JVM Ecosystem Report 2021

Table of contents

Introduction 3
Report highlights 4
JDKs in production 5
JDKs in development 7
JDK versions in production environments 9
JDK versions in development 11
Java, Changing Faster Than Ever After 26 Years 12
JVM languages used for applications in production 14
Most popular IDEs 18
Tools for building applications 20
Securing vulnerabilities in the Java ecosystem with Snyk 22
Application frameworks 25
Demographics 26
The State of Spring 29

Introduction

Welcome to our annual JVM ecosystem report! This report presents the results of the largest annual survey on the state of the JVM ecosystem. The survey was conducted over a period of six weeks through February and March 2021, gathering the responses of over 2000 Java developers.

This year's survey is a cooperation between Snyk and Azul and was slightly different from the previous surveys. We aimed for the survey to be more concise and focus only on the most important aspects of JVM developers today. Additionally, this year every participant was allowed to choose multiple options. We believe that the way the 2021 survey was designed, we have a better and more comprehensive view of the current JVM ecosystem. In addition, in this report, we also looked at different open data sources like GitHub and Google Trends to see how that data compares to the survey results.

We would like to thank everyone who participated and offered their insights on Java and JVM-related topics. For this survey, we teamed up with conferences and communities across the JVM ecosystem to reach as many developers as possible. Big shoutout to Foojay.io, the VirtualJUG, and other Java communities for the invaluable help. This massive effort results in an impressive number of developers participating in the survey, giving great insight into the current state of the JVM ecosystem. You can find all demographic information at the end of this report.

Another big shoutout to Josh Long from Tanzu VMWare, and Simon Ritter from Azul for supplying great highlight stories on their field of expertise. This is something we couldn't get from a survey alone.

Happy reading!

snyk All rights reserved. 2021 © Snyk

Report highlights

Before we start, here's a TL;DR overview of the main highlights in this report.

44% of developers use AdoptOpenJDK builds in production

60% of developers use Java SE 11 in production

25% of developers use Java SE 15 in development

1 in 6 developers uses Kotlin, making it the second most popular language on the JVM

90% of developers use Java for application development

50% of developers use Spring Boot

75% of developers use Maven to build their projects

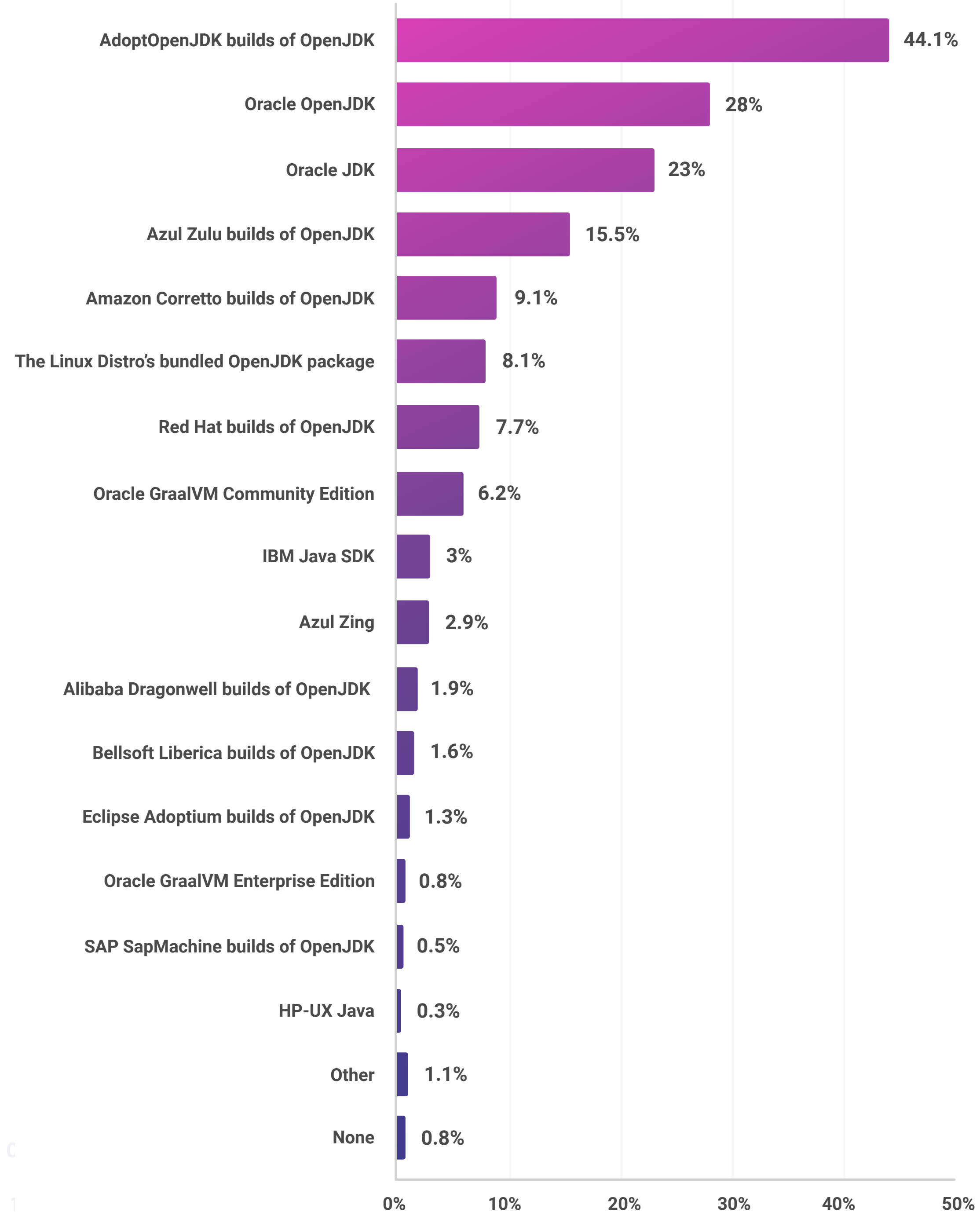
70% of developers use IntelliJ IDEA

 All rights reserved. 2021 © Snyk

JDKs in production

With a rising number of JDK suppliers and OpenJDK binaries, we traditionally kick off the report with the most critical question: Which JDKs are developers using in production?

Last year, we found out that many developers use more than one JDK in production. So this year, we changed this question (and almost all the other) to allow multiple answers. For this question, respondents could choose as many options as were applicable.



Multiple responses allowed.

snyk All rights reserved. 2021 © Snyk

Over 37% of our respondents say they are using at least two different JDKs, and 12.5% even use three or more different JDKs in production.

It is interesting to see that 44.1% of respondents use the free AdoptOpenJDK distribution in production, making it the most prevalent in our survey. However, we can also see that Oracle is still a big player in the market, with 28% for their OpenJDK build and 23% for the commercial Oracle JDK. The third most popular supplier of JDKs in production is Azul, at 15.5% adoption, and shouldn't be underestimated.

Google Trends indicate large interest in Oracle JDK

To see how our survey of usage compared with interest, we looked at Google search trends for AdoptOpenJDK, Oracle JDK, Oracle OpenJDK, and Azul Zulu over the first three months of 2021. Interestingly enough, Google Trends show a large community around Oracle JDK. On average, people search twice as much for Oracle JDK than the number two, AdoptOpenJDK. It is unclear, based on our data, whether this means that more people outside of this survey use Oracle JDK or that more people search for supporting documentation about Oracle JDK before getting started.

