Using Arquillian, Hoverfly, AssertJ, JUnit, Selenium, and Mockito

Testing Java Microservices

Alex Soto Bueno
Andy Gumbrecht
Jason Porter

SAMPLE CHAPTER

MANNING

Testing Java Microservices

by Alex Soto Bueno,
Jason Porter,
and Andy Gumbrecht

Chapter 1

brief contents

1 An introduction to microservices

This chapter covers

- Why move toward a new microservice architecture?
- What microservices are today, and where the future may lead
- The basic component makeup of a microservice
- Testing strategies

Traditional monolithic applications are deployed as a single package, usually as a web or enterprise-archive file (WAR or EAR). They contain all the business logic required to complete multiple tasks, often alongside the components required to render the *user interface* (UI, or GUI for *graphical user interface*). When scaling, this usually means taking a complete copy of that entire application archive onto a new server node (basically, deploying it to another server node in a cluster). It doesn't matter where the load or bottleneck is occurring; even if it's only in a small cross section of the application, scaling this way is an all-or-nothing approach. Microservices are specifically designed to target and change this all-or-nothing aspect by allowing you to break your business logic into smaller, more manageable elements that can be employed in multiple ways.

This book isn't intended to be a tutorial on the varied microservice architectures that are available today; we'll assume you have some understanding of the subject. Rather, we're going to help you overcome the challenges involved in *testing* the common features that all microservice applications share. In order to do that, in this chapter we'll establish some common ground about what a microservice is, so that you can relate to where we're coming from when we discuss these topics in later chapters.

Shifting toward the ever-more-popular microservice architecture means you need to adopt new strategies in development, testing, and restructuring/refactoring and move away from some of the purely monolithic-application practices.

Microservices offer you the advantage of being able to scale individual services, and the ability to develop and maintain multiple services in parallel using several teams, but they still require a robust approach when it comes to testing.

In this book, we'll discuss various approaches for using this new, more focused way of delivering tightly packaged "micro" services and how to resolve the complex testing scenarios that are required to maintain stability across multiple teams. Later chapters will introduce an example application and how to develop testing strategies for it; this will help you better understand how to create your own test environments.

You'll see and use many features of the Arquillian test framework, which was specifically designed to tackle many of the common testing challenges you'll face. An array of mature extensions have been developed over the years, and although other tools are available, Arquillian is our tool of choice—so expect some bias. That said, Arquillian also provides close integration with many testing tools you may already be familiar with.

A note about software versions

This book uses many different software packages and tools, all of which change periodically. We tried throughout the book to present examples and techniques that wouldn't be greatly affected by these changes. All examples require Java 8, although when we finished the book, Java 10 had been released. We haven't updated the examples because in terms of testing microservices, the release doesn't add anything new. Something similar is true for JUnit 5. All of the examples are written using JUnit 4.12, because when we started writing the book, JUnit 5 wasn't yet in development. At the time we finished the book, not all of the frameworks explained here have official support for JUnit 5, so we decided to skip updating the JUnit version. Other libraries, such as Spring Boot and Docker (Compose), have evolved as well during the development of the book, but none of these changes have a significant impact on how to write tests.

1.1 What are microservices, and why use them?

In this section, we present what we believe is a reasonably good interpretation of the currently available answers to these questions. What you learn will provide a solid basis for understanding the microservice architecture, but expect innovation over time. We won't make any predictions: as stated, our principle focus for the book is testing microservices, which is unlikely to change in any significant way.

It isn't important that you fully understand the microservice architecture at this point. But if, after reading this chapter, the term *microservice* is still a dark void for you, we encourage you to gather more information from your own sources.

TIP You may find it useful to join the open discussions at MicroProfile (http://microprofile.io). This is an initiative by the likes of IBM, London Java Community (LJC), RedHat, Tomitribe, Payara, and Hazelcast to develop a shared definition of Enterprise Java for microservices, with the goal of standardization.

1.1.1 Why use microservices?

Before we delve into the nature of microservices, let's answer the "why" question. Until recently, it's been commonplace to develop monolithic applications, and that's still perfectly acceptable for any application that doesn't require scaling. The problem with scaling any kind of monolithic application is straightforward, as shown in figure 1.1. Microservices aren't here to tell you that everything else is bad; rather, they offer an architecture that is far more resilient than a monolith to changes in the future.

