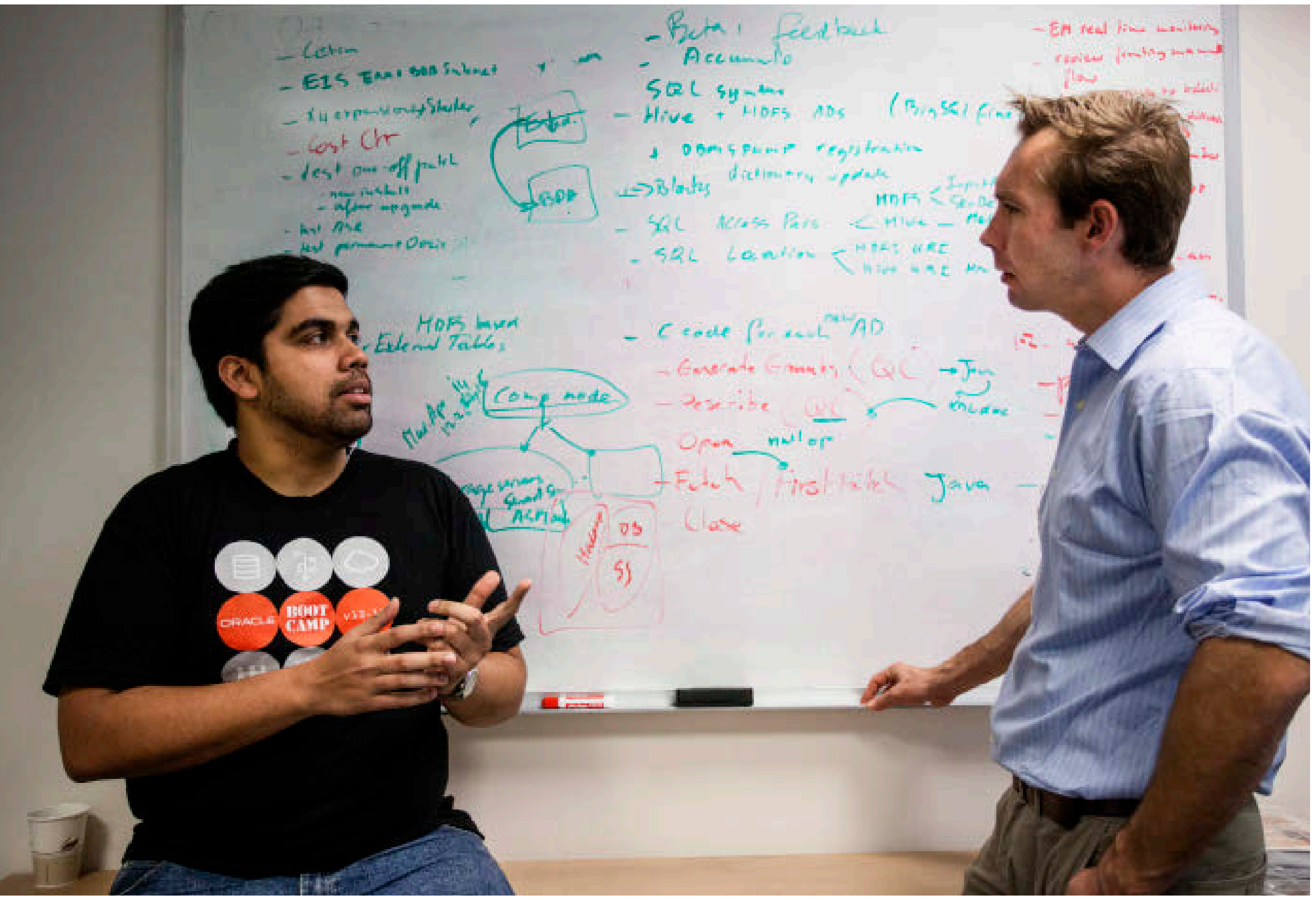
—Caroline Kvitka

A KEY PLAYER

You can't underestimate the importance of Hadoop. If you're building and running a big data system, pieces of the Hadoop ecosystem are central to it.

big data is important to us. We've been dealing with petabytes of data for years, and we view big data as an extension of the way people can view the world. Managing that data and making sure that these new systems allow for greater insight and play well with existing investments is important to us. Our aim is to provide the best possible integration points with our existing suite of data management products and, at the same time, drive standards in the broader open source community to meet the needs of consumers. We are participating in the Apache community to make sure that the

The Java Virtual Machine is mature, flexible, and scalable enough to handle massive volumes of data, says McClary.



Ali Rizvi, security specialist on the Oracle big data team, talks with McClary about setting up Apache Sentry authorization on Oracle Big Data Appliance.

contributions we make to Hadoop return to the community.

Java Magazine: What role does Hadoop play?

McClary: You can't underestimate the importance of Hadoop to big data; it's the word most associated with big data. Hadoop is at its core petabyte-scale processing and storage. But it's also part of a wider ecosystem of tools, which is constantly growing and evolving, that includes higher-level languages plus other types of data storage on top of Hadoop. So if you're building and running a big data system, pieces of the Hadoop ecosystem are central to it.

Java Magazine: What kinds of skills should Java developers cultivate to compete in the world of big data?

McClary: For application developers who mostly work in Java, the skills they should cultivate center mainly around thinking about how to decompose problems into parallel parts. When you write algorithms and implement them in systems such as MapReduce, you have to think about parallelism in a way that isn't necessarily obvious if you're just writing threaded applications that have shared memory.

If you're writing code to count or classify something, you have to understand that each of your processes that receives data and produces output will only see part of the picture. So when we think about these algorithms, we have to think about how to decompose them so that discrete units can be shared to produce a big picture. This is a conceptual challenge for developers.

A second set of skills to cultivate involves learning to really understand what you can get out of Java, understanding how I/O works, and understanding Java inside lots of parallel single-threaded processes. If you are able to think about these parallelized approaches and know how to get the most out of the Java environment, you're going to produce better code in a big data environment.

The easiest thing to do is to get some code—any of the core pieces of Hadoop or the surrounding languages and tools tend to have great examples packed into the source code. So you can pull down the source code, look at examples in the directory, and you'll find a lot that you need to know to get going.

Any Java developer who is interested in big data should be looking at what's being done in GitHub because it is publicly searchable, with lots of people contributing interesting bits and pieces. They should not just look at Java, but also look at other JVM languages, such as Scala and Clojure, and understand the different ways that the JVM is approaching these problems.

All in all, it's a great time to be a Java developer. </article>

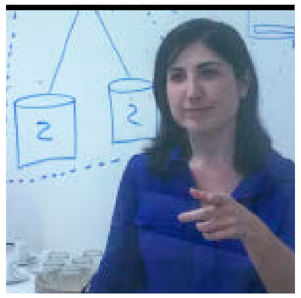
MORE ON TOPIC:

Big Data and Java

Timothy Beneke is a freelance writer and editor, best known for his books on gender. His interviews, which cover a wide range of topics, have appeared in many journals, including *Mother Jones*, the *East Bay Express*, and the *Chicago Reader*.

LEARN MORE

- The Data Warehouse Insider
- myNoSQL
- "An Engine for Big Data"
- Connecting Hadoop and Oracle Database with Oracle Big Data Connectors



Big Data Processing with Java

Combine knowledge about your data with knowledge about Hadoop to produce faster and more-efficient MapReduce jobs.

FABIANE NARDON AND FERNANDO BABADOPULOS

BIO

Big Data and Java

The big data revolution is happening now, and it's time to take part in it. Businesses are generating increasing volumes of data every day, and public data sets that can be repurposed to discover new information are widely available. Add the cheap, on-demand processing of cloud computing to the mix and you have a new world of possibilities.

It's not hard to imagine that many disruptive applications will emerge by leveraging the power of big data technologies in the cloud. Many startups that would not be financially viable just a few years ago are now delivering new and exciting applications. As Java developers, we are well equipped to take part in this revolution, because many of the most popular big data tools are Java-based. However, to build really scalable and powerful applications while keeping hosting costs under control, we have to rethink architectures while trying not to get lost in the myriad of tools available.

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

Hadoop and Other Tools

Apache Hadoop, a framework that allows for the distributed processing of large data sets, is probably the most well known of these tools. Besides providing a powerful MapReduce implementation and a reliable distributed file system—the Hadoop Distributed File System (HDFS)—there is also an ecosystem of big data tools built on top of Hadoop, including the following, to name a few:

- Apache HBase is a distributed database for large tables.
- Apache Hive is a data warehouse infrastructure that allows ad hoc SQL-like queries over HDFS stored data.
- Apache Pig is a high-level platform for creating MapReduce programs.
- Apache Mahout is a machine learning and data mining library.
- Apache Crunch and Cascading are both frameworks for creating MapReduce pipelines.

Although these tools are powerful, they also add overhead that won't pay off unless your data set is really big. As an exercise, try running the code examples provided with this article over a very small data set, such as a file with just one line. You will see that processing will take a lot more time than you would expect.

Hadoop and HDFS were not designed to work with small files, and they are inefficient when dealing with them.

There is also the obvious learning curve and the infrastructure work needed to put these tools into production. So before you decide to invest in them, make sure your data is really big.

How Big Is *Big*?

So, how do you determine whether you really have a big data problem? There is not a fixed size, but here are a few metrics you can use to decide whether your data is big enough:

- All your data will not fit on a single machine, meaning that you need a cluster of servers to be able to process your data in an acceptable time frame.
- You are dealing more with terabytes than gigabytes.
- The amount of data you are processing is growing consistently and it will probably double annually.

If your data is not really big,

keep things simple. You will probably save time and money processing your data with a traditional Java application or even with simpler tools, such as grep or Awk.

Efficient Big Data Processing with Hadoop

If you decided to use Hadoop for analyzing your data set, you will want to avoid performance bottlenecks as your data increases. There are many configuration tunings you can apply to a Hadoop cluster, and you can always add more nodes if your application is not processing your data as fast as you need it to. However, keep in mind that nothing will have a bigger impact on your big data application's performance than making your own code faster.

When implementing a big data processing application, your code will typically be executed millions or billions of times during each processing cycle. Consider, for example, that you have to process a 10 GB log file in which each line is 500 bytes long. The code that analyzes each line will run 20 million times. If you can make your code only 10 microseconds faster for each line, this will make processing the file 3.3 minutes faster. Because you will probably have to process many 10 GB files per day, over time, those minutes will represent a significant economy of time and resources.

The lesson here is that every microsecond counts. Choose the fastest Java data structures for your problem, use cache when possible, avoid unnecessary object instantiations, use efficient String manipulation methods, and use your Java programming skills to produce the fastest code you can.

Besides producing efficient code, it is important to be aware of how Hadoop works so you will avoid common pitfalls. Let's start with a common beginner's mistake: not reusing Writable objects.

The code shown in **Listing 1** is a very simple MapReduce example. It receives lines from a web-access log file and counts how many times a particular internet domain was accessed. Note that in this example, we are instantiating a new object org.apache.hadoop.io.Text every time we want to write a key/value pair to the mapper output. Doing this allocates many unnecessary short-lived objects and, as you know, object allocation in Java is an expensive operation. Instead, in this case, you can safely reuse your Writable objects, as shown in **Listing 2**, which will result in faster and more-efficient code.

Note: Listing 1 and **Listing 2** (as well as all the other listings shown in this article) are excerpts. The full listings are available in the source code file.

LISTING 1 LISTING 2

```
public static class Map extends Mapper<LongWritable, Text,
   Text, IntWritable> {

   private static final IntWritable one = new IntWritable(1);

   public void map(LongWritable key,
        Text value, Context context)
        throws IOException, InterruptedException {
     try {
        String line = value.toString();
        String[] parts = line.split(" " );
        String domain = new URL(parts[2]).getHost();
        context.write(new Text(domain), one);
     } catch (MalformedURLException e) {
     }
   }
 }

 public static class Reduce
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key,
        Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
      int count = 0;
      for (IntWritable value : values) {
        count = count + value.get();
      }
      context.write(key, new IntWritable(count));
    }
 }
```

Download all listings in this issue as text

Another easy-to-apply technique that can improve the performance of your MapReduce job is to use a Combiner. However, in order to take advantage of this technique, it is useful to first understand how Hadoop MapReduce works: When you run your job, Hadoop will split your input data in smaller chunks and send each chunk to be processed by a different Hadoop node.

Each chunk will be processed by a mapper function that will produce an output. The output produced by the mapper will be stored in a buffer. When this buffer reaches a threshold, a new spill file is saved on disk. When the map task has processed all its data, the spill files are merged and sorted into a single output file. The map output files are then sent to the reducer over the network. The reducer will process the data produced by the mappers and then output the final result.

In a distributed system, the bottleneck usually is caused by the amount of data going across the network. In our example, the mapper is writing one pair (for example, <domain, 1>) for each log line. This means that potentially millions of pairs will have to be written to disk and transferred to the reducers, which will then combine the pairs and count how many times a domain was accessed.

What if we could run a mini-reducer in the mapper node and combine the pairs before sending the data across the network? That is what combiners are for. If you associate a combiner with your MapReduce job, it can aggregate data in the map phase and then transfer smaller data sets to the reducers. Most of the time, the combiner itself can be the reducer function, as shown in **Listing 3**. As an exercise, try running this example over a large log file with and without a combiner. Then, compare the jobs statistics and you will notice the difference.

Now that you understand how Hadoop MapReduce works, you can optimize your code further using a few tricks and the knowledge you have about your data. For example, in our MapReduce job, we are counting domains and we know that the number of different domains will not be huge. There are probably a few hundred at most, and we can easily fit the list in memory. So, instead of using a combiner, you can use an in-memory map and count the domains in the mapper function. In the end, the mapper outputs only the domains and their counts, and then the reducer can sum the results. The cleanup method is called at the end of the mapper task, so we can use it to output the <domain, count> pairs, as shown in **Listing 4**. If you run this example and compare the statistics with the previous implementations, you will notice that less data was saved on disk and the job ran faster.

Combining the knowledge you have about your data with knowledge on how Hadoop works, you can produce faster and more-efficient MapReduce jobs, which will save you a lot of money later.

LISTING 3 LISTING 4

```
job.setMapperClass(Mapp.class);
job.setCombinerClass(Reduce.class);
job.setReducerClass(Reduce.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
job.submit();
```

Download all listings in this issue as text

MapReduce Pipelines

The examples we have seen so far are pretty simple: a single MapReduce job that receives an input and produces an output. In real life, however, big data problems are usually not so trivial. Frequently, you have to chain several different MapReduce jobs to reach the result you want. An output of one job becomes the input of another job and so forth. This is called a MapReduce *pipeline*.

Implementing efficient pipelines requires some work, however. You need to handle intermediate data, find the right granularity for each job, and so on. Fortunately, there are tools you can use to create your MapReduce pipelines. Here are a few examples:

- Pig: A scripting language that can chain different MapReduce tasks
- Cascading: A Java framework for creating data pipelines
- Crunch: A Java library that makes it easy to create MapReduce pipelines by providing optimizations and useful abstractions

To better understand how you can take advantage of pipelines, let's explore how Crunch works and see a few examples.

Crunch uses three interfaces representing distributed data sets: PCollection, PTable, and PGroupedTable. These interfaces have a method called parallelDo. When you invoke parallelDo, it applies a function to each element in the collection in parallel and returns as a result another collection (PCollection, PTable, or PGroupedTable). You can then chain several different functions using the collection output of one function as the input for the next. Crunch will take care of the intermediate data and also create an optimized execution plan for your pipeline.

The optimization phase provided by Crunch is important because it will try to run as few MapReduce jobs as possible. Thus, when it makes sense, Crunch will combine different functions (that would normally be implemented as different MapReduce jobs) into a single, more efficient job. The rationale behind this optimization strategy is that because I/O is expensive, once you read data from disk, you had better do as much as you can with it. Also, there is overhead when you start a MapReduce job, so if you can do more work with fewer jobs, it will be more efficient.

Functions that are part of your Crunch pipeline are subclasses of the DoFn class. Instances of these functions must be serializable, since Crunch will serialize the function and distribute it to all the nodes that will be running the task.

Let's see how our domain-access counting example could be implemented using Crunch. **Listing 5** shows the DoFn function we used to process the log file, and **Listing 6** shows how the pipeline is defined and executed. Note that the pipeline will be executed only when the pipeline.done() method is called.

If you look at the code shown in **Listing 6**, you will notice that Crunch provides several methods out of the box to implement common big data processing tasks. For example, instead of writing the Reducer code to sum the values collected, we used the Aggregators.<String>SUM_INTS() method to do the same.

MapReduce pipelines are powerful mechanisms to process your big data and, as your application gets

LISTING 5 LISTING 6

```
public class Listing5
    extends DoFn<String, Pair<String, Integer>> {

   public void process(String line,
      Emitter<Pair<String, Integer>> emitter) {

      String[] parts = line.split(" ");
      try {
         URL url = new URL(parts[2]);
         emitter.emit(Pair.of(url.getHost(), 1));
      } catch(MalformedURLException e) {}
   }

}
```

Download all listings in this issue as text

more complex, you will probably want to use them.

Big Data Processing in the Cloud

Deploying a big data application in the cloud has many advantages. You can buy more machines on demand as your data grows, and you can be prepared for peaks. However, to use cloud services without incurring prohibitive scaling costs, you have to architect your application with the peculiarities of the cloud in mind. First of all, more-efficient code means less processing time and, therefore, fewer hosting costs. Every time you have to add a new node to your cluster, you are adding more costs, so it is a good practice to make sure your code is as efficient as it can be.

When deploying big data applications in the cloud, it is important to consider using a *shared-nothing architecture*. Shared-nothing architectures are basically individual machines that are connected only by the network; they don't share any disk or memory. That is why this kind of architecture scales very well; no bottlenecks are caused by competing disk access or even by another process. Each machine takes care of all the work; the machines are independent of each other and self-sufficient.

Hadoop fault-tolerance features open the possibility of exploring even cheaper cloud machine offerings, such as Amazon spot instances (machines that you can lose when the price is higher than what you bid). When you use this kind of machine to run a TaskTracker, for example, you can afford losing the machine at any time, because Hadoop will rerun the job in another node as soon as it detects that you lost one or more nodes that had jobs running.

In fact, in many big data applications, it can even be OK to lose a small part of your data. If you are doing statistical processing, which is very common, a small data set that ends up not being processed will probably not affect the final result, and you can use this to your advantage when creating your architecture.

You can use a service that provides Hadoop in the cloud to host your application. Amazon EMR is a good example of such a service. Using a Hadoop hosting service will alleviate the burden of installing and maintaining your own Hadoop cluster. However, you can also install your own Hadoop solution in the cloud, if you need more flexibility.

Another advantage of using Hadoop in the cloud is that you can monitor how your jobs are behaving and automatically add nodes or remove nodes, as needed, even with your job running. The fault-tolerance features of Hadoop ensure that everything will keep working. The trick here is to preconfigure the master node with a range of allowed slave IP addresses. This is done using the conf/slaves file inside the Hadoop installation directory. With this configuration in place, you can start a new slave node with one of the preconfigured IP addresses, and it will join the cluster automatically.

Conclusion

We are just scratching the surface of what we will be able to accomplish with big data applications in the near future. Hadoop, HBase, Hive, Cascading, Crunch, and similar tools make Java the driving force behind the big data revolution.

When the size of the data set becomes part of the problem, we have to rethink architectures and find solutions that will make our code more efficient. Fortunately, that's the kind of challenge that we Java developers will gladly accept. </article>

MORE ON TOPIC:

Big Data and Java

LEARN MORE

- Apache Hadoop
- Introduction to Data Science

MAKE THE FUTURE JAVA

Java™

FIND YOUR JUG HERE

My local and global JUGs are great places to network both for knowledge and work. My global JUG introduces me to Java developers all over the world.

Régina ten Bruggencate
JDuchess

LEARN MORE

ORACLE®

TOM WHITE

BIO

Introduction to Hadoop

Write big data applications with Hadoop and the Kite SDK.

Big Data and Java

Writing applications that store and process large volumes of data is very much in demand these days, and Apache Hadoop is a common choice of platform for working with big data. It can be difficult to know how to get started with Hadoop, however. So in this article we're going to look at it from a Java developer's point of view, and write an application to store and process arbitrary event objects that are generated by an external source.

If you haven't heard of Hadoop, it's an Apache-hosted project that offers a distributed file system (called the Hadoop Distributed File System, or HDFS) and a batch computing model (called MapReduce) for processing large data sets stored in HDFS. At least, that was a good definition of Hadoop a few years ago, but nowadays Hadoop is generally taken to mean the stack of components that build on and around the HDFS and MapReduce core.