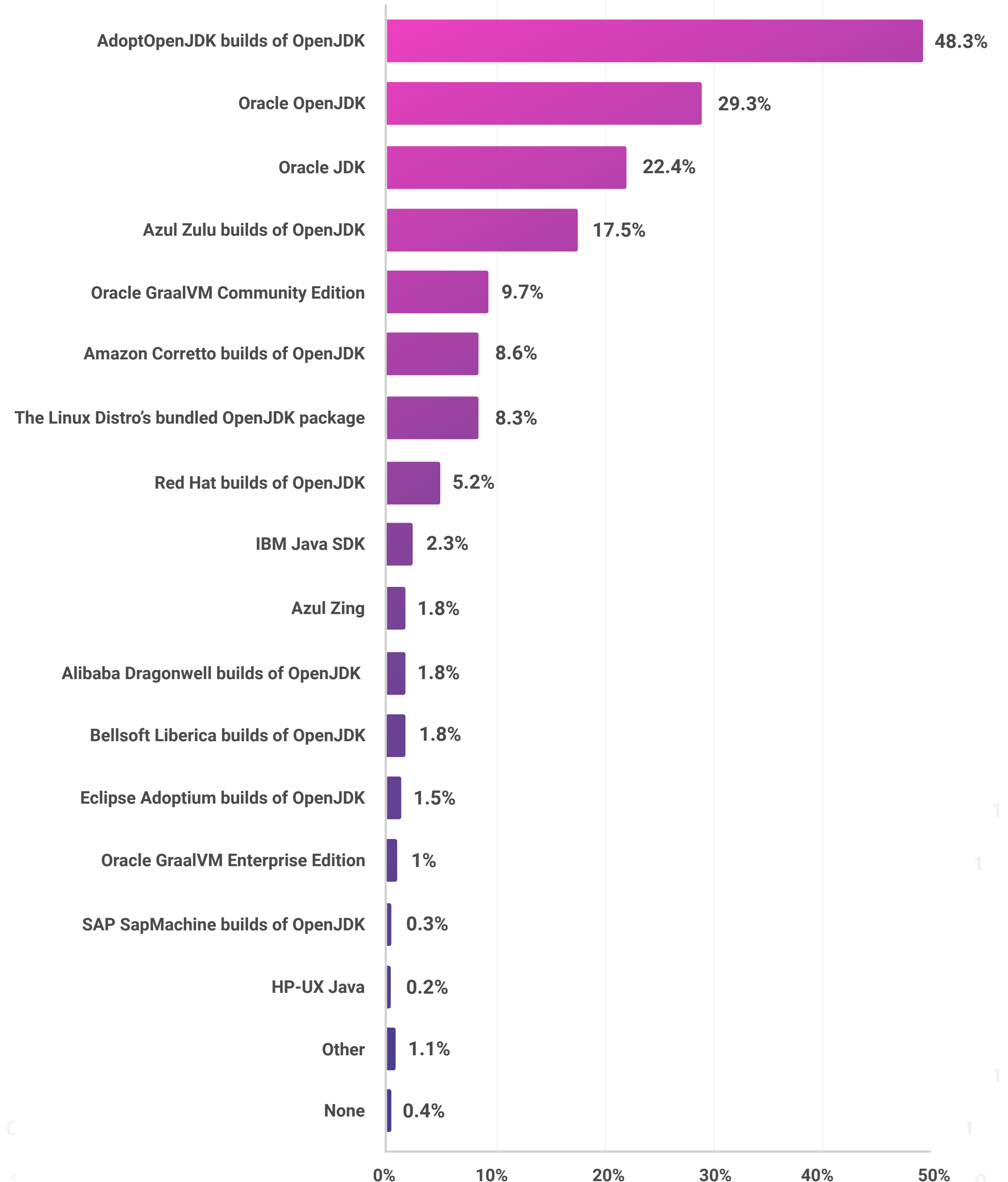
 All rights reserved. 2021 © Snyk

JDKs in development

We were also curious about the difference between the applications in production and development. Therefore, we asked the same question about the use of JDKs for development. Although we expected some differences between development environments and production, it turns out that the numbers are pretty similar.

Similar to production, the majority of the developers are using OpenJDK builds by AdoptOpenJDK in development. This accounts for almost half of the respondents!

In general, we see that the OpenJDK distributions are doing slightly better in development compared to production. Next to the 48.3% for AdoptOpenJDK, we also see slightly higher numbers for Oracle OpenJDK 29.3% and Azul Zulu 17.5%. The proprietary JDK builds like Oracle JDK 22.4% and Azul Zing 1.8% score slightly less than in development.



Multiple responses allowed.

snyk All rights reserved. 2021 © Snyk

All together, we still see that Oracle is still the most popular vendor with multiple different builds.

We also noticed a slight increase in the number of JDK's people used compared to production. Almost 39% of the respondents use two or more JDKs, and 14% use three or more different JDK suppliers in development. Still, all these numbers are just minor deviations, and the usage of JDKs in production and development can be considered very similar.

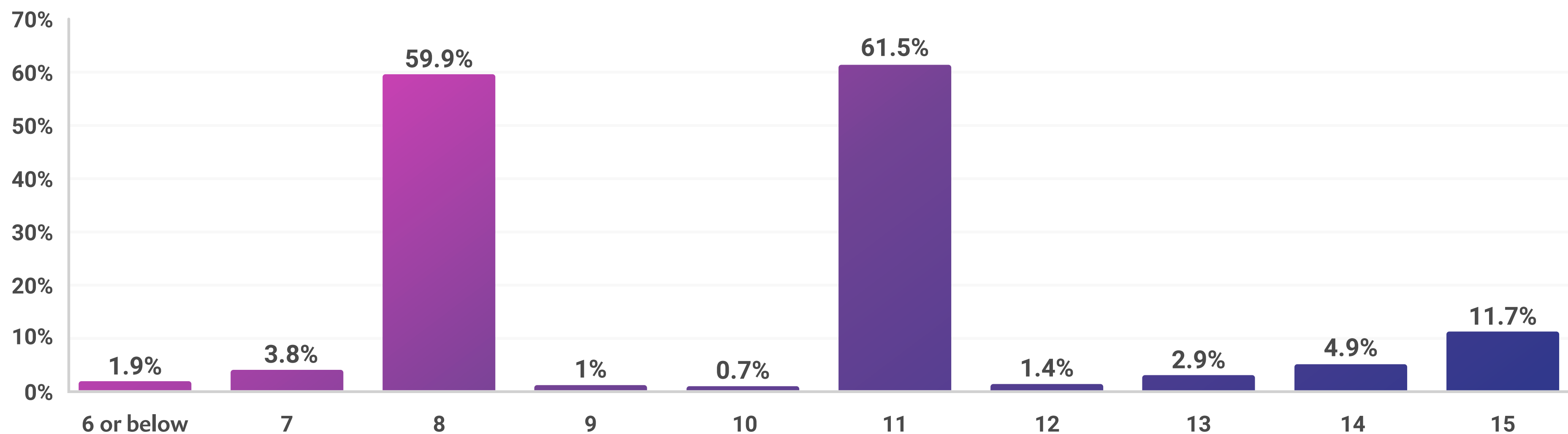
 All rights reserved. 2021 © Snyk

JDK versions in production environments

Another exciting question every year is the adoption rate of newer versions. Are developers working with the more recent version of Java, or are they unable or possibly unwilling to upgrade? In the last couple of years, we saw that developers were stuck at Java 8 and not adopting newer versions. This year we changed the question to submit the top three Java versions they use in production.

We found out that 40% of the survey participants use more than one Java version in production. Because of this, we can also conclude that more people than we realized do upgrade to versions beyond 8. Currently, 61.5% are using Java 11 somewhere in production, and almost 12% are using the latest release, which was Java 15 during the survey.

This is huge, because it shows that developers do upgrade their Java version beyond Java 8 to some extent. The mantra that most Java developers are comfortable staying on Java 8 seems to be slowly breaking apart.



Up to 3 responses allowed.

All rights reserved. 2021 © Snyk

We think it's great to see that older versions like Java 7 or below are no longer commonly used. Hopefully, just for that single legacy application hiding in a basement far, far away. We also see that non-LTS versions like Java 10, 12, and 13 are not heavily used in production. This might have something to do with the support of these versions. The general advice is either stay on the most recent LTS version, currently Java 11, or move to the latest release every six months. This trend is also clearly visible in the survey results.

We know that about 12% of the Java developers like to use the latest and greatest of Java. More than half of people who use Java 15 (7.8%) combined it with other Java versions. This means that almost 4% of Java developers are only using Java 15 in production and will most probably keep upgrading this to the latest version once it is released, which is quite astonishing.

It is also interesting to see that half of the Java 11 users — currently the most used version in production — also use Java 8 in their production stack. Cross-referencing the answers shows us that 30.2 % use both Java 8 and Java 11 in production. We can definitely conclude that the adoption of Java version beyond 8 is something that many developers want and try to do. That calls for a celebration! Also, it also confirms that developers need to maintain older applications that depend on legacy Java versions next to the newer, improved versions.

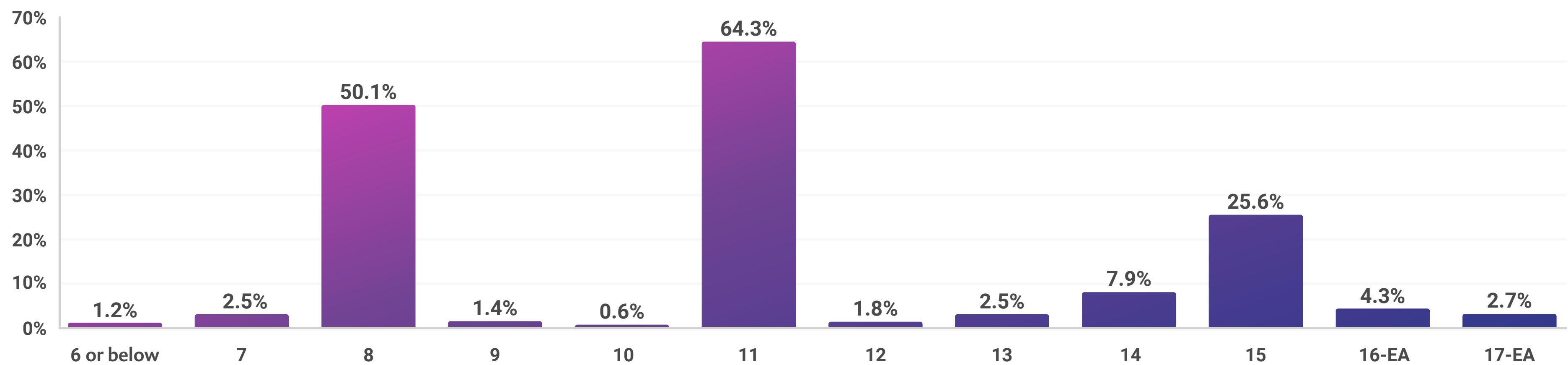
snyk All rights reserved. 2021 © Snyk

JDK versions in development

Similar to the JDK version in production, almost 65% of the Java developers use Java 11 in development next to possible other versions. Just like with the previous question, people were allowed to choose up to three options. Interestingly (and excitingly) enough, more people use Java 11 and fewer people use Java 8 in development.

This is a remarkable change in perspective and confirms again that developers want to migrate to newer Java versions beyond Java 8. Also, 25% of developers, which is quite a significant amount, are using the latest release version, Java 15, in development. The early access version Java 16-EA and Java 17-EA are unfortunately not yet as popular, but still used more than the ancient Java 7 and below.

With more than 25% of the Java developers using Java 15 and almost 65% using Java 11, we can confidently say that there is a clear shift away from Java 8. Although, Java 8 will undoubtedly be part of the developer's stack for some time.



Up to 3 responses allowed.

