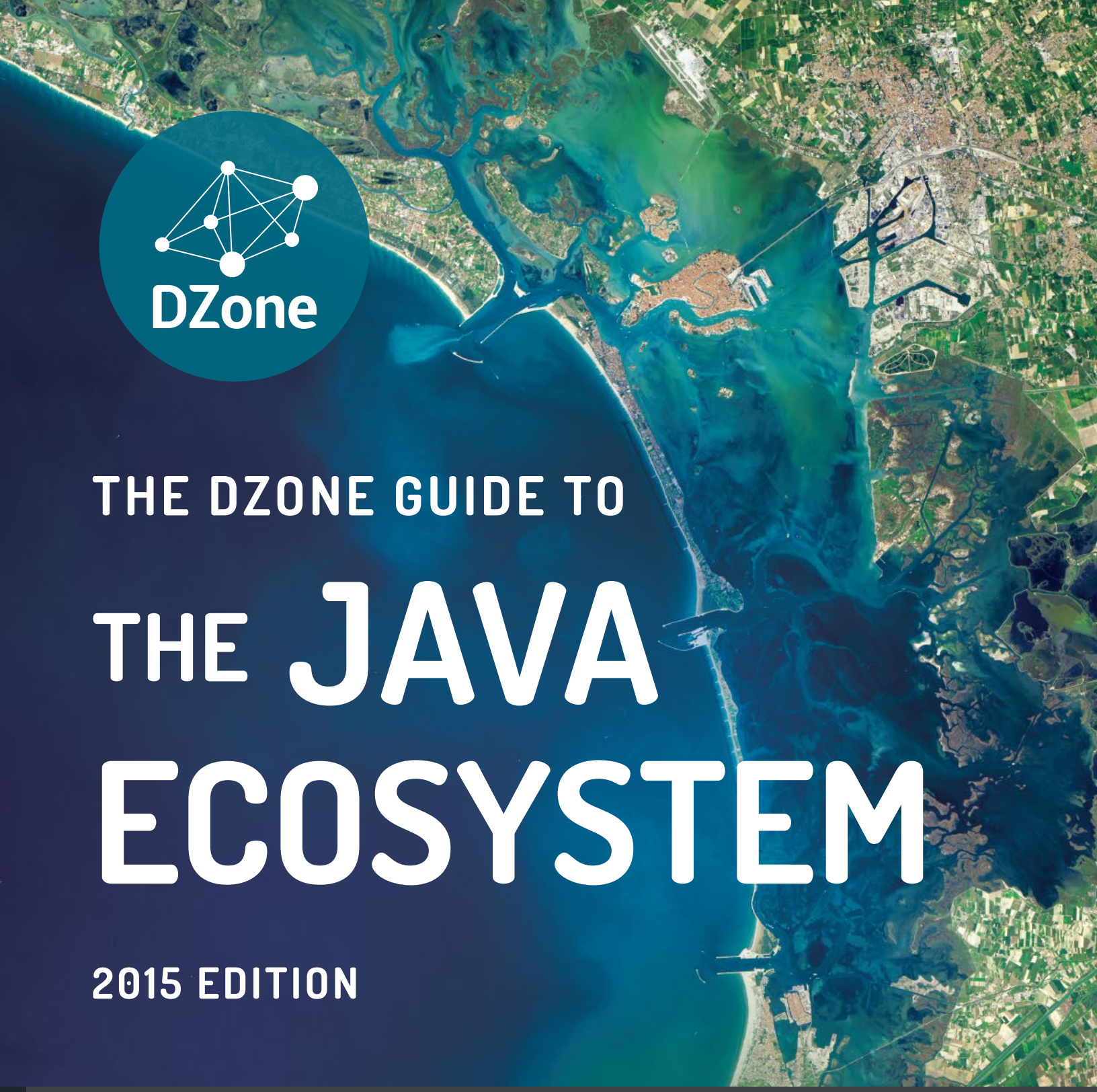


DZone

THE DZONE GUIDE TO THE JAVA ECOSYSTEM

2015 EDITION

BROUGHT TO YOU IN PARTNERSHIP WITH

CloudBees
The Enterprise Jenkins Company

JetBRAINS

JNBridge

New Relic.

Pivotal

riverbed

Dear Reader,

20 years—a lifetime for a programming language, never mind a whole platform. This year, we celebrate Java's 20th birthday, and what better way to celebrate than to release DZone's *first* Guide to the Java Ecosystem.

I can still remember the 10TH JavaOne. There was energy and a belief that Java would fulfill its mission to give developers a true chance to "Write Once, Run Anywhere." All this while Sun Microsystems struggled to find a place in the market amid a changing economy and a rapidly evolving Internet. Today, Java ***is*** everywhere, and it is the workhorse of enterprises and independent developers alike. It is open. It is more than just a language. And after all this time, Java remains the most popular language for developers (Tiobe) and one of the most active for job hunters (Indeed.com).

Long before DZone.com, we started Javalobby.org: an independent group of Java developers eager to talk about this new language that didn't require memory management and had the backing of some powerful organizations. We've evolved our vision of Javalobby over the years (now our Java topic portal), but it remains one of the largest sections of our site and one of the primary destinations for Java developers on the web. It's because of Java's role in the very foundation of DZone that finding the right title and focus for this guide was one of the hardest decisions our editorial team has had to make this year.

This guide, which you're hopefully enjoying today, focuses on trying to understand the trends about how developers such as yourself are using the Java platform. What version of the platform are you using? Are you using a full JEE stack or just using simpler tools like Spring? What percentage of developers prefer Tomcat over Websphere? Statistics like these—along with expert analysis and articles from industry leaders—await you in the *2015 Guide to the Java Ecosystem*.

Thanks, as always, for your support, and please continue to visit DZone.com daily.

P.S. We're always hiring—get in touch if you'd like to help build the platform that powers DZone.com and communities for LinkedIn, Microsoft, and a variety of other organizations.

MATT SCHMIDT
PRESIDENT AND CTO
MATT@DZONE.COM

TABLE OF CONTENTS

EDITORIAL

John Esposito
research@dzone.com
EDITOR-IN-CHIEF

G. Ryan Spain
DIRECTOR OF PUBLICATIONS

Mitch Pronschinske
SR. RESEARCH ANALYST

Matt Werner
MARKET RESEARCHER

Moe Long
MARKET RESEARCHER

John Walter
EDITOR

Allen Coin
EDITOR

Tom Smith
RESEARCH ANALYST

BUSINESS

Rick Ross
CEO

Matt Schmidt
PRESIDENT & CTO

Kellet Atkinson
VP & PUBLISHER

Matt O'Brian
DIRECTOR OF BUSINESS DEVELOPMENT

Jane Foreman
VP OF MARKETING

Alex Crafts
sales@dzone.com
DIRECTOR OF MAJOR ACCOUNTS

Chelsea Bosworth
MARKETING ASSOCIATE

Chris Smith
PRODUCTION ADVISOR

Jim Howard
SALES ASSOCIATE

Chris Brumfield
CUSTOMER SUCCESS ASSOCIATE

ART

Ashley Slate
DESIGN DIRECTOR

Yassee Mohebbi
GRAPHIC DESIGNER

Special thanks to our topic experts Paul Bakker, Reza Rahman, Markus Eisele, Arun Gupta, Josh Long, Ivan St. Ivanov, Marcus Lagergren, and our trusted DZone Most Valuable Bloggers for all their help and feedback in making this report a great success.

WANT YOUR SOLUTION TO BE FEATURED IN COMING GUIDES?
Please contact research@dzone.com for submission information.

LIKE TO CONTRIBUTE CONTENT TO COMING GUIDES?
Please contact research@dzone.com for consideration.

INTERESTED IN BECOMING A DZONE RESEARCH PARTNER?
Please contact sales@dzone.com for information.

Executive Summary

DZone surveyed more than 600 IT professionals and produced several expert articles for our 2015 Guide to the Java Ecosystem to give Java organizations greater insight into the effects of major trends on the larger Java development community and to discover common patterns and technology stacks currently in use across the Java industry. In this summary you will learn about the current state of the Java platform and the ecosystem of tooling around the Java language.

RESEARCH TAKEAWAYS

01 DEVELOPERS ARE MOVING TO NEW VERSIONS OF JAVA MORE QUICKLY

Data: 53% of respondents say they their new apps will use Java 8 in the next six months and 35% are going to start converting existing apps to Java 8 in that same time frame. 62% of respondents said their existing apps are in Java 7 and 20% already have existing apps in Java 8.

Implications: A slight majority of Java organizations seem to be moving to the latest version of Java for new applications. This shows a bit more forward movement in the Java community than there has been in years past. Large swaths of Java companies have sizable legacy codebases due to their organization's age and the constant progression of the language, which has left some of them behind. There will always be a large number of organizations where there isn't enough risk or enough resources to motivate an update to their legacy code, but it's surprising to see significant numbers of Java companies moving their apps to the newest version.

Recommendations: The fact that Oracle has declared Java SE 7 end-of-life is a significant contributor to this trend towards faster migration. To maintain a high level of security and take advantage of new language features that other programming languages have had for a long time, migration to the latest version Java is always the best option if an organization can afford it. Trisha Gee's article in this guide, "Why Java 8?" illustrates the positive impact Java 8 can have on your applications. Some key things enticing developers toward Java 8 are functional programming features and performance improvements.

02 INTEREST IN CONTAINERS & MICROSERVICES IS GROWING IN THE JAVA WORLD

Data: While nearly half of respondents haven't researched containers or microservices enough to comment on them (43% for microservices, 46% for containers), 21% want to implement containers (12% already have) and 15% want to implement microservices (10% already have). There are more respondents interested in containers—or actually using them—than those that aren't (12% not interested) and the same goes for microservices (20% not interested).

Implications: These trends have made a significant impact in the Java community over the last 2-3 years, but it will still take several more years for containers or microservices to obtain mainstream adoption in most of the organizations where they can be beneficial. However, many of the most successful web properties mentioned previously (Google, Netflix, Twitter, etc.) have already implemented containers and microservices where appropriate. These companies are driving trends by sharing some of their implementations.

Recommendations: Developers, architects, and business executives need to start researching the potential benefits that containers and microservices bring to their applications. Some may decide that they are not appropriate or mature enough for their systems, but more definitive research must be done so that a decision can be made. Several technologies have already moved to support microservices—such as Spring Boot, Wildfly Swarm, and Akka. Docker already has a dominant position in the containerization space. A good place to start researching Docker is by reading "Docker and Kubernetes in the Java Ecosystem" by Paul Bakker on DZone. For building microservices in the Java ecosystem, you should read Ivar Grimstad's article in this guide, "Building Microservices With Java."

03 JAVA IS ONE OF THE MOST POPULAR PROGRAMMING LANGUAGES

Data: There are several indexes in tech media that use various metrics to measure programming language popularity, including TIOBE, RedMonk, and IEEE. Java is ranked #1 or #2 in all of these major indexes. No other language has rankings that are as consistently high as Java's across all these indices. On the TIOBE index, one of the longest running indices, Java has been the number one language for 11 of the last 14 years, and it is currently number one by a significant margin. Only 3% of respondents in our Java ecosystem survey said they are pessimistic about the future of Java.

Recommendations: Java has reached a point where its popularity results in further popularity. More developers are using it, more open-source utilities exist around it, and there is more maturity and reliability around the platform and its ecosystem of tools. As a result, even more organizations and developers choose to build applications in Java.

Recommendations: While many thought Java would decline after the announcement of Oracle's acquisition of Sun Microsystems in 2009, it only took a few years to reestablish itself. Now with the release of Java 8, Java is finally catching up with the expressiveness of C# and now has the ability to be written with less verbosity than ever before. People are now optimistic about the future of Java, and it's being used in many of the most well-respected tech companies. Its popularity in the software industry is expected to continue and even grow further due to its advantages in data analysis and embedded systems, which will likely make Java a significant programming language in the Internet of Things industry.

Key Research Findings

More than 600 IT professionals responded to DZone's *2015 Java Ecosystem Survey*. Here are the demographics for this survey:

- Developers (42%) and Development Leads (25%) were the most common roles.
- 63% of respondents come from large organizations (100 or more employees) and 37% come from small organizations (under 100 employees).
- The majority of respondents are headquartered in Europe (44%) or the US (32%).
- Over half of the respondents (61%) have over 10 years of experience as IT professionals.
- We only surveyed developers who use Java in their organization. Other language ecosystems for these organizations include C# (31%), C/C++ (29%), Universal JavaScript (28%), PHP (26%), Python (26%), and Groovy (23%).

01. MOST DEVELOPERS ARE KEEPING UP FAIRLY WELL WITH NEW JAVA VERSIONS

A majority of respondents (58%) said their organization was using Java 8—the latest version of Java—in at least some of their new applications. 48% said some of their new apps would be built on Java 7. Only 10% of respondents indicated the use of Java 6 or below for any new apps, and a minuscule 1% said new apps would use Java 5 or lower. As far as existing apps are concerned, Java 8 has not caught up to Java 7; most organizations (62%) still have apps built on Java 7, and almost half (47%) have apps in Java 6. Only 20% of respondents said they had any existing apps built on Java 8. While this data shows that many developers are eager to move forward with Java as it evolves, they are hesitant to refactor existing applications already built on a previous platform. 18% of Java 7 users who have not moved to Java 8 have no plans to start using Java 8 in the next 6 months, for either new or existing apps.

02. JAVA EE AND SPRING USAGE IS ALMOST EQUAL

One of the significant rifts in the Enterprise Java community is the difference between organizations that use Spring and those that use Java EE. Our survey shows that the usage of some components of the two platforms is pretty close, with Java EE having a slight edge. Overall, 58% use some version of Spring and 67% use some version of Java EE. Very few respondents used Java EE 5 or lower, or Spring 2.x or lower. The breakdown of each version's usage can be found in the charts, but what's interesting is that almost one-third (32%) of respondents use both.

03. JVM LANGUAGES HAVE A STRONG SHARE OF JAVA DEVELOPERS

57% of respondents have written code in a non-Java, JVM-based language. Groovy (40% of all respondents) is the language that most have tried, with Scala (31%) close behind. The next highest languages were JRuby and Clojure, which were both around 6%. For each of the groups who had tried a specific JVM language, over 80% said they enjoyed using it. Only Groovy and Scala had large enough groups of users to make a statistically significant observation about user opinions. When asked if they would like to use these languages at their job, 45% of Groovy users said they would, and 55% of Scala users said they would.