The stack contains components with whimsical names such as Flume, Sqoop, Pig, and Oozie (all of which are Apache projects in their own right) and, indeed, Hadoop itself was named for the creator's son's stuffed yellow elephant toy. **Table 1** summarizes some of the more common Hadoop components relevant to this article.

Although Hadoop was unequivocally a batch-processing system in its initial incarnation, it's important to understand that newer components provide more-responsive, low-latency analysis of large data sets. For example, interactive SQL engines, such as Impala, bring the convenience of low-latency SQL queries to the platform; document search systems, such as Apache Solr, now run natively on Hadoop; and there are even in-memory processing engines, such as Spark (in the Apache Incubator), for running complex data processing pipelines expressed in programming languages such as Java, Python, and Scala.

The common foundation for all these components is HDFS, the distributed file system. Each component reads and writes data stored in HDFS, and this opens up the possibility of being able to use the same HDFS-resident data sets between different components.

Creating Hadoop Applications

Imagine you have the following scenario: a web property that generates a large number of events from multiple sources such as user

COMPONENT	DESCRIPTION
APACHE AVRO	A CROSS-LANGUAGE DATA SERIALIZATION LIBRARY
APACHE HIVE	A DATA WAREHOUSE AND SQL ENGINE
APACHE PIG	A DATAFLOW LANGUAGE AND EXECUTION ENGINE
APACHE FLUME	A STREAMING LOG-CAPTURE AND DELIVERY SYSTEM
APACHE OOZIE	A WORKFLOW SCHEDULER SYSTEM
APACHE CRUNCH	A JAVA API FOR WRITING DATA PIPELINES
CASCADING	AN API FOR WRITING DATA APPLICATIONS
PARQUET	A COLUMN-ORIENTED STORAGE FORMAT FOR NESTED DATA
CLOUDERA IMPALA	AN INTERACTIVE SQL ENGINE RUNNING ON HADOOP
HUE	A WEB UI FOR INTERACTING WITH HADOOP

Table 1

click streams, application events, machine log events, and so on. The goal is to capture, store, and analyze the events so you can discover changes in usage patterns and behavior over time.

Here is how you might build this with Hadoop: incoming event data is written to HDFS via Flume, while being indexed and searched from HDFS via Solr. Periodically, batch processing of portions of the data set (for example, the last day, week, month, or year) is carried out using a MapReduce-based tool, such as Crunch, Cascading, Hive, or Pig. Both the original data set and the derived data set are queried interactively using Impala through the command-line interface, using a web dashboard, or using a business intelligence package.

That's the high-level view; but how do you go about building such an application? That's where it's easy to get stuck as a Hadoop newcomer, because typically you'd have to dive into the details of each Hadoop component, while along the way answering questions such as the following:

- What file format should I use?
- How do I lay out my files in HDFS in date-based partitions?
- How do I describe the data model for entities in Flume, Solr, Crunch, Impala, and so on?

In short, how do you get all the components to play nicely together?

At Cloudera, we've been asked these questions for a while by users, customers, and partners, which led us to create the Kite SDK (formerly the Cloudera Development Kit, or CDK) to codify some of the best practices in Hadoop application development, and to provide an API and a set of examples that developers can use to get started quickly with their projects.

Let's look next at how to build an application on Hadoop using Kite.

Note: All the code for this article can be found on GitHub, along with instructions for running it on Cloudera's QuickStart VM, a freely downloadable virtual machine preinstalled with all the Hadoop services you need to get started.

Describing the Data

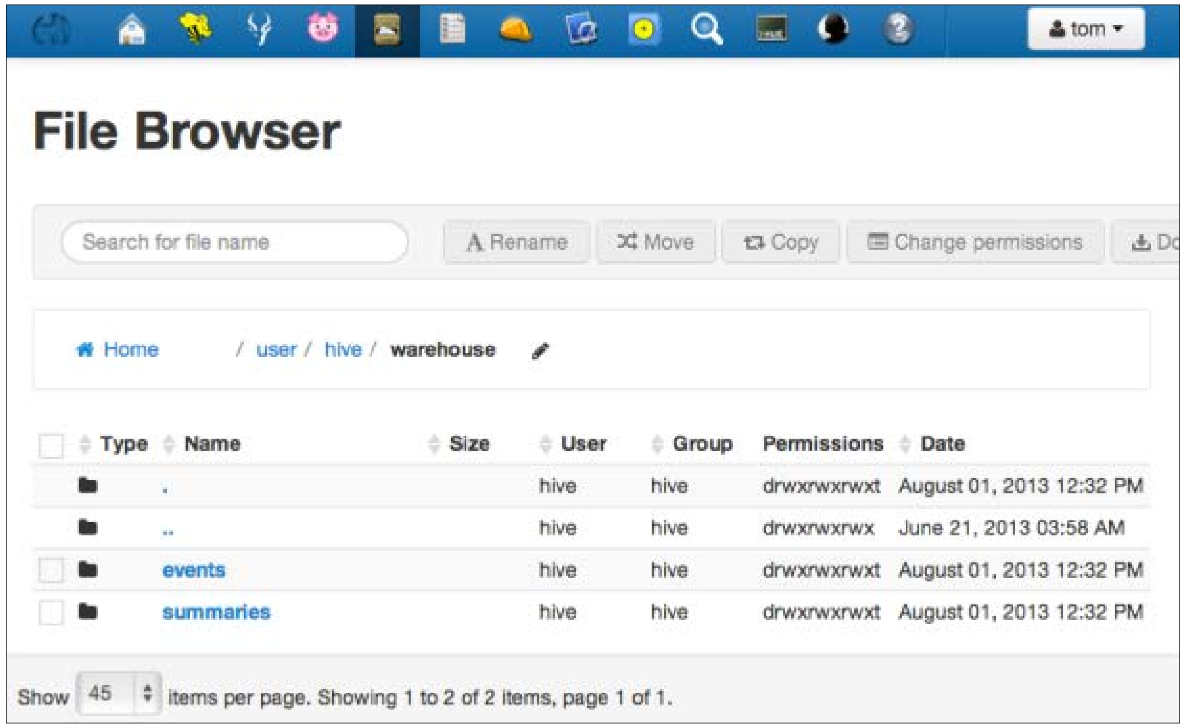
One of the cornerstones of Kite is its use of Avro as a common data model for the entities in the system. Avro defines a simple and concise way to describe your data, as well as a binary serialization format for Avro data. However, as an application developer, you don't have to delve into the internals of Avro, because you can define your entities as plain old Java objects (POJOs) and the system will do the mapping to Avro for you.

LISTING 1 LISTING 2

```
package com.tom_e_white.javamagazine;

import com.google.common.base.Objects;

public class Event {
  private long id;
  private long timestamp;
  private String source;

  public long getId() { return id; }
  public void setId(long id) { this.id = id; }

  public long getTimestamp() { return timestamp; }
  public void setTimestamp(long ts) { this.timestamp = ts; }

  public String getSource() { return source; }
  public void setSource(String source) { this.source = source; }

  @Override
  public String toString() {
    return Objects.toStringHelper(this)
        .add("id", id)
        .add("timestamp", timestamp)
        .add("source", source)
        .toString();
  }
}
```

Download all listings in this issue as text

Listing 1 shows a simple Event class, which has three fields—id, timestamp, and source—as well as associated getters and setters. **Listing 2** shows another POJO for a Summary class, which we'll use to produce simple summaries of collections of Event objects from the same source, bucketed into time intervals. Thanks to Avro, these two Java classes are a complete description of the data model.



File Browser

Type	Name	Size	User	Group	Permissions	Date
	.		hive	hive	drwxrwxrwxt	August 01, 2013 12:32 PM
	..		hive	hive	drwxrwxrwx	June 21, 2013 03:58 AM
	events		hive	hive	drwxrwxrwxt	August 01, 2013 12:32 PM
	summaries		hive	hive	drwxrwxrwxt	August 01, 2013 12:32 PM

Show 45 items per page. Showing 1 to 2 of 2 items, page 1 of 1.

Figure 1

Creating Data Sets

Now that we have defined our data model, the next step is to create somewhere to store our data sets. A *data set repository* is a physical storage location for data sets and their metadata, and Kite provides a few choices for the repository. We'll use a data set repository that stores its data in HDFS and its metadata in the Hive metastore—the latter choice means that the metadata will be accessible later from other systems, such as Impala, when we want to run SQL queries against the data.

A data set repository can be referred to by a URI with a scheme name of repo and a hierarchical part that describes the storage. For example, repo:hive refers to a Hive storage location where the data is stored in Hive's warehouse directory (in HDFS), and the metadata is stored in the Hive metastore. Alternatively, a URI such as repo:hdfs://namenode:8020/data refers to a storage location under the /data directory in HDFS, with metadata stored on the file system, too, under /data/.metadata.

Listing 3 shows a short program to create two data sets, called events and summaries. The program is invoked with a single argument to specify the data set repository URI (we'll use repo:hive), and this is used to create a DatasetRepository instance using the static open() method on DatasetRepositories.

LISTING 3

```
package com.tom_e_white.javamagazine;

import org.kitesdk.data.DatasetDescriptor;
import org.kitesdk.data.DatasetRepositories;
import org.kitesdk.data.DatasetRepository;

public class CreateDatasets {

 public static void main(String[] args) throws Exception {
  DatasetRepository repo = DatasetRepositories.open(args[0]);

  DatasetDescriptor descriptor =
    new DatasetDescriptor.Builder()
      .schema(Event.class).build();
  repo.create("events", descriptor);

  DatasetDescriptor summaryDescriptor =
    new DatasetDescriptor.Builder()
      .schema(Summary.class).build();
  repo.create("summaries", summaryDescriptor);
 }
}
```

Download all listings in this issue as text

As shown in **Listing 3**, a data set is described by a DatasetDescriptor, which ties down its schema, format, compression codec, and partitioning scheme, among other things. Here, we specify the schemas for our data sets indirectly from the POJO classes, and we create the data sets by calling create() on the DatasetRepository instance passing the name and the DatasetDescriptor instance.

The other settings—format, compression, and so on—are not set explicitly, so the defaults are used. The default format, for example, is Avro binary data, but we could choose Parquet for greater efficiency if our data sets had dozens of fields and the queries read only a few of them at a time.

After running CreateDatasets, we can peek in HDFS (using Hue, a web UI for Hadoop) to confirm that the data set directories have been created, as shown in **Figure 1**.

Populating a Data Set Using Flume

From the client's point of view, logging Event objects is easy: we just use an appropriately configured Apache log4j logger. **Listing 4** shows a Java application, GenerateEvents, that creates a new Event object every 100 milliseconds and logs it using an org.apache.log4j.Logger instance.

The log4j configuration, log4j.properties, is shown in **Listing 5**. It defines a log4j appender for GenerateEvents that is configured to send log event objects to a local Flume agent running on port 41415. Furthermore, the appender is data set–aware and, in this case, knows that the events should be written to the events data set in the repo:hive repository, which is the data set we created earlier.

WHAT HADOOP IS NOW

Although Hadoop was unequivocally a batch-processing system in its initial incarnation, it's important to understand that newer components provide more-responsive, low-latency analysis of large data sets.

Creating a Derived Data Set Using Crunch

After running GenerateEvents for a while, we see files being written to the events directory in HDFS. The next step is to process the event data to generate a derived data set called summaries, which as the name suggests is a shorter summary of the event data.

Listing 6 is a Crunch program that runs over all the event data to count the number of events for each source and for each minute of the day. If we generated events over a number of days, then by bucketing by minute of day like this, we would see any diurnal patterns in the data.

The first three lines of the run() method in **Listing 6** show how the events and summaries data sets that we created earlier are loaded. The rest of the program is written using the Crunch API, so before getting to that, we'll have a look at the basics of Crunch.

Crunch works on *pipelines*, which orchestrate the flow of data from one or more input *sources* to one or more output *targets*. Within a pipeline, data is transformed by an arbitrary chain of *functions*, represented by instances of org.apache.crunch.DoFn<S, T>, which define a mapping from a source type S to a target type T.

Functions operate on Crunch's PCollection<T> type, which represents a distributed, unordered collection of elements of type T. In many ways, it is the unmaterialized, distributed equivalent of java.util.Collection<T>.

PCollection has a small number of primitive transformation methods, including parallelDo() for applying a function to every element in the PCollection, and by() for extracting a key from each element using a function. The by() method returns a subclass of PCollection<T> called PTable<K, V>, which is a table of

LISTING 4 / LISTING 5 / LISTING 6

```
package com.tom_e_white.javamagazine;

import java.util.UUID;
import org.apache.log4j.Logger;

public class GenerateEvents {
 public static void main(String[] args) throws Exception {
  Logger logger = Logger.getLogger(GenerateEvents.class);
  long i = 0;
  String source = UUID.randomUUID().toString();
  while (true) {
   Event event = new Event();
   event.setId(i++);
   event.setTimestamp(System.currentTimeMillis());
   event.setSource(source);
   logger.info(event);
   Thread.sleep(100);
  }
 }
}
```

Download all listings in this issue as text

key-value pairs that offers grouping and combining methods to perform aggregation.

If you are familiar with MapReduce, you'll notice that these primitive operations are very similar. The major difference with Crunch is that you are not constrained to key-value pairs—you can use regular Java objects—and you don't have to manage the separate MapReduce jobs, because Crunch will compile the pipeline into a series of MapReduce jobs and run them for you.

In a typical Crunch program, then, a source data set will be read as a PCollection<S> instance; then be transformed using a chained sequence of methods to create another PCollection<T>, with a different element type; and then finally written to a target data set. Both the source and target are usually in HDFS.

Translating this into the code shown in **Listing 6**, the events data set is read into a PCollection<Event> instance (using a convenience method on CrunchDatasets to translate the Dataset into a Crunch source), then transformed into a PCollection<Summary>, and finally written to the summaries data set. (Note that we are using the same Java classes, Event and Summary, that we defined earlier for the data set.)

The first transformation is the key-extraction operation defined by the DoFn implementation GetTimeAndSourceBucket (see **Listing 7**) that extracts the minute of day time stamp (minuteBucket) from each Event object and returns the (minuteBucket, source) pair as the key by which to group.

The second transformation is the parallel operation over all groups, which is used here to summarize each time bucket. Looking at **Listing 8**, we see that MakeSummary is a function whose input is a Pair<Long, String>, Iterable<Event>, which just means that it gets the (minuteBucket, source) key pair from the previous grouping operation and an Iterable over all the Event objects for that key. The implementation simply uses this information to construct a Summary object that is the output of the function, returned by writing it to the Crunch Emitter object.

JUMP START

The Kite SDK codifies some of the best practices in Hadoop application development and provides an API and a set of examples that developers can use to get started quickly with their projects.

It's worth noting that Crunch functions can contain arbitrary Java code, using any Java libraries you choose (we used the Joda-Time API in GetTimeAndSourceBucket), and this brings great power to the types of analyses you can write.

LISTING 7 / LISTING 8 / LISTING 9

```
package com.tom_e_white.javamagazine;

import org.apache.crunch.MapFn;
import org.apache.crunch.Pair;
import org.joda.time.DateTime;
import org.joda.time.DateTimeZone;

class GetTimeAndSourceBucket extends
   MapFn<Event, Pair<Long, String>> {
  @Override
  public Pair<Long, String> map(Event event) {
   long minuteBucket = new DateTime(event.getTimestamp())
      .withZone(DateTimeZone.UTC)
      .minuteOfDay()
      .roundFloorCopy()
      .getMillis();
   return Pair.of(minuteBucket, event.getSource());
  }
}
```

Download all listings in this issue as text

Reading a Data Set from Java

It's easy to iterate over the entities in a data set using the Java API. This is illustrated in **Listing 9** for the summaries data set that we just wrote with the Crunch job. We load the data set as an instance of Dataset<Summary>, and then get a DatasetReader<Summary> by calling its newReader() method.

DatasetReader is a Java Iterable, so we can iterate over its entries in an enhanced for loop, printing each Summary object to the console as we go.

What we see is

```
Summary{source=973e9def...,
 bucket=1384965540000,
 count=212}
Summary{source=973e9def...,
 bucket=1384965600000,
 count=37}
```

Reading a Data Set from Impala Using JDBC

Because the data set metadata is stored in the Hive metastore, we can use Hive or Impala to query the data set using SQL. **Listing 10** shows a program that uses the JDBC API to dump the records that are in the summaries data set to the console. The Hive JDBC driver supports both Hive and Impala; in this example, we are connecting to Impala, which listens on port 21050 for JDBC connections.

The output on the console is very similar to before:

```
source=973e9def...,
 bucket=1384965540000,
 count=212
source=973e9def...,
 bucket=1384965600000,
 count=37
```

Of course, being JDBC it's possible to use any JDBC framework to access these data sets, for example, a web-based dashboard showing a few key queries.

There are lots of other things you can build with Kite; let's look at a few extensions in the following sections.

Updating Data Sets

The Crunch job we saw in **Listing 6** runs over the whole events data set in one go, but in practice, the events data set would be partitioned by time, and the job would be run periodically on new data set partitions.

The Kite Maven plug-in helps with this, by allowing you to package, deploy, and run applications—in this case, a Crunch job. An application is packaged a bit like a WAR file, with bundled library dependencies and configuration information (such as the Hadoop cluster to use), and it is run by Oozie, either as a one-off *workflow* or as a repeating *coordinator* application that runs on a defined schedule. The Kite examples include an example of running a repeating job.

Integrating Search

Another extension would be to add a Solr Flume sink to the system to index events before they are sent to HDFS. In this design, the Lucene index is stored in HDFS, and the

LISTING 10

```
package com.tom_e_white.javamagazine;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class ReadSummariesJdbc {

 public static void main(String[] args) throws Exception {
  Class.forName("org.apache.hive.jdbc.HiveDriver");

  Connection connection = DriverManager.getConnection(
    "jdbc:hive2://localhost:21050/;auth=noSasl");
  Statement statement = connection.createStatement();
  ResultSet resultSet = statement.executeQuery(
    "SELECT * FROM summaries");
  while (resultSet.next()) {
   String source = resultSet.getString("source");
   String bucket = resultSet.getString("bucket");
   String count = resultSet.getString("count");
   System.out.printf("source=%s, bucket=%s, count=%s\n",
     source, bucket, count);
  }

  resultSet.close();
  statement.close();
  connection.close();
 }
}
```

Download all listings in this issue as text

SolrCloud servers run in the same cluster as Hadoop. By using any libraries or frameworks that talk to the Solr API, we can then run searches against the data in "near real time," which means that events typically appear in the index seconds after ingest.

Handling Schema Evolution

In the real world, the data model changes, and it's vital that the code can adapt to such changes. Avro has clear rules about schema changes, and these are built into the data set API so that compatible updates are allowed, but incompatible updates are rejected. For example, adding a new field with no default is not allowed, because it would not be possible to read old data with the new schema, as there would be no default value for the new field.

For example, here is how you would update the Event class to add a new field, called ip.

```
public class Event {
  private long id;
  private long timestamp;
  private String source;
  @Nullable private String ip;
  ... getters and setters elided
}
```

The Nullable annotation is used to indicate that the field has a default value of null, so when an old event object is read into the new object, its ip field is null.

Conclusion

In this article we have seen how to get started with Hadoop, the de facto platform for big data, from a Java developer's point of view. Using Kite's Java APIs we have built a rich application for storing and processing data sets in a flexible manner, using techniques that you can use in your next big data project. </article>

Thanks to Ryan Blue for reading a draft of this article.

MORE ON TOPIC:

Big Data and Java

LEARN MORE

- Kite documentation
- Kite examples
- *Hadoop: The Definitive Guide*

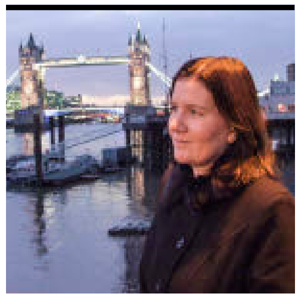
3 Billion Devices Run Java

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
or call 1.800.ORACLE.1



KIM ROSS

BIO

Introduction to Cassandra

Learn the many benefits of choosing Cassandra as a storage solution.

Apache Cassandra is a NoSQL persistence solution that offers distributed data, linear scalability, and a tunable consistency level, making it well suited for highly available and high-volume applications. Developed by Facebook and open sourced in 2008, Cassandra was influenced by the Bigtable and Dynamo white papers. Dynamo inspired the partitioning and replication in Cassandra, while Cassandra's log-structured Column Family data model—which stores data associated with a key in a schemaless way—is similar to Bigtable. Cassandra is used by many big names in the online industry including Netflix, eBay, Twitter, and reddit.