All rights reserved. 2021 © Snyk

Java, Changing Faster Than Ever After 26 Years

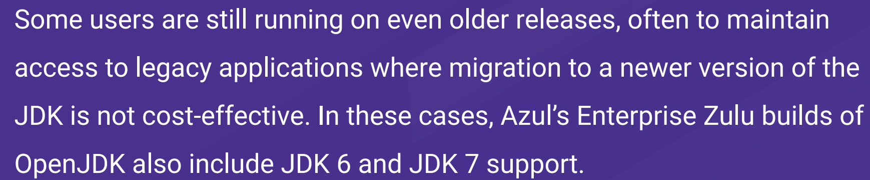
Simon Ritter
@speakjava
Deputy CTO at Azul Systems

It's amazing to think that Java turned 26 in May, and that the current release is JDK 16. Thanks to the change to a strict six-month release cadence, introduced in 2017, we are now seeing the platform develop faster than at any time in its history.

As developers, the challenge is keeping up to date with all the latest features and using them in our code. In this survey, we see that Java is still, by far, the most popular language for developing JVM-based applications. I'm sure the steady stream of minor enhancements to the core language will continue to maintain Java's popularity. I regularly give talks on how Java is evolving, and it is great to see how even small things can make developer's lives easier.

Records, introduced in JDK 14, is an excellent example of a powerful new feature added to the language without incurring any technical debt. We now have a very simple, straightforward way to define data classes in Java without any of the tedious boilerplate code we needed in the past. This feature also demonstrates another aspect of the new release cadence: preview features and incubator modules. Features like Records can be added to the JDK without immediately making them part of the Java SE standard. Based on feedback from developers, changes can be made where deemed appropriate (even to the point of removing the feature). Records were added as a preview feature and, in JDK 15, they were extended to allow the declaration of records inside a method (local records). This can be especially beneficial when using the Streams API and shows the power of a preview feature (which would not have been practical under the old, multi-year release cadence. In JDK 16, Records were promoted to a full language feature and included in the Java SE specification.

For those running enterprise-wide mission-critical applications, updating to a new version of Java every six months is unlikely to be an option. For those users, providers of OpenJDK distributions have followed the Oracle JDK to provide long-term support (LTS) releases. Thankfully, all OpenJDK distributions are aligned on which versions these are. JDK 11 is the current LTS, and we'll have the next one in September with the release of JDK 17. Many distributions also provide extended updates for JDK 8. The survey results show that over 60% of respondents are still using JDK 8 in production, so they will still require timely access to updates providing security patches and bug fixes.



Some users are still running on even older releases, often to maintain access to legacy applications where migration to a newer version of the JDK is not cost-effective. In these cases, Azul's Enterprise Zulu builds of OpenJDK also include JDK 6 and JDK 7 support.

To enable the widest selection of deployable Java runtimes, Azul offers what we call Medium Term Support (MTS) releases, currently for JDK 13 and 15. These can be deployed in production and include all updates until 18 months after JDK 17 is released. This enables users to take advantage of cool new features like Records and have plenty of time to migrate from JDK 15 to 17.

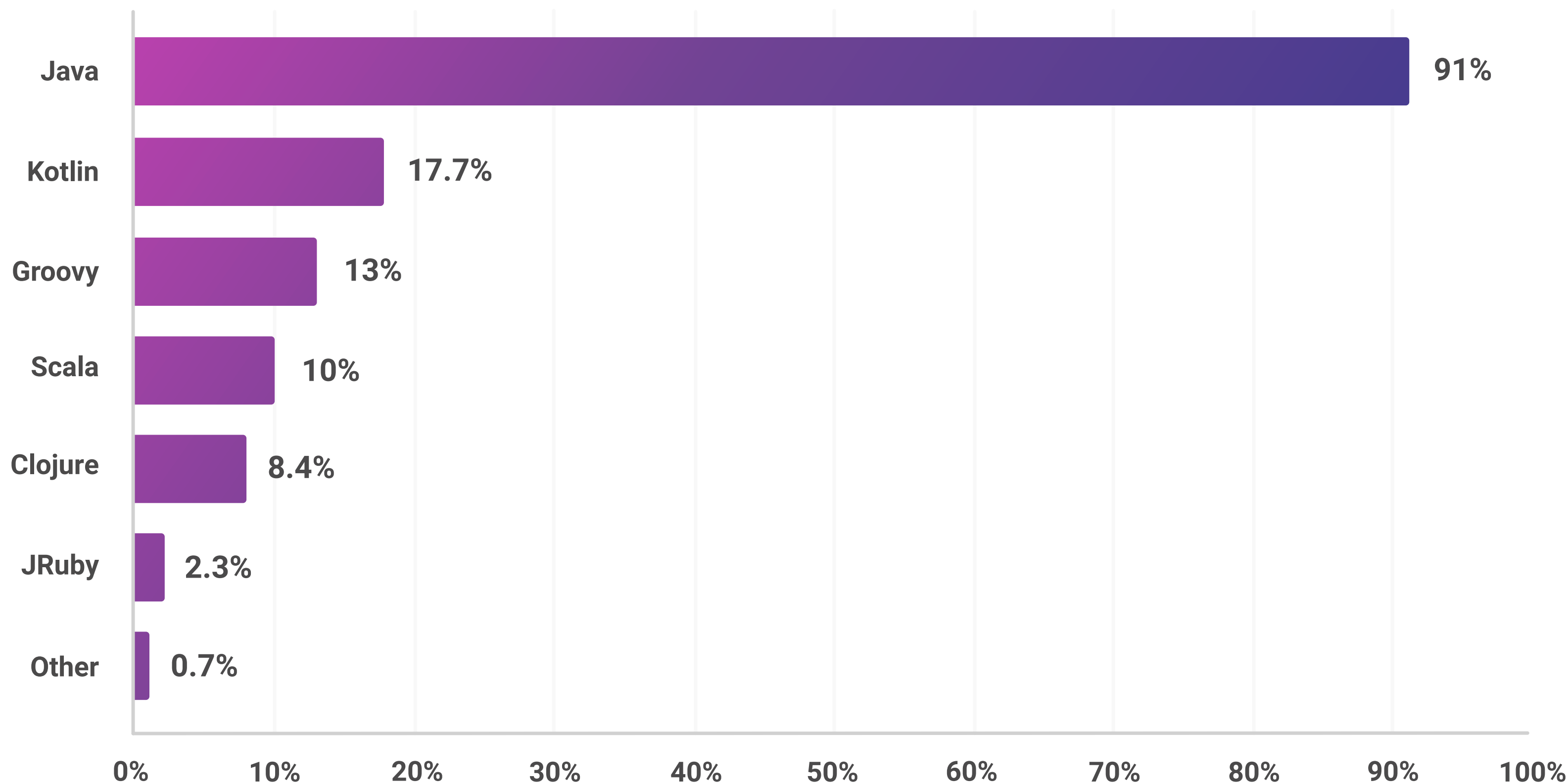
I'm sure I'm not the only person who believes that most of Java's continued popularity has resulted from the managed runtime nature of the Java Virtual Machine (JVM). Things like automated memory management and just-in-time (JIT) compilation delivers more reliable applications that often outperform statically compiled code. Developers are not constrained to Java but can compile languages like Kotlin and Scala to bytecodes and benefit from the JVM in the same way. That Java is easily the most popular JVM language is another testament to how well it was originally designed and has been thoughtfully developed.

This survey provides interesting insights into how both Java the language and Java the platform is being used. I think it's clear to see that, even after twenty-six years, Java shows no sign of slowing down or becoming less relevant to the needs of enterprise application developers. What's important with a new release every six months is to keep testing your application code with the latest version. This will make the switch to a new LTS release in production as simple as possible.

JVM languages used for applications in production

While the variety of JVM languages grew over the last couple of years, Java is very much on top. With over 90% of developers using Java, we can see that it remains a very popular language.

We see a consistently large percentage every year because many companies rely heavily on Java. In addition, we see that the constant development of the language helps maintain the level of popularity.



Multiple responses allowed.

All rights reserved. 2021 © Snyk

In last year’s report, we asked, “What is your main JVM language?” which allowed one pick. At that point, 86.9% chose Java. However, the follow-up questions here were, “Do you write or maintain any Java applications?” and “For those who don’t use Java in their main application, do they use it at all?” The combined result last showed us that 96% of the respondents used Java. This is very similar to the 91% we have this year and confirms the stable popularity and importance of the language.

The broader question we asked this year showed that the popularity of Kotlin is more extensive than we thought last year. With an impressive 17.7% of the developers using Kotlin in production, it has a steady second place. However, it is interesting to see that 15% of the developers use Kotlin together with Java. This is probably because of the great interoperability between Kotlin and Java, and likely the main reason for its success. It shows that Kotlin is not commonly used as the only JVM language in production for writing applications.

In general, we see that 33% use more than one JVM language in their stack, which is interesting because 58.8% only use Java without anything else. With 91% having Java in their stack, almost all users that use two or more languages have Java included.

 All rights reserved. 2021 © Snyk

Additional data sources

First, let's take a look at the TIOBE index. This index measures the popularity of all programming languages. TIOBE calculates the ratings based on the number of skilled engineers worldwide, courses, and third-party vendors. They use popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube, and Baidu to gather this data.

When looking at the top 50 languages in the April 2021 Tiobe index and filtering the JVM we see the following.