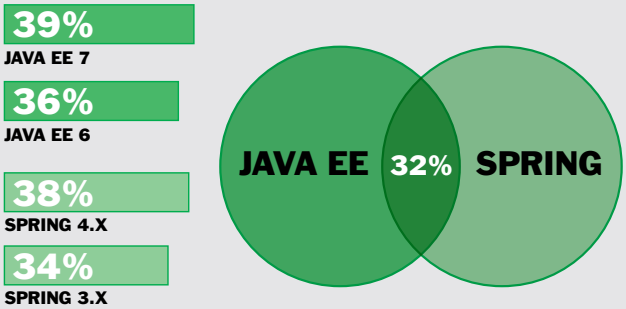
04. CERTAIN TOOLS DOMINATE IN MOST AREAS OF THE JAVA ECOSYSTEM

We asked respondents what their *primary* IDE, build tools, and CI engines were. For IDEs, Eclipse is still king with 54% of respondents using it. 30% use IntelliJ IDEA (most respondents [23%] actually use the paid version, with only 7% using the community edition). 13% use NetBeans. For build tools, Maven

01. **WHAT VERSIONS OF JAVA ARE BEING USED AT YOUR ORGANIZATION?**

VERSION	FOR NEW APPS	FOR EXISTING APPS
Java 5 or lower	1%	11%
Java 6	10%	47%
Java 7	48%	62%
Java 8	58%	20%

02. **WHICH OF THE FOLLOWING JAVA PLATFORMS DO YOU USE?**



(66%) is much more popular than other competitors like Ant (18%) and Gradle (11%). Jenkins absolutely dominates the CI space, with 62% using it. The next highest CI server has less than 10% share of respondents. 20% don't use a CI server. We also asked respondents to list all the Java persistence tools they use and found that standard JPA/Hibernate is the most common persistence solution for Java developers (64%), while a few others prefer JDBC—either standard (38%) or Spring JdbcTemplate (24%) versions.

05. JSF AND SPRING MVC ARE NECK-AND-NECK IN THE JAVA WEB FRAMEWORKS RACE

The Java web framework battle is much closer than the previously mentioned tooling competitions. In an August 2015 poll, DZone surveyed over 1,300 developers, and the results showed Spring MVC (34.2%) and Java Server Faces (34.5%) in a virtual tie. However, more Java developers are looking for front-end development tools outside of the Java ecosystem. When asked in the *2015 Java Ecosystem Survey* about the tools they use for application front-ends, respondents actually chose AngularJS (43%) more than Spring MVC (34%) and JSF (30%). All of Angular's direct competitors—React.js, Ember.js, Backbone.js—were under 10%.

06. FULL JEE APP SERVERS ARE NOT COMMON

Lightweight application servers like Tomcat (68%) and Jetty (27%) have been dominant in Java development for some time. For several years, the trend in Java development has been moving toward the use of only the components that you need, and JEE features in application servers are no exception [1]. We also found that most of these app servers are used in development more often than they are used in production. Only WebLogic and WebSphere were deployed in production more often than deployment.

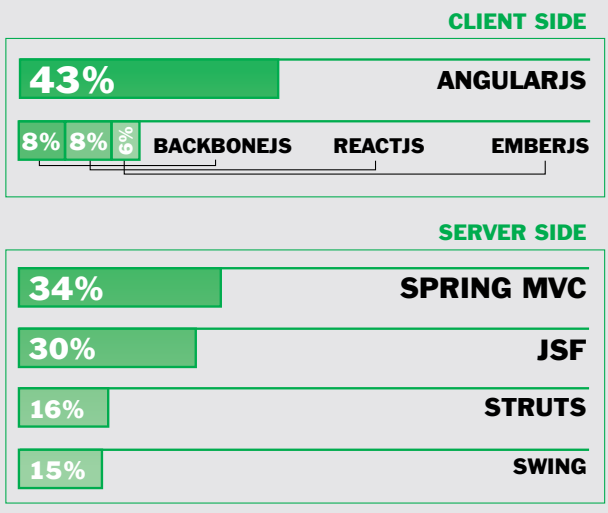
07. JAVA DEVS SLOWLY ADOPTING CLOUD, BUT MICROSERVICES AND CONTAINERS ARE FURTHER OFF

The adoption of cloud infrastructure is still modest in Java organizations, with only 37% running some percentage of their applications on the cloud during the development phase. 43% ran some percentage of their tests/QA on the cloud, and 42% are using some cloud infrastructure in their production environment. When we asked about more recent development trends like microservices and containers, only 10% were using microservices and only 12% were using containers (specifically, 9% were using Docker). Most respondents haven't researched these two trends enough to even think about using them (43% for microservices, 46% for containers), and some don't see a use for them currently or for the foreseeable future (20% for microservices, 12% for containers). However, the interest in these two trends is significant with 15% wanting to implement microservices and 21% wanting to implement Docker containers.

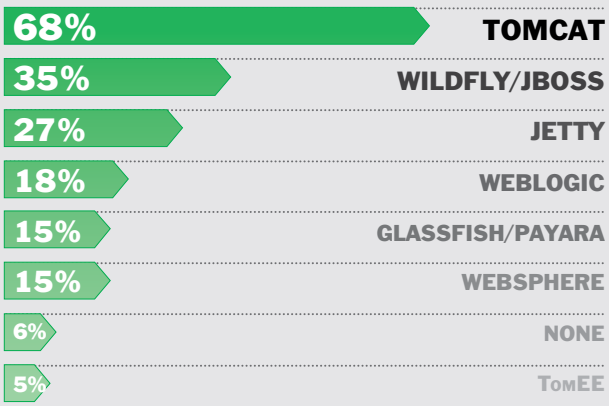
08. NEWER SERVER TECHNOLOGIES ARE STILL BLEEDING-EDGE AND RARE

We asked respondents about the server technologies they were using, and while most were still using Java EE or Spring primarily, we found small groups using some of the newer server technologies in the Java space. 11% said they are using a containerless method with Spring Boot and Dropwizard. However, Spring Boot and Dropwizard are used for more than just containerless deployment, so it's definitely not an indicator of the two products' overall popularity. 9% said they are using Scala and Akka, and 2% are using Vert.x.

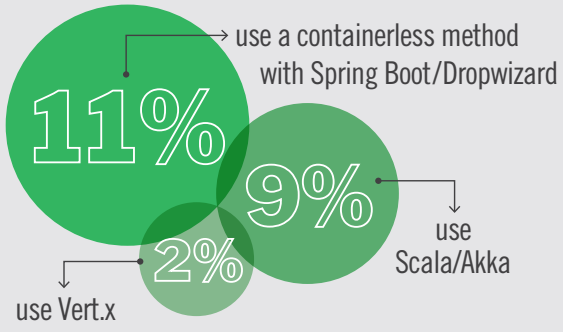
03. **WHAT DOES YOUR ORGANIZATION USE FOR APPLICATION FRONT-ENDS?**



04. **WHICH APP SERVER(S) DO YOU USE?**



05. **WHICH OF THE FOLLOWING SERVER TECHNOLOGIES DO YOU USE?**



Why Java 8?

QUICK VIEW

01 In many cases, Java 8 will improve application performance without any specific work or tuning.

02 Lambda expressions, the Streams API, and new methods on existing classes are some of the key productivity improvements.

03 Java 8's new `Optional` type gives developers significant flexibility when dealing with null values, reducing the likelihood of `NullPointerExceptions`

BY TRISHA GEE

Java 8 came out early last year—and Java 7 is now end of life—making Java 8 the only Oracle-supported option until Java 9 comes out at the end of next year. However, since organizations value stability over trendiness, many of us are still working with Java 7, or even 6.

Let's look at some features of Java 8, and provide some arguments to persuade your organization to upgrade.

IT'S FASTER

Here's a selling point that might please your boss, the business, or the operations guys: you'll probably find Java 8 runs your application faster. Generally speaking, applications that have moved to Java 8 see some sort of speed improvement without any specific work or tuning. This may not apply to an application that has been highly tuned to a specific JVM, but there are a number of reasons why Java 8 performs better:

Performance Improvements in Common Data Structures: Benchmarks of the ever-popular HashMap show that performance is better in Java 8. These sorts of improvements are very compelling—you don't need to learn the new Streams API or lambda syntax or even change your existing code to get speed improvements in your application.

Garbage Collector Improvements: Often "Java Performance" is synonymous with "Garbage Collection," and it is certainly true that poor garbage collection performance will impact an application's performance. Java 8 has substantial changes to GC that improve performance and simplify tuning. The most well-known of these changes is the removal of PermGen and the introduction of Metaspace.

Fork/Join Speed Improvements: The fork/join framework was new in Java 7, and was the latest effort to simplify concurrent programming using the JVM. A lot of work went into improving it further for Java 8. Fork/join is now the framework that's used under the covers for parallel operations in the Streams API (more on this later).

In addition, there are plenty more changes in Java 8 to support concurrency, and Oracle has summarized some of the performance improvements in JDK 8.

FEWER LINES OF CODE

Java is regularly accused of being heavy on boilerplate code. Java 8 addresses some of these issues by embracing a more functional style for the new APIs, focusing on what you want to achieve and not *how* to do it.

LAMBDA EXPRESSIONS

Lambda expressions in Java 8 are not just syntactic sugar over Java's existing anonymous inner classes—the pre-

Java 8 method of passing behavior around. Lambdas take advantage of Java 7's under-the-hood changes, so they perform well. To see examples of where using lambda expressions can simplify your code, read on.

NEW METHODS ON OUR FAVORITE COLLECTIONS

While Lambdas and Streams (which we'll cover next) are probably the top two selling points of Java 8, what's less well-known is that changes in Java 8 have allowed the language developers to add new methods to existing classes without compromising backwards compatibility. The result is that the new methods, combined with lambda expressions, allow us to drastically simplify our code. Take, for example, the common case of figuring out if an element already exists in a `Map` and creating a new one if not. Before Java 8, you might write something like:

```
private final Map<CustomerId, Customer>
customers = new HashMap<>();

public void incrementCustomerOrders(CustomerId
customerId) {
    Customer customer = customers.
get(customerId);
    if (customer == null) {
        customer = new Customer(customerId);
        customers.put(customerId, customer);
    }
    customer.incrementOrders();
}
```

This operation of "check if the item is in the map; if not, create it and add it" is so common that there's a new method on `Map` to support it: `computeIfAbsent`. This method takes as its second argument a lambda that states how to create the missing item:

```
public void incrementCustomerOrders(CustomerId
customerId) {
    Customer customer = customers.
computeIfAbsent(customerId,
            id -> new Customer(id));
    customer.incrementOrders();
}
```

In fact, there's another new feature in Java 8 called method references that makes this even shorter:

```
public void incrementCustomerOrders(CustomerId
customerId) {
    Customer customer = customers.
computeIfAbsent(customerId, Customer::new);
    customer.incrementOrders();
}
```

`Map` and `List` both have new methods in Java 8. It's worth checking them out to see how many lines of code they can save you.

> **Generally speaking, applications that have moved to Java 8 see some sort of speed improvement without any specific work or tuning.**

STREAMS API

The Streams API gives you flexibility to query and manipulate your data. This is a powerful tool. Check out some of the articles or books on the subject for a more complete view. Building fluent queries for your data is interesting in a Big Data world, but is just as useful for common operations. Let's say, for example, that you have a list of books and you want to get a list of unique authors for these books, in alphabetical order:

```
public List<Author>
getAllAuthorsAlphabetically(List<Book> books)
{
    List<Author> authors = new ArrayList<>();

    for (Book book : books) {
        Author author = book.getAuthor();
        if (!authors.contains(author)) {
            authors.add(author);
        }
    }
    Collections.sort(authors, new
Comparator<Author>() {
        public int compare(Author o1, Author
o2) {
            return o1.getSurname().
compareTo(o2.getSurname());
        }
    });
    return authors;
}
```

In the code above, we first iterate through the list of books, adding the book's author to the author list if it hasn't seen it before; then we sort the authors alphabetically by surname. This is exactly the sort of operation that streams have been designed to solve elegantly:

```
public List<Author>
getAllAuthorsAlphabetically(List<Book> books)
{
    return books.stream()
                .map(book -> book.
getAuthor())
                .distinct()
                .sorted((o1, o2) ->
o1.getSurname().compareTo(o2.getSurname()))
                .collect(Collectors.toList());
}
```

Not only is this fewer lines of code, it's arguably more descriptive—a developer coming to this code later can read it and understand that 1) it's getting authors from the books, 2) it's only interested in unique authors, and 3) the list that is returned is sorted by author surname. Combine the Streams API with other new features—method references and new methods on Comparator—and you get an even more succinct version:

```
public List<Author>
getAllAuthorsAlphabetically(List<Book> books)
{
    return books.stream()
                .map(Book::getAuthor)
                .distinct()
                .sorted(Comparator.
comparing(Author::getSurname))
                .collect(Collectors.toList());
}
```

Here it's even more obvious that the sorted method orders by the author's surname.

> **New methods, combined with lambda expressions, allow us to drastically simplify our code.**

EASY TO PARALLELIZE

We spoke about better out-of-the-box performance, and in addition to those earlier mentioned features, Java 8 can explicitly make use of more CPU cores. By simply replacing the method `stream` in the examples above with `parallelStream`, the JVM will split the operation into separate jobs and use fork/join to run them on multiple cores. However, parallelization is not a magic incantation to make everything faster. Doing operations in parallel always requires more work—splitting up operations and recombining results—and will therefore not always take less time. But this option is very interesting for areas that are suitable for parallelization.

MINIMIZE NULL POINTERS

Another new feature of Java 8 is the new `Optional` type. This type is a way of explicitly stating "I might have a value, or I might be null." Which means an API can now be explicit about either returning values that might be null vs. values that will always be non-null, minimizing the chances of running into a `NullPointerException`.

What's nice about `Optional` is the way you tell it to deal with nulls. For example, if we're looking for a particular book in a list, the new `findFirst()` method returns an `Optional`, which tells us it's not guaranteed to find a value. Given this optional value, we can then decide what to do if it's null. If we wanted to throw a custom Exception, we can use `orElseThrow`:

```
public Book findBookByTitle(List<Book> books,
String title) {
    Optional<Book> foundBook = books.stream()
            .filter(book -> book.getTitle().
equals(title))
            .findFirst();
    return foundBook.orElseThrow(() -> new
BookNotFoundException("Did not find book with
title " + title));
}
```

or you could return some other book:

```
return foundBook.orElseGet(() ->
getRecommendedAlternativeBook(title));
```

Or we could return an `Optional` so that callers of the method can make their own decision on what to do if the book is not found.

IN SUMMARY

Java 8 was a big release for Java, with syntax changes, new methods and types, and under-the-cover changes that will help your application even if you don't use the new language features. Java 7 is no longer supported by Oracle, so organizations are being pushed to migrate to Java 8. The good news is that Java 8 has many benefits for your business, your existing application, and for developers looking to improve their productivity.

TRISHA GEE has developed Java applications for a range of industries, including finance, manufacturing, technology; open source and non-profit—for companies of all sizes. She has expertise in Java high-performance systems, and is passionate about enabling developer productivity. Trisha is a Developer Advocate for JetBrains, a leader of the Sevilla Java User Group, a key member of the London Java Community, a MongoDB Master, and a Java Champion.

