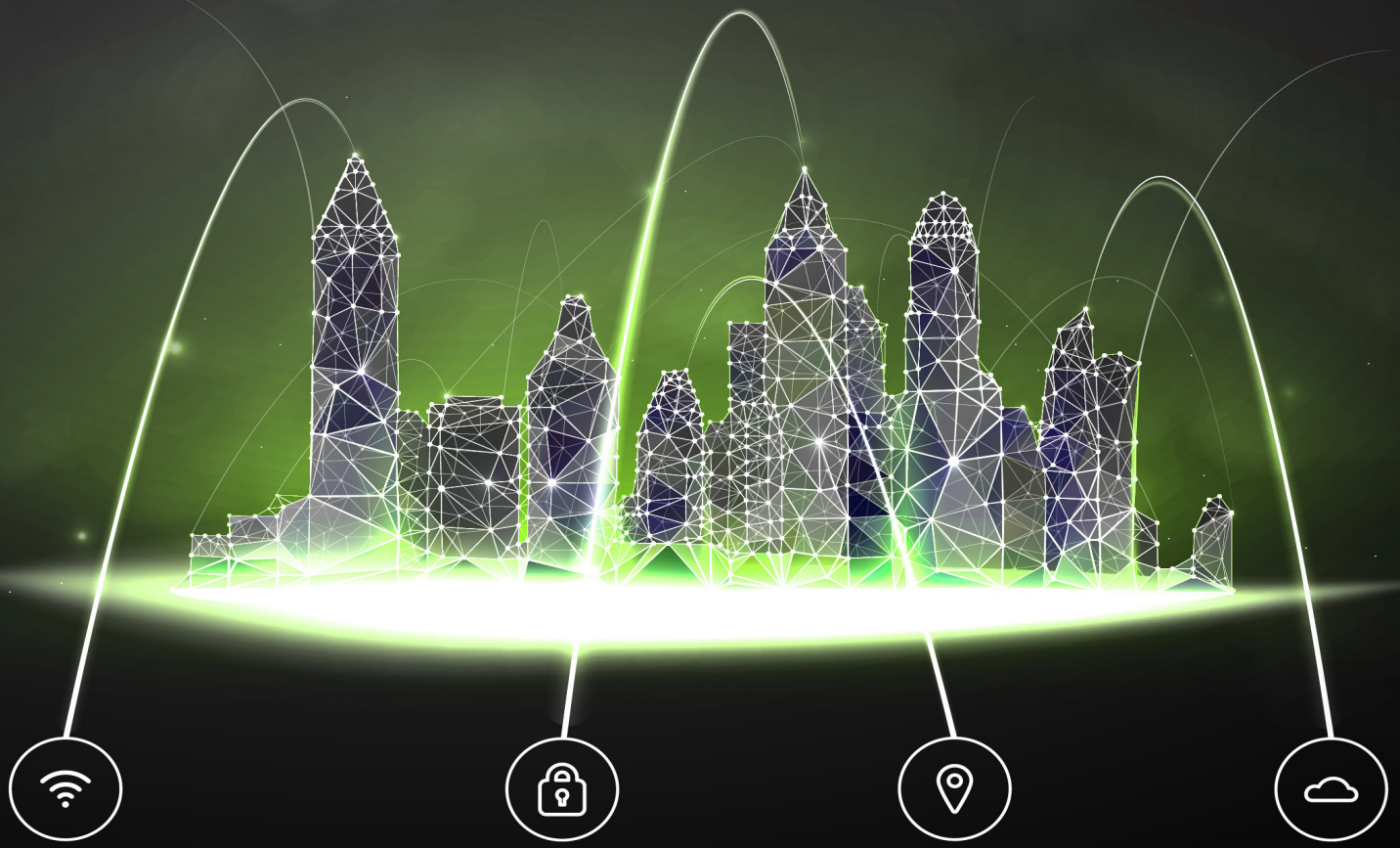
JRebel
by Perforce

Microservices Trends in Java Development Report



MICROSERVICES BACKGROUND

Microservices is an architectural software style where an application is built from lightweight, loosely-coupled services that allow developers to create, test, and release different functionalities separately. The improved agility and scale microservices provide have led to increased use within the Java ecosystem in the last several years over the traditional monolith architecture.