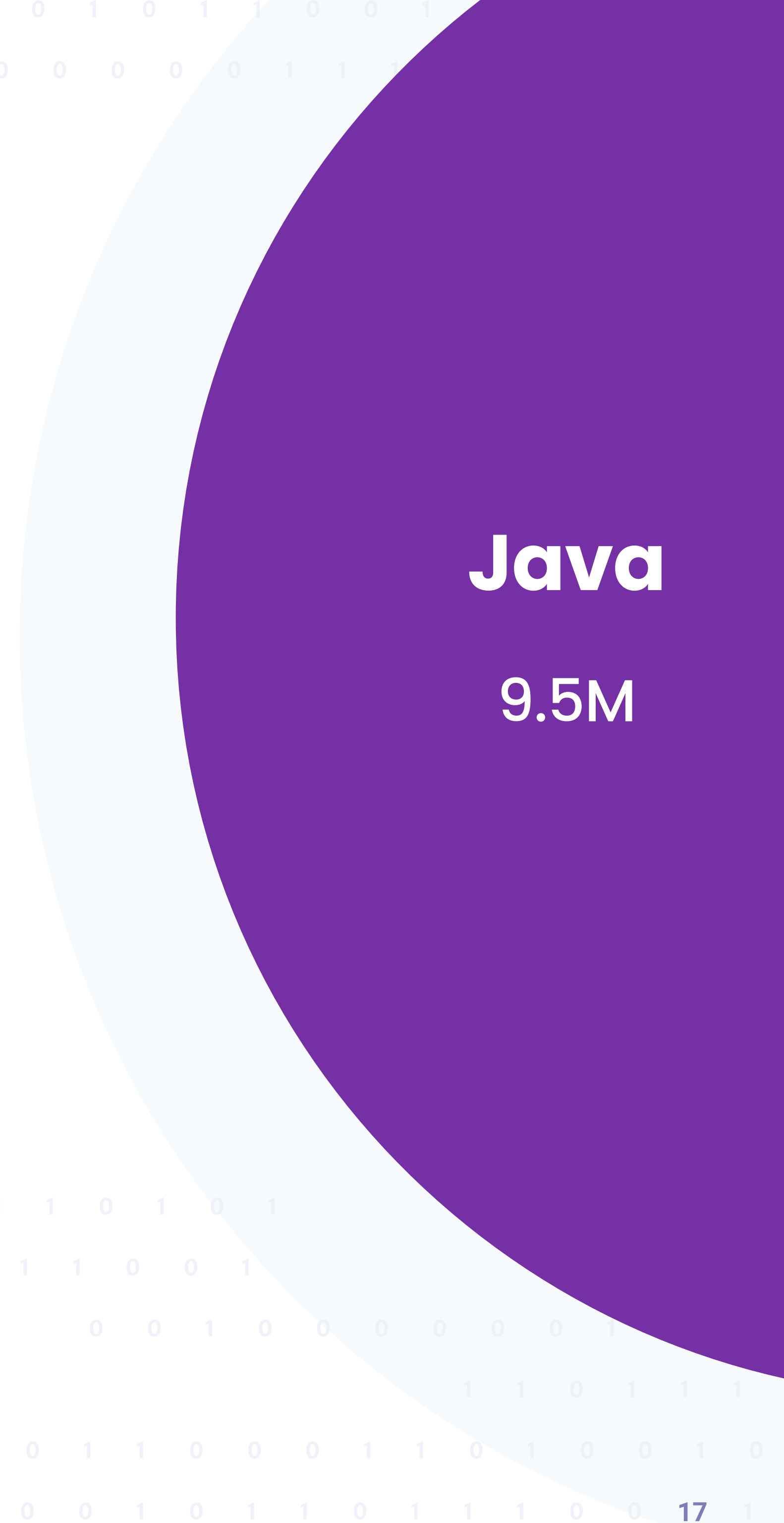
Position	Language	Rating
2	Java	11.23%
17	Groovy	1.04%
26	Scala	0.61%
39	Kotlin	0.32%
> 50	Clojure	

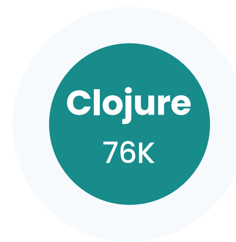
snyk All rights reserved. 2021 © Snyk

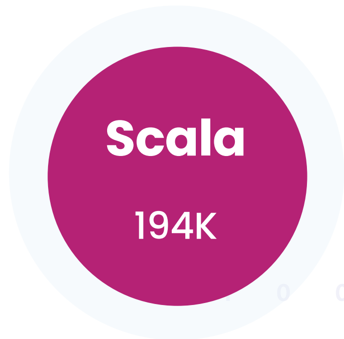
Number of repositories for JVM languages on Github

In early April, we also looked at GitHub and searched for the amount of repo's containing a certain JVM language. We used the GitHub API for this matter, as the results seemed more stable than a web search.

We can conclude that Java is, by far, the most popular language in all sources we consulted for this report, and it will probably stay this way in the foreseeable future. An interesting finding is the difference between the popularity of Scala, Groovy, and Kotlin. Both the amount of GitHub repositories and the survey results put Kotlin in second place. However, according to the TIOBE index, both Scala and Groovy are more popular. Nevertheless, we cannot deny that Kotlin is making a significant impact within the JVM landscape. And it undoubtedly stays this way or even grows more in the near future.







Groovy

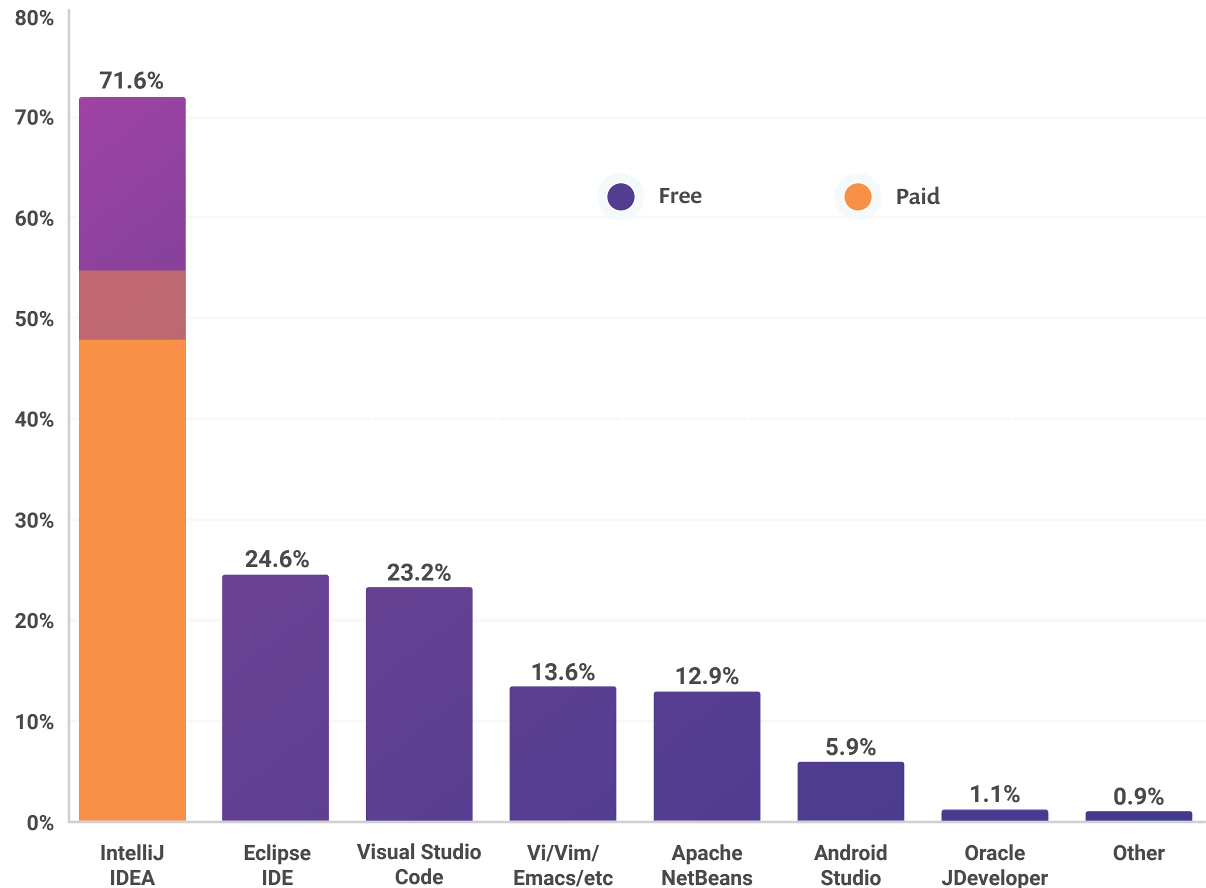
 All rights reserved. 2021 © Snyk

Most popular IDEs

For years, we've seen the dominance of JetBrains IntellIJ IDEA in the Java ecosystem. This year is not any different. IntelliJ IDEA is the most widely used within the JVM community. According to our survey, 51.3% use the Ultimate (paid) version and 27.3% the Community (free) version.