Microservices and Cloud Native Java

Spring Cloud Services for Pivotal Cloud Foundry 1.0 beta packages server-side components of Spring Cloud, found in projects such as Spring Cloud Netflix and Spring Cloud Config, and makes them available as native services inside Pivotal Cloud Foundry. Spring Cloud (projects.spring.io/spring-cloud) provides tools for Spring developers to quickly apply some of the common patterns found in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus). For DevOps teams working with microservices, this means simple install, configuration and production lifecycle management for critical microservice server infrastructure from NetflixOSS and Pivotal.

When authoring code to use these services, developers can use a familiar Spring programming model to implement common patterns found in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, and control bus). Microservice teams can wield NetflixOSS (and much more) with the groundbreaking productivity of Spring Boot. Taking advantage of these battle-tested microservice patterns, and of the libraries that implement them, can now be as simple as including a starter POM in your application's dependencies and applying the appropriate annotation.

Spring Cloud provides tools for Spring developers to quickly apply some of the common patterns found in distributed systems

This is this first and only Cloud Native Java end-to-end solution - a solution where microservices are deeply integrated into the application framework, runtime platform, and multicloud IaaS infrastructure automation.

WRITTEN BY PIETER HUMPHREY
PRODUCT MARKETER, **PIVOTAL**

Spring Cloud by Pivotal

Pivotal

Spring Cloud Services for Pivotal Cloud Foundry 1.0 beta packages server-side components of Spring Cloud projects and makes them available as services inside Pivotal Cloud Foundry.

CATEGORY
Java Framework for Distributed Computing

NEW RELEASES
As Needed

OPEN SOURCE?
Yes

CASE STUDY

Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon). Eureka instances can be registered and clients can discover the instances using Spring-managed beans, and an embedded Eureka server can be created with declarative Java configuration.

STRENGTHS

- Full Cloud Foundry service broker
- Threat modeling, supports OAUTH2, HTTPS, and CF UAA
- Simple installation and configuration with NetflixOSS
- Includes build-in event bus, messaging, and configuration management

PROMINENT TEAM MEMBERS

- Matt Stine
- Ben Hale
- Scott Frederick
- Chris Schaefer
- Craig Walls
- Roy Clarkson
- Will Tran
- Mike Heath

BLOG spring.io/blog

TWITTER @SpringCloudOSS

WEBSITE cloud.spring.io

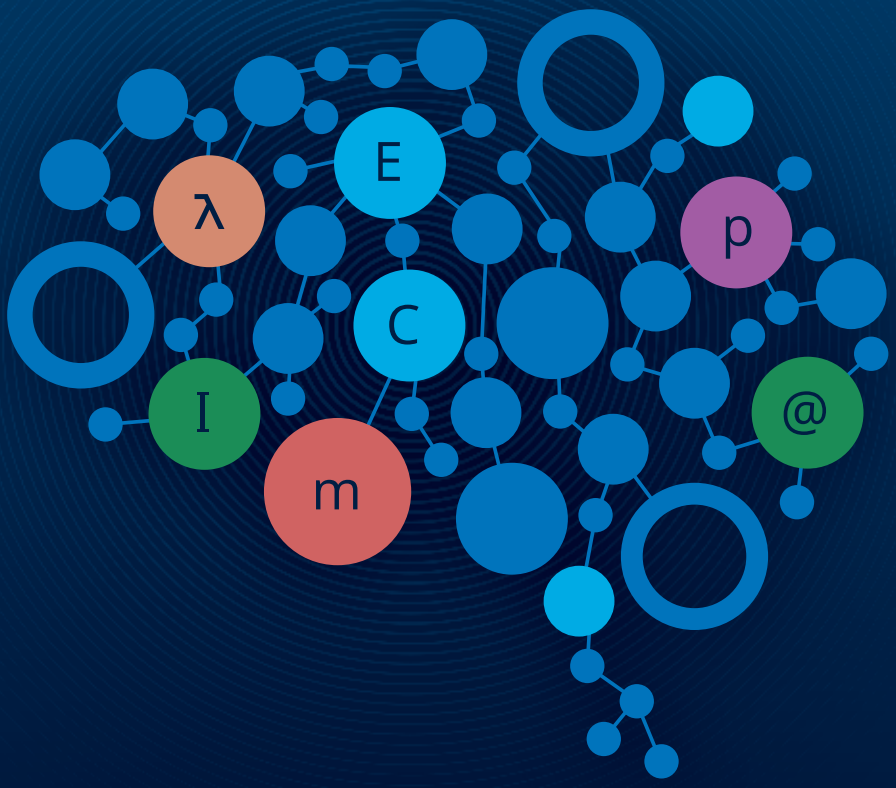
IntelliJ IDEA

The most intelligent Java IDE

for web, mobile and enterprise

Free and open-source edition included

WARNING: PROLONGED USE MAY CAUSE ADDICTION



JetBRAINS

Getting the Most Out of Code Review

Generations of developers have created multiple tools to automatically ensure code quality, yet none of these tools can guarantee you bug-free code. Whatever programming language you use, however small or large your project, problems of various severity pop up here and there that can only be spotted by a human. It is becoming clearer to many of us that building peer code review into the development process is crucial to having better quality code.

To get the most value out of code review, there are four key aspects one needs to have in mind:

- Code review does not mean that other ways of ensuring code quality should be neglected. You still need to write tests, use static code analysis, formatter, spellchecker, etc. Don't waste your reviewer's time on bugs that your IDE can find or can be otherwise automated.
- Distraction is the archenemy of code review; you need to be able to focus. Pick a code review tool that doesn't spam your mailbox with notifications and has a clean, uncluttered UI.
- Code changes are best understood in context. Try to provide meaningful comments in your own code for your reviewers, and use a tool that lets reviewers inspect changes as if they were working in their IDEs.
- Look for particular things as you review, be it business logic, security issues, SOLID principles, or something else. Scanning for general problems is surely helpful, but when given a chance to apply your expertise, take it.

Don't waste your reviewer's time on bugs that your IDE can find or can be otherwise automated.

The right code review tool is half the job. The rest is up to you!

WRITTEN BY MARIA KHALUSOVA
PRODUCT MARKETING MANAGER, **JETBRAINS**

Upsource by JetBrains

JetBRAINS

Upsource is a smart, lightweight code review and collaboration tool that provides unique IDE-level Java code insight.

CATEGORY
Code Review

NEW RELEASES
Semi-Annual

OPEN SOURCE?
No

CASE STUDY

Our team at Zando.co.za implemented a code review culture to improve the way we developed and maintained our eCommerce platform, and we were looking for the right tool: after watching the demo videos for Upsource on the day of its release, we installed it right away. As a team, we've come to know and trust the JetBrains brand and were excited to experiment with this new shiny toy! Installation was extremely simple, and we were up and running very quickly. JetBrains put a lot of thought into 2.0, and Upsource now boasts a lot of the features we've been waiting for since we began code reviewing intensely. Many thanks to JetBrains and the Upsource team for a great piece of software that gets things done, instead of getting in our way!

STRENGTHS

- Major VCSs are supported: Git, Mercurial, Subversion, and Perforce
- Upsource features Java code insight that helps you conduct code review faster and more easily
- With the IDE integration plugin, you do not have to leave your IDE to perform a review
- Comprehensive repository browsing and powerful search

NOTABLE CUSTOMERS

Companies all over the world, large and small, trust team collaboration tools from JetBrains.

BLOG blog.jetbrains.com/upsource | TWITTER @upsource_jb | WEBSITE jetbrains.com/upsource

First Steps in Java Microservices

BY IVAR GRIMSTAD

QUICK VIEW

01 Strategies for building microservices in the Java ecosystem include *container-less, self-contained,* and *in-container.*

02 Container-less microservices package the application, with all its dependencies, into a single "fat" JAR file.

03 Self-contained microservices also package a single fat JAR, but these also include an embedded framework with optional third-party libraries that will be compatible.

04 In-container microservices package an entire Java EE container and its service implementation in a Docker image.

Architectures based on microservices introduce new challenges for architects and developers. An ever-increasing list of languages and tools brings with it the capabilities to conquer this challenge. Java is no exception. This article explores different approaches to building microservices using the Java ecosystem.

INTRODUCTION

This article does ***not*** discuss whether microservices are good or evil, ***nor*** whether you should design your app for microservices upfront or extract the services as they emerge from your monolith application.

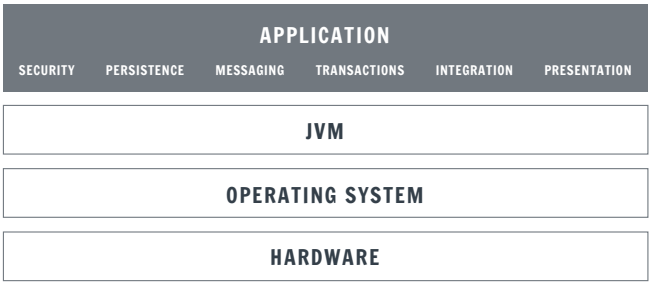
The approaches described here are not the only ones available, but they should give you a pretty good overview of several possibilities. Even though the Java ecosystem is the main focus in this article, the concepts should be transferrable to other languages and technologies.

I have named the approaches in this article *container-less, self-contained*, and *in-container*. These terms may not be entirely established, but they fulfill their purpose here to differentiate the approaches. I will describe what each means in the sections that follow.

CONTAINER-LESS

In the container-less approach, the developer treats everything on top of the JVM as a part of the application.

The container-less approach enables so-called *single JAR deployment* (also called a "*fat JAR deployment*"). This means that



the application, with all its dependencies, is packaged as a single JAR file and can be run as a standalone Java process.

```
$ java -jar myservice.jar
```

One advantage of this approach is that it is extremely easy to start and stop services as needed when scaling up or down. Another advantage is convenient distribution. You just need to pass one JAR file around.

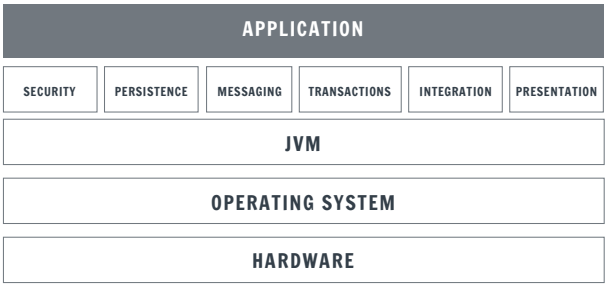
A downside of this approach is library compatibility. You are on your own for things like transaction support, or you need to bring in a third party library that provides support for this scenario. Later on—if you need support for something else, say persistence—you may need to fight compatibility issues between the libraries.

SELF-CONTAINED

Another variant of single JAR deployment is building your services with an embedded framework. In this approach, the framework provides implementations of the services needed and the developer can choose which to include in the service.

You may argue that this is exactly the same as the *container-*

less solution, but I like to distinguish them here since the *self-contained* approach actually gives you a set of third party libraries that you *know* are compatible.



This approach can involve tools like *Spring Boot* and *Wildfly Swarm*.

SPRING BOOT

Spring Boot and the Spring Cloud projects have excellent support for building microservices in Java. Spring Boot allows you to pick and choose various parts of the Spring ecosystem, as well as popular external tools and then package them along with your application in a JAR file. Spring Initializr allows you to do this with a simple checkbox list form. A simple *Hello World* service is shown in this example:

GIST SNIPPET: bit.ly/1YgUO9c

WILDFLY SWARM

A Java EE counterpart to Spring Boot is WildFly Swarm. It enables you to pick and choose which parts of the Java EE specification you need and package them and your application in a JAR file. The *Hello World* example looks like this:

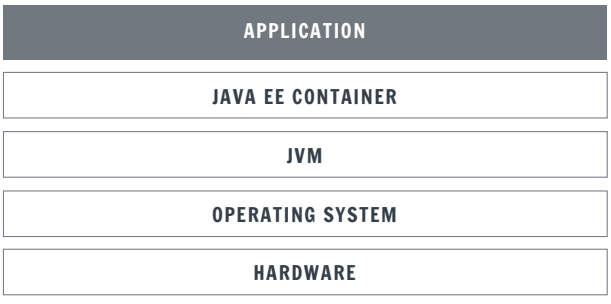
GIST SNIPPET: bit.ly/1LADHW9

The advantage of the *self-contained* approach is that you get to select only what you need in order for the service to run.

One disadvantage of this approach is that the configuration is a little more complex and the resulting deliverable JAR file is a bit bigger since it builds in the required container capabilities in the actual service.

IN-CONTAINER

While it seems like a lot of overhead to require an entire Java EE container to be able to deploy a microservice, keep in mind that some developers argue that the the 'micro' in microservice does not necessarily mean that the service is small or simple.



In these cases it may seem appropriate to treat the Java EE container as the required platform. Thus, the only dependency you need is the Java EE API. Note that the dependency is provided since the implementation is *provided* by the container. That means that the resulting WAR file is extremely lean.

The implementation of the service is the same as the Wildfly Swarm example above. See it here:

GIST SNIPPET: bit.ly/1ik7Hya

The advantage of this approach is that the container provides tested and verified implementations of standard functionality through standard APIs. Thus, you as a developer can focus entirely on the business functionality and leave the plumbing out of the application source. Another advantage of this approach is that the actual application code does not depend on the Java EE application server it is deployed to, whether it is GlassFish, WildFly, WebLogic, WebSphere, or any other Java EE compatible implementations.

The disadvantage is that you need to deploy the service into a container and thus increase the complexity of the deployment.

DOCKER

This is where Docker comes in. By packaging the Java EE Container and the service implementation in a Docker image, you achieve more or less the same result as you would with a single JAR deployment. The difference is that now the service is contained in a Docker image and not a JAR file.

Dockerfile

```
FROM jboss/wildfly:9.0.1.Final
ADD myservice.war /opt/jboss/wildfly/standalone/
deployments
```

The service is started by starting the Docker image in the Docker engine.

```
$ docker run -it -p 8081:8080 myorganization/myservice
```

SNOOP

The observant reader may have noticed the `@EnableEurekaClient` annotation in the Spring Boot code snippet from before. This annotation registers the service with Eureka, making it discoverable by service consumers. Eureka is a part of the Spring Cloud bundle and is an extremely easy-to-use and configure service discovery solution.

Java EE does not offer this functionality out of the box, but there are several open-source solutions available. One such solution is Snoop, which functions in a similar way to Eureka. The only thing needed to make a Java EE microservice available for service lookup is the `@EnableSnoopClient` annotation as shown in this example:

GIST SNIPPET: bit.ly/1Kp2PP1

CONCLUSION

Java is an excellent choice when building microservices. Any of the approaches described in this article will get things done. The most appropriate method for your particular case depends on the requirements of the service. For simpler services, a *container-less* or *self-contained* service is the better choice, but more advanced services may be faster and easier to implement with the power of an *in-container* implementation. Either way, Java *is* a proven ecosystem for implementing microservices.