Microservices enable you to isolate and scale smaller pieces of your application, rather than the *entire* application. Imagine that you've extracted some core business logic in your application to services A and B. Let's say service A provides access to an inventory of items, and B provides simple statistics. You notice that on average, service A

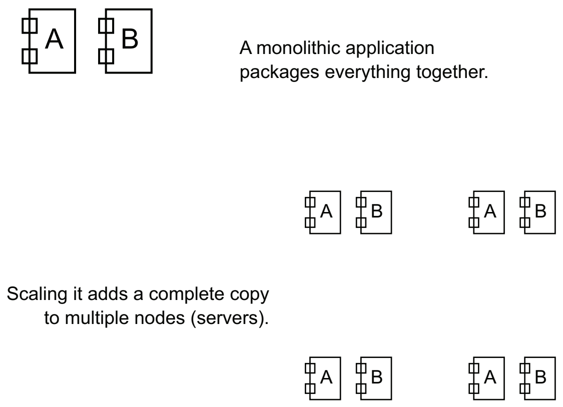
A
B
A monolithic application packages everything together.
Scaling it adds a complete copy to multiple nodes (servers).
A
B
A
B
A
B
A
B

Figure 1.1 Scaling a monolithic application

is called one million times per hour and service B is called only once per day. Scaling a monolithic application would mean adding a new node with the application that includes both services A and B.

Wouldn't it be better if you only needed to scale service A? This is where the potential of microservices becomes apparent: in the new architecture, shown in figure 1.2, services A and B become microservices A and B. You can still scale the application, but this additional flexibility is the point: you can now choose to scale where the load is greatest. Even better, you can dedicate one team of developers to maintaining microservice A and another to microservice B. You don't need to touch the application to add features or fix bugs in either A or B, and they can also be rolled out completely independently of each other.

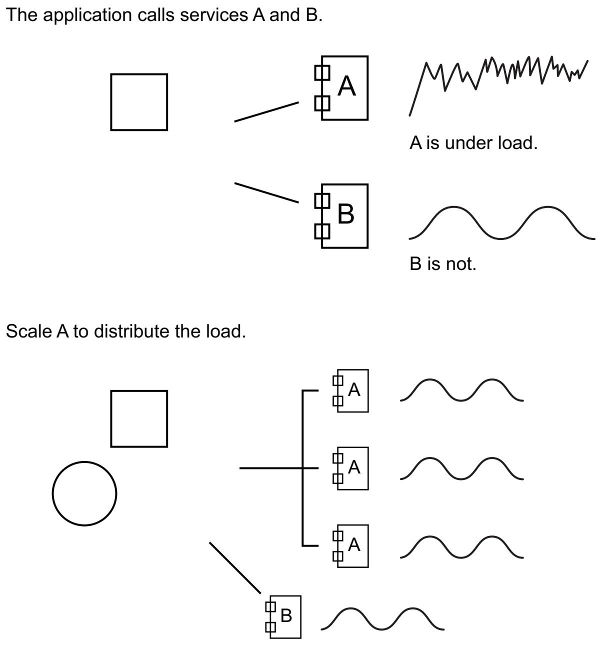
The application calls services A and B.
A
A is under load.
B
B is not.
Scale A to distribute the load.
A
A
A
B

Figure 1.2 Scaling a microservice independently of the main application

Companies like Netflix, Google, Amazon, and eBay have based much of their platforms on a microservice architecture, and they've all been kind enough to share much of this information freely. But although considerable focus is placed on web applications, you can apply a microservice architecture to any application. We hope this whets your appetite!

1.1.2 What are microservices?

At first glance, the term *micro* may conjure up images of a tiny application with a small footprint. But regarding application size, there are no rules, other than a rule of thumb. A microservice may consist of several, several hundred, or even several thousand lines of code, depending on your specific business requirements; the rule of thumb is to keep the logic small enough for a single team to manage. Ideally, you should focus on a single endpoint (which may in turn provide multiple resources); but again, there's no hard-and-fast rule. It's your party.

The most common concept is that a single application should be the uppermost limit of a microservice. In the context of a typical application server running multiple applications, this means splitting applications so they're running on a single application server. In theory, think of your first microservice as a single piece of a jigsaw puzzle, and try to imagine how it will fit together with the next piece.

You can break a monolithic application into its logical pieces, as shown in figure 1.3. There should be just enough information within each piece of the puzzle to enable you to build the greater picture. In a microservice architecture, these pieces are much more loosely coupled; see figure 1.4.