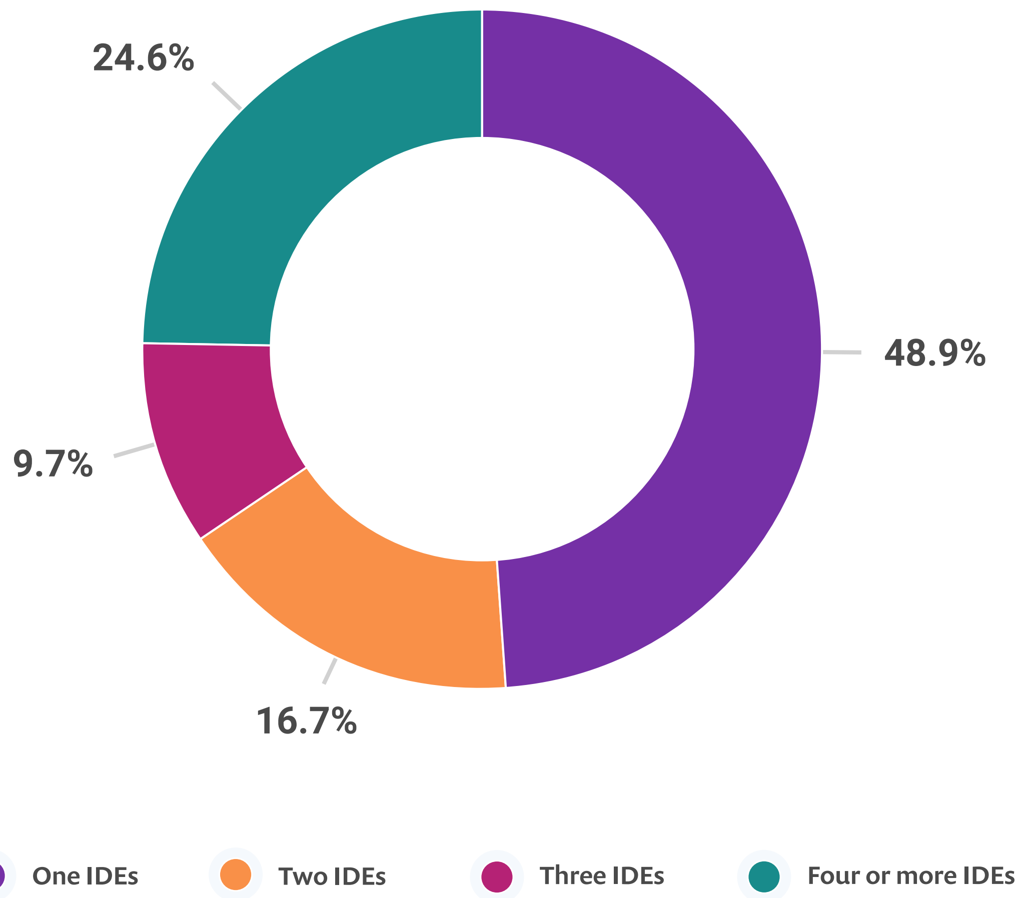
In the previous year, we only asked for "your main IDE," but we noticed that more and more people are using multiple IDEs for several reasons. Specific tooling that a developer uses for a project might build on top of the Eclipse IDE or Apache Netbeans, while for another project the developer might prefer VS Code or IntelliJ IDEA.

The results are quite amazing and we see that adoption of Visual Studio Code — currently 23.2% and 2% last year — grew tremendously. The same holds for the popularity of Apache Netbeans, with almost 13% today and only 1% last year. We believe the reason for these increases are because of the ability to answer with more than one answer.



Multiple responses allowed.

 All rights reserved. 2021 © Snyk



When looking at the amount of different IDEs a Java developer uses, we found the following statistics.

We see that almost half of the developers use a single IDE to do all of their work. On the flip side, more than half of the JVM community sees fit for multiple IDEs, and almost a quarter needs to use four or more different IDE's. The only conclusion we can draw for this is that none of the IDEs work well enough on their own for at least the 24.6% that use four or more. Why else would you use so many?

Also interesting to note is that the amount of Eclipse users is relatively steady, with 24.6% this year and 20% last year. This appears to indicate that most Eclipse users consider this IDE their main tool for development. For IntelliJ, there is a small percentage of people (7%) that use both the free Community version and the Ultimate version. As the Ultimate (paid) version is an upgrade of the free community version, the question is why?

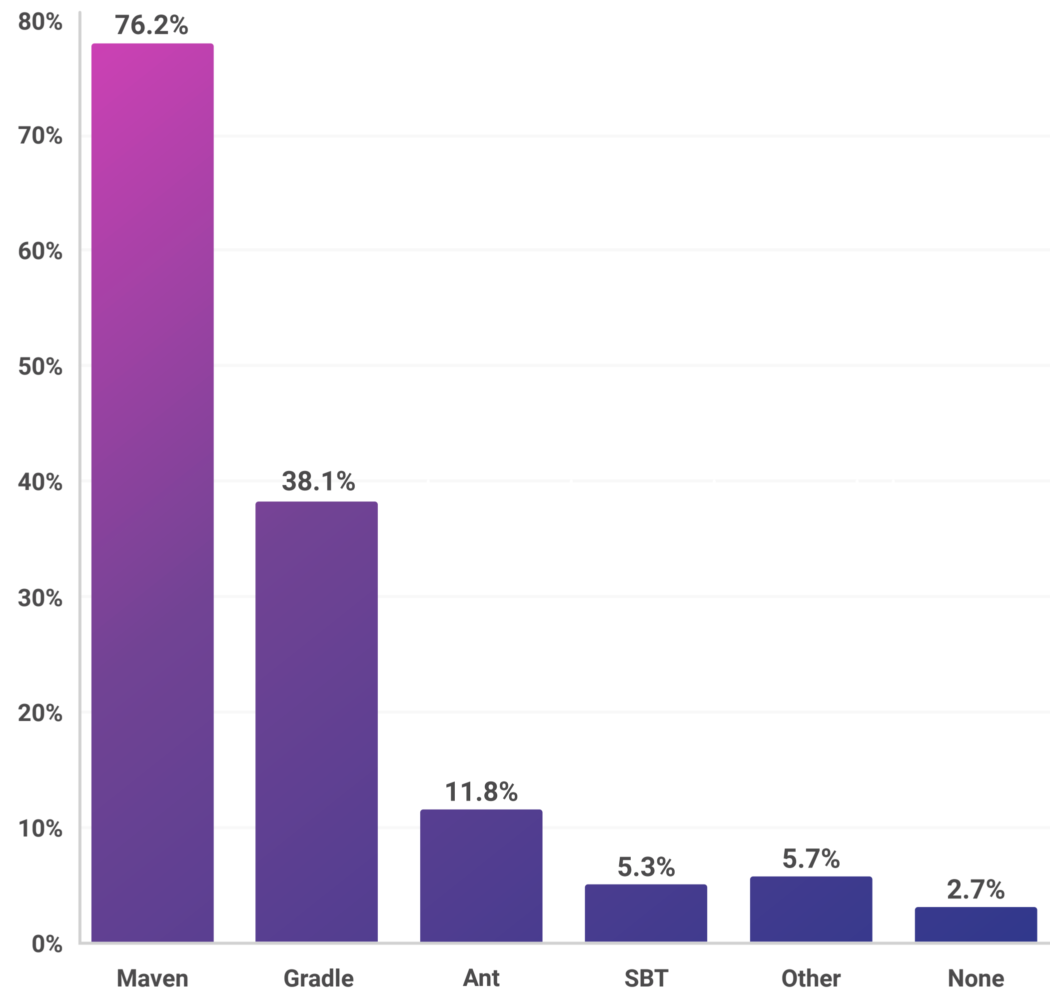
snyk All rights reserved. 2021 © Snyk

Tools for building applications

Maven is still the number one build system for the Java ecosystem. With more than 76% of developers using Maven, it is even higher than in last year's survey. Gradle is comfortably holding seconds place with 38.1%, also scoring higher than it did last year.

Similar to the other questions, people were allowed to put in multiple answers. This, however, didn't substantially impact the relative popularity of the tools.

We find it interesting that 5.7% reported using another build system. It is worth mentioning that there were quite a few people mentioning Leiningen and Clojure CLI tools. Both tools are used by Clojure developers, with Leiningen used by about 2.3% of the respondents and Clojure CLI by 1.1%.



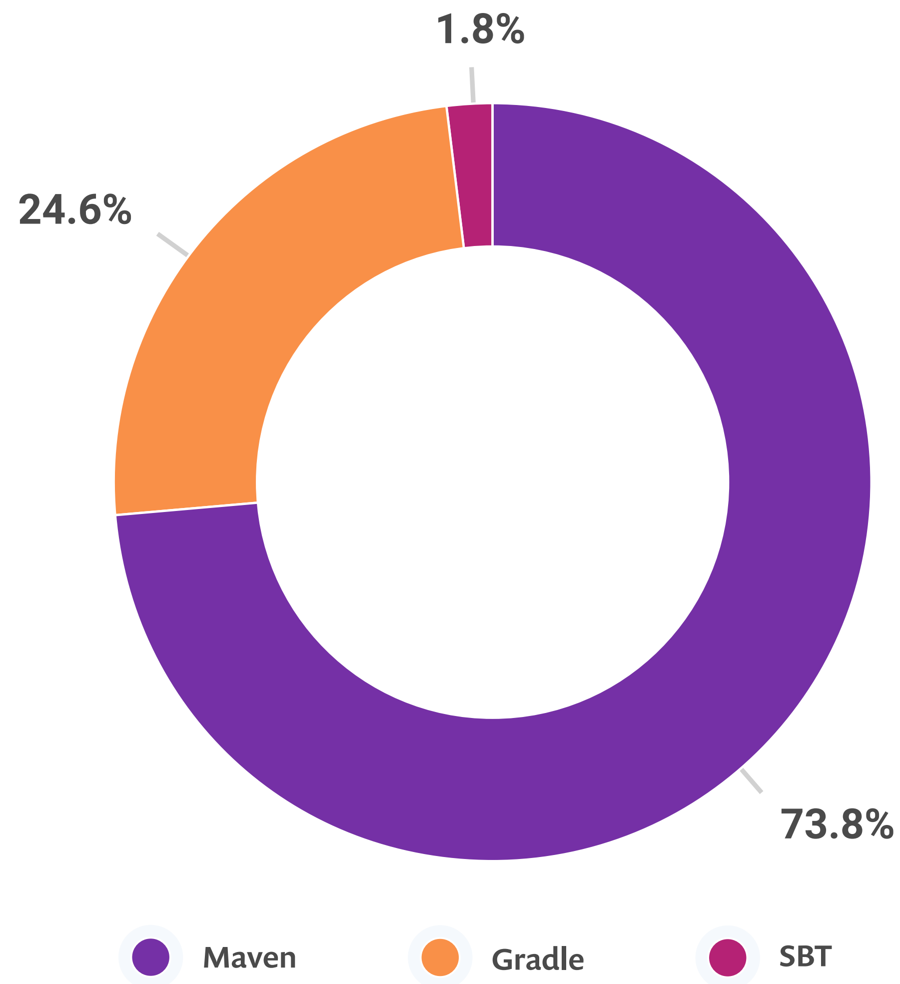
Multiple responses allowed.

snyk All rights reserved. 2021 © Snyk

Package manager distribution within Snyk

We see the same distribution of package managers if we look at the data we collected from Snyk. In the Java ecosystem, we can scan for open source vulnerabilities for three different package managers: Maven, Gradle, and SBT.