Advantages

There are many benefits to choosing Cassandra as a storage solution; one of the most compelling is its speed. Cassandra is known for its exceptionally fast writes but is also extremely competitive with its fast read speed. It is also highly available. Its decentralized peer-to-peer architecture means there are no master components that could become single points of failure, so it is extremely fault-tolerant—especially if you spread your nodes over multiple data centers.

Cassandra also offers linear scaling with no downtime. If you need *x* number of nodes for *y* number of requests, 2*x* nodes will cater to 2*y* requests. With the introduction of virtual nodes in version 1.2, the load increase normally experienced while increasing capacity is spread across all the nodes making scaling while under load a non-issue. It is very simple to scale both up and down.

Another benefit is the flexibility achieved by the ability to tune the consistency level for each request according to the needs of your application.

Cassandra has an active community, with big players such as Netflix and Twitter, contributing open source libraries. Cassandra is actively under development, which means that there are frequent updates with many improvements.

Disadvantages

Although Cassandra is highly effective for the right use cases and has many advantages, it is not a panacea.

One of the biggest challenges facing developers who come from a relational database background is the lack of ACID (atomicity, consistency, isolation, and durability) properties—especially with transactions, which many developers have come to rely on. This issue is not unique to Cassandra; it is a common side effect of distributed data stores. Developers need to carefully consider not only the data that is to be stored, but

Kim Ross describes why Cassandra was ideal for a use case at her social games startup.

Big Data and Java

also the frequency and access patterns of the data in order to design an effective data model. To ensure that data access is efficient, data is normally written to multiple column families according to how it is read. This denormalization, which is frequently used to increase performance, adds extra responsibility onto the developer to ensure that the data is updated in all relevant places. A poorly thought-out and poorly accessed data model will be extremely slow. If you are unsure how you will need to access the data (for example, for an analytics system where you might need a new view on old data), you typically will need to combine Cassandra with a MapReduce technology, such as Hadoop. Cassandra is best suited to accessing specific data against a known key.

Cassandra is a relatively young technology and although it is very stable, this lack of maturity means that it can be difficult to find staff who have expertise. However, there is a wealth of information, and conference videos are available online, which makes it easy to develop Cassandra skills quickly. So, you should not let this put you off.

Also, there is a small amount of administration to run Cassandra, because you need to ensure that repairs are run approximately weekly. Repairs ensure that all the data is synchronized across all the appropriate nodes.

READY FOR VOLUME
Cassandra is a NoSQL persistence solution that **offers distributed data, linear scalability, and a tunable consistency level,** making it well suited for highly available and high-volume applications.

Structure

Let's take a look at the structure of Cassandra. The data is stored in and replicated across nodes. *Nodes* used to refer to a server hosting a Cassandra instance, but with the introduction of virtual nodes, one server will now contain many (virtual) nodes. The number of nodes the data is duplicated to is referred to as the *replication factor*. The nodes are distributed around a token ring, and the hash of the key determines where on the token ring your data will be stored.

Cassandra provides a number of replication strategies, which, combined with the *snitch*, determine where the data is replicated. The snitch finds out information about the nodes, which the strategy then uses to replicate the data to the correct nodes. For instance, the NetworkTopologyStrategy lets you specify a replication factor per data center, thereby ensuring that your data is replicated across data centers and providing availability even if an entire data center goes down.

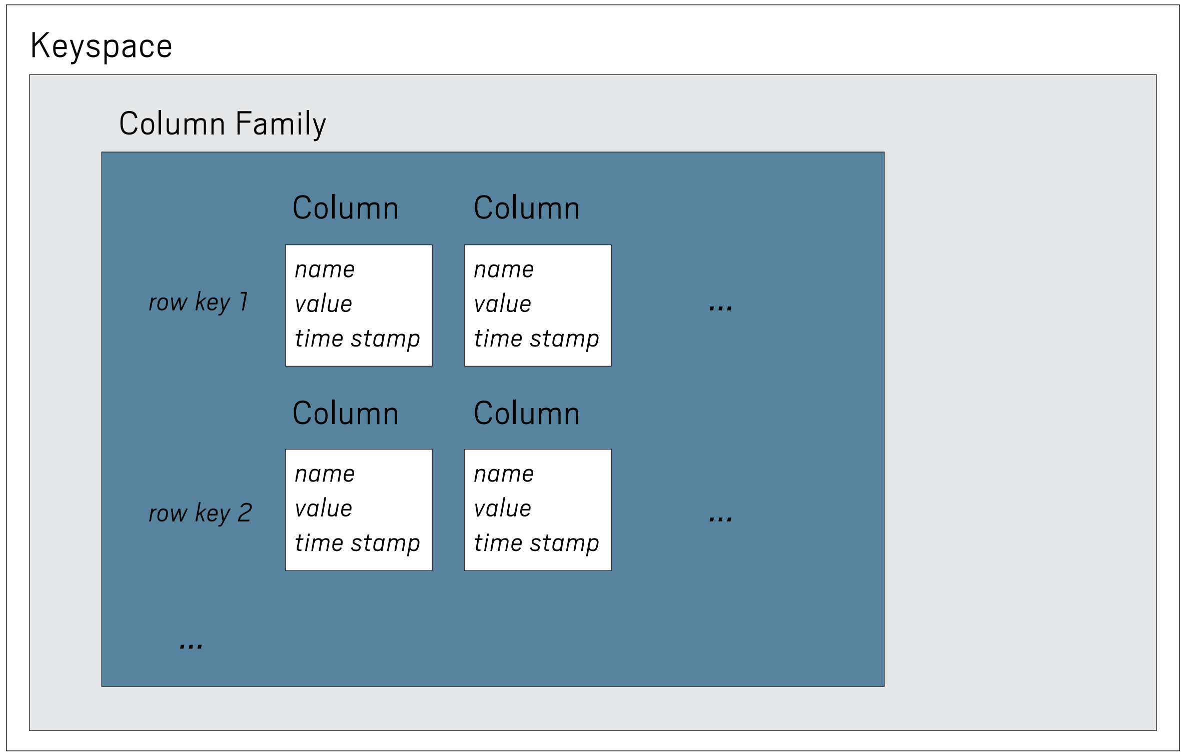


Figure 1

Data Structure

A *keyspace* is the namespace under which Column Families are defined, and it is the conceptual equivalent of a database (see **Figure 1**). The replication factor and strategy are defined at the keyspace level.

A *column family* contains a number of rows, indexed by a row key, with each row containing a collection of columns. Each column contains a column name, value, and time stamp. One important thing to note is that each row can have a completely unique collection of columns.

The ordering of the columns in a row is important; to achieve optimal performance you should read columns that are close to each other to minimize disk seeks. The ordering of columns and the datatype of each column name are specified by the comparator. The datatype of each column value is specified by the validator. It is possible to make columns automatically expire by setting the time to live (TTL) on the column. Once the

TTL period has passed, the column will automatically be deleted.

There are also a couple of special types of columns and column families. The Counter column family can contain only counter columns. A *counter* is a 64-bit signed integer and is valuable because you can perform an atomic increment or decrement on it. The only other operations you can perform on a counter column are read and delete. It is the only column type that has the ability to perform an atomic operation on the column data and, as such, incrementing a counter column is typically slower than updating a regular column unless data is written at consistency level one.

Supercolumn families can contain only supercolumns. A *supercolumn* is a column of columns and allows you to store nested data, as shown in **Figure 2**. For example, a "home" supercolumn might contain the columns "street," "city," and "postcode." Supercolumns are limited in that they can be nested only to one level. If you require more-dynamic nesting, you should use composite columns.

Composite columns are normal columns in which the column name consists of multiple, distinct components (see **Figure 3**), allowing for queries over partial matches on these names. A comparator of comparators would then be used to ensure the ordering. Composite columns allow you to nest columns as deeply as you want. Due to this ability and the performance gains composite columns had over supercolumns in earlier versions of Cassandra, they, rather than supercolumns, have become the standard for nested data.

Distributed Deletes

Distributed deletes pose a tricky problem. If Cassandra were to simply delete the data, it is possible that the delete operation would not be successfully propagated to all nodes (for example, if one node is down). Then, when a repair was performed, the data that wasn't removed would be seen as the newest data and would be replicated back across the appropriate nodes, thereby reviving the deleted data.

So instead of deleting the data, Cassandra marks it with a *tombstone*. When a repair is run, Cassandra can then see that the tombstoned data is the latest and, therefore, replicate the tombstone to the node that was unavailable at the time of the delete operation.

Tombstones and associated data are automatically deleted after 10 days by default as part of the compaction process. Therefore, it is important that a *repair* (which is a manual process) be run more frequently than the compaction process (which is normally done every 7 days) to ensure that the tombstones are replicated to all appropriate nodes before the tombstones are cleaned up.

LINEAR SCALABILITY

Cassandra is a brilliant tool if you require a scalable, high-volume data store. Its linear scalability at virtually no load cost is hard to beat when you have a sudden surge in traffic.

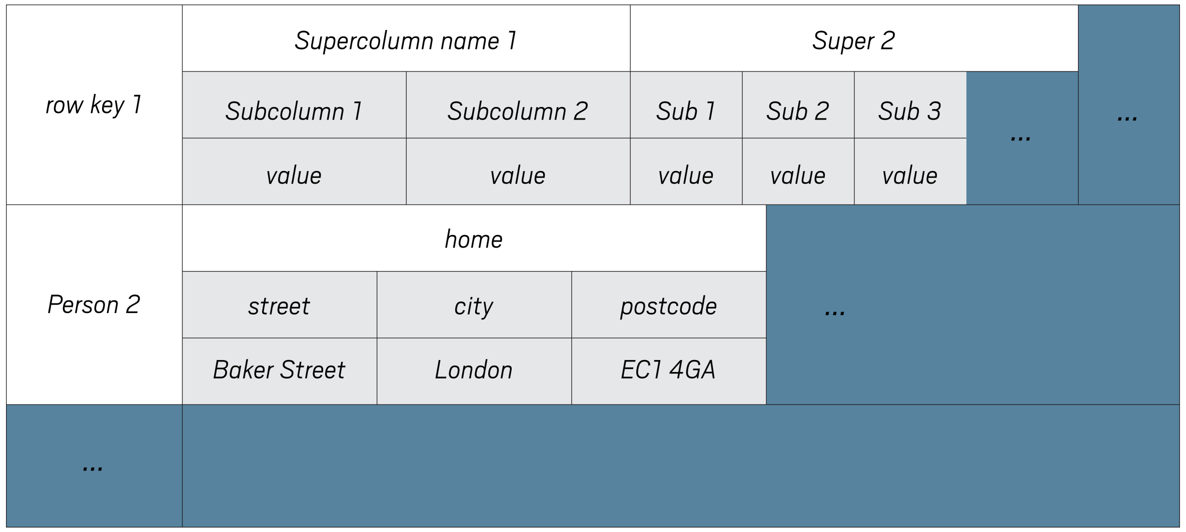


Figure 2

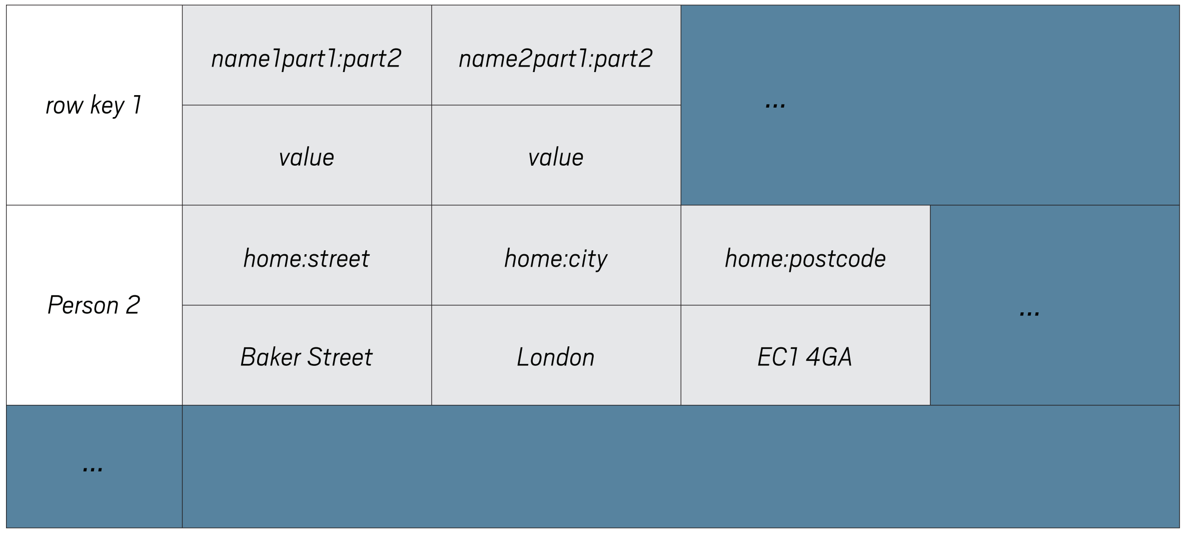


Figure 3

Consistency

Cassandra is frequently labeled as an "eventually consistent data store," although the consistency level is tunable. A database can be considered consistent when a read operation is guaranteed to return the most-recent data.

Note: When working with Cassandra, it is important to understand the CAP theorem. Out of the three elements—consistency, availability, and partition tolerance—you can achieve a maximum of two of the three.

When writing or reading data, you can set your consistency level with each data request. Consistency levels include one, two, all, and quorum, among others.

Let's assume your replication factor is 3. If you write and read at consistency level one, it is possible that the node you read from has not yet been updated for the write operation that occurred and you could read old data. It is important to note that if you write at level one, it doesn't mean that the data will be written only to one node; Cassandra will attempt to write to all the applicable nodes. All it means is that the operation will be returned as soon as Cassandra has successfully written to one node.

The quorum consistency level is 1 more than half the replication factor. For example, if the replication factor is 3, quorum would be 2.

If you read and write at the quorum consistency level, data consistency is guaranteed. If data is written to nodes A and B but could not be written to node C, and then data is read from B and C, Cassandra will use the time stamps associated with the data to determine that B has the latest data, and that is what will be returned. While giving you consistency, the quorum consistency level also allows for one of the nodes to go down without affecting your requests.

If your consistency level is all, your reads and writes will be consistent but will not be able to handle one node being unavailable. It is also worth remembering that the more nodes you need to hear from to achieve the specified consistency level, the slower Cassandra will be. Reading at the quorum consistency level (with a replication factor of 3), your speed will be that of the second-fastest node, whereas with a consistency level of all, Cassandra will need to wait for the slowest node to respond. It is worth considering whether stale data is acceptable for some of your use cases.

SPEED DEMON

There are many benefits to choosing Cassandra as a storage solution; one of the most compelling is its speed. Cassandra is known for its exceptionally fast writes but is also extremely competitive with its fast read speed.

Accessing Cassandra

Cassandra has a command-line interface, cqlsh, which you can use to directly access the data store. cqlsh takes commands in CQL, which is very similar to SQL. Don't let the syntactic similarities fool you into thinking that the background processing is the same.

There are a number of different libraries for accessing Cassandra through Java (as well as through other languages). There is a Java driver developed by Datastax that accepts CQL. Prior to the development of this driver, Java spoke to Cassandra only through the Thrift API. There are a number of libraries to support the Thrift API, such as Hector and Astyanax, among others. Both the Thrift API and the Java driver are now supported.

Conclusion

Cassandra is a brilliant tool if you require a scalable, high-volume data store. Its linear scalability at virtually no load cost is hard to beat when you have a sudden surge in traffic.

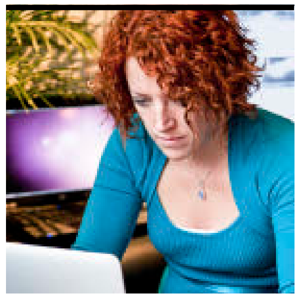
It is relatively easy to get up and running quickly, but you must pay your dues by spending time getting to know your access patterns, so you can model your data appropriately. It is worth looking under the hood of Cassandra because doing this can provide great insights into the optimal usage patterns. Given today's growing data requirements, Cassandra is a valuable addition to any developer's toolkit. `</article>`

MORE ON TOPIC:

Big Data and Java

LEARN MORE

- "Cassandra Data Modeling Best Practices, Part 1"
- "Cassandra Data Modeling Best Practices, Part 2"
- "Virtual Nodes in Cassandra 1.2"



TRISHA GEE

BIO

MongoDB and Big Data

Unlike relational databases, the document-oriented structure of MongoDB provides flexibility to make working in this big data world easier.

MongoDB is a NoSQL database whose main features include document-oriented storage, full index support, replication and high availability, auto-sharding, querying, fast in-place updates, Map/Reduce, and GridFS.

In this issue of *Java Magazine*, you'll see that *big data* is an umbrella term that covers a wide range of business problems and technical solutions. It refers to the volume, variety, and velocity of data that organizations increasingly have to deal with. MongoDB addresses these challenges in the following ways:

- Massive data volumes—MongoDB is designed to scale horizontally on commodity hardware, making it easy to add nodes to a cluster as data volumes and velocities increase. I'll cover the fundamentals of this in the section on auto-sharding.
- Real-time demands—As well as supporting the fundamental querying that you'd expect from an operational database, MongoDB's native Map/Reduce and aggregation framework support real-time analytics and insights. I'll cover simple querying in this article, but MongoDB provides much more in terms of mining your data for information.
- Rapid evolution—An organization's needs should change as it responds to the information gained from all the data it has access to. MongoDB's dynamic document schema enables applications to evolve as the needs of the business evolve. This is covered in the first section, "Document-Oriented Storage."

In this article, we'll also cover how MongoDB supports replication and high availability, requirements that are fundamental to ensuring you keep all that important data.

Document-Oriented Storage

MongoDB is a document database, meaning that it stores data in semistructured, JSON-style documents in *collections*, rather than storing data in tables that consist of rows and columns, as is done with relational databases such as MySQL. MongoDB collections are similar to tables in relational databases, although collections are a lot less rigid, as we'll discover later.

The example in **Listing 1** is a structured set of keys and their values. As well as top-level properties, such as name, the structure supports subdocuments (for example, address) and arrays (for example, the IDs of the books the patron has borrowed). Compared to relational databases, this structure is much closer to what we as object-oriented developers are used to working with. You can almost directly translate **Listing 1** into a couple of Java classes, as shown in **Listing 2**.

GOT QUERIES?
You can build up reasonably complex queries with MongoDB, which is one of the nice things about it—it's designed to be queried.

The subdocument address in **Listing 1** translates into a separate Address class in **Listing 2**, and the array in the document in **Listing 1** corresponds to a List in **Listing 2**.

Another nice thing about this document structure is that it's not

Big Data and Java

PHOTOGRAPH BY JOHN BLYTHE

fixed, which gives you a dynamic schema for your data. This is great for startups or new projects when you don't quite know how an application is going to evolve.

Let's take our Patron in **Listing 2**: this person is a borrower in an application for a library. When we first learn about the patron, we might know only his name:

```
patron = {
  _id: "joe",
  name: "Joe Bookreader",
}
```

Perhaps we require him to provide his address only when he checks out his first set of books, and we record the IDs of the books he borrowed, so his document will be like the one shown in **Listing 1**.

Suppose that at a later date, the library decides to start lending out music. It's easy to add a new field in MongoDB (see **Listing 3**). Then the application can just record the list of CD IDs when the patron checks out his first set of CDs.

The document structure can easily change to adapt to the evolving needs of the application. Instead of having to run a script that updates a relational database table with a new column, if a MongoDB document requires new information, you can simply add a new field to the document.

This dynamic representation is not only useful for changing requirements. It is also very useful when storing similar, but not quite the same, items. For example, although books and CDs have different characteristics (for example, a book has properties such as author, title, and ISBN, and a CD has properties such as artist, title, and track listing), you can still store books, CDs, and any other items in the same collection despite the fact that the documents have different fields. In a relational database table, you might handle these different characteristics using columns that have optional null values, or you might use multiple tables and joins. In a document database, this isn't necessary—all the documents with their different fields can live in the same collection.

With this dynamic document schema, MongoDB supports the increasing variety of data that organizations need to store when dealing with big data.