IVAR GRIMSTAD is an experienced software architect and conference speaker focusing on Enterprise Java. He is a member of the JCP and currently works on JSR 371 (MVC 1.0), JSR 375 (Java EE Security API), and JSR 368 (JMS 2.1). Working as a Java developer since the language's very beginning, he has built applications using everything from JavaEE to Spring and a variety of other open-source products.

How to detect & fix problems 10x faster

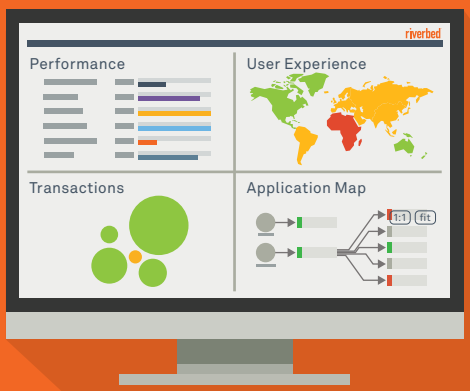
Challenges

75% of IT organizations suffer from slow apps — IDG, 2013

70% learn about problems from end users — Gartner, 2012

31% performance issues take more than a month to resolve — Forrester, 2013

Take control of app performance with SteelCentral AppInternals



> "We're now able to look inside of the developers' code – without having to modify the code – while it's running in our production environment. That's fantastic. I can't imagine someone running a site of any real size without this capability."
>
> Eric McCraw, Global Web Systems Manager for IT, National Instruments

> "AppInternals has drastically reduced the amount of time required to pinpoint issues and allowed us to pinpoint where the problems exist."
>
> Eric Saxe, IT Manager, Asurion

Top 5 benefits cited by users

1. Improved app performance
2. Reduced time spent in 'war rooms'
3. Reduced cost of fixing apps
4. Improved app user satisfaction
5. Faster development of new apps and features

TechValidate ID: 958-DAB-5CA

Source: Techvalidate survey of Riverbed users of SteelCentral and Performance Management

Try AppInternals today at www.appinternals.com

riverbed®

The Application Performance Company™

Developing and Delivering High-Performing Applications

Application performance equals business performance. Apps have transformed how businesses operate, have increased productivity, and have enabled numerous innovative markets. Developers make this possible.

As indispensable as applications have become, users expect more features as needs evolve and depend on developers to deliver these frequently, without compromising reliability or around-the-clock availability. Failing to do so will drive users to competitors or alternatives.

How do developers expedite release cycles and maximize time spent on new features as opposed to maintaining existing code? It comes down to two practices:

ENSURE RELEASE QUALITY PRIOR TO RELEASE

Identifying all erroneous code during testing helps developers fix issues early. Detailed transaction tracing can help QA expose and diagnose bottlenecks so that developers have the information they need to resolve them quickly. This helps make releases production ready.

RESOLVE ERRORS IN PRODUCTION ASAP

Monitoring apps in production ensures that you are the first know when problems strike. Recording every transaction from user to back-end—along with its associated system metrics and call tree details—helps developers reconstruct incidents and eliminate the root cause quickly.

> **WE'RE NOW ABLE TO LOOK INSIDE OF THE DEVELOPERS' CODE – WITHOUT HAVING TO MODIFY THE CODE – WHILE IT'S RUNNING IN OUR PRODUCTION ENVIRONMENT. THAT'S FANTASTIC. I CAN'T IMAGINE SOMEONE RUNNING A SITE OF ANY REAL SIZE WITHOUT THIS CAPABILITY.**
> – GLOBAL WEB SYSTEMS MANAGER, NATIONAL INSTRUMENTS

Helping developers stay focused on new features is important for businesses to stay competitive. Application performance monitoring products arm them with the diagnostics they need to minimize the time spent fixing bugs, maximize release quality, and keep them focused on delivering business-impacting applications.

WRITTEN BY KRISHNAN BADRINARAYANAN
SR. PRODUCT MARKETING MANAGER, **RIVERBED TECHNOLOGY**

SteelCentral by Riverbed Technology

riverbed

Trace all transactions from user device to back-end. Get comprehensive visibility and analytics across apps, networks and infrastructure.

CATEGORY
APM

NEW RELEASES
Quarterly

OPEN SOURCE?
No

CASE STUDY

The web systems team at National Instruments is tasked with ensuring that its public website, www.ni.com, runs optimally. They used to spend 1000s of hours each year troubleshooting issues caused by newly released apps. Their inability to quickly find causes of app performance problems created tension between them and the developers, and often hurt user experience. The web systems team now uses AppInternals to quickly diagnose root causes of app performance problems. Troubleshooting time is down by 90%, along with MTTR. Developers use AppInternals to test their code resulting in 20% to 30% fewer issues introduced into production. Scheduled updates are up from 16 to 120 per year. ni.com is more stable and delivers better user experience.

STRENGTHS

- Major VCSs are supported: Git, Mercurial, Subversion, and Perforce
- Upsource features Java code insight that helps conduct code review faster and easier
- With IDE integration plugin, you do not have to leave your IDE to perform a review
- Comprehensive repository browsing and powerful search

NOTABLE CUSTOMERS

- Asurion
- National Instruments
- Shell
- Visa

BLOG riverbed.com/blogs/

TWITTER @riverbed

WEBSITE appinternals.com

Production Debugging Is Not a Crime

BY ALEX ZHITNITSKY

QUICK VIEW

01 Developers need as much context information as possible in their logs when debugging in production. The data extraction needs to be planned and built in before deployment to production so that context data is not lost as the stack frames collapse.

02 For bugs like deadlocks or heavy performance bottlenecks, jstack is a great debugging tool, but developers need to modify it for preemptive execution if they want to collect information at the exact time the bug occurs.

03 Java agents are an important debugging tool. They retrieve data straight from the source in a lightweight way, allowing access to the exact variable values that caused each error.

04 Successful production testing strategies can include duplicating live traffic into new app versions or creating a canary server with varying degrees of features and controls.

Today more than ever, speed plays a larger role in the software development lifecycle. We see ReD teams who want to push code faster to production environments with rising complexity, and this amplifies a vulnerability that must be addressed.

Those few hours after a new deployment set the tone for its success. Every once in a while, things go wrong, no matter how strict your tests are. When your code is out in production and it meets the real-world architecture and scale of your system, with real data flowing through the application, things can go south pretty quickly. In order to be resilient and stay on top of things, a strategy needs to be implemented that allows you to:

- Identify when there's an error happening
- Assess the error's severity to prioritize it
- Draw out the state that caused the error
- Trace back and solve the root cause
- Deploy a hotfix

In this article we'll cover some of the most useful practices to allow you to assemble a time-critical response and "weaponize" your application.

THE CASE FOR DISTRIBUTED LOGGING

With production environments spread across multiple nodes and clusters, it's easy for a transaction that starts on one machine or service to cause an error someplace else. When an exception happens, there's a need to be able to trace back this type of distributed transaction, and the logs are often the first place to look for clues.

This is why, for every log line printed out, we need to be able to extract the full context to understand exactly what happened there. Some data might come from the logger itself and the location the log is created in; other data needs to be extracted at the moment of the event. A good way to trace such errors to their origin would be generating UUIDs at every thread's application entry point.

A useful yet underutilized feature here is using thread names to provide a window for this precious context, right before the stack collapses and the data is lost. You can format your thread name to something like:

```
Thread.currentThread().
setName(prettyFormat(threadName, getUUID(), message.
getMsgType(), message.getMsgID(), getCurrentTime()));
```

So instead of an anonymous name like *"pool-1-thread-17"* your application now produces smart stack traces that start this way: *"threadName: pool-1-thread-17, UUID: AB5CAD, MsgType: AnalyzeGraph, MsgID: 5678956, 30/08/2015 17:37"*

This works well when handling caught exceptions, but what if there's an uncaught exception involved? A good practice is to set a default uncaught exception handler, both to cover for that and to help extract any useful data you need. Other than thread names, additional places we can use to store hints about what happened are the TLS (Thread Local Storage) and the MDC (Mapped Diagnostic Context, which is provided by your logging framework). All other data gets lost as the stack frames collapse.

LEANING ON THE JVM TOOL BELT

Some more complex bugs like deadlocks or heavy performance

bottlenecks require a different approach. Take jstack, for instance: a powerful tool that ships together with the JDK. Most of you are probably already familiar with it in some way. Basically, jstack allows you to hook into a running process and output all the threads that are currently running in it. It will print each thread's stack trace; frames—either Java or native; locks they're holding; and all sorts of other metadata. It can also analyze heap dumps or core dumps of processes that have already ended. It's a longstanding and super useful toolkit.

The problem here is that jstack is mostly used in retrospect. The condition you're looking to debug has already happened, and now you're left searching through the debris. The server isn't responding, the throughput is dropping, database queries are taking forever: a typical output would be a few threads stuck on some nasty database query, with no clue of how we got there. A nice hack that would allow you to get the jstack output where it matters most is to activate it automatically when things start tumbling down. For example, you can set a certain throughput threshold and get jstack to run at the moment it drops [1].

Combined with using smart thread names, we can now know exactly which messages caused us to get stuck, and we can retrace our steps back, reproduce the error, isolate it, and solve it.

> **Unless we prepare our applications and environment in advance, there will not be much insight to recover after getting hit by errors and performance issues.**

USING JAVA AGENTS TO SKIP LOGGING ALTOGETHER

The next step in this process is gaining visibility into your application during runtime. Logs are inspected in retrospect and only include the information that you've decided to put there in advance. We've seen how we can enrich them with stateful data, but we also need a way to access the exact variable values that caused each error to get down to the real root cause. Java agents give us the ability to get to the data we need straight from the source without writing to disk and using huge log files, so we can extract only the data we'll actually be using.

One interesting approach is using BTrace, an open-source Java agent that hooks up to a JVM and opens up a scripting language that lets you query it during runtime. For instance, you can get access to things like ClassLoaders and their subclasses, and load up jstack whenever some troubled new class is instantiated. It's a useful tool for investigating specific issues and requires you to write scripts for each case you want to cover.

You could also write your own custom Java agent, just like BTrace. One way this helped our team at Takipi was when a certain class was instantiating millions of new objects for some reason. We wrote an agent that hooks up to the constructor of that object. Anytime the object was allocated an instance, the agent would extract its stack trace. Later we analyzed the results and understood where the load was coming from. These kinds of problems really pique our team's interest. On our day-to-day we're building a production grade agent that knows how to extract the variable values that cause each exception or logged error, all across the stack trace, and across different machines.

TESTING IN PRODUCTION: NOT FOR THE FAINT-HEARTED

Jokes aside, testing in production is a serious practice that many companies are taking part in. They don't cancel the testing phase completely, but they understand that staging environments are not enough to mimic full-blown distributed environments, no matter how much time and effort you put into setting them up. The only real testing takes place in production, with real data flowing through the system and unexpected use cases being thrown at it.

There are several approaches you can adopt for performing controlled production testing, depending on what kind of functionality it is that you're trying to test. One option is duplicating and routing live traffic both through the current system and through the new version of the component that you're testing. This way you can see how the new component behaves and compare it directly to the current version without risking the delivery of wrong results back to the user if, for example, it's some data crunching task.

Another option is segmenting your traffic and releasing new features gradually. One way to do this is to use a canary server, which is a single node in your system updated with the new version you'd like to roll out (just like a canary in a coal mine). Sometimes it's also helpful to add more fine-grained monitoring and logging to the canary server. Another option is to add more abstraction on top of the canary setup, implementing and making use of gradual rollouts with feature switches, or A/B testing small changes in your application to see how they impact performance.

FINAL THOUGHTS

Debugging Java applications in production requires a creative and forward-thinking mindset. Unless we prepare our applications and environment in advance, there will not be much insight to recover after getting hit by errors and performance issues.

[1] https://github.com/takipi/jstack

ALEX ZHITNITSKY is an engineer working at Takipi on a mission to help Java and Scala developers solve bugs in production and rid the world of buggy software. Passionate about all things tech, he is also the co-founder & lead of GDG Haifa, a local developer group. Alex holds a B.Sc from the Technion, Israel's Institute of Technology.

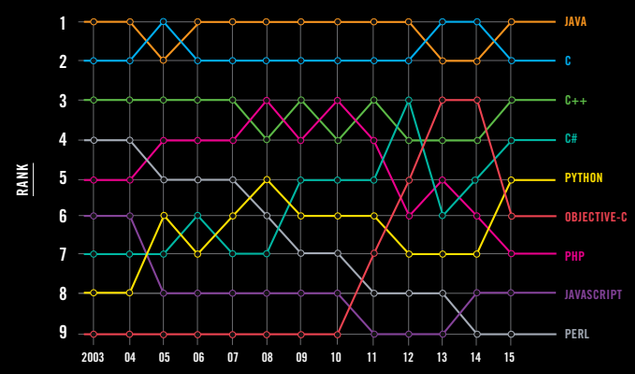
Java Popularity: By the Numbers

Anyone who's spent a little time in the software industry knows how widespread the use of Java is, but how popular is it exactly? There are multiple places where we can look to help answer this question. Google searches, job postings, open source repositories, Stack Overflow questions, and social media are all good sources to give us a fairly accurate picture of language popularity overall. Here are some charts to show how amazingly popular Java is.

01 TIOBE Programming Community Index

RANKINGS OVER TIME

Methodology:
Measures language search trends on Google, Google Blogs, Amazon, Bing, Yahoo!, Baidu, Wikipedia, and YouTube.



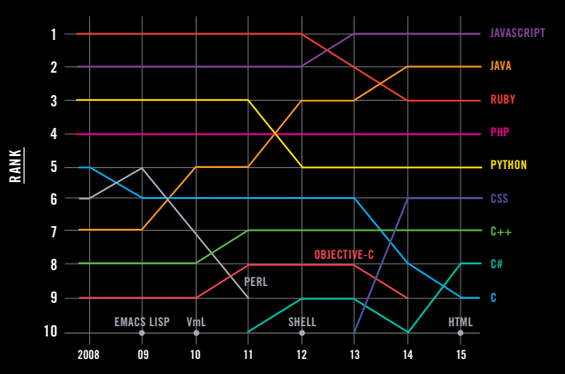
Java's Current Rank: #1

SOURCE tiobe.com/index.php/content/paperinfo/tpci/

02 GitHub Language Rankings

RANKINGS OVER TIME

Methodology:
Measures the number of public and private GitHub repositories using each language (excluding forks) as detected by Linguist.