Looking at the distribution within Snyk Open Source for Java, we can conclude that the numbers from the survey are consistent with our results. Maven is still clearly on top of the rest, with over 70% usage. It also appears that the Maven vs Gradle war is over and both have found their place within the JVM ecosystem.



snyk All rights reserved. 2021 © Snyk

Securing vulnerabilities in the Java ecosystem with Snyk

Brian Vermeer
@BrianVerm
Developer Advocate at Snyk

It is incredible to see that after more than 25 years, the Java ecosystem is still relevant. I would even say more relevant than ever before, as we've watched the JVM and its ecosystem evolve significantly over the years. And while Java is still the most popular, other languages have been created — and widely adopted — that run on the JVM, such as Scala, Groovy, and more recently, Kotlin. This flexibility could help explain Java's ongoing popularity. But with increased popularity, we need to focus on increased security concerns.

Even as Java adoption grows, we still see people clinging on to older versions. We see that people are slowly moving away from Java 8 towards more recent versions, which is good news. But we still have a long way to go as more recent versions are appearing more often in development environments, and not in production.

We also see that people are using many frameworks and libraries to create their applications. Spring Boot is currently the most popular framework, probably because it already comes with many different features. This also comes with certain problems.

The problem is that the use of many libraries comes with a price. You have to maintain these libraries just as much as your own custom code. Large frameworks have a lot of transitive dependencies and all the dependencies end up in your code. You have to be well aware that these dependencies are well maintained, healthy and most importantly free of security vulnerabilities. Updating both your Java distribution and your dependencies is essential, but unfortunately, not always a priority for many development teams.

Java and the JVM ecosystem are continually advancing and releasing new language features regularly. Together with the growth of open source, developers need to be empowered with actionable security advice that helps them find and fix security issues. This aligns well with Snyk's mission to help developers worldwide to build securely.

Snyk for Java security

Java developers using Snyk can scan their Maven, Gradle, and SBT projects and find and fix security issues in the open source libraries they import to their manifest files. They can monitor their projects, and detect vulnerabilities in containerized Java applications, whether impacting the Java runtime, or the open source libraries internal to the operating system of the container image. With the Snyk Code IntelliJ plugin, developers can experience a developer-first approach to static application security testing, leveraging security findings and actionable insights right from their IDE.

Java and the JVM ecosystem are crucial for Snyk, and one of the reasons we actively contribute to the Java community. For instance by publishing insightful resources about secure development in Java such as:

- Java security best practices cheat sheet
- 10 best practices to build a Java container with Docker
- Explaining the Java deserialize vulnerability
- Fixing vulnerabilities in Maven projects
- And many more blogs and videos

In addition, we actively contribute to communities like Foojay.io, the Virtual JUG, and several local Java User Groups. But also, on the product side, you see Java plays an important role. Because more developers and companies use Snyk with Java, new features are developed with the primary focus on Java.

Java developers affinity to security

As we said earlier, with any popular ecosystem, as it grows, so do security concerns. With Java, even though it's evolving rapidly, not everyone is updating to the latest version. The same holds for Java applications. We tend to use a lot of dependencies, and these packages and frameworks we use, depend on other libraries. All direct and indirect packages end up in your application. Unfortunately, we see that a lot of developers could do a better job at maintaining their dependencies. Once a dependency is declared in our manifest file, it tends to stay there forever. Updating is not always a priority, and so is removing unused packages.

With that in mind, let's look at the top ten most common vulnerable packages in the Java ecosystem in 2021 so far:

01. org.slf4j:slf4j-api
02. com.fasterxml.jackson.code:jackson-core
03. com.fasterxml.jackson.code:jackson-annotations
04. com.fasterxml.jackson.code:jackson-databind
05. org.springframework:spring-beans
06. org.springframework:spring-core
07. commons-codec:commons-codec
08. commons-logging:commons-logging
09. org.springframework:spring-context
10. org.springframework:spring-aop

We calculated this list by looking at the number of projects we found a vulnerable package in, during the first quarter of 2021.

Most of these packages **do not have** any top-level vulnerabilities. The problems are in the underlying transitive dependencies. In addition, in most cases the vulnerabilities are already solved. For instance, the Jackson folks are doing an excellent job in fixing their libraries quickly. This also holds for the Spring libraries. In most cases, there was already a newer version available that didn't have the issues.

Taking care of your dependencies is essential. Snyk offers numerous integrations, like the ones for the most popular IDEs: IntelliJ IDEA, Eclipse, and VS Code. But you can also integrate with your favorite build tool, whether you prefer Maven or Gradle.

Development at Snyk for the Java Ecosystem

With the continued growth of the Java ecosystem, we believe that enabling Java and other JVM developers with security information is essential. We try to achieve this by providing tooling and security insights in all steps of your development lifecycle. Picking the tooling that fits your current process is essential. For almost all new features we develop at Snyk, we treat Java as a first-class citizen. New features like Snyk Code, our SAST tooling where we help you secure your custom code, and prioritization are developed with Java support from the start.

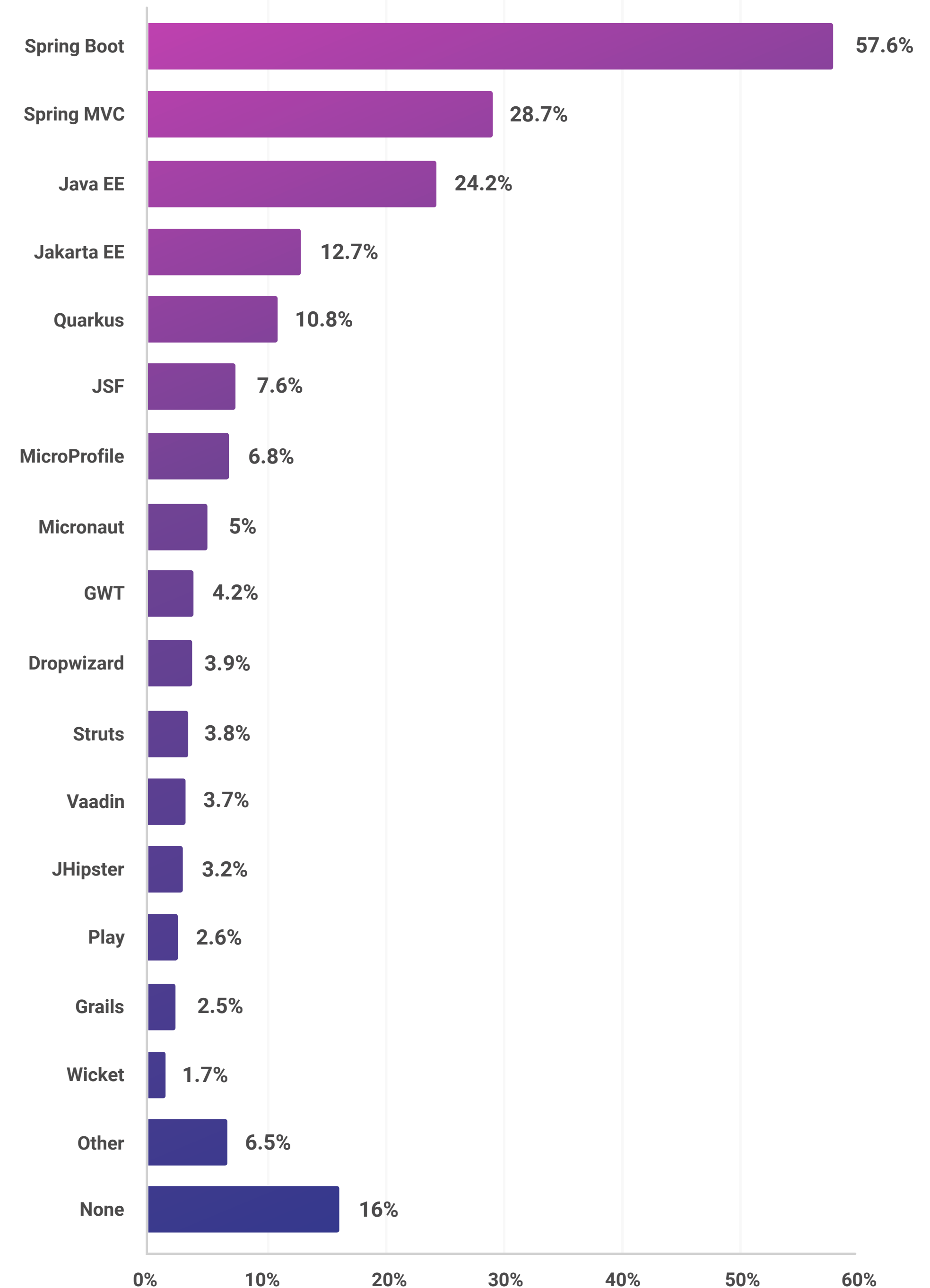
Application frameworks

The Java world is still a Spring-dominated world, with over half of the market using Spring Boot and almost a third using Spring MVC. But Java EE and its successor Jakarta EE are doing well, at nearly 24.2% and 12.7%.