Executive Summary

What does the microservices landscape look like for these developers?

In May 2019, a survey conducted by the JRebel team at Perforce aimed to discover how Java users have adopted microservices. We wanted to better understand the widespread microservices adoption and how developers are using these services within their project scope.

The survey concluded that while there is still a lot to learn in regard to microservices, this architecture does have a positive impact on reducing build and restart times during development.

Survey Methods

A total of 69 respondents participated in the survey, 49 of which were using microservices in their main project and Java as their main language. This group of 49 people make up the findings below. The people surveyed were mostly not users of our products, so product biases should be avoided.

The sample size is not huge and the results have a considerable margin of error, if formally calculated. However, we believe this serves as a valuable temperature check of this hot and rapidly-evolving technology segment.

The Microservices Architecture

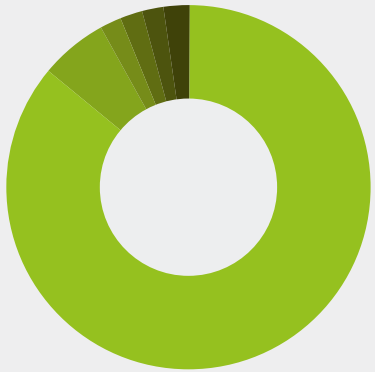
We asked respondents to select one from a list of frameworks they were using for microservices, with an option to add a free-text answer.

A significant number (86%) indicated Spring Boot as their framework of choice. This response did not come as a surprise, as Spring Boot is the fastest-growing Java framework on the market with built-in microservices support.

Another expected finding is that Docker has a large footprint among microservices users. Docker fits nicely with a lot of patterns and best practices that microservices are known for. Each service should be:

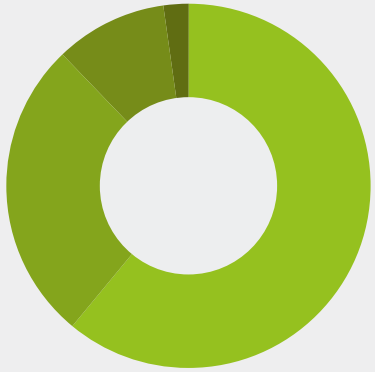
- Isolated.
- On its own JVM.
- Able to start and stop on its own.
- Scalable independently of other services.

Q: Which framework are you using for microservices?



86%	Spring Boot
6%	DropWizard
2%	Micronaut
2%	Only singleton's & Java EE 8
2%	In-house service generator + Tomcat
2%	Customized framework built on Java, .NET and Node.js

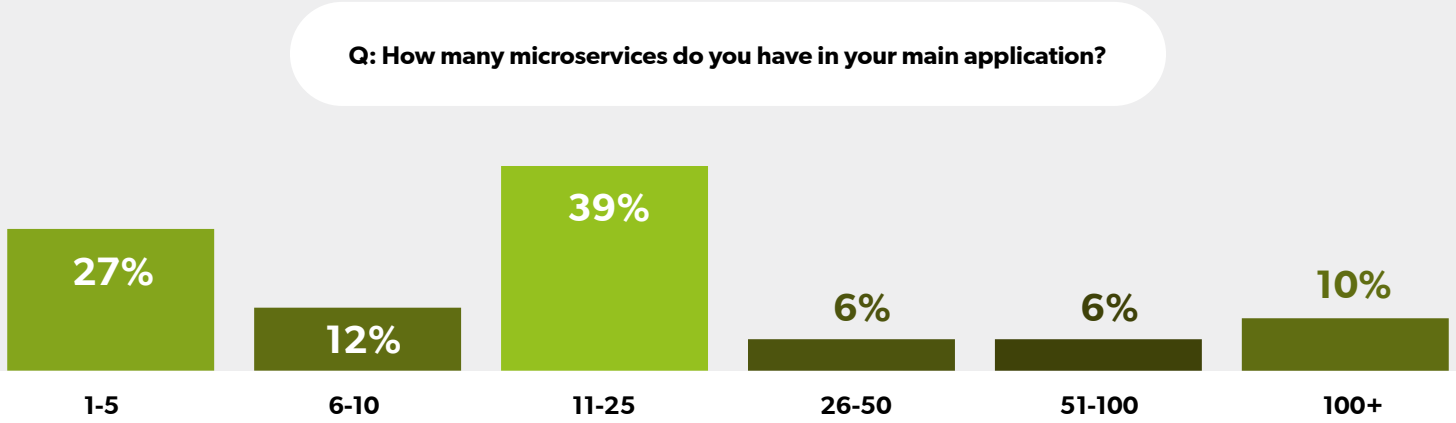
Q: Are you using containers in your development environment?