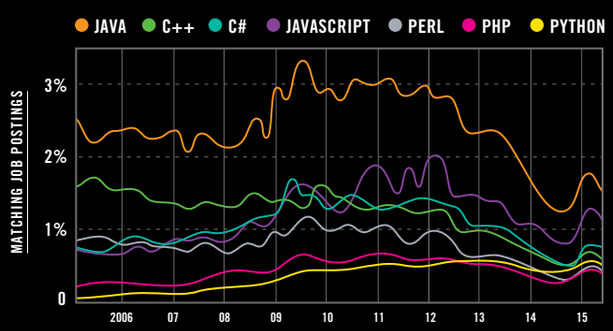
Java's Current Rank: #2

SOURCE github.com/blog/2047-language-trends-on-github

03 Indeed Job Keyword Trends

RANKINGS OVER TIME

Methodology:
Measures Indeed.com job posting keyword search results for selected languages. C was not included because there were too many false positives.



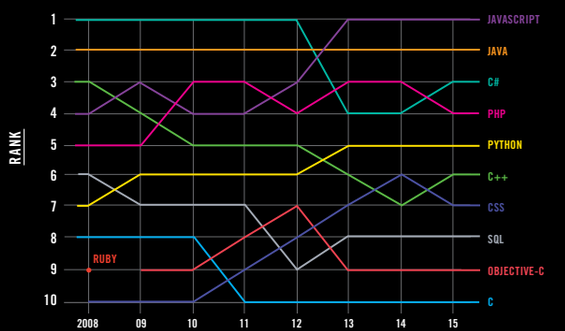
Java's Current Rank: #1

SOURCE bit.ly/indeed-rankings

04 Stack Overflow Language Rankings

RANKINGS OVER TIME

Methodology:
Measures the number of questions on Stack Overflow with each language tag.



Java's Current Rank: #2

SOURCE makingdataeasy.com/stackoverflow-trends

05 IEEE Language Rankings 2015

CURRENT RANKINGS

Methodology:
Measures Google searches, social media trends, GitHub and SO projects/tags plus activity, and recent job postings on Career Builder and Dice.

1	JAVA	100	2	C	99.9
3	PYTHON	99.5	4	C++	99.4
5	C#	91.3	6	R	84.8
7	PHP	84.5	8	JAVASCRIPT	83.0
9	RUBY	76.2	10	MATLAB	72.4

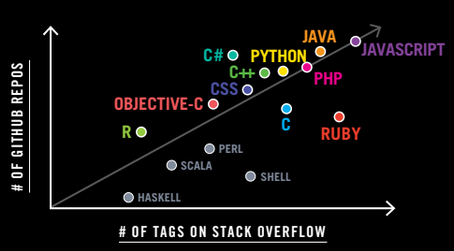
Java's Current Rank: #1

SOURCE spectrum.ieee.org/ns/IEEE_TPL/methods.html

06 RedMonk '15 Program. Language Rankings

CURRENT RANKINGS

Methodology:
Measures the number of repositories on GitHub using each language and the number of questions on Stack Overflow with each language tag.



Java's Current Rank: #2

SOURCE bit.ly/redmonk-rankings

Reactive Trends on the JVM

BY JONAS BONÉR

QUICK VIEW

01
Reactive Systems rely on a foundation of asynchronous message-passing to create loosely-coupled systems that are responsive, resilient, and elastic.

02
Analytics are being pushed into the stream (via Spark), which is emerging as the de facto approach for sub-second query response times across billions of rows of data.

03
Fast Data, or streaming data, can make systems more responsive. Reactive Streams can make Fast Data manageable by intelligently controlling the rate of data consumption.

04
Systems become very resilient when they are backed by a database with the full history of an application, which is built by event logging that is being used as the "Service of Record."

Two years ago, I collaborated with a few other developers to define a new set of architectural principles in the enterprise. It was clear at that time that everything from emergent deployment environments to user expectations to the size of datasets had outgrown previous patterns for building software. Basically what was typically "breaking" tied back to most software having synchronous call request chains and poor isolation, yielding single points of failure and too much contention.

In the Reactive Manifesto we created a new set of key principles that described the responsiveness, resilience, elasticity, and message-driven characteristics that we believed defined effective application architectures in the modern era of applications, running on everything from mobile devices to cloud-based clusters with thousands of multi-core processors.

Over the last two years, the response from the community has ranged from enthusiastic support (12,000+ signatures on the Reactive Manifesto) to eye-rolling (my favorite, in a Slashdot comment: "We want a machine that makes things cold. We don't care how it's built. We'll call this... The Refrigerator Manifesto."). Some felt that Reactive encapsulated key attributes that had long been embraced in their internal development philosophy, in the same way that some companies did Agile software development before it was "Agile." Some felt Reactive too prescriptive, while others felt it was too generic.

SO WHAT IS REACTIVE?

The Reactive Manifesto's goal is to condense the knowledge around designing highly-scalable and reliable applications into a set of four required architecture traits:

RESPONSIVE

The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behavior in turn simplifies error handling, builds end user confidence, and encourages further interaction.

RESILIENT

The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems—any system that is not resilient will be unresponsive during and after failure. Resilience is achieved by replication, containment, isolation, and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

ELASTIC

The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central

bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

MESSAGE-DRIVEN

As the foundation to the three traits above, Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to reify and delegate failures as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location-transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to more efficient system utilization.

> **"What was typically "breaking" tied back to most software having synchronous call request chains and poor isolation, yielding single points of failure and too much contention."**

NEW PATTERNS DRIVING REACTIVE INNOVATION

Over the last couple of years, I believe the three most interesting new trends driving Reactive innovation are:

MICROSERVICES

In traditional Java EE apps, services are written in a very monolithic way. That ties back to a strong coupling between the components in the service and between services. App servers (WebLogic, JBoss, Tomcat, etc.) are encouraging this monolithic model. They assume that you are bundling your service JARs into an EAR file as a way of grouping your services, which you then deploy—alongside all your other applications and services—into the single running instance of the app server, which manages the service "isolation" through class loader tricks; a very fragile model.

Today we have a much more evolved foundation for isolation—from the ground up—starting with, for example, Docker containers, isolated all the way up through better hardware and communication protocols. I'm excited about the Microservices momentum because it makes isolation first class, which is a necessity for resilience. You can't build a Reactive system without isolating failures and having a separate context outside the failed component to react to the failure. You need isolation in order to avoid cascading failures.

FAST DATA—THE WORLD IS GOING STREAMING

The whole movement towards Fast Data and real-time data requires closed feedback loops for getting data into and out of the system. The benefit of Fast Data is that you get systems that are more responsive and adaptive, allowing you to feed the results of real-time data processing back into the running system, which allows it to react to change. This capability can also be used to make these systems more resilient and scalable, but with reduced complexity.

One of the early pitfalls for streaming data in Fast Data scenarios, for example, was the lack of back-pressure. If a processing stage produced data faster than the next stage could consume that data, it would lead to a failure in the consumer, which would cause cascading failures throughout the entire processing pipeline. Reactive Streams address that problem by bringing back-pressure to streams to control the rate of data consumption. In general, the back-end systems that scale Big Data and IoT are a perfect fit for Reactive architectures.

EVENT LOGGING AS THE SERVICE OF RECORD

Another area in which I'm seeing a lot of Reactive innovation, specifically on the JVM, is when event logging is being used as the "Service of Record." In event logging, each state change to the application is materialized as an event in the log. What you get is a database with the full history of the application; a database of facts, rather than the traditional SQL database approach that only works with a "cache of the subset of the log," as Pat Helland aptly put it. If your durable state is based on an event log, it can be easily replicated and replayed somewhere else to bring the system or component up to speed wherever it is. This is a great pattern for failure handling in distributed stream processing—if one thing fails it can be brought back up to speed and continue. Architectural patterns making use of the event log include Event Sourcing and CQRS. This way of thinking about durable state works very well with Microservices, where each service can have its own isolated, strongly consistent, event log-based, durable storage that relies on eventual consistency between the services for scale and availability. Fast Data is the foundation for this durable state and streaming architecture.

REACTIVE EVOLVING FROM PRINCIPLES TO IMPLEMENTATION PATTERNS

I believe that Reactive is on a similar arc to the one the Agile methodology followed. A lot of companies were using Agile methodologies without calling it Agile years before the term was coined. But calling it Agile made it easier to talk about and communicate the philosophy, so the Agile Manifesto was created. When this set of principles became more familiar in the developer community, you started to see the adoption of Scrum for distilling the essence of the processes and XP for supporting programming principles and practices like test-driven development, pair programming, and continuous integration.

Today we're seeing the same shift for Reactive. The core message of Reactive is aimed at core principles rather than tools and techniques. But Microservices, Fast Data, and Event Logging are great examples of how implementation patterns within the Reactive movement are starting to get more definition and momentum.

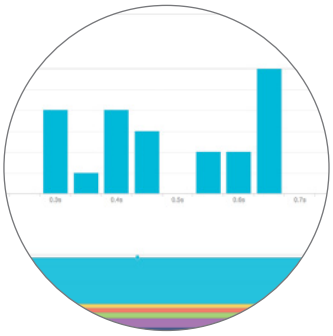
JONAS BONÉR is the co-founder and CTO of Typesafe. He is also the inventor of Akka, a JVM-based toolkit and runtime that uses the actor model to build concurrent, distributed applications. Jonas is a Java Champion and a co-author of the Reactive Manifesto.

Constantly monitoring your applications so you don't have to.

New Relic's SaaS-based Application Performance Monitoring helps you build, deploy, and maintain great web and mobile software.

Start your free trial now.
newrelic.com/java

- **Application Response Times**
- **Error Tracking & Analytics**



- **Transaction Tracing**
- **Thread Profiling**

- **JVM Performance Analysis**
- **Deployment Comparisons**

New Relic®

Interoperability Tools Help Cure Developer Fatigue

"Developer fatigue," that frustrated exhaustion you feel trying to keep up with the constant flood of new languages, libraries, frameworks, platforms and programming models, is a real problem. While there's nothing wrong with trying to stay up-to-date, a number of downsides work against you: loss of productivity, lost cycles spent on immature tools, short half-life and the risk of making bad bets.

This problem has been around since the dawn of computers, but it's getting worse. Causes include the push for "full stack" developers who must "do more with less," the rising number of open source projects, the proliferation of new types of hardware and the pressure to introduce the new in order to stay ahead of the competition.

How can you possibly keep up? Perhaps the best way is to take it slowly: integrate the features of new technologies as you need them. In my experience, the need to use a technology is the best reason to start learning about it — and the best way to retain the skills once learned. Jumping in and learning a new technology for its own sake (aka "warehousing") is popular, but much less effective.

With gradual introduction to new technologies and a focus on the aspects that can help solve immediate problems, you can stay productive while still using familiar tools, and introduce new stuff one feature at a time. While not possible in all cases, if your familiar platform is Java or .NET, and the new technology is .NET or Java-based, JNBridgePro can help.

Developer fatigue is frustrating and costly. Stay productive using familiar tools and incremental integration with new technologies as needed.

JNBridge has examples that show how a familiar legacy technology like .NET or Java can be integrated with a newer technology like Hadoop, Groovy, Python or Clojure. The examples focus on how you can bring in new technologies at your own pace, without having to assimilate the entirety of a new language, API, or platform. Our goal is to help you avoid developer fatigue while keeping current on the ever-increasing number of new technologies being introduced.

WRITTEN BY WAYNE CITRIN
CTO, JNBRIDGE

JNBridgePro by JNBridge

Exponentially faster than web services, JNBridgePro can fully expose any Java or .NET API or binary—services-enabled or not.

CATEGORY
Java and .NET Interoperability

NEW RELEASES
Semi-Annual

OPEN SOURCE?
No

CASE STUDY

Customers use JNBridgePro to connect Java and .NET in all kinds of applications. Adobe adds .NET integration services into Java-based ColdFusion. Swiss Re delivers a standardized numerical analysis Java API for their actuaries to code into Excel. A media company calls Apache Phoenix, a relational database layer over HBase, from an IIS application. ShapeTech uses the Java-based NASA World Wind API in their C# defense training apps. Ebiexperts integrates Java-based SAP BusinessObjects into their .NET-based BI product. A financial services firm accesses server-side EJBs for client data and screen pops it into their call center's .NET desktop telephony display. An aerospace company embeds .NET-based video controls into a Swing GUI user control.

STRENGTHS

- Access Java classes from .NET as if Java were a .NET language (C#, VB, etc)
- Access .NET classes (written in C#, VB, F#...) from Java as if they were Java classes
- Gain full access to any API on the other side, whether it's service-enabled or not
- Expose any Java or .NET binary, no source code required
- Deploy anywhere: same process, separate processes, separate devices, across a network

NOTABLE CUSTOMERS

More than 600 global enterprises and software development houses rely on JNBridge products in a variety of applications, including aerospace, financial services, healthcare, manufacturing, media, retail, telecommunications, and more.

BLOG jnbridge.com/jnblog | TWITTER @jnbridge | WEBSITE jnbridge.com

Java Ecosystem Executive Insights

BY TOM SMITH

QUICK VIEW

01
The Java ecosystem is massive. Don't build from scratch until you've searched the open-source communities. If you don't see what you're looking for, ask the community.

02
Give back. The Java community thrives based on the contributions of others. By sharing your code, you're ensuring the future of Java.

03
Even if you think you already know Java, look again. The open-source community is adding new elements daily to enable Java to remain relevant in all applications.

In order to more thoroughly understand the state of the Java ecosystem today, and where it's going, we interviewed 11 executives with diverse backgrounds and experience with Java technologies, projects, and clients.

Specifically, we spoke to:

Fred Simon, Co-Founder and Chief Architect, JFrog · Brandon Allgood, PhD, CTO, Numerate · Dr. Andy Piper, CTO, Push Technology · Gil Tene, CTO, Azul Systems · Anthony Kilman, Tech Lead, AppDynamics · Bhartendu Sharma, Vice President of Operations, Chetu · Ray Auge, Senior Software Architect, Liferay · Jonas Bonér, Founder and CTO, Typesafe · Toomas Römer, CTO and Founder, ZeroTurnaround · Michael Hunger, Lead Developer Advocate, Neo Technology · Charles Kendrick, CTO and Chief Architect, Isomorphic Software

The Java ecosystem is massive. Everyone we spoke with has been working in the ecosystem throughout their careers, and most have positive feelings about how the platform has evolved to one that is open. Even though Java is not described as a "bright and shiny object," it continues to have a very bright future—assuming Oracle and Google can resolve their differences.

Here's what we've learned from the conversations:

01

There's **no definitive agreement on the most important part of the Java ecosystem**. Perhaps this is a function of the age, diversity, and size of the ecosystem.

Ubiquity, reliability, and performance of the core platform were mentioned most frequently as benefits and the reasons why Java is the platform of choice for large, well-established enterprises. The object-oriented nature of the platform enables large development teams to work on multiple layers. Platform independence enables it to interface with other JVM languages. The JVM serves as a platform for new languages outside of Java like Scala, Clojure, Groovy, and many more.