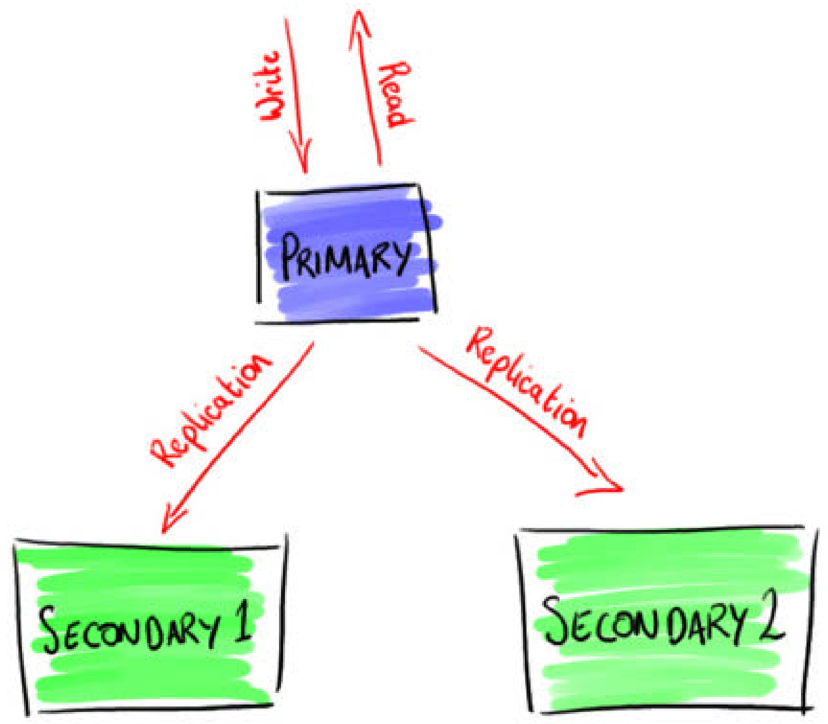


Figure 1

LISTING 1 / LISTING 2 / LISTING 3

```
{
 _id: "joe",
 name: "Joe Bookreader",
 address: {
  street: "123 Fake St",
  city: "Faketon",
  zip: 12345
 }
 books: [ 27464, 747854, ...]
}
```

Download all listings in this issue as text

Replication and High Availability

Traditionally developers have left worrying about replication and high availability (HA) to the operations guys. With the rise of DevOps and the increasing need for systems that are up 24/7, we can no longer afford to do this. Fortunately MongoDB supports replication and HA out of the box, and it's easy to set up, too.

As shown in **Figure 1**, MongoDB supports HA with *replica sets*. A replica set has a *primary*, which, by default, is where you read your data from and write all your data to, although you can configure where you read your data from. The *secondaries* will ensure that they have a copy of all the data written to the primary, taking care of the replication and providing HA. If the primary disappears from view at any time, an appropriate secondary will be voted as the new primary.

Check out the replication documentation for more details.

Auto-Sharding

Replica sets are fine for providing HA and replication, but they don't help you scale to deal with the massive data volumes and velocity prevalent in big data. That's where *sharding* comes in. With MongoDB, you can select a collection to be sharded across multiple replica sets on a given shard key, as shown in **Figure 2**.

Figure 2 shows what happens when you shard the patron collection by name. In a sharded configuration, you have more than one replica set (shown as Replica Sets 1, 2, and 3; the secondaries have been abbreviated to S1 and S2), one or more MongoS servers that act as a router between each of the replica sets, and three configuration servers that keep track of which data is in which replica set. In our example, we've chosen to shard the patron collection by patron last name. MongoDB works out how to evenly distribute the documents. MongoDB will figure out whether the distribution changes and shuffle things onto different servers as appropriate, invisible to the application. The MongoDB driver that your application uses will talk to a MongoS instance, which will do all the work in figuring out how to route your query to the correct servers.

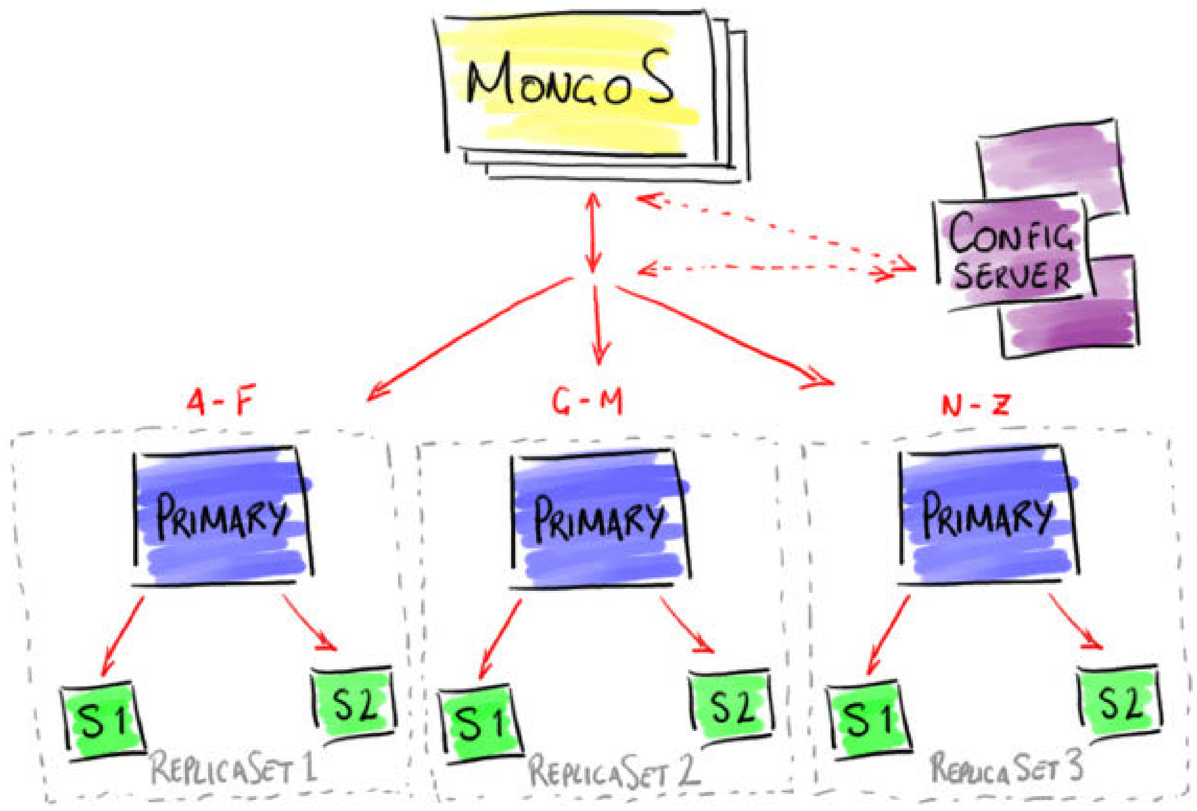


Figure 2

LISTING 4

```
 MongoClient mongoClient = new MongoClient
(new MongoClientURI("mongodb://localhost:27017"));
 DB database = mongoClient.getDB("library");
 DBcollection collection = database.getCollection("patrons");
```

Download all listings in this issue as text

Querying

We've covered the background for MongoDB, how it's a document database, and how it runs on many servers to provide availability and scalability. Now let's go into detail about how to actually interact with the server as a Java developer.

As a developer on the Java Virtual Machine (JVM), you have a lot of choice about how to interact with MongoDB. I'm going to cover the most basic method, which is using the Java driver that MongoDB Inc. supports.

When accessing databases, we often talk about CRUD operations: create, read, update, and delete. Although MongoDB is a NoSQL database and, therefore, it doesn't make much sense to use JPA or JDBC to interact with it, getting Java applications to perform CRUD operations is pretty straightforward.

Getting started. If you want to run the following examples or play with MongoDB, you first need to install MongoDB. It's lightweight enough to install and run on your laptop without worrying about it hogging all the resources.

Note: The source code for the examples described in this article can be downloaded here.

Connecting. For these examples, we're going to connect to a database called "library," and our patrons are going to be in a collection called "patrons."

To get access to the database and the collection in our Java application, we can do something like the code shown in **Listing 4**. If this database and collection don't already exist, they'll be created when you insert something into them. You don't need to explicitly create them.

Creating. Let's take the example patron we outlined earlier. If we have a patron and we want to insert it into the database, we can do something like the code shown in **Listing 5**.

Documents are basically maps that map the name of a field to a field value. In order to save something to the database, you need to turn your domain object (in this case, the Patron and its Address) into a DBObject, which is a map-like data structure. You don't have to do this yourself; there are object document mappers (ODMs) that will automatically convert a Java object into a DBObject for you, but in this article I'm covering the basics of doing it yourself. After insert is called, the document is in MongoDB.

Reading. Now that you have something in the database, you'll probably want to read it back out. You can build up reasonably complex queries with MongoDB, which is one of the nice things about it—it's designed to be queried.

To find a specific patron in the database by name, you can do something like the code shown in **Listing 6**. Queries are documents, too. In this case, we've created a query to find all documents in which the field name matches the value Sam. The results come back as a cursor that you can iterate over.

If you know there should be only a single result (for example, if you're querying by ID) or if you care only about the first result, you can do something like the code shown in **Listing 7**.

You can build up more-complex queries than this simple query by example. Let's say we want to find all the users who haven't borrowed books. You can use the $exists operator to find out whether the books array exists in the document (see **Listing 8**). And you can do operations such as sort, skip, and limit, the way you'd expect to be able to with any database, as shown in **Listing 9**.

This is a really brief overview of the sorts of queries you can create. There's a lot more information about querying on the MongoDB website.

NO COMPROMISES

MongoDB is designed to run in a scalable, highly available way out of the box, which lets developers create applications that deal with the demands of big data without having to compromise their design.

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

```
BasicDBObject addressAsDocument = new BasicDBObject
("street", patron.getAddress().getStreet())
            .append("city", patron.getAddress().getCity())
            .append("phone", patron.getAddress().getPhone());
DBObject patronAsDocument = new BasicDBObject
("_id", patron.getId())
            .append("name", patron.getName())
            .append("address", addressAsDocument)
            .append("books", patron.getBooks());
collection.insert(patronAsDocument);
```

Download all listings in this issue as text

Updating. As with reading, there are options for updating a document, too. In particular, you can either replace the whole document or simply update the fields that you care about.

If we were to update the first line of a patron's address, we could do something like **Listing 10**. In this operation, we find the document we want to update by creating the query findCharlie, which locates a patron (Charlie) by his unique ID. We also create a second document, updateCharlieAddress, which states the update operation—in this case, we want to set the street field in the address subdocument to a new value. Finally, to run this update, we pass in the query to locate the document (findCharlie) and the details of the update (updateCharlieAddress) to the update method on the collection that contains our document.

This is perfect if we have a single new value that needs to be added to or updated in our document. But often what we have is an updated domain object, and for simplicity we might want to persist this whole object back to the database with all its new values (or with values removed that we want to remove from the database). In such a case, we might have a Patron, for example, Charlie again, and we want to persist the updated Charlie object into MongoDB, overwriting the old Charlie. Code for doing this is shown in **Listing 11**. Notice that it is simpler than the code shown in **Listing 10**.

I've added a toDBObject() method onto my domain object to simplify these examples. However, LISTING 11 LISTING 12 LISTING 13

```
  Patron updatedCharlieObject = //our updated object
  DBObject findCharlie = new BasicDBObject
("_id", charlie.getId());
  collection.update
(findCharlie, updatedCharlieObject.toDBObject());
```

Download all listings in this issue as text

it would be good practice to either do this transformation in a data access object (DAO) or use one of the ODMs I mentioned earlier.

Deleting. It's not all about putting things into a database and forgetting about them; sometimes you need to remove things, too. You're probably beginning to spot a pattern in how MongoDB does things, so it shouldn't surprise you that to delete, you create a document that represents the documents to remove (see **Listing 12**).

The code in **Listing 12** will delete all documents that match the given criteria. In this case, the match will be by ID, so a single document will be removed. But you can pass in a query that matches more than one document and they will all be deleted. For example, if you wanted to remove all the patrons who lived in London, you could use the code shown in **Listing 13**.

Conclusion

MongoDB provides a document structure that is similar to the way developers think about their data: documents map well to domain objects. The database is designed to run in a scalable, highly available way out of the box, which lets developers create applications that deal with the demands of big data without having to compromise their design. In addition, the Java driver has been designed to make it simple to interact with the database using pure Java. It is also the basis of a number of other projects that create Java and JVM language APIs, so there is a wide choice of tools for interacting with MongoDB if you are a Java or JVM developer. </article>

MORE ON TOPIC:

Big Data and Java

LEARN MORE

• MongoDB ecosystem

Java SE 8 Date and Time

Why do we need a new date and time library?

BEN EVANS AND RICHARD WARBURTON

BIO

BEN EVANS PHOTOGRAPH BY JOHN BLYTHE

A long-standing bugbear of Java developers has been the inadequate support for the date and time use cases of ordinary developers.

For example, the existing classes (such as java.util.Date and SimpleDateFormatter) aren't thread-safe, leading to potential concurrency issues for users—not something the average developer would expect to deal with when writing date-handling code.

Some of the date and time classes also exhibit quite poor API design. For example, years in java.util.Date start at 1900, months start at 1, and days start at 0—not very intuitive.

These issues, and several others, have led to the popularity of third-party date and time libraries, such as Joda-Time.

In order to address these problems and provide better support in the JDK core, a new date and time API, which is free of these problems, has been designed for Java SE 8. The project has been led jointly by the author of Joda-Time (Stephen Colebourne) and Oracle, under JSR 310, and will appear in the new Java SE 8 package java.time.

Core Ideas

The new API is driven by three core ideas:

- Immutable-value classes. One of the serious weaknesses of the existing formatters in Java is that they aren't thread-safe. This puts the burden on developers to use them in a thread-safe manner and to think about concurrency problems in their day-to-day development of date-handling code. The new API avoids this issue by ensuring that all its core classes are immutable and represent well-defined values.
- Domain-driven design. The new API models its domain very precisely with classes that represent different use cases for Date and Time closely. This differs from previous Java libraries that were quite poor in that regard. For example, java.util.Date represents an instant on the timeline—a wrapper around the number of milliseconds since the UNIX epoch—but if you call toString(), the result suggests that it has a time zone, causing confusion among developers.

  This emphasis on domain-driven design offers long-term benefits around clarity and understandability, but you might need to think through your application's domain model of dates when porting from previous APIs to Java SE 8.
- Separation of chronologies. The new API allows people to work with different calendaring systems in order to support the needs of users in some areas of the world, such as Japan or Thailand, that don't necessarily follow ISO-8601. It does so without imposing additional burden on the majority of developers, who need to work only with the standard chronology.

SURPRISE!
The existing classes aren't thread-safe, leading to potential concurrency issues for users—**not something the average developer would expect.**

LocalDate and LocalTime

The first classes you will probably encounter when using the new API are LocalDate and LocalTime. They are local in the sense that they represent date and time from the context of the observer, such as a calendar on a desk or a clock on your wall. There is also a composite class called LocalDateTime, which is a pairing of LocalDate and LocalTime.

Time zones, which disambiguate the contexts of different observers, are put to one side here; you should use these local classes when you don't need that context. A desktop JavaFX application might be one of those times. These classes can even be used for representing time on a distributed system that has consistent time zones.

Creating Objects

All the core classes in the new API are constructed by fluent factory methods. When constructing a value by its constituent fields, the factory is called of; when converting from another type, the factory is called from. There are also parse methods that take strings as parameters. See **Listing 1**.

Standard Java getter conventions are used in order to obtain values from Java SE 8 classes, as shown in **Listing 2**.

NEW API

Java SE 8 will ship with a new date and time API in java.time that offers greatly improved safety and functionality for developers. The new API models the domain well, with a good selection of classes for modeling a wide variety of developer use cases.

You can also alter the object values in order to perform calculations. Because all core classes are immutable in the new API, these methods are called with and return new objects, rather than using setters (see **Listing 3**). There are also methods for calculations based on the different fields.

The new API also has the concept of an *adjuster*—a block of code that can be used to wrap up common processing logic. You can either write a WithAdjuster, which is used to set one or more fields, or a PlusAdjuster, which is used to add or subtract some fields. Value classes can also act as adjusters, in which case they update the values of the fields they represent. Built-in adjusters are defined by the new API, but you can write your own adjusters if you have specific business logic that you wish to reuse. See **Listing 4**.

Truncation

The new API supports different precision time points by offering types to represent a date, a time, and date with time, but obviously there are notions of precision that are more fine-grained than this.

The truncatedTo method exists to support such use cases, and it allows you to truncate a value to a field, as shown in **Listing 5**.

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

```
LocalDateTime timePoint = LocalDateTime.now(
    );    // The current date and time
LocalDate.of(2012, Month.DECEMBER, 12); // from values
LocalDate.ofEpochDay(150);  // middle of 1970
LocalTime.of(17, 18); // the train I took home today
LocalTime.parse("10:15:30"); // From a String
```

Download all listings in this issue as text

Time Zones

The local classes that we looked at previously abstract away the complexity introduced by time zones. A time zone is a set of rules, corresponding to a region in which the standard time is the same. There are about 40 of them. Time zones are defined by their offset from Coordinated Universal Time (UTC). They move roughly in sync, but by a specified difference.

Time zones can be referred to by two identifiers: abbreviated, for example, "PLT," and longer, for example, "Asia/Karachi." When designing your application, you should consider what scenarios are appropriate for using time zones and when offsets are appropriate.

- ZoneId is an identifier for a region (see **Listing 6**). Each ZoneId corresponds to some rules that define the time zone for that location. When designing your software, if you consider throwing around a string such as "PLT" or "Asia/Karachi," you should use this domain class instead. An example use case would be storing users' preferences for their time zone.
- ZoneOffset is the period of time representing a difference between Greenwich/UTC and a time zone. This can be resolved for a specific ZoneId at a specific moment in time, as shown in **Listing 7**.

Time Zone Classes

- ZonedDateTime is a date and time with a fully qualified time zone (see **Listing 8**). This can resolve an offset at any point in time. The rule of thumb is that if you want to represent a date and time without relying on the context of a specific server, you should use ZonedDateTime.

- OffsetDateTime is a date and time with a resolved offset. This is useful for serializing data into a database and also should be used as the serialization format for logging time stamps if you have servers in different time zones.
- OffsetTime is a time with a resolved offset, as shown in **Listing 9**.

There is an existing time zone class in Java—java.util.TimeZone—but it isn't used by Java SE 8 because all JSR 310 classes are immutable and time zone is mutable.

Periods

A Period represents a value such as "3 months and 1 day," which is a distance on the timeline. This is in contrast to the other classes we've looked at so far, which have been points on the timeline. See **Listing 10**.

ANSI SQL	JAVA SE 8
DATE	LOCALDATE
TIME	LOCALTIME
TIMESTAMP	LOCALDATETIME
TIME WITH TIMEZONE	OFFSETTIME
TIMESTAMP WITH TIMEZONE	OFFSETDATETIME

Table 1

Durations

A Duration is a distance on the timeline measured in terms of time, and it fulfills a similar purpose to Period, but with different precision, as shown in **Listing 11**.

It's possible to perform normal plus, minus, and "with" operations on a Duration instance and also to modify the value of a date or time using the Duration.

Chronologies

In order to support the needs of developers using non-ISO calendaring systems, Java SE 8 introduces the concept of a Chronology, which represents a calendaring system and acts as a factory for time points within the calendaring system. There are also interfaces that correspond to core time point classes, but are parameterized by Chronology:

- ChronoLocalDate
- ChronoLocalDateTime
- ChronoZonedDateTime

These classes are there purely for developers who are working on highly internationalized applications that need to take into account local calendaring systems, and they shouldn't be used by developers without these requirements.

LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11

```
ZoneOffset offset = ZoneOffset.of("+2:00");
```

Download all listings in this issue as text

Some calendaring systems don't even have a concept of a month or a week and calculations would need to be performed via the very generic field API.

The Rest of the API

Java SE 8 also has classes for some other common use cases. There is the MonthDay class, which contains a pair of Month and Day and is useful for representing birthdays. The YearMonth class covers the credit card start date and expiration date use cases and scenarios in which people have a date with no specified day.

JDBC in Java SE 8 will support these new types, but there will be no public JDBC API changes. The existing generic setObject and getObject methods will be sufficient.