An up and coming framework in this year's list is Quarkus. With more than 10%, not a bad score at all, we will have to see how this continues over the next couple of years.

We find it quite astonishing that there's a substantial amount of respondents — 16% — that don't use any framework. These may be the die-hard developers that require control over everything that happens in their application. That is a heavy burden and deserves a lot of respect. In general, we see that we live in a highly Spring-dominated universe, which appears to indicate that the Spring folks are doing a great job serving the community.

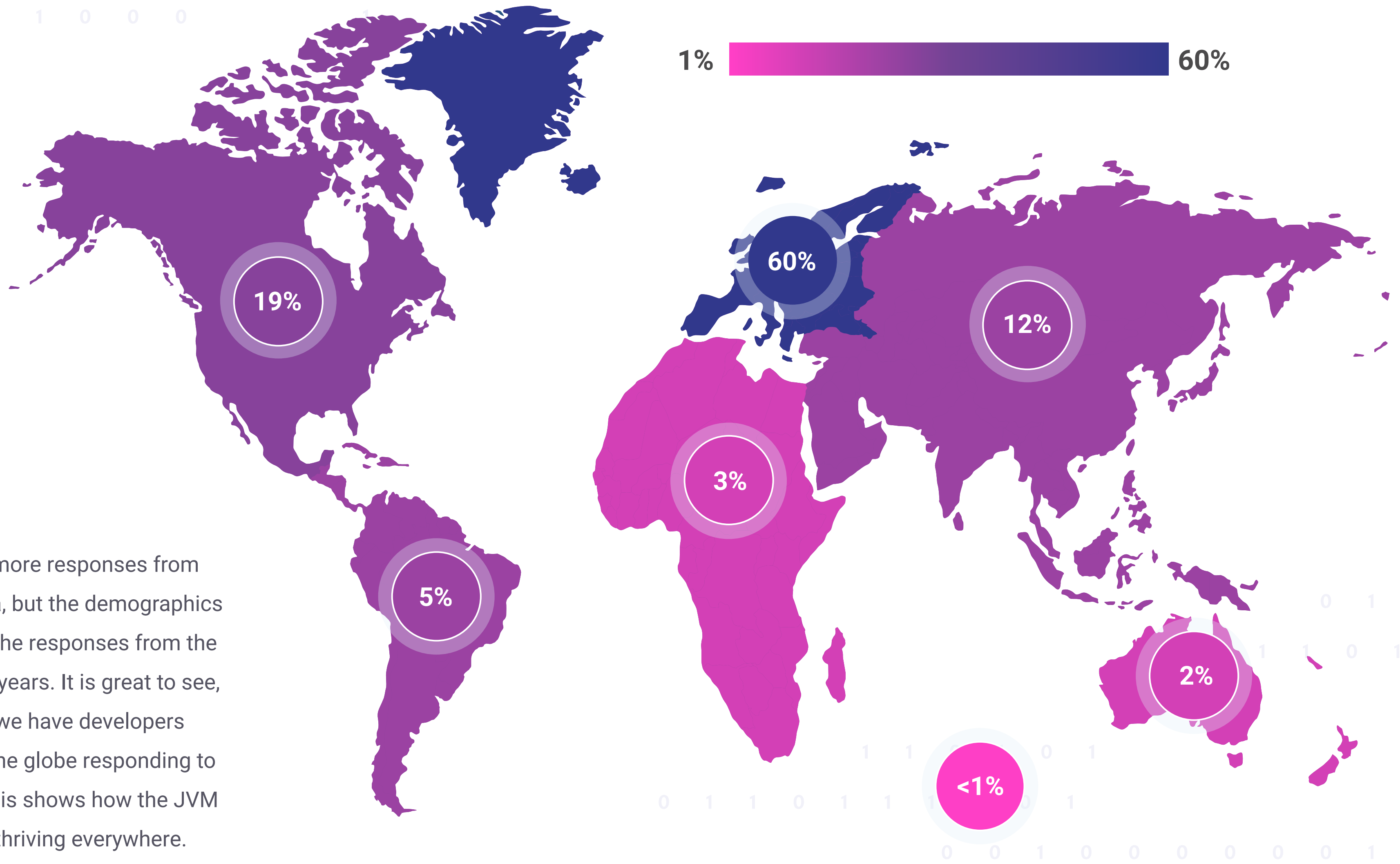
Only time will tell if this can be considered a dynasty and if Spring will stay the king of Java frameworks. Honestly, I do not expect much change shortly, but who knows?!



Multiple responses allowed.

All rights reserved. 2021 © Snyk

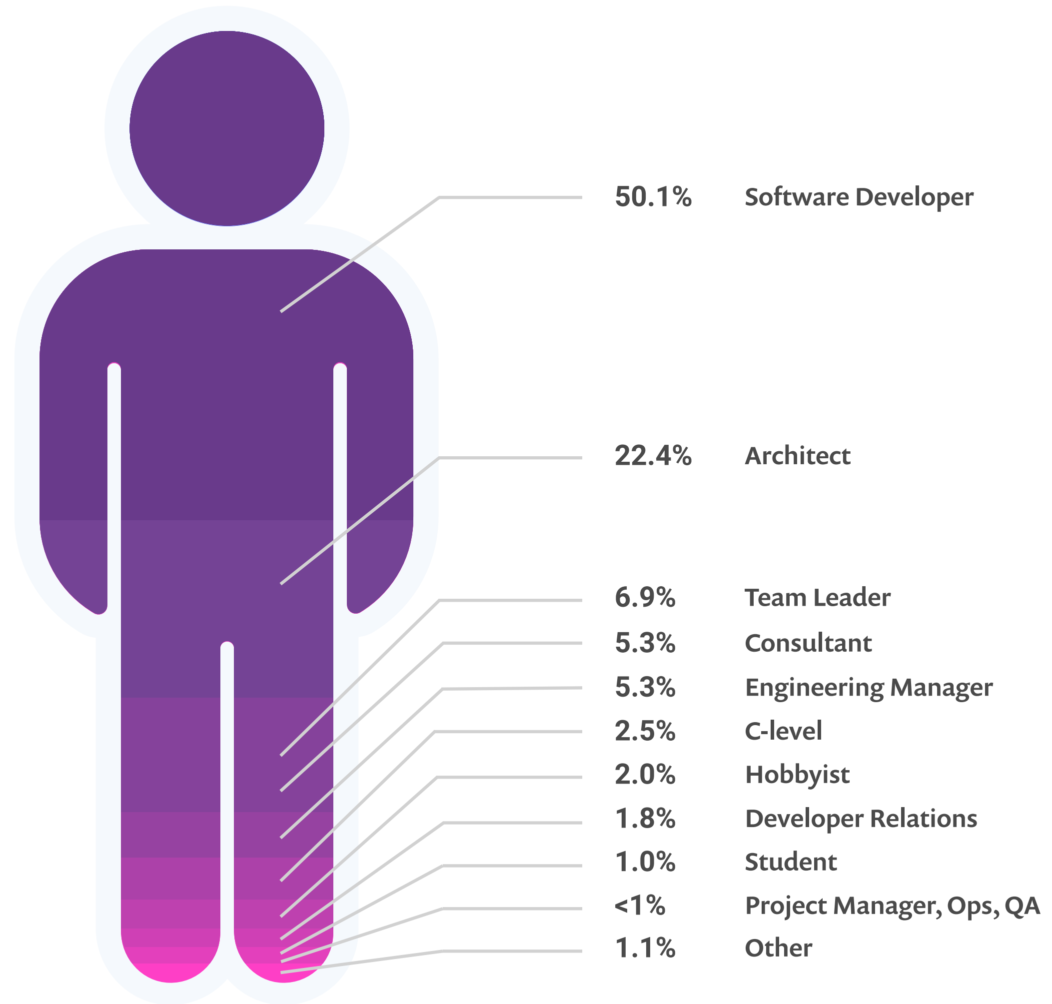
Demographics



We expected more responses from North America, but the demographics are similar to the responses from the last couple of years. It is great to see, however, that we have developers from all over the globe responding to this survey! This shows how the JVM ecosystem is thriving everywhere.