61%	I'm using Docker
27%	No
10%	I'm using other container technology
2%	Web and EJB containers

MICROSERVICES VS. MINISERVICES

Next, we asked respondents how many microservices they have in their main application:



Lots of people are actually using a rather small amount of microservices in their main application. Having 5 or less services in the application hints that these might not be the "true" microservices (small and single-purposed).

A term sometimes used for that is "miniservices" – the big monolithic app has been broken down into smaller components, but some of the architectural constraints often associated with microservices have been relaxed (strictly one feature per service, etc.).

Whether a distinction between microservices and miniservices is actually meaningful likely depends on the user. For development and local deployment, miniservices might be pretty similar to working with a monolithic application. For debugging, performance testing, monitoring, or scaling the whole system, the complexity will increase immediately even if you have just a fistful of services.

How are Engineers Working with Microservices?

From the Java tooling point of view, we wanted to determine whether developers are focusing on one service at a time or consistently making changes across services. We asked how many microservices are typically being modified simultaneously.

To build on the previous question regarding the number of microservices used in an application, the responses here signal how small or interdependent each service is.

The median number of services worked on simultaneously was three, so the developer tooling should make redeploying multiple microservices fast and easy.

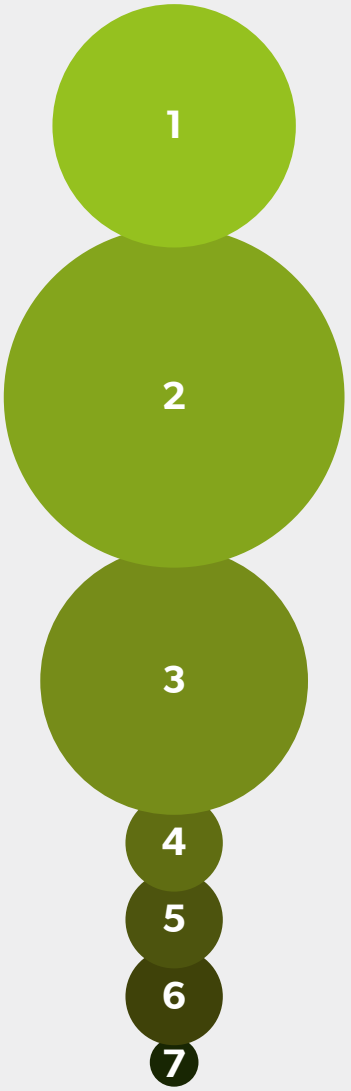
DEVELOPMENT ENVIRONMENTS

One of the most important questions we asked was how developers see the effects of changes in development. The intention was to gain insight into how developers are setting up their environment when working with microservices:

- Running full applications locally (common in monolithic architectures).
- Running one or two services on a local Docker while modifying.
- Relying on unit tests and staging, skipping local deployment altogether.

The differences between these options are significant for any Java tooling.

Q: How many microservices are you typically modifying simultaneously?