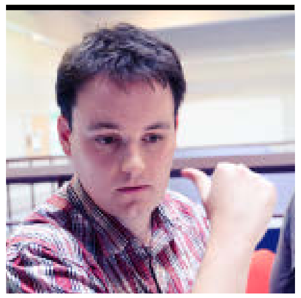
These types can be mapped to vendor-specific database types or ANSI SQL types; for example, the ANSI mapping looks like **Table 1**.

Conclusion

Java SE 8 will ship with a new date and time API in java.time that offers greatly improved safety and functionality for developers. The new API models the domain well, with a good selection of classes for modeling a wide variety of developer use cases. </article>

LEARN MORE

• JSR 310



JULIEN PONGE
BIO

Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM

Scenarios for using Oracle Nashorn as a command-line tool and as an embedded interpreter in Java applications

PHOTOGRAPH BY
MATT BOSTOCK/GETTY IMAGES

Until Java SE 7, JDKs shipped with a JavaScript scripting engine based on Mozilla Rhino. Java SE 8 will instead ship with a new engine called Oracle Nashorn, which is based on JSR 292 and invokedynamic. It provides better compliance with the ECMA normalized JavaScript specification and better runtime performance through invokedynamic-bound call sites.

This article is an introduction to using Oracle Nashorn in several ways. It covers using the standalone engine through the jjs command-line tool as well as using Oracle Nashorn as an embedded scripting engine inside Java applications. It shows the Java-to-JavaScript interoperability and how Java types can be implemented and extended from JavaScript, providing a seamless integration between the two languages.

The examples can be run using a recent JDK 8 early-access release. You can also use a custom build of OpenJDK 8. This is very simple to do thanks to the new OpenJDK build infrastructure (for example, sh configure && make images on a Mac OS X operating system with the XCode command-line tools installed).

It's Just JavaScript

A simple way to get started with Oracle Nashorn is to run JavaScript programs from the command line. To do so, builds of Oracle's JDK or OpenJDK include a command-line tool called jjs. It can be found in the bin/ folder of a JDK installation along with the well-known java, javac, or jar tools.

The jjs tool accepts a list of JavaScript source code files as arguments. Consider the following hello.js file:

```
var hello = function() {
  print("Hello Nashorn!");
};

hello();
```

Evaluating it is as simple as this:

```
$ jjs hello.js
Hello Nashorn!
$
```

Oracle Nashorn is an ECMA-compliant implementation of the language; hence, we can run more-elaborate snippets, such as the one shown in **Listing 1**, which prints a filtered list in which only the even numbers remain from the original list. It also prints the sum of those even numbers:

```
2,4,6,8,10
30
```

While Oracle Nashorn runs ECMA-compliant JavaScript, it is important to note that objects normally accessible in a web browser are not available, for example, console, window, and so on.

Scripting Extensions

If you run jjs -help to get a comprehensive list of the jjs command-line tool commands, you will notice a few interesting features:

- It can run scripts as JavaFX applications.

- JavaScript strict mode can be activated.
- Additional classpath elements can be specified for the Java Virtual Machine (JVM).
- An intriguing *scripting mode* can be enabled.

The scripting mode is interesting if you plan to take advantage of jjs to run system scripts written in JavaScript, just as you would do in Python, Ruby, or Bash. The scripting mode mainly consists of two language extensions: *heredocs* and *shell invocations*.

Heredocs. Heredocs are simply multiline strings, and they use a syntax that is familiar to Bash, Perl, and Ruby programmers (see **Listing 2**). Text begins with << followed by a special termination marker, which is EOF in our case. The formatting is left intact until the termination marker. Also, JavaScript expressions can be embedded in ${...} expressions. Running this program yields the output shown in **Listing 3**.

Note that in scripting mode, double-quoted strings can embed expressions that will be evaluated: "Hello ${name}" will evaluate against the value of name, while 'Hello ${name}' will not.

Shell invocations. Shell invocations allow the invocation of external programs in which the command is put between back-tick characters. Consider the following example:

```
var lines =
'ls -lsa'.split("\n");
for each (var line in lines) {
  print("|> " + line);
}
```

It runs the ls -lsa command. A shell invocation returns the standard console output as a string, which enables us to split lines and print them prepended with "|> ", as shown in **Listing 4**. If you need more-elaborate control over invoked processes, you should know that a $EXEC function exists, which provides access to the standard input, output, and error streams.

Other goodies. The scripting mode provides further goodies:

- The $ENV variable provides the shell environment variables.
- The $ARG variable is an array of the program command-line arguments.
- Comments can start with #, which is useful for making scripts executable on UNIX-like systems.
- exit(code) and quit() functions can terminate the current JVM process.

Consider the following executable .js file:

```
#!/usr/bin/env jjs -scripting
print(
"Arguments (${$ARG.length})");
for each (arg in $ARG) {
  print("- ${arg}")
}
```

We can make it executable and invoke it (arguments are passed after --), as shown in **Listing 5**.

Embedding Oracle Nashorn

The public API to embed Oracle Nashorn is simply javax.script. When Oracle Nashorn is available, its scripting engine is accessible through the nashorn identifier.

Listing 6 shows how you can access Oracle Nashorn from a Java application to define a sum function, call it, and then display the result.

The scripting engine object is the sole entry point to the Oracle Nashorn interpreter. It can be cast to the javax.script.Invocable interface, too:

```
Invocable invocable = (
Invocable) engine;
System.out.println(
invocable.invokeFunction(
"sum", 10, 2));
```

The Invocable interface also provides a method to convert evaluated code to a reference on a Java interface. Suppose that there exists some interface Adder as follows:

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

```
var data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var filtered = data.filter(function(i) {
  return i % 2 == 0;
});
print(filtered);

var sumOfFiltered = filtered.reduce(function(acc, next) {
  return acc + next;
}, 0);
print(sumOfFiltered);
```

Download all listings in this issue as text

```
public interface Adder {
 int sum(int a, int b);
}
```

The evaluated code defines a sum function with two arguments; hence, we can use it as an implementation as follows:

```
Adder adder =
invocable.getInterface(
 Adder.class);
System.out.println(
 adder.sum(2, 3));
```

This is a convenient way to extend Java types from JavaScript, but fortunately it's not the only one, as we will see in the next sections.

Not every JavaScript code is to be evaluated from a String: java.io.Reader; instances can be used, too, as shown in **Listing 7**.

You should consult the complete javax.script APIs for more details, including the information about the ability to define scopes and bindings of script engines.

mustache.js

Let's now call a real-world JavaScript library from a Java application. To do so, let's use the popular mustache.js template library, which is commonly used to render view fragments in HTML applications. Briefly, given a JSON data object {"name":"Bean"} and a template "Hello {{name}}", Mustache renders "Hello Bean". The template engine can do more than that, though, because it also supports conditions, collection iteration, and more.

Suppose that we downloaded mustache.js. **Listings 8a** and **8b** show our Java integration example.

After getting a scripting engine reference for Oracle Nashorn, we evaluate the mustache.js code. We then define the Mustache template as a String. The data model needs to be a JSON object. In our case, we first have to define it as a String and call JSON.parse to have it as a JSON object. We can then call Mustache.render. Running this program yields the following output, calling mustache.js for template rendering:

```
$ java sample2.Mustache
Email addresses of Mr A:
- contact@some.tld
- sales@some.tld

$
```

Java Seen from Oracle Nashorn

In most cases, calling Java APIs from Oracle Nashorn is straightforward, with the resulting code being Java written in JavaScript.

Basic example. We can call the System.currentTimeMillis() static method, as shown in **Listing 9**. And Java objects can be instantiated using the new operator:

LISTING 7 / LISTING 8a / LISTING 8b / LISTING 9

```
engine.eval(new FileReader("src/sample1/greeter.js"));
System.out.println(invocable.invokeFunction("greet", "Julien"));
```

Download all listings in this issue as text

```
var file =
new java.io.File("sample.js");
print(file.getAbsolutePath());
print(file.absolutePath);
```

Note that although java.io.File does not define an absolutePath method or public field, Oracle Nashorn inferred a property for it, so the expression file.absolutePath is equivalent to file.getAbsolutePath(). In fact, Oracle Nashorn treats the getXY() and setXY(value) methods as properties.

Dealing with arrays. The following snippet populates a queue as an instance of java.util.LinkedList:

```
var stack =
new java.util.LinkedList();
[1, 2, 3, 4].forEach(function(item) {
  stack.push(item);
});

print(stack);
print(stack.getClass());
```

This produces the following output, confirming that we are directly manipulating Java objects from JavaScript:

```
[4, 3, 2, 1]
class java.util.LinkedList
```

We can also take a tour through the new Java 8 stream APIs to sort the collection, although in this case, this is not the most efficient way to do so:

```
var sorted = stack
  .stream()
  .sorted()
  .toArray();
print(sorted);
```

This time, it prints something like [Ljava.lang.Object;@473b46c3, which indicates a Java native array. However, a Java array is not a JavaScript array. Internally, Oracle Nashorn provides JavaScript arrays using a custom class that also implements java.util.Map. Conversions can be performed using the to and from methods of the Oracle Nashorn–provided Java object:

```
var jsArray = Java.from(sorted);
print(jsArray);

var javaArray =
Java.to(jsArray);
print(javaArray);
```

which prints:

```
1,2,3,4
[Ljava.lang.Object;@23a5fd2
```

Imports. By default, all references to Java types need to be fully qualified (for example, java.lang.String, java.util.LinkedHashSet, and so on). Oracle Nashorn does not import the java package by default, because references to String or Object conflict with the corresponding types in JavaScript. Hence, a Java string is java.lang.String, not String.

Mozilla Rhino was the predecessor of Oracle Nashorn as the JavaScript engine implementation provided with Oracle's JDK releases. It featured a load(path) function to load a third-party JavaScript file. This is still present in Oracle Nashorn. You can use it to load a special compatibility module that provides importClass to import a class (like an explicit import in Java) and importPackage to import a package (like a wildcard import in Java):

```
load(
"nashorn:mozilla_compat.js");

importClass(java.util.HashSet);
var set = new HashSet();

importPackage(java.util);
var list = new ArrayList();
```

It is important to note that these functions import the symbolic references into the global scope of the JavaScript code being interpreted. While they are still supported for compatibility reasons, the use of mozilla_compat.js and importClass is discouraged. Instead, it is recommended that you use another function coming from the Mozilla Rhino heritage—JavaImporter—as shown in **Listing 10**.

LISTING 10

```
var CollectionsAndFiles = new JavaImporter(
  java.util,
  java.io,
  java.nio);

with (CollectionsAndFiles) {
 var files = new LinkedHashSet();
 files.add(new File("Plop"));
 files.add(new File("Foo"));
 files.add(new File("w00t.js"));
}
```

Download all listings in this issue as text

JavaImporter takes a variable number of arguments as Java packages, and the returned object can be used in a with statement whose scope includes the specified package imports. The global JavaScript scope is not affected, making JavaImporter a much better alternative to importClass and importPackage.

Overloaded methods. Java allows *method overloading*, that is, the definition within a single class of several methods that have the same names but different signatures. The java.io.PrintStream class is a good example, providing many print and println methods for objects, strings, arrays, and primitive types.

Oracle Nashorn properly selects the most suitable target at runtime on a per-invocation basis. This means that you should never have to worry about overloaded methods when dealing with Java APIs. Still, there is a way to precisely select the required target if you need to. This mainly occurs with ambiguous parameters when you are passing a function object in which different interface types are permitted by overloaded methods, such as with the submit methods of java.util.concurrent executors.

In the following code, the first call to println will select the println(String) overloaded method. The second call uses a JavaScript object property to access the println(Object) variant. The string to be passed provides a signature that Oracle Nashorn uses at resolution time. Note that as an exception, classes from the java package need not be qualified; hence, we can write println(Object) instead of the valid, but longer, println(java.lang.Object).

```
var stdout =
java.lang.System.out;
stdout.println("Hello");
stdout["println(Object)"](
"Hello");
```

Type objects. The Java.type function can be used to obtain references to precise Java types. These include not just objects but also primitive types and arrays:

```
var LinkedList = Java.type(
"java.util.LinkedList");
var primitiveInt = Java.type(
"int");
var arrayOfInts = Java.type(
"int[]");
```

The returned objects are an Oracle Nashorn–specific representation of mappings to Java types. It is important to note that they differ from instances of java.lang.Class. Type objects are useful as constructors and for instanceof-based comparisons. Let's look **Listing 11**.

It is possible to go back and forth between a type object and a Java class reference. The class property of type objects returns their java.lang.Class counterpart. Similarly, the static property is made available to java.lang.Class instances to get their corresponding type objects.

The code in **Listing 12** would print the following:

```
class java.util.LinkedList
[JavaClass java.util.LinkedList]
true
true
```

Extending Java Types

Oracle Nashorn provides simple mechanisms for extending Java types from JavaScript code. It is important to be able to provide interface implementations and concrete subclasses.

Implementing interfaces. Given a Java interface, a simple way to provide an implementation is to instantiate it, and pass its constructor function a JavaScript object in which the methods to be implemented are given as properties.

Listing 13 provides a concrete implementation of java.util.Iterator, giving implementations of the next and hasNext methods (the remove method is provided by a default method in Java 8). We can run it, and check that it works as expected (see **Listing 14**).

When interfaces consist of a single method, a function object can be directly given with no need to perform an explicit new operator call. The example in **Listing 15** illustrates this on collection streams.

Running the code in **Listing 15** prints the following:

```
>>> 2
>>> 4
>>> 6
>>> 8
```

Note that Oracle Nashorn also provides a language extension in the form of *Oracle Nashorn functions*, which provides an abridged syntax for small lambda functions. This works everywhere a single abstract-method type is expected

LISTING 11 / LISTING 12 / LISTING 13 / LISTING 14 / LISTING 15

```
var list = new LinkedList;
list.add(1);
list.add(2);
print(list);
print(list instanceof LinkedList);

var a = new arrayOfInts(3);
print(a.length);
print(a instanceof arrayOfInts);
```

Download all listings in this issue as text

from Java APIs, too. Therefore, we can rewrite the following code from **Listing 15**:

```
var odd = list.stream().filter(
  function(i) {
  return i % 2 == 0;
});
```

Like this:

```
var odd = list.stream().filter(
  function(i) i % 2 == 0);
```

This language extension is useful when dealing with the new Java SE 8 APIs that provide support for lambda expressions, because JavaScript functions can be used wherever a Java lambda is expected. Also, note that this shorter form is to be supported by JavaScript 1.8 engines.

The case of abstract classes is the same as interfaces: you provide a JavaScript object with the required method implementations to its constructor function. Or, directly pass a function when an instance of a single abstract-method class is required.

GET GOING
A simple way to get started with Oracle Nashorn is to run JavaScript programs from the command line.

Using instance-bound implementations. To extend concrete classes, you have to use the Java.extend function. It takes a type object as a first argument to denote the base class to be extended. If the parameter is an interface type, it assumes that the base class is java.lang.Object. Further types can be given as extra parameters to specify a set of implemented interfaces.

Consider the example shown in **Listing 16**. The Java.extend function returns a type object, also called an *extender*. It can be invoked to create concrete subclasses; in our case, instance is a subclass of java.lang.Object that implements the two interfaces java.lang.Comparable and java.io.Serializable. Implementations are passed to instances being created through a JavaScript object passed to the constructors.

Running the code in **Listing 16** yields the following console output:

```
true
true
-1
0
1
```

Using class-bound implementations. Instances created from the same extender type share the same class although their implementations differ on a per-instance basis (see **Listing 17**).

While this is fine in many cases, passing the implementation to each instance might not always be convenient. Indeed, there are cases where objects must be instantiated through some form of inversion-of-control mechanism, such as those found in dependency injection APIs. In such cases, the third-party APIs typically require a reference to the implementation class, which makes the previous extender mechanism unsuitable.

LISTING 16 LISTING 17

```
var ObjectType = Java.type("java.lang.Object");
var Comparable = Java.type("java.lang.Comparable");
var Serializable = Java.type("java.io.Serializable");

var MyExtender = Java.extend(
ObjectType, Comparable, Serializable);
var instance = new MyExtender({
  someInt: 0,
  compareTo: function(other) {
    var value = other["someInt"];
    if (value === undefined) {
      return 1;
    }
    if (this.someInt < value) {
      return -1;
    } else if (this.someInt == value) {
      return 0;
    } else {
      return 1;
    }
  }
});

print(instance instanceof Comparable);
print(instance instanceof Serializable);
print(instance.compareTo({ someInt: 10 }));
print(instance.compareTo({ someInt: 0 }));
print(instance.compareTo({ someInt: -10 }));
```

Download all listings in this issue as text

LISTING 18 LISTING 19

```
var Callable = Java.type("java.util.concurrent.Callable");

var FooCallable = Java.extend(Callable, {
 call: function() {
  return "Foo";
 }
});

var BarCallable = Java.extend(Callable, {
 call: function() {
  return "Bar";
 }
});

var foo = new FooCallable();
var bar = new BarCallable();

// 'false'
print(foo.getClass() === bar.getClass());

print(foo.call());
print(bar.call());
```

Download all listings in this issue as text

Fortunately, Java.extend allows implementations to be bound to a class definition rather than being specified for each instance. To do so, you simply need to pass an implementation JavaScript object as the last parameter.

Consider **Listing 18**, which defines two extender types: FooCallable and BarCallable. When creating instances foo and bar, there is no need to pass implementations. We can also check that instances do not have the same class definition. In fact, FooCallable.class or BarCallable.class can be passed to third-party Java APIs that need instances of java.lang.Class definitions.

Although not illustrated by this example, classes defined with class-bound implementations provide constructors inherited from their superclass. In this example, our objects implicitly extend java .lang.Object and implement java .util.concurrent.Callable; hence, the corresponding class definition simply has a public no-arguments constructor.

Using instance-bound and class-bound implementations. Last but not least, it is possible to combine both instance-bound and class-bound implementations. You can refine the class-bound implementation of all or some of the methods by passing an implementation object to its constructor, as shown in **Listing 19**.

Conclusion

This article covered various scenarios for using Oracle Nashorn as a command-line tool and as an embedded interpreter in Java applications. It also covered the interoperability between Java and JavaScript, including the ability to implement and extend Java types from JavaScript.

Oracle Nashorn is an excellent way to take advantage of a scripting language for polyglot applications on the JVM. JavaScript is a popular language, and the interaction between Java and JavaScript is both seamless and straightforward for a wide range of use cases. </article>

LEARN MORE

- OpenJDK 8
- Project Nashorn

JOHAN VOS

BIO

Part 2

Building Rich Client Applications with JSR 356: Java API for WebSocket

Create a client-side WebSocket application and integrate it with the JavaFX platform.

In a previous article, we explored JSR 356, the Java API for WebSocket. We created a simple chat server using GlassFish 4 and an HTML5 chat application that uses JavaScript to communicate with the WebSocket protocol inside the GlassFish application server.

As mentioned in the previous article, the Java API for WebSocket contains two parts: a client component and a server component. In the previous article, we examined the server component. In this article, we will see how you can leverage the client part of the Java API to create a standalone JavaFX application that uses WebSockets to communicate with the server component we introduced in the previous article.

The client component of the Java API for WebSocket is a subset of the server component. It contains the same functionality, except for the ServerEndpoint functionality that allows developers to register an endpoint that listens for incoming requests. As a consequence, there are no additional dependencies for the client component.

Because we want to visualize the chat messages, we need a user interface. The JavaFX platform provides a perfect framework for visualizing the chat information, and because it is pure Java, it integrates very well with the client API for WebSocket.

Scenario

We will implement the following scenario:

- An end user starts the JavaFX application.
- The application opens a WebSocket connection to the server and retrieves the list of active chatters.
- A login screen is shown.
- The end user enters his nickname.
- The nickname is sent to the server.
- The server sends a login message with the nickname to all active clients.
- The clients process the login message and add the new user to the list of active chatters.
- The end user creates a new chat message, which is then sent to the server.
- The server distributes the new chat message to all active clients.
- When the end user leaves the chat, the client WebSocket connection to the server is closed.
- The server detects the closed connection, and sends a logout message to the remaining clients.
- The clients process the logout message and remove that specific user from the list of active chatters.