The transparency of the JDK enables the open-source community to make innovative additions to the ecosystem, thereby making Java more interesting and relevant. The JCP is vibrant and active with participants actively sharing contributions that add to the usefulness and improve runtime performance of the platform.

02

As the owner of Java, and the JDK, **Oracle is clearly the most important player** in the ecosystem, producing the "official" elements of the platform; however, it is not the only player. In fact, there are at least 60 million Java developers. The Java Advisory Committee is actively overseeing the evolution of Java and ensuring standards and best practices are being maintained. The JCP is driving evolution in every sector.

Google is important because of Android. IBM is committed to leading development and standards while serving on the governing board for OpenJDK. Pro-open-source organizations like Pivotal/Spring, Apache, Typesafe, and Red Hat are adding more interest and innovation around Java. Azul is leading JVM development while SAP is staying involved in development as well. Open-source communities around JVM languages like JRuby, Groovy, Clojure, and Scala are driving additional innovation.

03

Twitter is the most popular way for respondents to stay up-to-date on the deluge of Java ecosystem trends, with most respondents following specific thought leaders and using

Twitter to find the most relevant and timely blog posts. The Java community and elements thereof, as well as developer communities—like DZone's Java Zone (formerly Javalobby), InfoQ, Hacker News, and StackExchange—were also mentioned as being great sources of knowledge and information.

04

The greatest value of the Java ecosystem is its ubiquity. Java can be used for big servers, Big Data, IoT, and large websites. You can use the same language for both mobile on the client-side and Big Data-crunching server-side. This makes it easy to integrate between multiple services, platforms, and distributed transactions to get things done quickly. It's easy to find developers who know Java. It builds the safest, most stable enterprise software that can scale. It has a tremendous library ecosystem and a strong open-source community behind it.

05

The biggest recent changes respondents have seen in the Java ecosystem seem to be the **introduction of Java 8 and the involvement of the open-source community**. Java 8 enables easier parallel computations, backwards compatibility, and new language features like Lambda expressions—a powerful technique in a developer's toolbelt. Open source has led to tremendous innovation, more languages, and opportunities across the mobile space due to Android also being open source.

One respondent expressed concern that Sun's lack of leadership and major missteps (citing JavaFX and JSF) have led to a number of conflicting approaches in basic areas of the platform like UI and data binding.

06

While it may seem contradictory, the **biggest obstacle to Java's success *is* its success**. Members of the IT community have an inherent bias against anything that's been around for more than a few years. Java has been around for 20 years and is the legacy platform for most large enterprises. But there's a reason why: it's good, it's secure, it's scalable, it's flexible, and it's seeing a return to relevance with the additional elements being contributed by the open-source community. While developers and startups may want to use "bleeding edge" technology stacks such as Node.js (it was Ruby on Rails in 2008), established businesses are more interested in software that can get the job done reliably. They don't care about the technology stack until it's causing problems.

A more specific concern is the poor expressiveness of the Java language, which can result in code that takes longer to write, is harder to read, and tends to be rigid in the face of evolving requirements. Java continues to struggle from JAR hell, a problem similar to DLL hell, which .NET solved years ago. Various solutions to this issue keep getting pushed out. Project Jigsaw, which is planned for Java 9, should alleviate the problem; however, a definitive solution has not yet been found.

07

We asked respondents **what they value in Java developers**, and—like everything else in the Java ecosystem—there was a diverse group of opinions:

- Understand the software and its architecture. Have standard design patterns, like Flyweight and Observer, down pat.
- Master parallelization and be knowledgeable about how to interact with threads.
- Be a team player that is patient, empathetic, and stays abreast of how the ecosystem is evolving.
- Don't try to reinvent the wheel. There's already so much written in Java that's available to the Java community; you can speed up development time by building on top of something that's already been built and proven to work.

08

Concerns with the Java ecosystem revolve around the omnipresent tension between Oracle and Google. Java will benefit greatly when these technology behemoths check their egos at the door and begin collaborating to improve Java. If Oracle does not resolve their differences with Google, this could cause many partners to look for other solutions, causing Java to stagnate.

Most of the skepticism around Oracle's support of Java following its acquisition of Sun in 2009 has dissipated as users have seen Oracle promoting the Java community while providing fair access. Some even commented that Java is being run more professionally with Oracle behind it.

09

The future of the Java ecosystem lies in IoT, mobile, and enterprise app development. The Java platform is the best solution for building multi-core distributed applications, and it can help normalize these evolving computing environments. Additionally, as the importance of personalization, privacy, and security become even greater around IoT and Big Data, more enterprises will realize these are the foundational elements upon which Java was built.

10

Some parting thoughts for developers: The Java ecosystem is massive, and the Java community has already solved a lot of problems. **Do not begin creating solutions from scratch until you've thoroughly researched pre-existing solutions.** This will reduce redundant work, speed up development, and increase your knowledge of the platform and the ecosystem.

There are 60 million developers using Java. Make sure to share what you learn and what you produce. The more developers that are participating in the community, the deeper and richer the ecosystem will be.

You can always learn more. As one respondent said, "If you think you know Java very well, you haven't looked deep enough."

The executives we spoke with are working on their own products and serving clients. We're interested in hearing from developers, and other IT professionals, to see if these insights offer real value. Is it helpful to see what other companies are working on from a more industry-level perspective? We welcome your feedback at research@dzone.com.

TOM SMITH is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

Continuous Delivery is Eating DevOps as Software is Eating the Business

What if you could release dozens of tested software improvements to production daily? Could your business respond to market pressures more quickly? The answer is yes - and continuous delivery makes it possible.

Continuous delivery involves automation of software delivery pipelines - from code to production. With automation, releases are delivered more quickly than is possible with legacy processes. Delivery can occur as often as several times each hour, day or week. Deployment becomes a non-event.

WHETHER YOU DEVELOP IN JAVA OR NOT, THE TREND CALLED CONTINUOUS DELIVERY IS OCCURRING ALL AROUND YOU. YOU NEED TO LEARN ABOUT IT.

THE VALUE OF CONTINUOUS DELIVERY

Software is at the heart of everything we do today. As software eats the world, businesses must deliver new capabilities faster and faster. Continuous delivery empowers an organization to respond to market opportunities more swiftly – with the ability to instantly learn what works and what doesn't.

A common misconception associated with automation is that risk increases and quality decreases. On the contrary, continuous testing is inherent in continuous delivery practices. You're actually building quality into the process. Organizations utilizing effective continuous delivery practices produce higher quality software.

BOTTOM LINE: WHY YOU SHOULD CARE ABOUT CONTINUOUS DELIVERY?

As software eats the business, so continuous delivery eats software development and deployment. Yet, moving from legacy development processes to continuous delivery is not easy. Smart firms dip their toe in the water, trying continuous delivery on a small project. Lessons learned from that initial project are then applied to broader initiatives. Initial apprehensions about potential risks associated with continuous delivery practices disappear as speed, transparency, automation and cross-team collaboration actually reduce risk, spare the enterprise new risks and catapult application delivery forward. In the process, applications are optimized–and so is the business.

WRITTEN BY SACHA LABOUREY
CEO AND FOUNDER, **CLOUDBEES, INC**

Jenkins Platform by CloudBees

Based on the time-tested, proven, extensible, and popular Jenkins automation platform. Advanced CD Pipeline execution and management.

CATEGORY
Continuous Deployment and CI Platform

NEW RELEASES
As Needed

OPEN SOURCE?
Yes

STRENGTHS

- Proven CI/CD platform
- Built-in high availability
- Role-based access control
- Advanced analytics
- Enterprise scale-out features

CASE STUDY

CHALLENGE: Orbtiz needed to shorten delivery times for more than 180 applications that power 13 different Web sites.

SOLUTION: Refine software delivery processes and implement open source Jenkins and CloudBees solutions for continuous delivery to automate tests, ensure reliable builds and increase build consistency across the organization.

BENEFITS: Release cycles cut by more than 75%; teams focused on high-value tasks; user experience enhanced through increased multivariate testing.

NOTABLE CUSTOMERS

- Netflix
- Nokia
- TD Bank
- Orbitz
- Lufthansa
- Apple

BLOG cloudbees.com/blog | TWITTER @cloudbees | WEBSITE cloudbees.com

diving deeper

INTO THE JAVA ECOSYSTEM

TOP 10 #JAVA TWITTER FEEDS

 @MREINHOLD

 @SPRINGROD

 @LUKASEDER

 @PETERLAWREY

 @REZA_RAHMAN

 @TRISHA_GEE

 @ADAMBIEN

 @JODASTEPHEN

 @JBONER

 @JSTRACHAN

DZONE JAVA ZONES

Java

dzone.com/java

The largest, most active Java developer community on the web. With news and tutorials on Java tools, performance tricks, and new standards and strategies that keep your skills razor-sharp.

Web Dev

dzone.com/webdev

Web professionals make up one of the largest sections of IT audiences; we are collecting content that helps web professionals navigate in a world of quickly changing language protocols, trending frameworks, and new standards for user experience. The Web Dev Zone is devoted to all things web development—and that includes everything from front-end user experience to back-end optimization, JavaScript frameworks, and web design. Popular web technology news and releases will be covered alongside mainstay web languages.

DevOps

dzone.com/devops

DevOps is a cultural movement, supported by exciting new tools, that is aimed at encouraging close cooperation within cross-disciplinary teams of developers and IT operations/system admins. The DevOps Zone is your hot spot for news and resources about Continuous Delivery, Puppet, Chef, Jenkins, and much more.

TOP JAVA REFCARDZ

Core Java
Gives you an overview of key aspects of the Java language and references on the core library as well as the most commonly used tools.

Java Performance Optimization
Covers JVM internals, class loading, garbage collection, troubleshooting, monitoring, concurrency, and more.

Getting Started with Scala
Covers creating a new Scala project, a tour of Scala's features, an introduction to classes and objects within Scala, and much more.

TOP JAVA WEBSITES

ProgramCreek
programcreek.com

IBM DeveloperWorks
ibm.com/developerworks/java

Baeldung
baeldung.com

TOP JAVA TUTORIALS

Vogella
vogella.com

Programming By Doing
programmingbydoing.com

Java (Beginner) Programming Tutorials
bit.ly/YouTubeJava

The Power, Patterns, and Pains of Microservices

QUICK VIEW

01 Microservices make software faster to release and easier to maintain, but also invite the complexity inherent in distributed systems.

02 Distributed computing problems are extremely hard; but many have been solved already.

03 Use known distributed systems patterns to make your microservices-based applications more resilient and more robust.

BY JOSH LONG

SURVIVAL IS NOT MANDATORY

It is not necessary to change. Survival is not mandatory. - W. Edwards Deming

The principles of The Agile Manifesto, now non-controversial and well accepted, speak to how to both write and deploy software more quickly and more safely—to production. Indeed, the very measure of success is defined by how quickly software is delivered to customers working—and working reliably. It's easy to forget this, but unless your customers can use it, it's not shipped. Software in production provides a vital feedback loop that helps businesses better react to market forces. Software in production is the only differentiator for any software business and—as my friend Andrew Clay Shafer reminds us—"you are either building a software business, or you will be losing to someone who is."

Is this news? No, of course not. A full-throated advocate of winning knows that the one constant in business is change. The winners in today's ecosystem learned this early and quickly.

One such example is Amazon. They realized early on that they were spending entirely too much time specifying and clarifying servers and infrastructure with operations instead of deploying working software. They collapsed the divide and created what we now know as Amazon Web Services (AWS). AWS provides a set of well-known primitives, a *cloud*, that any developer can use to deploy software faster. Indeed, the crux of the DevOps movement is about breaking down the invisible wall between what we knew as developers and operations to remove the cost of this back-and-forth.

Another company that realized this is Netflix. They realized that while their developers were using TDD and agile methodologies, work spent far too long in queue, flowing from isolated workstations—product management, UX, developers, QA, various admins, etc.—until finally it was deployed into production. While each workstation may have processed its work efficiently, the clock time associated with all the queueing meant that it could sometimes be *weeks* (or, *gulp*, more!) to get software into production.

In 2009, Netflix moved to what they described as the *cloud-native architecture*. They decomposed their applications and teams in terms of features; small (small enough to be fed with *two pizza-boxes!*) collocated teams of product managers, UX, developers, administrators, etc., tasked with delivering one *feature* or one independently useful product. Because each team delivered a set of free-standing services and applications, individual teams could iterate and deliver as their use cases and business drivers required, independently of each other. What were in-process method invocations

became independently deployed network services. Microservices, done correctly, hack Conway's law and refactor organizations to optimize for the continuous and safe delivery of small, independently useful software to customers. Independently deployed software can be more readily scaled at runtime. Independently deployed software formalizes service boundaries and domain models; domain models are forced to be internally consistent, something Dr. Eric Evans refers to as a *bounded context* in his epic tome, Domain Driven Design.

Independent deployability implies agility but *also* implies complexity; as soon as network hops are involved you have a distributed systems problem!

RIDE THE RIDE

Thankfully, we don't have to solve the common distributed systems problems ourselves! The giants of the web who've come before and won have shared a lot of what they've done, and the rest of us—forever on the cusp of the next big viral mobile app or social-network runner—can learn from and lean on what they've provided. Let's look at some of the common patterns and various approaches to using them.

CONSISTENCY IMPROVES VELOCITY

My friend and former colleague Dave McCrory coined the idea of *data gravity*—the inclination for pools of data to attract more and more data. The same thing—*monolith gravity*—exists with existing monolithic applications; any existing application will have inertia. A development team will face less friction in adding new endpoints and tables in a large SQL database to that application if the cost associated with standing up new services is significant. For many organizations, standing up new services can be a daunting task indeed! Many organizations have wiki pages with dozens of steps that must be carried out before a service can be deployed, most of which have little or nothing to do with the services' business value and drivers!

> **Thankfully, we don't have to solve the common distributed systems problems ourselves**

Microservices are APIs, typically REST APIs. How quickly can you stand up a new REST service? Microframeworks like Spring Boot, Grails, Dropwizard, Play framework, and Wildfly Swarm are optimized for quickly standing up REST services with minimum fuss. Extra points go to technologies that make it easy to build smart, self-describing *hypermedia* APIs as Spring Boot does with Spring HATEOAS.

Services are tiny, ephemeral, and numerous. The economics that made deploying lots of applications into the same JVM and application server interesting 15 years ago are long since gone for most of us, and most environments these days embrace *process concurrency* and isolation. What this means, in practice, is self-contained *fat jars*, which all of the aforementioned web frameworks will easily create for you.