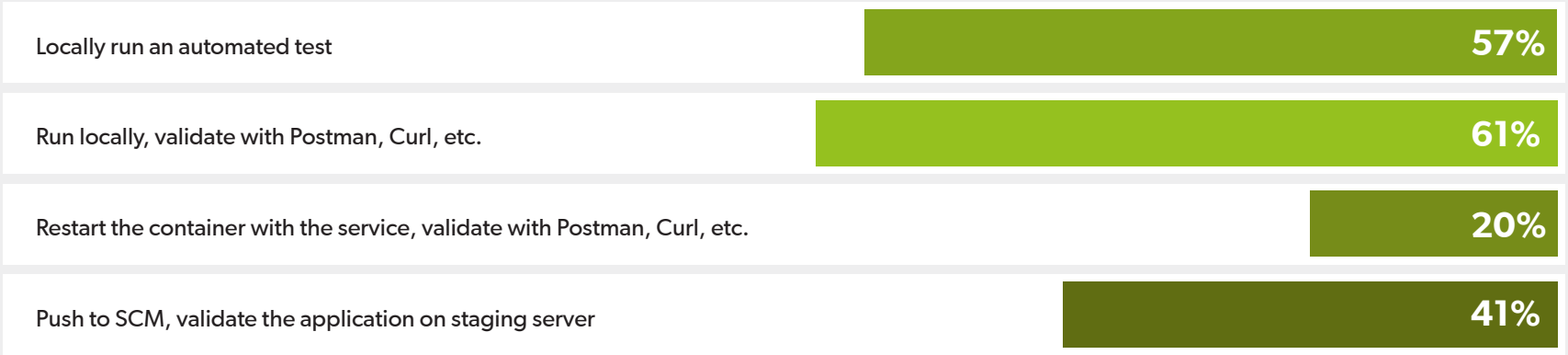
Respondents could select more than one option between the following: locally run an automated test; run locally and validate with Postman, Curl, etc.; restart the container with the service and validate with Postman, Curl, etc.; and push to SCM and validate on a staging server.

Most participants selected multiple answers, with an even split between unit tests vs. running apps locally.

A quarter (25%) indicated that they completely rely on unit testing and staging, known as a test-driven development (TDD).

About half of developers were taking advantage of unit tests, about half were still running the app locally. A quarter of responders didn't run the application neither on host machine locally nor inside Docker.

Q: How do you see the effect of your changes in development?



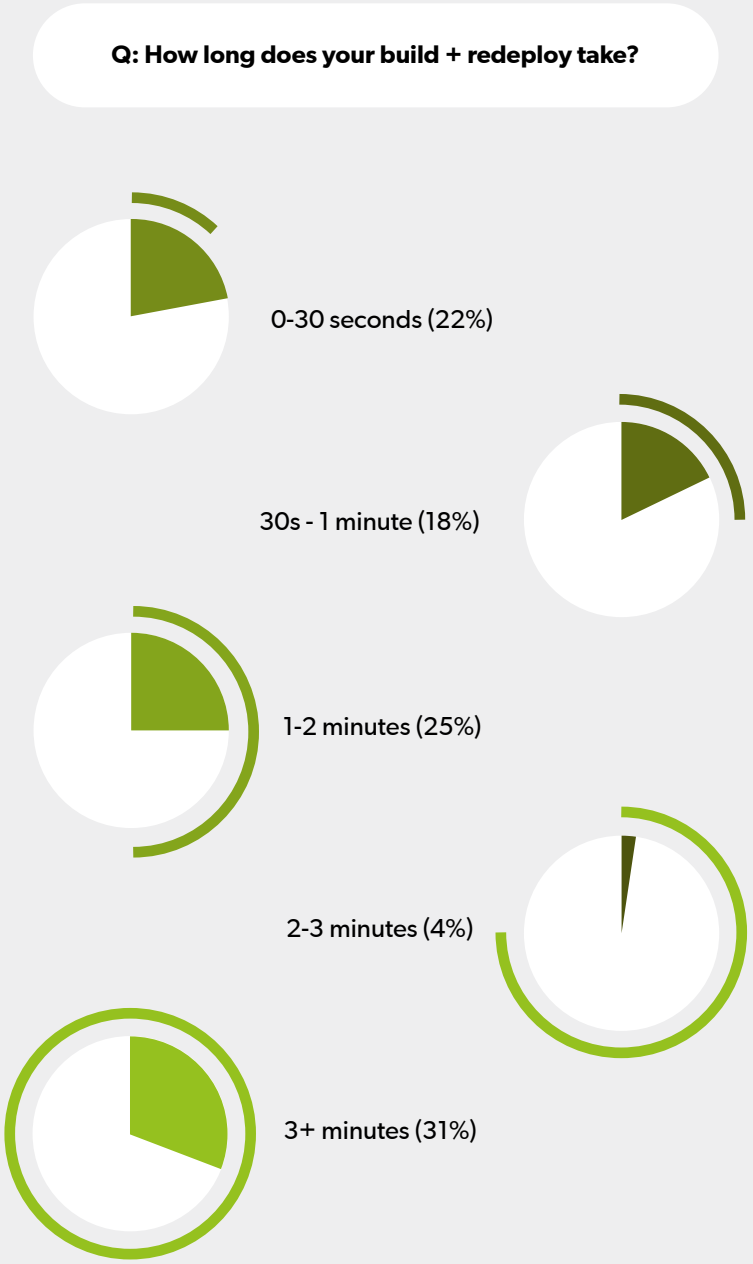
Time Spent Restarting Microservices Apps