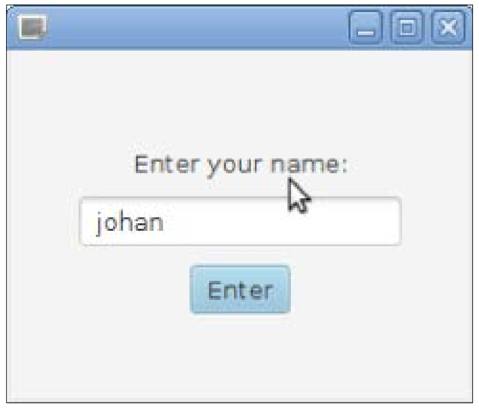


Figure 1

We have to create two views to support this scenario:

- A login screen, allowing the user to choose a nickname and to enter the chat (see **Figure 1**)
- A chat screen, showing the list of active users, the chat messages, and the input field for entering new chat messages (see **Figure 2**)

PHOTOGRAPH BY TON HENDRIKS

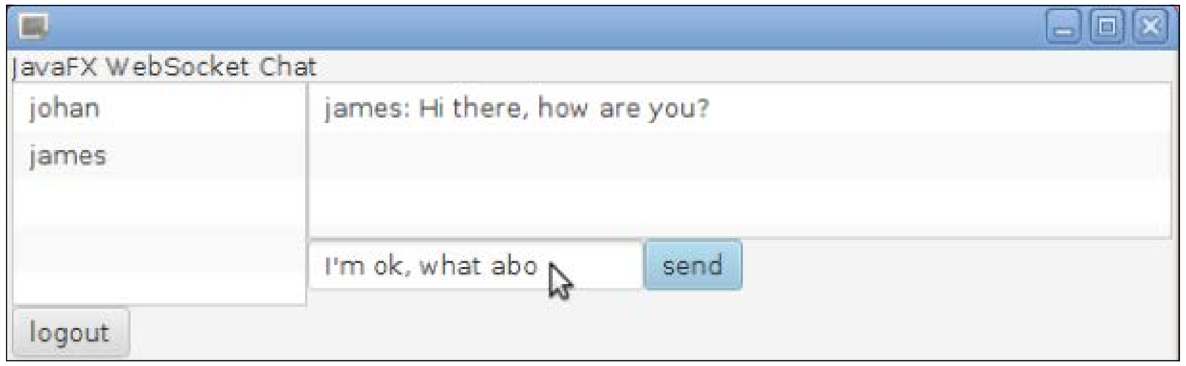


Figure 2

Once the user clicks the Enter button in the login screen, the chat screen is shown. If the user clicks the logout button in the chat screen, the login screen is shown.

Because we want to explore the concepts of the WebSocket client API, we will not pay attention to the graphical aspects of the application. However, it is very easy to customize the look and feel of the JavaFX application by using cascading style sheets (CSS).

Before we dive into the details of the WebSocket client API, we will explore some of the related JavaFX concepts we will use in this application.

The JavaFX Client Application

In many cases, it is good practice to have the user interface components in JavaFX be driven by the data components. Thanks to the concepts of *observables* and *properties*, JavaFX controls and components can react to changes in data structures.

In our application, we maintain a list of active users and a list of chat messages. Both lists are instances of ObservableList. They are visualized using instances of ListView and, as a consequence, the user interface component will change when the data in the ObservableList instance changes.

Because both UI components and the WebSocket code need to be able to interact with the data, we create a Model class and follow the singleton approach. There are a number of other ways to do this, and different developers might prefer different patterns. Discussing these patterns is outside the scope of this article, though.

Our Model class contains the ObservableList instances for maintaining the active chatters and the chat messages, as shown in **Listing 1**.

The list of chat messages contains a number of instances of ChatMessage. The ChatMessage class encapsulates the relevant information for a particular chat message:

```
public class ChatMessage {

  private String text;
  private String uid;
 ...
}
```

The user interface components that render this data are ListView instances, and they are created as part of the createChatPane() method in the main class App.java (see **Listing 2**).

By default, a ListView will render its content by showing the String representation of the contained objects, but this can be overridden using CellFactories. The default behavior is fine for rendering the names of the active chatters, but for rendering chat messages, we

LISTING 1 LISTING 2

```
public class Model {

  private static Model instance = new Model();

  private static ObservableList<String> chatters =
FXCollections.observableArrayList();
  private static ObservableList<ChatMessage> chatMessages =
FXCollections.observableArrayList();

  public static Model getInstance() {
    return instance;
  }

  public ObservableList<String> chatters() {
    return chatters;
  }

  public ObservableList<ChatMessage> chatMessages() {
    return chatMessages;
  }

}
```

Download all listings in this issue as text

want to show the author of the chat message and its content. In order to do so, we have to set a new CellFactory on the ListView instance as shown in **Listing 3**.

The code in **Listing 3** will make sure that instances of ChatListCell will be used for rendering the content of the list. The ChatListCell, which extends the javafx.control.cell.ListCell, does the real work. We override the updateItem method, which will be called when the content of the cell needs to be changed, as shown in **Listing 4**.

As you can easily detect, the name of the chatter (which is contained in the uid field on ChatMessage) is shown, followed by a colon and the text of the chat message.

WebSocket Client Interaction

Before we can send and receive messages, we need to establish a WebSocket connection to the server application.

In the start method of our JavaFX application, we call the createWebSocket method, which we implement as shown in **Listing 5**.

First, we need to obtain a WebSocketContainer. The Java API for WebSocket provides static methods for obtaining an instance of an implementation of the WebSocketContainer interface. Our sample application uses Tyrus, which is the Reference Implementation of JSR 356, but any other JSR 356–compliant implementation should work as well. The Tyrus client .jar file contains the required metainformation for informing the javax.websocket.ContainerProvider about the concrete implementation classes.

Once we have an instance of the WebSocketContainer, we can create a WebSocket connection (a session) to the server endpoint. There are two ways to create a session:

- Implementing the WebSocket lifecycle methods in a class that extends EndPoint
- Decorating a class with @ClientEndPoint and also decorating the relevant lifecycle methods with the correct annotations

Both approaches are valid, and it is up to the developer to make a choice between programmatic endpoints and annotated endpoints. In our sample application, we use the annotated endpoints strategy. A session is thus obtained, as shown in **Listing 6**.

The connectToServer method on the WebSocketContainer requires two parameters:

- The class of the annotated endpoint, which should be decorated with @ClientEndPoint
- A URI containing the address information for the remote endpoint; in our case, the URI is constructed using new URI ("ws://localhost:8080/chatserver/endpoint");

LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6 / LISTING 7

```
@Override
 public void updateItem(ChatMessage item, boolean isEmpty) {
   if (isEmpty || (item == null)) {
      return;
   }
   setText(item.getUid() + ": " + item.getText());
 }
```

Download all listings in this issue as text

The Client Endpoint

The lifecycle of the WebSocket is monitored by the ChatClientEndpoint class. This class contains a number of methods that are annotated with javax.websocket annotations (see **Listing 7**). The ChatClientEndpoint itself should be annotated with @ClientEndpoint.

The onConnect method is decorated with the @javax.websocket.OnOpen annotation, and will be called when the WebSocket connection is established. We do not need to perform any action when the connection is established; hence, this method is not doing anything useful except for printing an informative message on the console.

The onMessage method is decorated with the @javax.websocket.OnMessage annotation, and it will be called when a WebSocket message is received by this endpoint.

In the previous article, we saw how the server component translates a ChatCommand into a JSON string and sends it to the connected clients. We have two options for handling the incoming messages:

- We can use a Decoder for translating the incoming JSON text into a ChatCommand instance and accept ChatCommand messages as part of the onMessage method.

- We can process the incoming JSON text directly as a String instance in the onMessage method.

While developing the server component in the previous article, we explored the encoder/decoder approach; here, we will process the JSON content directly in the onMessage method.

There is no JSON standard defined in Java SE 7, but the Java EE 7 specifications do define a JSON API. We will use this API and, as a consequence, the demo application should work with any JSON parser that is compliant with the JSON API defined in Java EE 7. The Java EE 7 specification defines only the API, and an implementation is still needed at runtime. In our demo application, we use the Reference Implementation for the JSON API that is part of the GlassFish application server.

First of all, we need to determine what type of message we are receiving. That information is part of the JSON structure—in the node named "command"—and it is obtained as shown in **Listing 8**.

Our chat application contains four different commands:

- login: When this command is received, we are notified about a user who joined the chat.
- logout: When we receive a logout command, a user has left the chat screen.
- allusers: The allusers command is used when the server sends us a list of all active chatters.
- message: This command is received when the server sends us a chat message.

Depending on the command, different actions have to be performed. In the case of a login command, the code in **Listing 9** will be executed.

The name of the user that joined the chat is contained in the "uid" node. We will add this name to the list of chatters, which is maintained in the chatters ObservableList in the Model class. If we modify an ObservableList that is used by a component in the JavaFX SceneGraph (for example, a ListView), we need to make sure that this modification is done by the JavaFX application thread. This is achieved using the Platform .runLater() construction shown in **Listing 9**.

Note that adding the name of the new chatter is all we have to do in order to make the new chatter show in the user interface. This is because the JavaFX ListView showing all the active chatters is a JavaFX control that will be notified when changes in the underlying list occur.

When a logout command is received, we have to remove the user from the list of active chatters. The code required to do this is similar to the code required for adding a user, except that we have to remove the user from the list of chatters (see **Listing 10**).

When an allusers command is received, we first clear the list of active chatters, and then populate it with the names we receive in the "uids" nodes in the incoming JSON text, as shown in **Listing 11**.

Finally, when a message command is received, we construct a ChatMessage object and add that to the list of chat messages (see **Listing 12**).

Similar to the ListView rendering the active chatters, the ListView rendering the chat messages will automatically display new messages once they are added to the ObservableList that is used as the backing list for the chat messages.

LISTING 8 LISTING 9 LISTING 10 LISTING 11 LISTING 12

```
JsonReader reader = Json.createReader(
new ByteArrayInputStream(msg.getBytes()));
final JsonObject node = reader.readObject();
String command = node.getString("command");
```

Download all listings in this issue as text

Sending Messages

So far, we have seen how to process lifecycle events and messages originating from the server. We also need to implement the reverse scenario, where the client sends messages to the server. This is required in two cases:

- The user logs in, and we need to send a login message to the server containing the user's name.
- The user sends a message, and we need to send the user's name and the content of the message to the server.

Sending a message to a remote endpoint is possible once the session is established. We obtained a WebSocket session as a result of the container.connectToServer() method.

In our simple example, we will send only full text messages. The

LISTING 13 LISTING 14

```
RemoteEndpoint.Basic remoteEndpoint = session.getBasicRemote();
```

Download all listings in this issue as text

Java API for WebSocket is very powerful and can handle partial messages as well, and it can also deal with binary messages.

In order to send a text message, we need to obtain the remote endpoint from the session as shown in **Listing 13**. Then, sending text over the WebSocket to the server is simply done by using the following:

```
remoteEndpoint.sendText(...);
```

Because our server component expects messages in JSON format, we need to make sure we format our messages accordingly. Again, we use the Java EE 7–defined JSON processor for converting a Java object into a JSON message.

Sending a text message to the server can be done as shown in **Listing 14**.

Conclusion

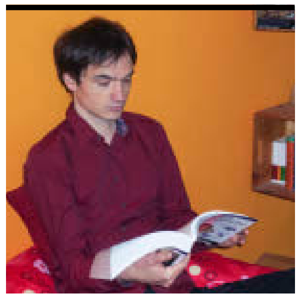
In this article, we saw how we can leverage the client part of the Java API for WebSocket for creating a client-side WebSocket application and connecting it to a server component that has a WebSocket endpoint.

We also saw how easily the client API can be integrated with the powerful JavaFX platform. In this example, we ignored the look and feel, but, it should be clear that combining JavaFX and the WebSocket client API produces a highly attractive, real-time Java client application. </article>

LEARN MORE

- "JSR 356, Java API for WebSocket"
- Johan Vos' blog
- Part 1 of "Building Rich Client Applications with JSR 356: Java API for WebSocket"

MAKE THE FUTURE JAVA
Java™
FIND YOUR JUG HERE
CEJUG is celebrating our 10-year anniversary in 2012! We follow Java technology with passion, share knowledge with pleasure, and create opportunities for students and professionals with ambition.
Hildeberto Mendonça
The Ceará Java User Group (CEJUG)
LEARN MORE
ORACLE



PETER KARICH

BIO

GraphHopper Maps: Fast Road Routing in 100-Percent Java

How to effectively manage gigabytes of OpenStreetMap data with Java

Big Data and Java

To make a computer understand the real world, using graphs as the data structures is often the simplest and most natural choice. In the case of road networks, *junctions* then are vertices connected by streets, which are the *edges*. **Figure 1**, which shows a map extract that has been converted into a graph, illustrates the modeling process.

A good road routing engine determines the best route from one vertex to another out of all possible combinations of streets. This set easily gets big for large distances, but the engine has to return an instant answer.

Several open source solutions in Java handle large graphs, including the popular graph databases Neo4j and OrientDB. Graph processing frameworks such as Apache Giraph and Gephi are also gaining more traction. Most of these systems already support spatial analysis, but none of them fulfills the special needs of a fast road routing engine, especially if you have to handle queries over large geographic areas or offline on mobile devices.

I wanted to create something similar to what the Apache Lucene search engine offers, but for road networks: a highly memory-efficient, scalable, and fast routing library in Java. This idea led me to create GraphHopper.

Why Java?

The most important benefit of using Java for GraphHopper was the development speed or time to market. Java compiles faster than C/C++ and has a larger open source community than C#. Java also offers advanced IDEs such as NetBeans, which has fast and precise autocompletion, easy-to-use debugging, compile on save, and unit testing. Unit testing is especially important if you evolve your API rapidly and don't want to start from scratch every time you encounter a major limitation. If you've ever tried to set up unit tests for C/C++, you appreciate the simplicity and speed of JUnit or TestNG.

But the benefits of working

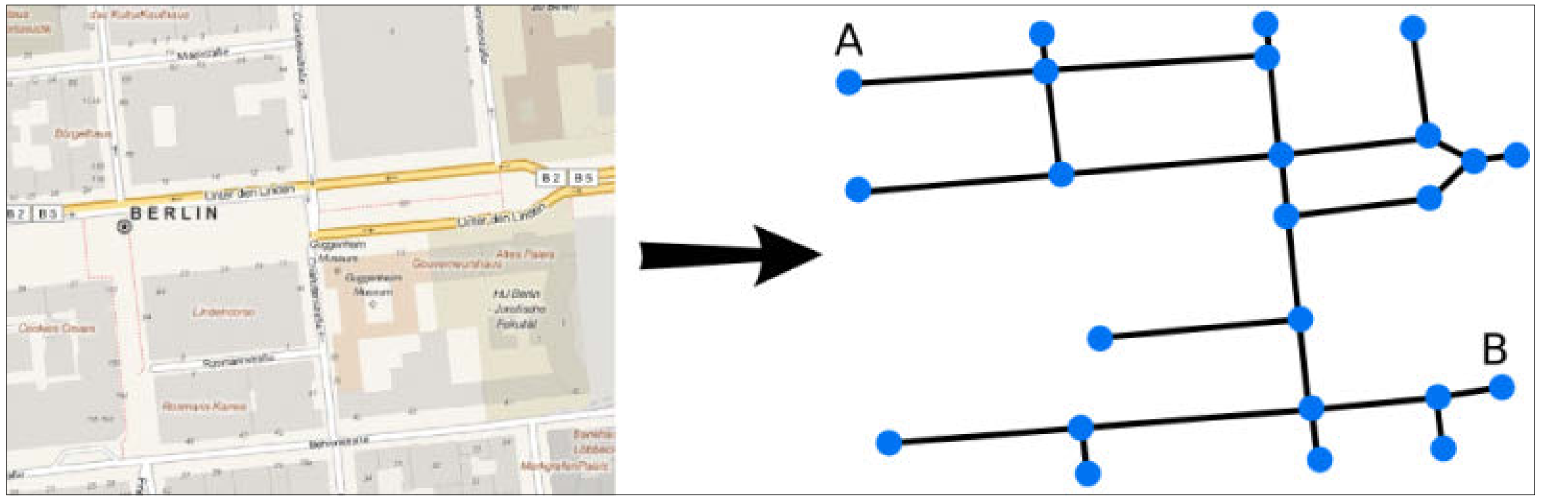


Figure 1

in Java don't end here. Java is standards-based, so you can speed project development by using tools such as Maven, which simplifies project setup for other developers who work on different systems or IDEs. And if you want to find bugs in software that manages big data, you sometimes have no other choice than to debug or profile the code directly on a remote machine, which is easily possible with NetBeans or the YourKit profiler.

Storage Comes First

The first component I needed was a storage solution, but none of the existing frameworks suited my needs. For example, Neo4j implements transactions, which can't be disabled and which create unnecessary overhead for a routing engine when you import large amounts of data or access a relatively static road network graph.

To illustrate the big data problem, let's look at the planet-wide OpenStreetMap data. As of November 2013 the compressed XML file is larger than 30 GB; it grows approximately 30 percent per year and contains more than *200 million streets*. GraphHopper can import that data in one hour with 20 GB of RAM and a commodity disc (and even these hardware requirements will be further optimized in the near future), but other storage solutions require days and multiple times more memory.

Routing in Java

An OpenStreetMap *ID* is a unique 64-bit number, and those numbers might have large gaps between them because of removal or update operations. Every object in OpenStreetMap has such an ID. A *node* is just a single geospatial point with associated GPS coordinates (latitude, longitude) and some optional attributes. A *way* can be used to model objects of the real world such as roads, houses, or areas. For routing with GraphHopper, we need only OpenStreetMap ways, which in GraphHopper are roads. From these we can calculate the connections (or edges) between the junctions, which are the internal nodes in our routing graph.

Graph Storage

Instead of a database table in which you query an index before getting relationships for a node (for example), with GraphHopper, I needed a data structure in which it's easy to jump from a node to all of the node's neighbors directly. It turns out that this data structure can be a simple array, in which an internal node ID is the array index (think of it as a pointer). For each node, the array stores a list of edges to neighboring nodes. Every edge contains properties such as speed and vehicle access.

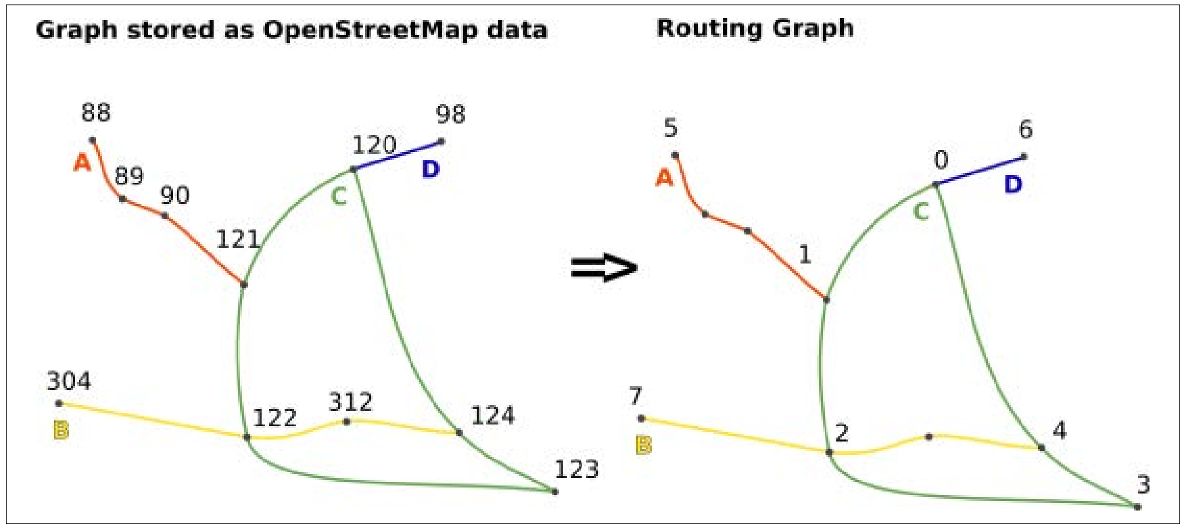


Figure 2

The OpenStreetMap node IDs aren't used as the internal node IDs; if they were, you'd have a very sparse array. Instead, the OpenStreetMap node IDs are used only during import time, and internal IDs are calculated from them. **Figure 2** shows the OpenStreetMap nodes and how they're used to create the routing graph with internal node IDs from 0 to 7.

However, a simple array limits parts of the graph to only 2 GB. A growth operation is too memory intensive, because you have to copy the existing data into the newly allocated space. That means you'd use at least twice the size of your already big array just to increase it by, say, 20 percent.

The solution is to create a *segmented version* of an array, which is basically a list of arrays to which you add a new array (or segment) when you want to expand it. You can access this list using a virtual index of type long, which is then propagated to the correct array and its index. The details are hidden behind the DataAccess API.