You can't fix what you can't measure: how quickly can a service expose application state—metrics (gauges, meters, histograms, and counters), health checks, etc.—and how easy is it to report microservice state in a joined-up view or analysis tool like StatsD, Graphite, Splunk, the ELK (Elastic Search/Logstash/Kibana) stack, or OpenTSDB? One framework that brought metrics and log reporting to the forefront is the Dropwizard microframework. Spring Boot's *Actuator* module provides many of the same capabilities (and in some cases more) and transparently integrates with the Dropwizard Metrics library if it's on the `CLASSPATH`. A good platform like Cloud Foundry will also make centralized log collection and analysis dead simple.

Getting all of this out of the box is a good start, but it's not enough. There is often much more to be done before a service can get to production. Spring Boot uses a mechanism called auto-configuration that lets developers codify things—identity provider integrations, connection pools, frameworks, auditing infrastructure, literally *anything*—and have it stood up as part of the Spring Boot application (*if* all the conditions stipulated by the auto-configuration are met) just by being on the `CLASSPATH`! These conditions can be anything, and Spring Boot ships with many common and reusable conditions: is a library on the `CLASSPATH`? Is a bean of a certain type defined (or not defined)? Is an environment property specified?

Starting a new service need not be more complex than a `public static void main` entry-point and a library on the `CLASSPATH` if you use the right technology.

CENTRALIZED CONFIGURATION

The 12 Factor Manifesto provides a set of guidelines for building applications with good cloud hygiene. One of the guidelines is to externalize configuration from the build so that one build of the final application can be promoted from development, QA, integration testing, and finally to a production environment. Environment variables and `-D` arguments, externalized `.properties`, and `.yml` files—which Dropwizard, Spring Boot, Apache Commons Configuration and others readily support—are a good start, but even this can become tedious as you need to manage more than a few instances of a few types of services. This approach also fails several key use cases. How do you change configuration centrally and propagate those changes? How do you support symmetric encryption and decryption of things like connection credentials? How do you support *feature flags*, which toggle configuration values at runtime, without restarting the process?

Spring Cloud provides the Spring Cloud Config Server, which stands up a REST API in front of a version controlled repository of configuration files. Spring Cloud also provides support for using Apache Zookeeper and Hashicorp Consul

as configuration sources, as well as various clients for all of these so that all properties—whether they come from the Config Server, Consul, a -D argument, or an environment variable—work the same way for a Spring client. Netflix provides a solution called Archaius, which acts as a client to a pollable configuration source. This is a bit too low-level for many organizations and lacks a supported, open-source configuration source counterpart, but Spring Cloud bridges the Archaius properties with Spring's as well.

SERVICE REGISTRATION AND DISCOVERY

Applications spin up and down, and their locations may change. For this reason DNS—with its time-to-live expiration values—may be a poor fit for service discovery and location. It's important to decouple the client from the location of the service; a little bit of indirection is required. A service registry adds that indirection. A service registry is a phonebook, letting clients look up services by their logical names. There are many such service registries out there: some common examples include Netflix's Eureka, Apache Zookeeper, and Hashicorp Consul. Modern platforms like Cloud Foundry don't necessarily need a separate service registry because of course it already knows where services live and how to find them given a logical name. At the very least, all applications will read from a service registry to inquire where other services live. Spring Cloud's DiscoveryClient abstraction provides convenient client-side API implementations for working with all manner of service registries, be they Apache Zookeeper, Netfix Eureka, Hashicorp Consul, Etcd, Cloud Foundry, Lattice, etc. It's easy enough to plug in other implementations since Spring is a framework and a framework is (to borrow the Eiffel definition) *"open for extension."*

CLIENT-SIDE LOAD BALANCING

A big benefit of using a service registry is client-side load balancing. Client-side load balancing lets the client find all the relevant registered instances of a given service—if there are 10 or a thousand, they're all discovered through the registry—and then choose from among the candidate instances which one to route requests to. The client can programmatically decide, based on whatever criteria it likes—capacity, round-robin, cloud-provider availability-zone awareness, multi-tendency, etc.—to which node a request should be sent. Netflix provides a great client-side load balancer called Ribbon. Spring Cloud readily integrates Ribbon at all layers of the framework, so that whether you're using the RestTemplate, declarative REST clients powered by Netflix Feign, the Zuul microproxy, or anything else, the provided Ribbon load balancer strategy is in play automatically.

EDGE SERVICES: MICROPROXIES AND API GATEWAYS

Client-side load balancing is only used within the data center, or within the cloud, when making requests from one service to another. All of these services live behind the firewall. Services that live at the edge of the data center, exposed to public traffic, are exposed using DNS. An HTML5, Android, Playstation, or iPhone application will not use Ribbon. Services exposed at the edge have to be more defensive; they cannot propagate exceptions to the client. Edge services are intermediaries, and an ideal place to insert API translation or protocol translation. Take for example an HTML5 application. An HTML5 application can't run afoul of CORS restrictions: it must issue requests to the same host and port. A possible route might be to add a policy to every backend microservice that lets the client make requests. This of course is untenable and unscalable as you add more and more microservices. Instead, organizations like Netflix use a microproxy like Netflix's Zuul. A microproxy like Zuul simply forwards all requests at the edge service to the backend microservices as enumerated in a registry. If your application is an HTML5 application, it might be enough to stand up a microproxy, insert HTTP BASIC or OAuth security, use HTTPS, and be done with it.

Sometimes the client needs a coarser-grained view of the data coming from the services. This implies API translation. An edge service, stood up using something like Spring Boot, might use Reactive programming technologies like Netflix's RxJava, Typesafe's Akka, RedHat's Vert.x, or Pivotal's Reactor to compose requests and transformations across multiple services into a single response. Indeed, all of these implement a common API called the reactive streams API because this subset of problems is so common.

CLUSTERING PRIMITIVES

In complex distributed systems, there are many actors with many roles to play. Cluster coordination and cluster consensus is one of the most difficult problems to solve. How do you handle leadership election, active/passive handoff, or global locks? Thankfully, many technologies provide the primitives required to support this sort of coordination, including Apache Zookeeper, Redis,and Hazelcast. Spring Cloud's Cluster support provides a clean integration with all of these technologies.

> **"Transactions are a stop-the-world approach to state synchronization and slow the system as a whole"**

MESSAGING, CQRS, AND STREAM PROCESSING

When you move into the world of microservices, state synchronization becomes more difficult. The reflex of the experienced architect might be to reach for distributed transactions, a la JTA. Ignore this urge at all costs. Transactions are a stop-the-world approach to state synchronization and slow the system as a whole—the worst possible outcome in a distributed system. Instead, services today use *eventual* consistency through messaging to ensure that state eventually reflects the correct system worldview. REST is a fine technology for *reading* data but it doesn't provide any guarantees about the propagation and

eventual processing of a transaction. Actor systems like Typesafe Akka and message brokers like Apache ActiveMQ, Apache Kafka, RabbitMQ, or even Redis have become the norm. Akka provides a supervisory system that guarantees a message will be processed at least once. If you're using messaging, there are many APIs that can simplify the chore, including Apache Camel, Spring Integration, and—at a higher abstraction level and focusing specifically on the aforementioned Kafka, RabbitMQ, and Redis—Spring Cloud Stream. Using messaging for writes and using REST for reads optimizes reads separately from writes. The Command Query Responsibility Segregation, or CQRS, design pattern specifically describes this approach.

CIRCUIT BREAKERS

In a microservice system it's critical that services be designed to be fault-tolerant: if something happens, then the services should gracefully degrade. Systems are complex, living things. Failure in one system *can* trigger a domino effect across other systems if care isn't taken to isolate them. One way to prevent failure cascades is to use a *circuit-breaker*. A circuit-breaker is a stateful component around potentially shaky service-to-service calls that—when something goes wrong—prevents further traffic across the downed path. The circuit will slowly attempt to let traffic through until the pathway is closed again. Netflix's Hystrix circuit-breaker is a very popular option, complete with a usual dashboard which can aggregate and visualize potentially open circuits in a system. Wildfly Swarm, as of this writing in Q3 2015, has support for using Hystrix in master, and the Play Framework provides support for circuit breakers. Naturally, Spring Cloud also has deep support for Hystrix, and we're investigating a possible integration with JRugged.

DISTRIBUTED TRACING

A microservice system with REST, messaging, and proxy egress and ingress points can be very hard to reason about in the aggregate: how do you trace—correlate requests across a series of services and understand where something may have failed? This is very difficult without a sufficient upfront investment in a tracing strategy. Google's Dapper first described such a distributed tracing tool. Google's Dapper paper paved the way for many other such systems, including one at Netflix, which they have not open-sourced. Apache HTRace is also Dapper-inspired. Twitter's Zipkin is open-source and actively maintained. It provides the infrastructure and a visually appealing dashboard on which you can view waterfall graphs of calls across services. Spring Cloud has a module called Spring Cloud Sleuth that provides correlation IDs and instrumentation across various components. Spring Cloud Zipkin integrates Twitter Zipkin in terms of the Spring Cloud Sleuth API. Once added to a Spring Cloud module, requests across messaging endpoints using Spring Cloud Stream, REST calls using the RestTemplate, and HTTP requests powered by Spring MVC are all transparently and automatically traced.

SINGLE SIGN-ON

Security is hard. In a distributed system, it is critical to ascertain the provenance and authenticity of a request in a consistent way across all services, quickly. On the open web, OAuth and OpenID Connect are very popular. In the enterprise, technologies like SAML are very popular. OAuth 2 provides explicit integration with SAML. API gateway tools like Apigee and SaaS identity providers like Stormpath can act as a security hub, exposing OAuth (for example) and connecting the backend to more traditional identity providers like ActiveDirectory, SiteMinder, or LDAP. Finally, Spring Security OAuth provides an identity server, which can then talk to any identity provider in the backend. Whatever your choice of identity provider, it should be trivial to protect services based on some sort of token. Spring Cloud Security makes short work of protecting any REST API with tokens from *any* OAuth 2 provider—Google, Facebook, the Spring Security OAuth server, Stormpath, etc. Apache Shiro can also act as an OAuth client using the Scribe OAuth client.

> "Independently deployed software formalizes service boundaries and domain models"

DON'T REINVENT THE RIDE'S WHEELS

We've looked at a fairly extensive list of concerns that are unique to building cloud-native applications. Trust me: you do *not* want to reinvent this stuff yourself. There's a lot of stuff to care for, and unless you've got Netflix's R&D budget (and smarts!), you're not going to get there anytime soon. Building the pieces is one thing, but pulling them together into one coherent framework? That's a whole other kettle of fish and few pull it off well. Indeed, even the likes of Netflix, Alibaba, and TicketMaster are using Spring Cloud (which builds on Spring Boot) because it removes so much complexity and lets developers focus on the essence of the business problem.

In a sufficiently distributed system, it is increasingly futile to optimize for high availability and paramount to optimize for reduced time-to-remediation. Services *will* fall down; the question is: how quickly can you stand them up again? This is why microservices and cloud-computing platforms go hand-in-hand: as the platform picks up the pieces, the software needs to be smart enough to adapt to the changing landscape, and to degrade gracefully when something unexpected happens.

JOSH LONG (@starbuxman) is the Spring Developer Advocate at Pivotal. Josh is a Java Champion, author of 5 books (including O'Reilly's upcoming "Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry") and 3 best-selling video tutorials (including "Building Microservices with Spring Boot Livelessons" with Phil Webb), and an open-source contributor (Spring Boot, Spring Integration, Spring Cloud, Activiti, and Vaadin). Josh has been a keynote presenter at dozens of software shows worldwide.

CHECKLIST

JAVA BEST PRACTICES

This checklist includes a mix of general advice and specific pitfalls that you need to look out for when developing Java applications.

GENERAL

☐ USE ENCAPSULATION

None of the instance fields in your class should be public. Utilize getters to allow access to the fields, and when appropriate, add setters to allow writing.

☐ CLOSE STREAMS

Any time you open a stream to read or write data, ensure that you close that stream afterwards. If you're using Java 7 or later, you can use the "try with resources" mechanism.

☐ USE EQUALITY OPERATORS

Remember that when comparing strings you should use the `.equals` method rather than the == operator, which compares the addresses of objects rather than the contents.

If you're using `String.intern()` you can use the == operator, but you should only use `.intern` sparingly since it can increase overhead during garbage collection.

PERFORMANCE

☐ USE PRIMITIVE TYPES

Java has a number of wrapper classes available for dealing with primitives, but use these only if they are absolutely necessary for your application. Wrapper classes are slow, while primitive types are simple values.

☐ CONCATENATE WITH STRINGBUILDER

When building up a string in your program, use `StringBuilder` rather than simple concatenation (with the + operator), which would create a new `String` object each time it is used.

☐ CONSTRUCT OBJECTS LAZILY

Almost all the time, you should prefer lazy initialization of your objects. If your object is constructed with data from an external system, initialization will be slowed, so delaying construction until it is required will result in faster overall execution.

EXCEPTION HANDLING

☐ BE SPECIFIC WITH EXCEPTIONS

As you create methods that throw exceptions, think of yourself as an API developer, and make the `throws` clause as specific as possible. It's better to list three potential exceptions rather than having one generic exception.

☐ ELIMINATE EMPTY CATCH BLOCKS

Empty catch blocks are useless for the readability of your class. You should always attempt to do something in your catch blocks. At the very least this should include logging the exception and should possibly inform the user.

☐ USE "TRY WITH RESOURCES"

In try-catch blocks that include some work with I/O resources, use the "try with resources" syntax to ensure that all resources are closed when the block has completed execution. If you are using a version of Java below Java SE 7, then use the *finally* block to clean up resources.

☐ NEVER THROW EXCEPTIONS FOR NORMAL CONTROL FLOW

You should never rely on exceptions to catch issues such as an array index out of bounds, when instead you can use defensive programming in your code to check if the index is within the size of the array. Exception checks run slower than the checks you can do within your own code.

WRITTEN BY: James Sugrue is the CIO at Carma, and has been a Zone Leader at DZone since 2008. This list was compiled from James' experiences with Java across a 14 year career, both using Java in practice and reading countless books and articles on the language. He recommends that every Java developer should read *Effective Java* by Joshua Bloch.