Average build and redeploy times have decreased with the move from the monolith to the microservices architecture style. Over half (65%) reported spending two or less minutes on builds and redeploys.

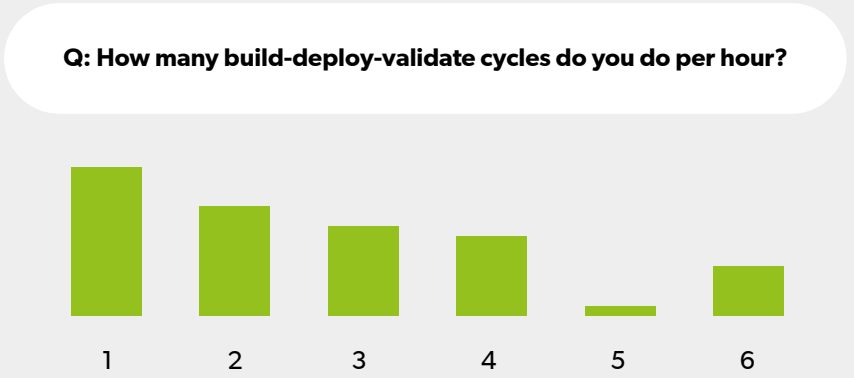
This is a decrease from the median redeploy times of our userbase on WebLogic and WebSphere (used in monolithic enterprise apps), both of which are over three minutes.

Still, over half of the developers are spending more than a minute each time they want to see the effect of their code changes. Over the course of the day, these short interruptions add up to a significant amount of time wasted.

Also, a third of engineers wait over three minutes for each change, which is way too much. Tools like JRebel will still be useful in the context of microservices to eliminate this wasted time.



While the time spent deploying has declined, microservices haven't had much of an impact on the number of build-deploy-validate cycles. In a previous survey we conducted, the median number of cycles was three per hour. The average value based on the current survey is 2.7 (with the median value being two). Independent of architecture, nobody wants to do blind coding for long.



Conclusion

As we continue to learn more about the microservices architecture and how it assists developers, we will also get a clearer picture on the best practices and tooling to support it.

This survey provides preliminary insight into how this up-and-coming architecture is being employed and what impact it is having on how engineers do their work.

Microservices add a layer of efficiency for Java developers, allowing them to split their application up into smaller, more maintainable components. They are also helping reduce time wasted on rebuilding and redeploying applications. Yet, we saw that more than half of engineers still spend more than one minute per each redeploy.

Ready to Get Started?

JRebel speeds up microservices application development, immediately reloading changes and eliminating the need for redeploys. To learn more, request a free trial of JRebel today.

START FREE TRIAL

jrebel.com/software/jrebel/trial