With that abstraction, adding implementations—such as one based on memory-mapped files using ByteBuffer—is easy. And you can access a graph that is hundreds of megabytes on devices with a small heap space.

The Utilities

This segmentation trick also helped in the import process, where I needed a HashMap to associate an OpenStreetMap node ID (type long) with an internal graph node ID (type int). Rehashing splits big rehash operations into more but much smaller operations—from gigabytes to megabytes. Although rehashing also increases the number of garbage collections, it reduces the pressure on memory resources.

A further step to reduce memory usage was to use primitive collections from Trove4j, because this library avoids autoboxing to store primitive types, such as int or long. It also can be faster than the standard Java collection library.

Road Routing in Java

Once the routing graph is available in memory or through a memory-mapped file, you can do routing using algorithms (Dijkstra or A*).

Dijkstra explores the nodes "in a circle" around your start, and it stops when the end is reached. The bidirectional version of Dijkstra improves the running time roughly by a factor of two, because it searches at the same time from the start and the end and stops when both circles "overlap."

However, the A* algorithm can be an improvement over Dijkstra if you can guide the search toward the end.

Memory-Efficient Traversal API

To explore the graph, the node- and edge-traversal must use CPU and memory resources efficiently. Another problem is how to make a nice but efficient API possible—that is, how can you iterate over all the edges of one node without allocating a new object for every edge? One solution is a stateful iterator using the flyweight pattern, which makes accessing edge properties, such as velocity or distance, very easy (see **Listing 1**).

LISTING 1

```
EdgeExplorer explorer = graph.
   createEdgeExplorer().
   setBaseNode(nodeX);
// now iterate through all
// edges of nodeX
while(iter.next()) {
  double distance =
      iter.distance();
  ...
}
```

Download all listings in this issue as text

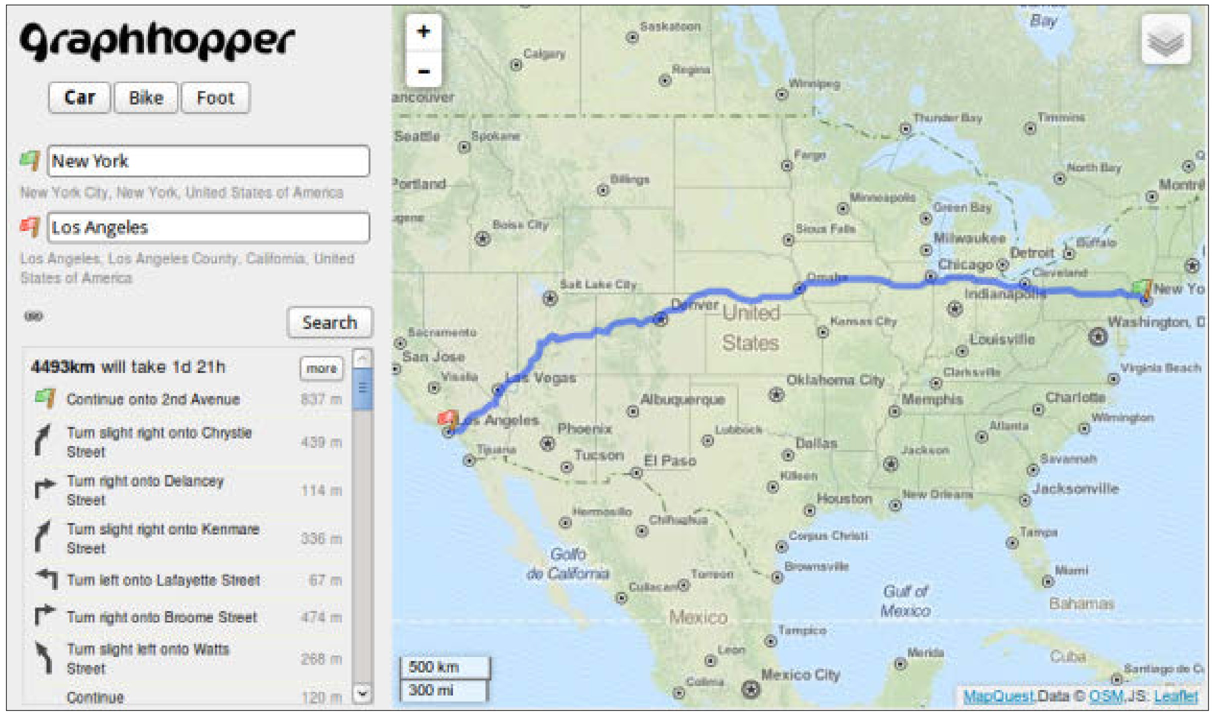
Contraction Hierarchies

You have two choices for making routing on road networks fast. You can use approximative algorithms, accept inaccurate results for some situations, and invest a lot of time tuning them. Or you can prepare the graph in a special manner.

I chose the second method. The preparation algorithm, Contraction Hierarchies, introduces additional shortcut edges to the graph, which makes it possible to ignore most of the other edges when doing a bidirectional Dijkstra. This technique makes routing up to 200 times faster. **Figure 3** shows a route that was calculated in 35 milliseconds, with 200 milliseconds of network latency.

Conclusion

In the field of routing engines—which have to be fast, memory friendly, and available for mobile devices—improving memory use must be done right from the beginning, because of the massive amount of data that's involved. Java lets you do this—and brings all the other advantages of Java to routing algorithm architects. </article>



New York
New York City, New York, United States of America
Los Angeles
Los Angeles, Los Angeles County, California, United States of America

4493km will take 1d 21h

Continue onto 2nd Avenue	837 m
Turn slight right onto Chrystie Street	439 m
Turn right onto Delancey Street	114 m
Turn slight right onto Kenmare Street	336 m
Turn left onto Lafayette Street	67 m
Turn right onto Broome Street	474 m
Turn slight left onto Watts Street	268 m
Continue	120 m

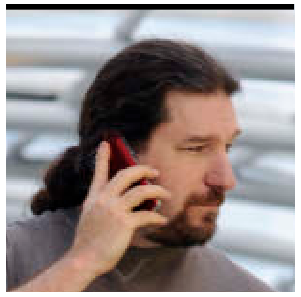
Figure 3

MORE ON TOPIC:

Big Data and Java

LEARN MORE

- GraphHopper developer resources



TED NEWARD
BIO

PHOTOGRAPH BY
PHIL SALTONSTALL

Part 1

Take Time to Play

Learn how the Play Framework simplifies life for server-side Web developers.

In many ways, servlets were the start of the modern enterprise Java stack. From servlets, we went to JavaServer Pages (JSPs) and "Model 2" designs, and from there came Enterprise JavaBeans (EJBs) and transactional processing. Then, in response to increased complexity, came "lightweight" containers such as Spring, and before long, Spring itself was really complicated, particularly when combined with its buddy Hibernate.

As a result, many Java developers miss the old days when there were (just) servlets.

Enter Play

As the movement in the web world toward simpler server-side infrastructure took deeper root, a group of developers in the Scala world decided to build an HTTP stack for Scala and Java developers that encompassed all the core elements that a Web 2.0 application would want. It came to be known as the Play Framework, and it specifically advertises itself as a "lightweight, stateless, web-friendly architecture."

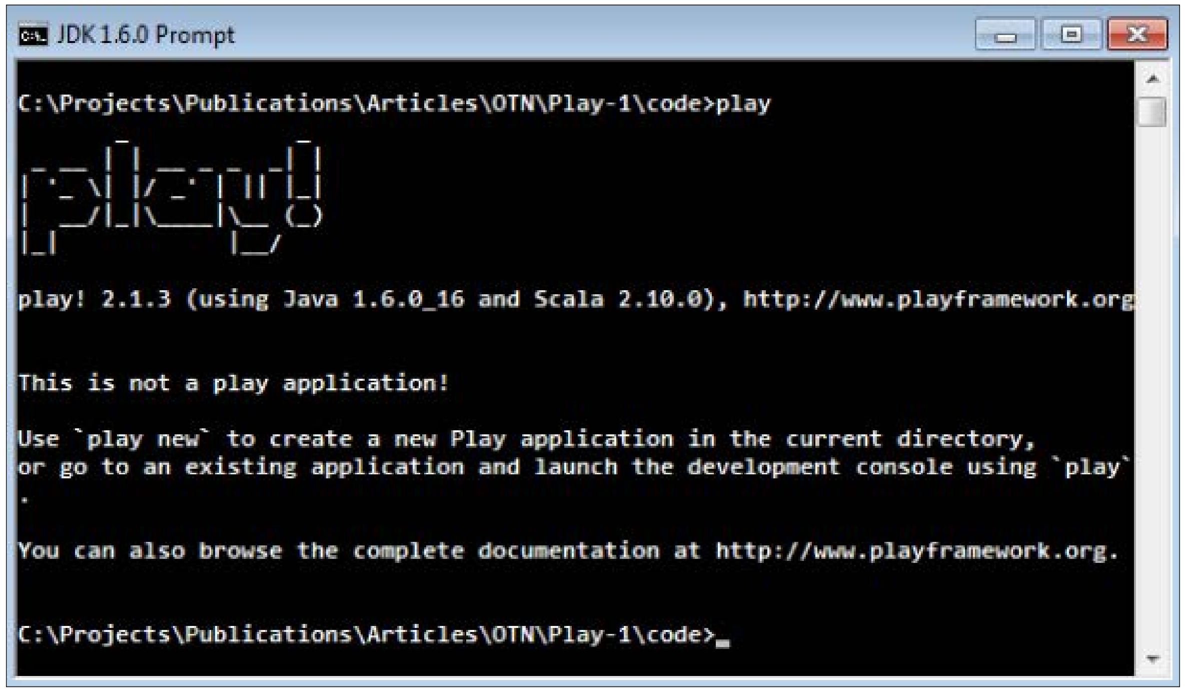
It's built on top of Scala (so it incorporates features from Scala such as type safety and functional approaches), Akka (so it's able to scale and parallelize well), and the Java Virtual Machine (JVM; so it's on top of the runtime you know and love). Plus, it's got some other goodness baked in that makes it worth a look as a replacement for the traditional Java EE app server and stack for your next JVM-based server-side application host.

It's hosted at playframework.com, but there are two different ways to get started with Play:

- Doing a "self-installation" by downloading a bundle from the Play website
- Using the Typesafe Activator

Doing a self-installation. One easy way to install Play is to download the Play .zip bundle from the website and expand it to a known location on your website. Then, assuming you already have Java installed on your machine and in your PATH, in a command-line window, add the /bin folder of the Play installation to your PATH and type play. This brings up a command-line console for interacting with the Play environment, which will be a core way in which developers interact with the server and environment.

Using the Typesafe Activator. Typesafe, the company that



JDK 1.6.0 Prompt

```
C:\Projects\Publications\Articles\OTN\Play-1\code>play
       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.1.3 (using Java 1.6.0_16 and Scala 2.10.0), http://www.playframework.org

This is not a play application!

Use `play new` to create a new Play application in the current directory,
or go to an existing application and launch the development console using `play`
.

You can also browse the complete documentation at http://www.playframework.org.

C:\Projects\Publications\Articles\OTN\Play-1\code>_
```

Figure 1

backs Akka and Play, has created a downloadable component it calls the Typesafe Activator, which is something of a lightweight integrated development environment (IDE), module repository, build system, and installer all rolled into one.

From the Typesafe website, you can download the Typesafe Activator, expand it onto your machine, launch the batch or shell script file, and use that to create a Play (or Akka or Scala) application from templates stored on the Typesafe website. It's a full IDE, too, so developers looking for an ultra-lightweight experience can avoid downloading Eclipse, IDEA, NetBeans, or any of the thousands of different text editors out there.

For the purposes of this article, because it's a bit lower-level (and, thus, it's a bit easier to see the various moving parts that are involved), I'm going to use the first approach, but anyone working from the Typesafe Activator will find it easy enough to follow along as well.

Getting Started

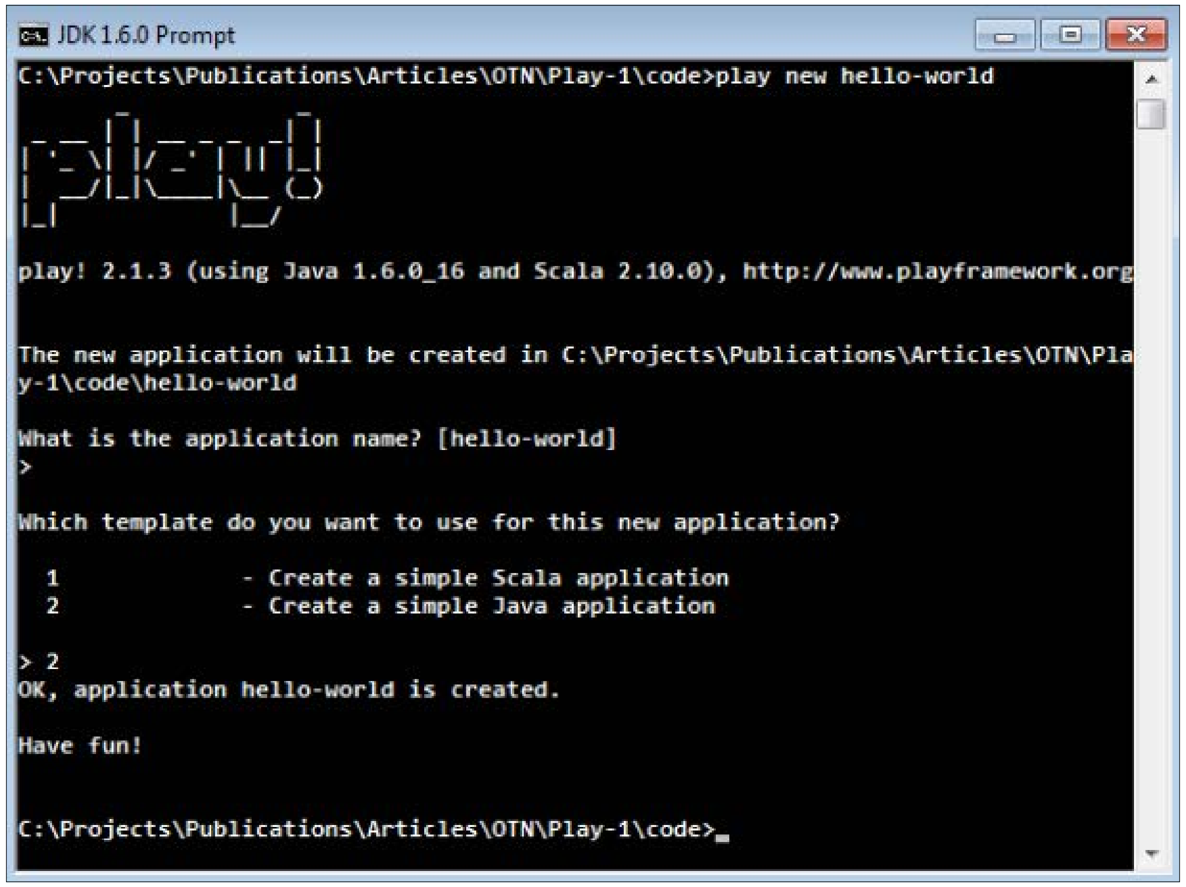
Assuming that Java is in your PATH and the root of the Play installation is also in your PATH, kicking off the console is as easy as typing play at the command prompt. Depending on whether this is the first time the console has been launched on this machine, it might fetch some additional bits from the internet, but eventually it will present you with this console response shown in **Figure 1**.

As you can see, play was ready to take some kind of action if the contents of this directory held a working application codebase—and, helpfully, it tells us that to create a new Play application is as simple as kicking off play new. So let's do that.

The first application one must build whenever coming to a new platform or language is always Hello World, so let's take a crack at building a simple Hello World application, which means we want to type play new hello-world at the command line, as shown in **Figure 2**.

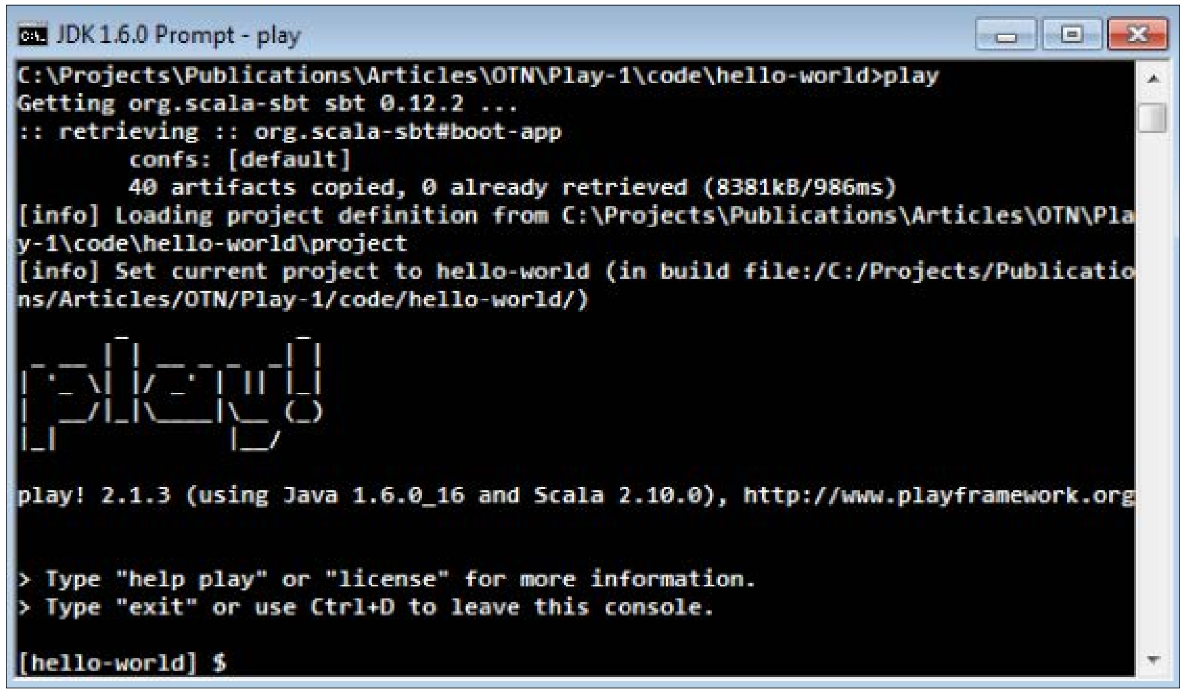
Notice that the console prompted twice: once for the name of the application (as opposed to the name of the directory we wanted to create—here, I just went with the default) and then for which of the two supported languages (Scala or Java) we wanted to use. For this first example, I chose Java, just to prove that the framework supports either language as a first-class development option. (In other words, despite being written in Scala, Play doesn't force you to learn Scala.)

Once it's finished with those two prompts, the console has scaffolded out some components, and the only thing left to do is to kick it off and see what's there. Doing



```
JDK 1.6.0 Prompt
C:\Projects\Publications\Articles\OTN\Play-1\code>play new hello-world
       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.1.3 (using Java 1.6.0_16 and Scala 2.10.0), http://www.playframework.org

The new application will be created in C:\Projects\Publications\Articles\OTN\Pla
y-1\code\hello-world

What is the application name? [hello-world]
>

Which template do you want to use for this new application?

  1             - Create a simple Scala application
  2             - Create a simple Java application

> 2
OK, application hello-world is created.

Have fun!

C:\Projects\Publications\Articles\OTN\Play-1\code>_
```

Figure 2



```
JDK 1.6.0 Prompt - play
C:\Projects\Publications\Articles\OTN\Play-1\code\hello-world>play
Getting org.scala-sbt sbt 0.12.2 ...
:: retrieving :: org.scala-sbt#boot-app
        confs: [default]
        40 artifacts copied, 0 already retrieved (8381kB/986ms)
[info] Loading project definition from C:\Projects\Publications\Articles\OTN\Pla
y-1\code\hello-world\project
[info] Set current project to hello-world (in build file:/C:/Projects/Publicatio
ns/Articles/OTN/Play-1/code/hello-world/)
       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.1.3 (using Java 1.6.0_16 and Scala 2.10.0), http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[hello-world] $
```

Figure 3

so is a simple matter of using cd to change to the generated hello-world directory and kicking off the console again by entering play, as shown in **Figure 3**.

As you can see, the console is now active, and to support running and building the application, it fetched some additional bits from the internet. The console, it turns out, is resting on top of sbt, the Scala build tool, which is popular in the Scala world. While developers don't need to learn sbt in depth, a passing familiarity with some kind of build system will be necessary to understand how to use sbt.