Solutions Directory

This directory contains Java web frameworks, build automation/repository management tools, in-memory data grids, Java monitoring tools, application servers, IDEs, and portals/CMS. It provides free trial data and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

PRODUCT	CATEGORY	FREE TRIAL	WEBSITE
AngularFaces	AngularJS + JSF Framework	Open Source	angularfaces.com
Apache Ant	Build Automation	Open Source	ant.apache.org
Apache Ignite	In-memory Data Grid	Open Source	ignite.apache.org
Apache Ivy	Dependency Manager	Open Source	ant.apache.org/ivy
Apache Maven	Build, Reporting and Documentation Automation	Open Source	maven.apache.org
Apache MyFaces	JSF Framework	Open Source	myfaces.apache.org
Apache Struts	Java Web Framework	Open Source	struts.apache.org
Apache Tapestry	Component-oriented Java Web Framework	Open Source	tapestry.apache.org
Apache Tomcat	Java Web Server	Open Source	tomcat.apache.org
Apache Wicket	Java Web Framework	Open Source	wicket.apache.org
AppDynamics	APM	Free Tier Available	appdynamics.com
Artifactory by JFrog	Binary/Artifact Repository	Open Source	jfrog.com/open-source
Big Memory Max by Terracotta	In-memory Data Grid	90 Days	terracotta.org/products/bigmemory
CA Application Monitoring	APM	30 Days	ca.com
Censum by jClarity	GC Analysis	7 Days	jclarity.com
Codenvy IDE	SaaS IDE	Free Tier Available	codenvy.com
CUBA Platform by Haulmont Technologies	Java Framework	Free Tier Available	cuba-platform.com
DCHQ	Cloud Automation Platform for Container-based Applications	Free Tier Available	dchq.io
DripStat	Java + Scala APM	Free Tier Available	dripstat.com
Dropwizard	Java REST Web Services Framework	Open Source	dropwizard.io
Dynatrace Application Monitoring	APM	30 Days	dynatrace.com
Eclipse	IDE	Open Source	eclipse.org

PRODUCT	CATEGORY	FREE TRIAL	WEBSITE
Finatra	Scala HTTP services built on Twitter-Server and Finagle	Open Source	twitter.github.io/finatra
GemFire by Pivotal	In-memory Data Grid	Open Source	pivotal.io/big-data/pivotal-gemfire
Google Web Toolkit	Single-page Web Framework	Open Source	gwtproject.org
Gradle	Build Automation	Open Source	gradle.org
Grails	Groovy Web Framework	Open Source	grails.org
GridGain	In-memory Data Grid	Free Tier Available	gridgain.com
Hazelcast Enterprise Platform	In-memory Data Grid	30 Days	hazelcast.com
Heroku Platform	PaaS	Free Tier Available	heroku.com
IceFaces by IceSoft	JSF Framework	Open Source	icesoft.org
Illuminate by jClarity	Java Performance Monitoring	14 Days	jclarity.com
Infinispan by Red Hat	In-memory Data Grid and Cache	Open Source	infinispan.org
IntelliJ IDEA by JetBrains	IDE	Free Tier Available	jetbrains.com/idea
ItsNat	Component-oriented Java Web Framework	Open Source	itsnat.sourceforge.net
Java Server Faces by Oracle	Java Web Framework	Open Source	oracle.com
JBoss Data Grid by Red Hat	In-memory Data Grid	Free Tier Available	redhat.com
JDeveloper by Oracle	IDE	Free Product	oracle.com
Jenkins Platform by CloudBees	Continuous Integration Platform	2 Weeks	cloudbees.com
jHiccup by Azul Systems	Java Performance Monitoring	Open Source	azulsystems.com
JMS Adapters for .NET or BizTalk by JNBridge	JMS Integration with .NET or BizTalk	30 Days	jnbridge.com
JNBridgePro	Java and .NET Interoperability	30 Days	jnbridge.com
JRebel by ZeroTurnaround	Java Class Reloader	Free Tier Available	zeroturnaround.com/software/jrebel
Lift	Scala Web Framework	Open Source	liftweb.net
MyEclipse by Genuitec	IDE	30 Days	genuitec.com/products/myeclipse
NetBeans by Oracle	IDE	Open Source	netbeans.org
New Relic	APM	14 Days	newrelic.com
Nexus by Sonatype	Binary/Artifact Repository	Open Source	sonatype.org/nexus
Oracle Coherence	In-memory Data Grid	Open Source	oracle.com

PRODUCT	CATEGORY	FREE TRIAL	WEBSITE
Payara Micro	Java EE Deployment Container	Open Source	payara.fish/home
Play! Framework	Java + Scala Web Framework	Open Source	playframework.com
Plumbr	Memory Leak Detection, GC Analysis, Thread & Query Monitoring	Available Upon Request	plumbr.eu
PrimeFaces	JSF Framework	Open Source	primefaces.org
Race Catcher by Thinking Software	Race Condition Debugger	N/A	thinkingsoftware.com
RichFaces by Red Hat	JSF Framework	Open Source	richfaces.jboss.org
Ring	Clojure Web Framework	Open Source	github.com/ring-clojure/ring
Ruxit APM	APM	30 Days	ruxit.com
Scalatra	Minimalistic Scala Web Framework	Open Source	scalatra.org
SmartGWT by Isomorphic Software	Single-page Web Framework + Prototyping Tools	60 Days	smartclient.com
Spray	Akka/Scala Web Framework	Open Source	spray.io
Spring Boot	Minimalistic Spring Bootstrapping Framework	Open Source	projects.spring.io/spring-boot
Spring Cloud by Pivotal	PaaS	Open Source	cloud.spring.io
Spring MVC	Spring Web Framework	Open Source	spring.io
SteelCentral by Riverbed Technologies	APM	14 Days	riverbed.com
Takipi	Large-scale Java/Scala Production Debugging	Free Tier Available	takipi.com
Tasktop Dev	IDE Task Manager	30 Days	tasktop.com/tasktop-dev
Tayzgrid by Alachisoft	In-memory Data Grid	Open Source	tayzgrid.com
Upsource by JetBrains	Code Review	Free 10-User Plan	jetbrains.com/upsource
Vaadin	JVM Single-page Web Framework	Open Source	vaadin.com
Vert.x	Event-driven, non-blocking JVM App Framework	Open Source	vertx.io
VisualVM by Oracle	JVM Monitoring	Open Source	visualvm.java.net
WebSphere eXtreme Scale by IBM	In-memory Data Grid	8 Hours	ibm.com
WildFly Swarm by Red Hat	Minimalistic Application Server	Open Source	wildfly.org/swarm
XRebel by ZeroTurnaround	Java Profiler	14 Days	zeroturnaround.com
YourKit	Java & .NET Profiler	15 Days	yourkit.com
Zing by Azul Systems	JVM Platform	Free Tier Available	azulsystems.com
ZK Framework by ZKoss	Java Web Framework	Open Source	zkoss.org

diving deeper

INTO FEATURED JAVA ECOSYSTEM SOLUTIONS

Looking for more information on individual Java Ecosystem solutions providers?
Nine of our partners have shared additional details about their offerings, and we've summarized this data below.

If you'd like to share data about these or other related solutions, please email us at research@dzone.com.

CloudBees Jenkins Platform
BY CLOUDBEES

DESCRIPTION The CloudBees Jenkins Platform is a software solution from CloudBees Inc. based on the Jenkins automation engine technology which enables IT organizations to easily adopt the Continuous Delivery model for the application development and delivery lifecycle. Continuous Delivery is often the foundation for a DevOps transformation. The CloudBees Jenkins Platform enables CD at enterprise scale.

OPEN SOURCE? Yes

PRICING cloudbees.com/products/pricing

Heroku Platform
BY SALESFORCE

DESCRIPTION Heroku is a cloud platform based on a managed container system, with integrated data services and a powerful ecosystem, for deploying and running modern apps. The Heroku developer experience is an app-centric approach for software delivery, integrated with today's most popular developer tools and workflows.

FREE TRIAL Free tier available

PRICING heroku.com/pricing

jHiccup
BY AZUL SYSTEMS

DESCRIPTION jHiccup is an Open Source tool developed by Azul CTO Gil Tene that provides a quick picture of your applications as they run in production. jHiccup generates "hiccup charts" that quickly show the latency inherent in your system, to the 99.9999th percentile. Hiccup charts also show the system latency across the entire test interval. jHiccup has minimal overhead, is written in Java, and it is free.

FREE TRIAL Open source

PRICING azulsystems.com/product/jHiccup

JMS Adapters for .NET and BizTalk
BY JNBRIDGE

DESCRIPTION Quickly create .NET-based apps (in C#, VB, etc.) that exchange messages with your existing JMS infrastructure using the JNBridge JMS Adapter for .NET.

Quickly integrate BizTalk Server into your existing JMS infrastructure using the JMS Adapter for BizTalk.

The Adapters work with any vendors JMS implementation, including WebSphere, WebLogic, ActiveMQ, GlassFish/ OpenMQ, and JBoss OpenJMS.

FREE TRIAL 30 day trial, full-featured

PRICING jnbridge.com/purchase

JRebel
BY ZEROTURNAROUND

DESCRIPTION JRebel is a JVM Java Agent that integrates with application servers, making classes reloadable with existing class loaders. Only changed classes are recompiled and instantly reloaded in the running application. It plugs into IDEs and build systems. Classes and static resources are loaded straight from the workspace and integrates with over 100 frameworks.

FREE TRIAL 14 day trial, full-featured

PRICING zeroturnaround.com/software/jrebel/pricing/

New Relic
BY NEW RELIC

DESCRIPTION New Relic APM surfaces issues you cannot see coming, helping your team reduce resolution times so they can focus on writing new code, not troubleshooting it. New Relic gives teams visibility into their microservice architectures, transactions, and code. The tool is secure and features alerts that integrate with popular chat software as well as SLA reporting.

FREE TRIAL Free tier available, with a 14 day trial

PRICING newrelic.com/application-monitoring/pricing

Spring Cloud
BY PIVOTAL

DESCRIPTION Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).

FREE TRIAL Open source

PRICING cloud.spring.io

SteelCentral
BY RIVERBED TECHNOLOGIES

DESCRIPTION SteelCentral is the only end-to-end solution that combines user experience, application, infrastructure, and network monitoring for comprehensive visibility and diagnostics, and centralized control. Development, operations, and support teams rely on SteelCentral to quickly expose, diagnose, and fix problems throughout the application lifecycle.

FREE TRIAL 14 day trial, full-featured

PRICING riverbed.com/appinternals/buy

Upsource
BY JETBRAINS

DESCRIPTION Upsource is an on-premises code review tool that supports Git, Mercurial, Subversion, and Perforce. It helps development teams improve code quality, learn from each other, and build up collective code ownership through effective lightweight code reviews and transparent collaboration. It is the only code review tool that provides Java code insight to drastically simplify the code review process.

FREE TRIAL Free for up to 10 users, 60 days evaluation available on request for unlimited number of users

PRICING jetbrains.com/upsource/buy

glossary

APPLET A program written in Java for use on the web, usually embedded in an HTML page.

BEAN Also known as JavaBeans, these are objects that conform to some simple conventions such as property naming and data encapsulation.

ENTERPRISE JAVABEANS (EJB) A key, annotation-based Java EE API that provides component services to POJOs.

GARBAGE COLLECTION An automatic memory management system used by the JVM to free heap space occupied by unused objects.

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) An application that provides a code editor, debugger, compiler, and many other possible features that make developers more productive and less likely to create errors when building software.

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS) A Java EE specification that uses annotations to help map resource classes (a POJO) as RESTful web resources, and also to pull information out of requests.

JAVA API FOR XML WEB SERVICES (JAX-WS) A Java EE specification that uses annotations to help map Java objects (POJOs) to SOAP-based web services.

JAVA ARCHIVE FILE (JAR) A package file format that bundles multiple Java class files along with embedded metadata in order to distribute applications and libraries on the Java platform.

JAVA ARCHITECTURE FOR XML BINDING (JAXB) A Java EE API for marshalling and unmarshalling Java objects into and out of XML.

JAVA CARD Provides a secure environment for applications that run on smart cards and other limited-memory devices.

JAVA COMMUNITY PROCESS (JCP) A process by which Java community members propose, discuss, and finalize specifications to improve and expand the Java platform.

JAVA COMPATIBILITY KIT (JCK) A bundle of tools, requirements, and a test suite used to certify a new Java platform implementation.

JAVA DATABASE CONNECTIVITY (JDBC) A Java standard that provides a call-level API that can access most SQL-based databases.

JAVA DEVELOPMENT KIT (JDK) A complete set of binaries that contain the JRE along with tools for developing, debugging, and monitoring Java applications.

JAVA ENTERPRISE EDITION (EE) An API and runtime environment for developing and running enterprise software, including web services and large scale network applications.

JAVA MESSAGING SERVICE (JMS) A Java EE API and messaging standard that facilitates message-oriented middleware functionality between components based on Java EE.

JAVA MICRO EDITION (ME) A solution for Java applications that run on embedded and mobile devices in the Internet of Things, including micro-controllers, sensors, set-top boxes, and printers.

JAVA NAMING AND DIRECTORY INTERFACE (JNDI) A set of APIs that enable object and data discovery and lookup for Java software clients.

JAVA PERSISTENCE API (JPA) A standard specification for the management of relational data in the form of object-relational mapping.

JAVA RUNTIME ENVIRONMENT (JRE) A set of binaries that allows you to execute Java programs on your machine.

JAVA SERVER PAGES (JSP) A tool in the Java platform that allows developers to build and deliver HTML, XML, and other documents. JSPs are translated into servlets at runtime.

JAVA SPECIFICATION REQUEST (JSR) Written descriptions specifying proposed and final specifications for features of the Java platform.

JAVA STANDARD EDITION (SE) The most common platform used for development and deployment of Java applications for desktop and server environments.

JAVA VIRTUAL MACHINE (JVM) The runtime engine of the Java platform which takes programs written in Java and compiles them to bytecode. The JVM provides a way for developers to write code that is platform independent.

LAMBDA EXPRESSION In Java 8 terms: an anonymous method replacing a clunky inner class with lighter-weight, more readable syntax. Makes functional programming in Java much easier.

MICROSERVICES A system or application consisting of small, lightweight services that each perform a single function according to your domain's bounded contexts. The services are independently deployable and loosely coupled.

OBJECT-RELATIONAL MAPPING (ORM) An object-oriented programming technique for converting data between incompatible type systems. An ORM lets developers interact with a relational database from within a programming language without having to write direct code for the database.

OPENJDK The official open-source Java SE reference implementation.

PLAIN OLD JAVA OBJECT (POJO) A term used to emphasize that a given Java object is not a special object such as those defined by the EJB framework. A simple Java object that does not implement a framework interface or extend a framework class. As of Java EE 5, EJBs are merely annotated POJOs.

SERVLET A small Java program that runs within a web server and extends the capabilities of a server.

STREAM API A new Java 8 API for declarative, automatically parallelizable data processing on Collections. Massively simplifies SQL-like data querying and manipulation.

TECHNOLOGY COMPATIBILITY KIT (TCK) A suite of tests that are built and submitted in order to certify a JSR for compliance.

WEB APPLICATION ARCHIVE FILE (WAR) A type of JAR that bundles together resources used to build a web application. A WAR is deployable on any Java servlet.

CloudBees®

The Enterprise Jenkins Company

Continuous Delivery Powered by Jenkins

Your DevOps Foundation

Build → Test → Stage → Deploy

The Enterprise Jenkins Company

Learn more: https://www.cloudbees.com/jenkins-workflow