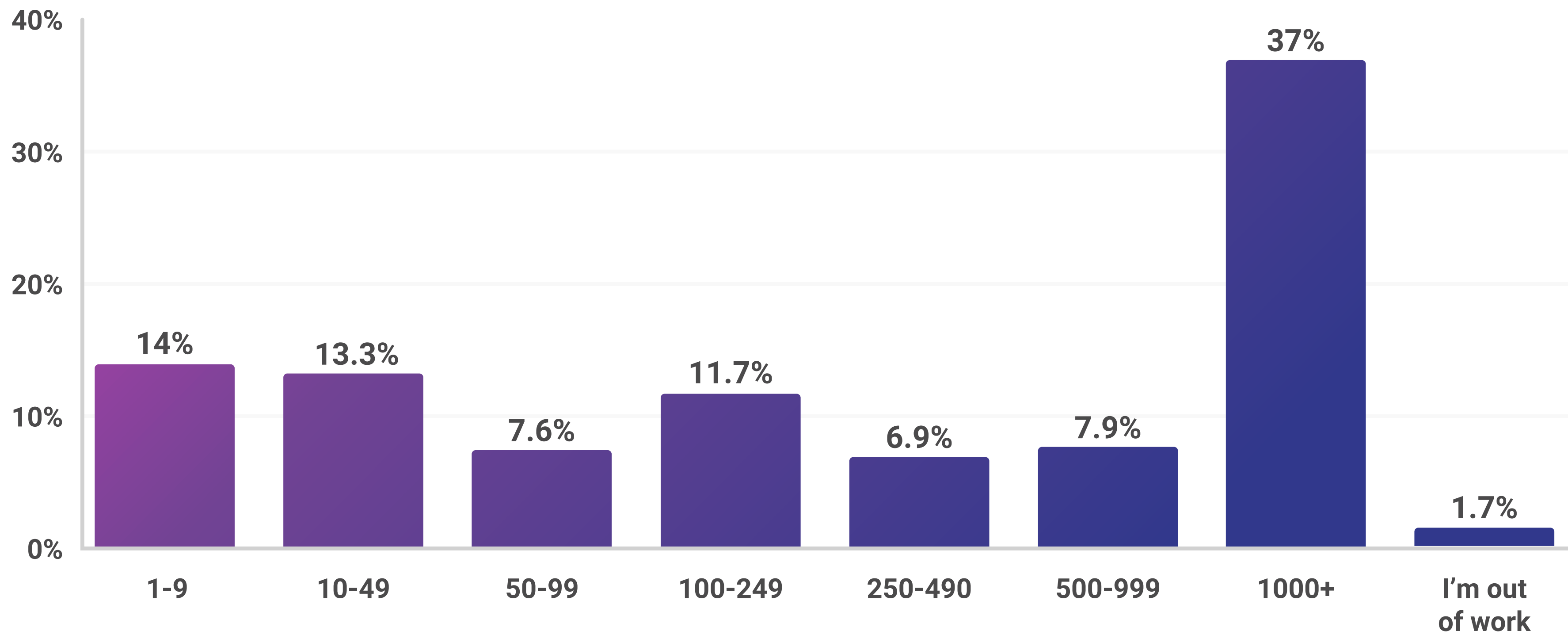
snyk All rights reserved. 2021 © Snyk

The vast majority of respondents come from a technical background with 90% of them being either developers, team leaders, or architects, consultants, or engineering managers. More than half state they are software developers, and a not insignificant number of C-level employees took time out of their busy schedule to participate in our survey.



 All rights reserved. 2021 © Snyk

With almost 35% of respondents working for companies that have less than 100 employees, we see that Java continues to have a significant role in startups and in small-to-medium businesses. Although the majority works and larger enterprises it is safe to say that Java has a role to play everywhere.



snyk All rights reserved. 2021 © Snyk

The State of Spring

Josh Long
@starbuxman
Spring Developer Advocate at
Tanzu VMware

Capturing the state of Spring is a big task! If you want to dive deep into the details, I must direct you to my weekly column on the Spring blog, called This Week in Spring. In this report, I'll just outline some of the tentpole themes I see in the Spring ecosystem.

Integration

Spring has popularized various design patterns through use case-specific frameworks like Spring Data, Spring Security, Spring Batch, Spring Integration, Spring Cloud Stream, Spring Cloud Data Flow, Spring Webflux, and Spring MVC.

The cloud platform integrations that have sprung up are essential areas of innovation. The Spring team works closely with the teams working on the various Spring integrations for Alibaba Cloud, AWS, Google Cloud, and Microsoft Azure.

The Spring team works particularly closely with Microsoft. We've even jointly developed a platform, Azure Spring Cloud, optimized for deploying Spring-based applications and microservices.

Kubernetes

Kubernetes is production for a good many organizations today. Kubernetes is a foundational layer. I tend to think, like Kelsy Hightower (@kelseyhightower), that Kubernetes is a foundational piece on top of which to layer application-focused abstractions and platforms. It's no wonder that people use things like Cloud Foundry, Istio, and KNative, and not raw Kubernetes. Spring Boot is an ideal application framework for Kubernetes-bound workloads.

Kubernetes-specific configuration

Spring Boot applications know when they're running on Kubernetes. A Spring Boot application may conditionally activate configuration (`spring.config.activate.on-cloud-platform=kubernetes`) and objects (`@ConditionalOnCloudPlatform(CloudPlatform.KUBERNETES)`) based on the cloud provider in which you're running.

External configuration with ConfigMap

Spring can read configuration from all sorts of places, like JNDI, environment variables, the Spring Cloud Config Server, Apache Zookeeper, Hashicorp Consul, and more. And it can also read configuration from Kubernetes `ConfigMaps` and `Secrets` mounted as either a directory (e.g.: **`spring.config.import=configtree:/mnt/my/config`**) or as the environment.

Probes

Kubernetes has this concept of a *readiness probe* — an HTTP endpoint that tells Kubernetes if the application is ready to handle traffic — and a *liveness probe* — an HTTP endpoint that tells Kubernetes if the application is still able to accept the traffic. Spring Boot can provide these for you automatically if you're running in Kubernetes or specify **`management.endpoint.health.probes.enabled=true`.**

Graceful shutdown

There are a lot of reasons why Kubernetes might need to shut down and destroy your pod: perhaps you deployed a new pod, or the liveness probe for an existing one has failed. What will happen to inflight transactions when Kubernetes destroys those pods? Spring Boot supports *graceful* shutdowns (server.shutdown=graceful), and *immediate* shutdowns (`server.s hutdown=immediate`). Spring Boot will stop accepting any new requests in graceful shutdown and wait for a certain interval (which you also need to configure in the Kubernetes pod specification) before it shuts down the container.

GraalVM and Spring Native

GraalVM native images are promising. They can take your JVM application and turn it into a lightweight, fast, architecture-specific application binary by doing static analysis of your application, determining which classes the runtime will load, and discarding *everything* else. *Everything*. The trouble is that it sometimes chucks out things your application needs at runtime, making some dynamic behavior typical in a Java application — like JNI, resource loading, serialization, proxies, and reflection — more complicated.

GraalVM is what we in the high falutin framework business call a "party pooper." You can specify which additional classes to retain. Enter Spring Native: it knows all about the dynamic behavior of your typical Spring Boot application.

A Spring Native application starts up *much* faster (milliseconds, not seconds or minutes), but unless you're doing serverless (with Spring Cloud Function and Spring Native, perhaps?), then startup speed misses the actual value. GraalVM native images are very efficient at runtime - tens of megabytes, not hundreds. Getting started is easy: go to the Spring Initializr and add the *experimental* (as of Q2 2021) Spring Native support and try it out. You can review the generated **`pom.xml`** by clicking `Explore`.

Buildpacks

The core conceit of the Cloud Native Foundation's Buildpacks project is that there are only so many different ways to containerize a given application artifact, be it a `.jar`, a `.war`, a .NET `.exe`, etc. So why reinvent the wheel? The Paketo project features a simple CLI that you can point at your application binary, and it'll containerize it: `pack build`. It couldn't be easier! Or could it? Spring Boot integrates buildpacks directly into the Maven and Gradle plugins (e.g.: `mvn spring-boot:build-image`). The plugins build Docker images for your local container registry that you can `docker tag`, and then `docker push`, to your container registry of choice (Dockerhub, Google Container Registry, or even VMware's Harbor project). The Spring Native support also uses buildpacks.

Reactive Programming

The Spring team was one of the cofounders of the Reactive Streams specification, with Lightbend, the Eclipse Foundation, and Netflix. We built the Reactor project, which serves as the foundation for all things reactive in the Spring ecosystem. There is support for reactive programming in virtually every Spring project. Reactive programming is a big deal because it supports three main qualities: resource efficiency, ease of composition, and robustness.

Reactive programming supports **resource efficiency** by allowing us to spend as little time on a thread as possible.

Reactive programming supports ease of composition because it gives us a single abstraction that allows us to deal with small batches of work or large (potentially unbounded) batches of work. It lets us deal with slow or fast data. Just remember the reactive streams' `Publisher<T>` interface and the Reactor project's `Mono<T>` and `Flux<T>`. These types compose nicely, freeing you to think about the higher-order algorithms and business you're trying to express, and not the tedious glue code so typical of distributed systems.

Reactive programming also gives a single unified abstraction, a central place to add support for **reliable services** by supporting backpressure, or flow control, to allow the consumer to control the consumption rate from the producer. These reactive types also include operators that support easy error handling, timeouts, retries, etc. This robustness is one reason Microsoft has issued guidance for all their Java Azure SDK teams — whether those SDKs are specifically for Spring users or not — to build their clients on Project Reactor.

RSocket

Reactive programming is fantastic! Many protocols and client libraries map to reactive types, but HTTP is a bit more complex, as there's no way to communicate backpressure at the protocol level. RSocket is a binary protocol designed by engineers at Netflix and Facebook to get around this and other imitations in HTTP 2 and GRPC. It is a multiplexed, stateful, payload-agnostic, binary protocol that supports different message exchange patterns. The official RSocket Java client builds on Reactor, and Spring offers a component model. There's integration with Spring Security, Spring Integration, and more. We're also contributing to the RSocket broker.

Next Steps

You can always look at the Spring Projects' experimental Github organization — github.com/spring-projects-experimental — to see the things on which we're working. I see Spring Retrosocket, Spring Native, a from-the-ground-up authentication and authorization server, GraphQL support, and so much more.

Spring's ecosystem is constantly growing. Check out the Spring Guides to learn more. The Spring team is usually on Gitter.im/spring-projects or Gitter.im/spring-cloud if you want to talk. We also monitor StackOverflow. Most of our Github projects have issues labeled to indicate we'd welcome a first-time contributor.

As always, you can find me, Josh Long, your humble Spring Developer advocate, on Twitter (@starbuxman).

Develop fast. Stay secure.

Report author

Brian Vermeer (@BrianVerm)

Report design

Growth Labs (@GrowthLabsMKTG)