To get the console to start up an HTTP server listening for incoming requests, we enter run (or, if you don't want to take these two steps individually, you can enter play run at the command line). Again, if the necessary bits to compile and run the application aren't on your machine yet, the console will fetch them from the appropriate corners of the internet. Eventually, the console will display "play – Listening for HTTP on /0.0.0.0:9000," which is its way of telling you to open a browser, browse to http://localhost:9000, and see what's there.

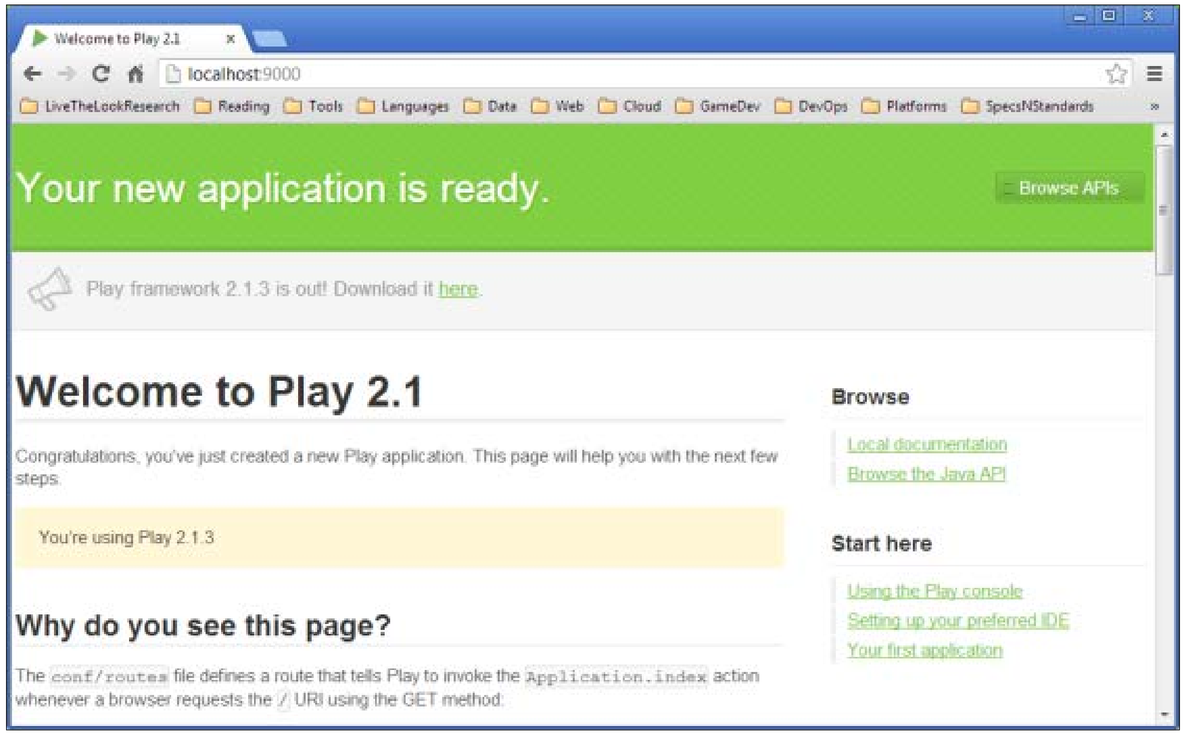
By the way, the first request that comes in will tell the server to compile the various files that make up the application (Play supports hot-reloading, just as most modern web-server stacks do these days), so the first request might take a few minutes. This act is pretty visible in the window that's hosting the server—it'll say "Compiling 4 Scala sources and 2 Java sources to . . ." as a way of letting you know what's going on.

Then you'll see a screen similar to **Figure 4**. Success! As you can see, the "Hello World" Play application is actually a pretty content-rich experience, designed to help developers get started with learning the environment.

Play Concepts

Like many (if not all) of the server-side web frameworks that emerged in the post-Rails era, Play is built around a model-view-controller (MVC) approach: *models* represent the data and entities that are being manipulated and used as part of the application; *views* display the models (or the parts of them that are relevant); and *controllers* do the necessary work on the models, as well as providing other logic (input validation, stores to and fetches from the database, and so on). If this is your first experience with MVC, numerous other resources on the web describe the MVC paradigm in greater detail.

Thus, one of the first things a new Play developer will want to do is find where these constructs


Welcome to Play 2.1

localhost:9000

LiveTheLookResearch · Reading · Tools · Languages · Data · Web · Cloud · GameDev · DevOps · Platforms · SpecsNStandards

Your new application is ready.

Browse APIs

Play framework 2.1.3 is out! Download it here.

Welcome to Play 2.1

Congratulations, you've just created a new Play application. This page will help you with the next few steps.

You're using Play 2.1.3

Why do you see this page?

The `conf/routes` file defines a route that tells Play to invoke the `Application.index` action whenever a browser requests the `/` URI using the GET method.

Browse
- Local documentation
- Browse the Java API

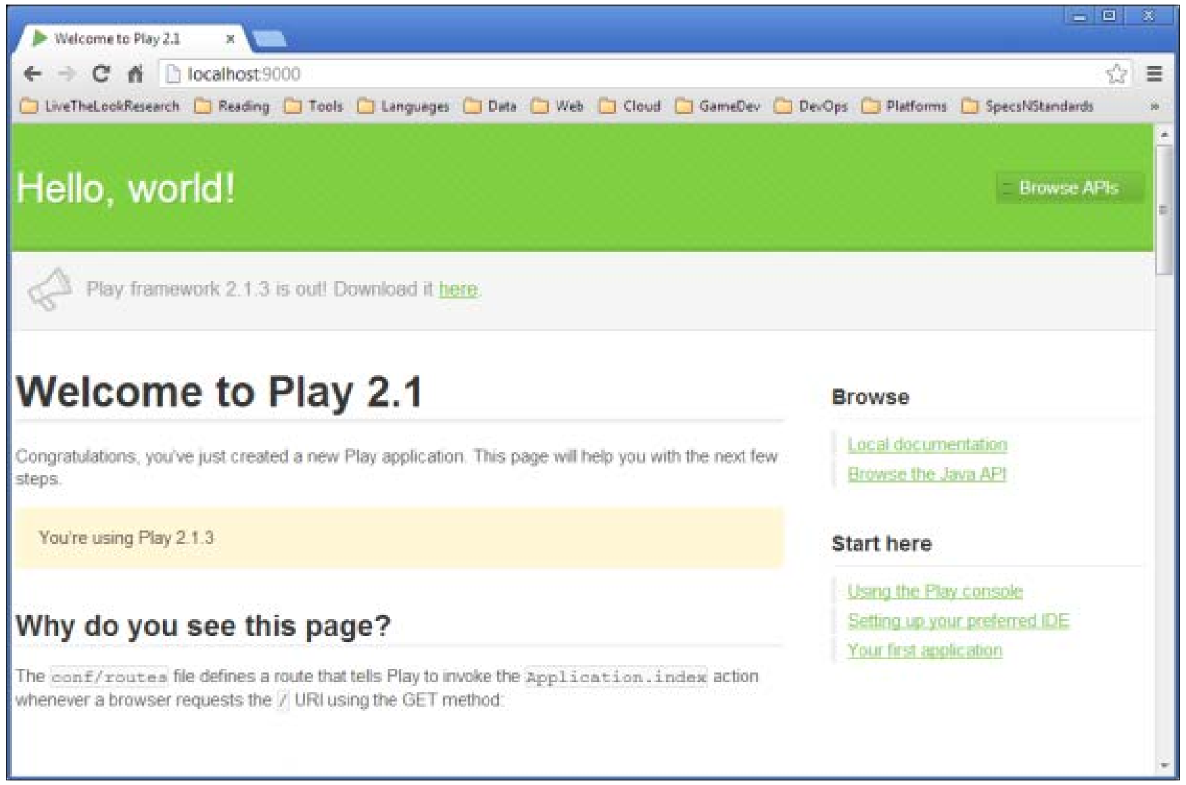
Start here
- Using the Play console
- Setting up your preferred IDE
- Your first application

Figure 4

live. In the generated hello-world directory, the framework created a number of subdirectories (including a generated .gitignore file for those of you who store your code in Git repositories). Most of these are pretty self-explanatory: public is for browser-accessible static resources such as JavaScript files, images, cascading style sheets (CSS), and so on; conf contains configuration files for the application; logs contains diagnostic logs; and so on.

In addition, application code resides inside the app subdirectory. It's important to realize that because this is the "root" of the code, this is where package names are rooted. When we look at the generated Java code, we'll see that the controllers are in a controllers package, because they're in a subdirectory called controllers inside of app, and so on. What this also implies is that if you don't like this particular naming scheme—perhaps you're a fan of the com.companyname.departmentname.applicationname.developername package-prefixing scheme that was all the rage back in 1997—you can use that scheme here as well, as long as you fix up the imports and package declaration names in the generated files.

Because this application is simple enough to not have any models, the console didn't feel the need to create a models directory, but controllers and views are there,



Hello, world!

Play framework 2.1.3 is out! Download it here

Welcome to Play 2.1

Congratulations, you've just created a new Play application. This page will help you with the next few steps.

You're using Play 2.1.3

Why do you see this page?

The `conf/routes` file defines a route that tells Play to invoke the `Application.index` action whenever a browser requests the `/` URI using the GET method.

Browse

Local documentation
Browse the Java API

Start here

Using the Play console
Setting up your preferred IDE
Your first application

Figure 5

LISTING 1

```
package controllers;

import play.*;
import play.mvc.*;

import views.html.*;

public class Application extends Controller {

  public static Result index() {
    return ok(index.render("Your new application is ready."));
  }

}
```

Download all listings in this issue as text

and inside of each we find a few generated files.

Inside of controllers lies the generated controller for the home page, called Application.java. (The names of the source files and their classes aren't set in stone, as we'll see in a bit.) Opening it up, the generated Java class looks like **Listing 1**.

As you can see, this is a pretty bare-bones bit of code. A controller class must extend the base Controller class, but beyond that, it's just a set of methods. Given that there's really zero work that this controller needs to do (there's no input validation, no database data retrieval or storage, and no business logic to execute), this is a good thing. In fact, all it really needs to do is tell the framework which view to display. It does so by asking the index class (which it imported from the views.html package, which corresponds to the index.scala.html file in the views subdirectory under app) to render itself with the passed String "Your new application is ready." And, yes, if we change that String constant to read "Hello, world!" instead, we get a slightly different response (see **Figure 5**).

We didn't even have to do any manual recompilation of the Java files—the running server took care of that. The corresponding view file, by the way, makes it fairly clear where that String constant goes, and how the remainder of the content gets generated:

```
@(message: String)

@main("Welcome to Play 2.1") {

  @play20.welcome(
 message, style = "Java")

}
```

We'll get into the syntax of the views in a bit, but even without detailed explanation, it's fairly obvious that there's an incoming "message" that's getting passed into a pre-existing component called play20.welcome, which is generating the rest of the content. If you like, replace it with the following if you prefer a more minimalist hello-world app:

```
@(message: String)

<h1>@message</h1>
```

Wait a Minute

Readers who've worked with servlets or just about any other JVM-based web framework are immediately going to jump on several things I seem to have overlooked. **How did we end up here?** In traditional Java EE applications, developers had to create an explicit URL-to-servlet mapping that told the servlet container how to find which servlet (out of potentially hundreds in a medium-sized application) to execute in response to the URL sent to the server. In a Play application, this routing information is described in the routes file in the conf directory, and the one generated for us looks like **Listing 2**.

The syntax here is fairly straightforward to understand: taking each of these lines in a top-down fashion, the first line that matches the HTTP verb and URL pattern given to the server invokes the method (called a *controller action* or *controller action method*) on the right-hand side. So, the http://localhost:9000/ request is a GET request (because it's not coming from a form POST), which maps to the index() method of the controllers.Application class, which then calls the render method on the index view.

Note that the framework considers any typos or other syntax violations to be compilation errors, so it's safe to assume that the file is (to a degree) compiled along with everything else. Note also that the controller-mapping part of the routing can include parameters to the controller, as demonstrated by the Assets .at() call in **Listing 2**.

Static. Controller action methods, such as index in **Listing 2** and, in fact, most methods written in Play, are always static, because the creators of the framework wanted to emphasize that HTTP and the web are intrinsically stateless. In other words, rather than wrestle with the "instance data versus session data versus singleton versus Strategy versus . . ." debate that seemed to plague most servlet frameworks and applications, methods are (nearly) always static.

This means that there is no temptation to associate a controller object instance with a user's session, no need to "pool" controller instances, no need to wonder about how and where the threads are going to interact with the controller instances, and so on. If everything is a static, not only does this fit better with the functional style that Scala brings to the JVM, but it also helps developers conceptualize more effectively how the web works.

NOT REQUIRED
Despite being written in Scala, **Play doesn't force you to learn Scala.**

Now before panic sets in, know that the framework (like every other web framework in the world) provides an abstraction around the user session, usually tied to a cookie stored in the user's browser, that can be used to store objects across requests. Play also provides another scope, called the *flash scope*, that is stored for only the next request received from that user on this server, which makes it primarily useful for Ajax-based responses.

However, with the good news comes the bad news: unlike other frameworks, Play doesn't store session data on the server—instead, it's stored in the cookie in the user's browser, which means that session data is limited in size (to 4 KB, the maximum size of a cookie) and in type (to strings only).

Obviously, it would be fairly trivial for a developer to use the session space to store a key in server-stored data (in a database, perhaps) that could store any kind of data, but this is not something that the framework offers "out of the box," again, largely because this kind of data reduces the inherent scalability of HTTP, your application, and the web as a whole. (By and large, the web world is moving toward a model of RESTful interaction with the server. REST doesn't use cookies to manage state, and Play pretty unashamedly wants to adhere to REST's principles.)

Session scope is available through the session() method, which returns an Http.Session object, which (as is fairly common) derives from java.util .HashMap<String,String>, making its use pretty obvious. Flash scope is similarly obtained—via the flash() method.

The ok method. At the end of the controller's execution, the controller needs to signal back to the browser one of the predefined HTTP status codes: 200 for "OK," 400 for "client error," and so on. The framework abstracts this down

LISTING 2

```
# Home page
GET    /              controllers.Application.index()

# Map static resources from the /public folder
 to the /assets URL path
GET    /assets/*file   controllers.Assets.at(
 path="/public", file)
```

Download all listings in this issue as text

into a fairly simple system: the ok method sends back a 200 and other methods send back different responses, most of which are pretty self-explanatory. For example, the forbidden method sends back 403, unauthorized sends back 401, redirect and seeOther send back 303, notFound sends back 404, and so on. (See the HTTP specification or the framework Javadocs for the exact mappings of method to status code and the exact meaning of each status code.)

However, because many browsers still display content that's returned by a non-200 status code, these methods will accept content in a variety of forms (String, InputStream, and so on) that will be echoed back down the pipe to the browser and, hence, the common trick of passing the results of calling the render() method on the view as the parameter to the response method.

Method on the view. Views are compiled into Scala classes that expose a single method, render(), but depending on what is declared at the top of the view, the method can take additional parameters, such as the message parameter that was declared in index.scala.html. Unlike some of the other web frameworks out there, though, Play embraces the strongly type-safe nature of the Scala language, so the parameters are verified throughout the application's code. If, for some reason, a developer came around later and tried to convince the view that message was an Int, the passed parameter from the application controller isn't going to match up, and the code won't compile.

EMBRACING TYPE-SAFE

Unlike some of the other web frameworks out there, **Play embraces the strongly type-safe nature of the Scala language,** so parameters are verified throughout the application's code.

Scala makes a big deal about type safety, more so in many respects than Java does, and the developers of Play are fully on board with that approach. As a result, a developer using this framework can expect a lot more in the way of compilation errors during development than other frameworks (Grails, Rails, Node, and so on) will generate. The tradeoff, of course, is that the compiler will catch easy type-mismatch errors without requiring explicit unit tests or integration tests written by the developer.

Tests. In recent years (since the "Test Infected" article published in 2001, anyway) there has been widespread adoption of developer test code in a variety of forms, and Play supports (if not outright encourages) this.

One of those generated directories is the test directory, and within it we find two different test files: one called ApplicationTest, which contains a pregenerated test method to verify that the index.scala.html view renders correctly, and another called IntegrationTest, which contains pregenerated code to do a more end-to-end test, essentially starting up a mock database and faked application, issuing an HTTP request, and examining the returned page source for the desired text. Best of all, executing the tests consists only of typing test at the console. (To stop the console from running the HTTP server, just press Ctrl-D in that terminal window, and control will return to the console shell.)

Conclusion

The Play Framework simplifies many things in the server-side web development world. It brings a new angle to the world of website development: using statically typed, compiler-verified practices to ensure code correctness, but without losing the productivity and code terseness that we find in major dynamic-language-based frameworks such as Ruby on Rails, Grails, and Node.js. Note that static compilation won't obviate the need for unit tests, but it certainly will reduce the number of tests that need to be written, because now, rather than being consigned to the dustbin of history, the compiler is an active participant and validator in the development cycle, and can leverage its considerable knowledge of type systems and correctness to make sure that you, the developer, aren't doing something really stupid. Which, frankly, I think is a good thing.

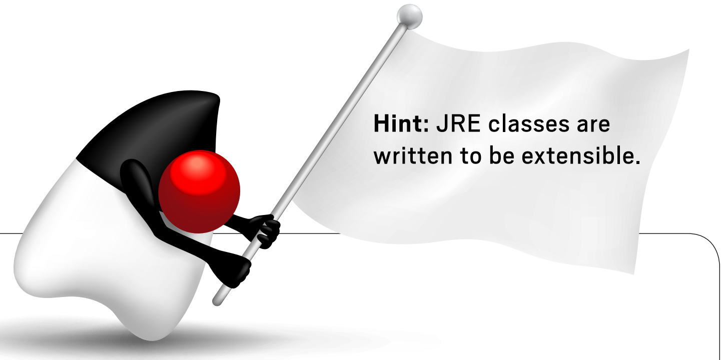
Be sure to check out the documentation on playframework.com. (Also, once your application is running locally, point your browser to http://localhost:9000/@documentation to work with a local copy of the same documents.)

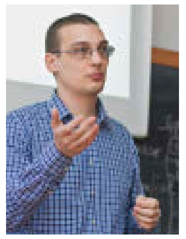
In Part 2 we will examine how Play supports the construction of modern REST-only server applications, such as what might be hiding behind a mobile application or a client-side JavaScript framework. Stay tuned, and as the Play console suggests, have fun! `</article>`

LEARN MORE

• Play

//fix this /





In the November/December 2013 issue, Oracle ACE and NuBean Consultant Deepak Vohra presented us with a generics code challenge. He gave us a class Sample and asked what needed to be fixed. The correct answer is #4: Remove one of the first two methods and add the type parameters K and V to the third method.

Method setArrayList(ArrayList<T> arrayList, T t) has the same erasure, setArrayList(ArrayList, Object), as another method setArrayList(ArrayList arrayList, Object obj) in type Sample. You can't have two methods with the same erasure in the same class. Also K and V cannot be resolved to a type.

This issue's challenge comes from Attila Balazs, a polyglot developer from Cluj-Napoca, Romania, who presents us with an error-logging challenge.

1 THE PROBLEM

A developer tries to ensure that exceptions thrown while executing runnables in a ThreadPoolExecutor are properly logged.

2 THE CODE

He comes up with the following code, which requires all callsites that submit runnables to be modified. Is there a simpler solution?

```
final Future<?> runnableSubmission = es.submit(new Runnable() {
    @Override
    public void run() {
        int computed = 1 / 0;
    }
});

es.submit(new Runnable() {
    @Override
    public void run() {
        try {
            runnableSubmission.get();
        } catch (InterruptedException | ExecutionException e) {
            LOG.log(Level.SEVERE, "Exception while executing
task!", e);
        }
    }
});
```

3 WHAT'S THE FIX?

1) Add a ThreadFactory to the ThreadPoolExecutor, which sets an UncaughtExceptionHandler on the created thread.

2) Extend ThreadPoolExecutor and override the afterExecute method.

3) Extend ThreadPoolExecutor and override the Future<?> submit(Runnable task) method to add the checking task after each runnable.

4) Extend ThreadPoolExecutor and override all three variations of submit declared in the ExecutorService interface.

GOT THE ANSWER?

Look for the answer in the next issue. Or submit your own code challenge!

ART BY I-HUA CHEN