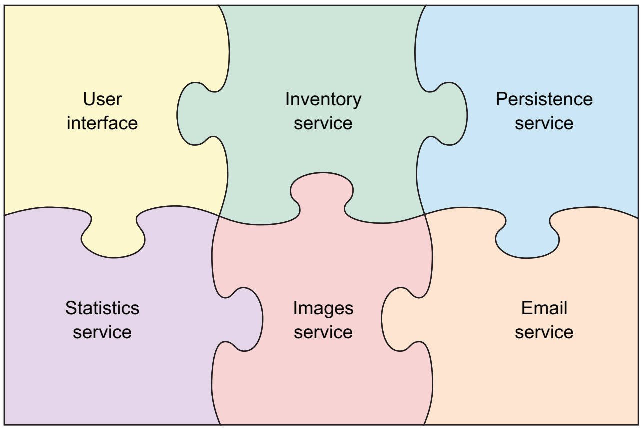

Figure 1.3 Each service is part of the big picture.

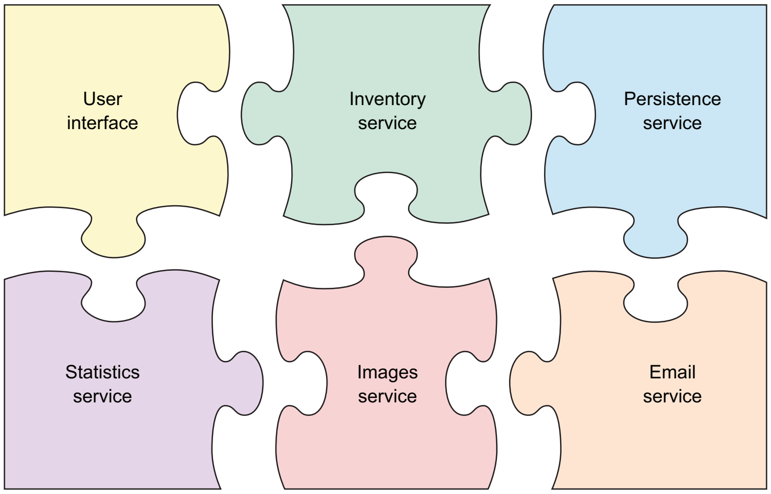

Figure 1.4 Each microservice is still part of the picture but is isolated within a separate environment.

1.1.3 Continuous integration, deployment, and Docker

The *decoupling* of application elements into scalable microservices means you'll have to start thinking about the *continuous integration (CI)* and *continuous delivery (CD)* pipelines from an early stage. Instead of one build script and one deployment, you'll need multiple independent builds that must be sewn together for integration testing and deployment to different hosts.

You'll find that far less work is involved than you may think. This is largely due to the fact that a microservice is, for all intents and purposes, an application like any other. The only difference is that a microservice packages the application together with its runtime environment. The easiest and most recognized way to do this today is to create and deploy a microservice as a Docker image (www.docker.com).

NOTE Docker is the world's leading software-containerization platform. If you're not sure what Docker is, then at some point please visit www.docker.com and follow the "What is Docker?" tutorial. Don't worry, though—we'll guide you through this pipeline when we put all the microservice elements together toward the end of the book.

The heavyweight CI/CD contenders are Travis (https://travis-ci.org), Bamboo (https://de.atlassian.com/software/bamboo), and Jenkins (https://jenkins.io). They all provide great support for microservices and deployment pipelines for Docker images; but in this book, we'll use Jenkins, because it's open source and has a huge community. It's not necessarily the easiest to use, but it offers by far the most features via plugins. In chapter 8, we'll highlight all the involved technologies in detail and guide you through the development of a viable CI/CD pipeline.

1.2 Microservice networks and features

Microservices are *loosely coupled*, which leads to new questions. How are microservices coupled, and what features does this architecture offer? In the following sections, we'll look at some answers. But for all intents and purposes, each microservice is isolated by a network boundary.

1.2.1 Microservice networks

Microservices are most commonly integrated over a RESTful (Representational State Transfer) API using HTTP or HTTPS, but they can be connected by anything that's considered a protocol to access an endpoint to a resource or function. This is a broad topic, so we're only going to discuss and demonstrate Java REST using JAX-RS.

TIP If you're unfamiliar with RESTful web services using JAX-RS (https://jax-rs-spec.java.net), now would be a good time to read up on these topics.

With this information, your initial ideas for microservices should be starting to take form. Let's continue with our earlier example. Microservice A, the inventory service, is isolated by a network layer from the UI and from microservice B, the statistics service. B

Hypermedia

Services should be developed with *hypermedia* in mind. This is the latest buzzword; it implies that services should be self-documenting in their architecture, by providing links to related resources in any response. Currently there's no winner in this category, and it would be unfair to start placing bets now, but you can take a look at the front runners and make an educated guess: JSON-LD (http://json-ld.org), JSON Hypertext Application Language (HAL, https://tools.ietf.org/html/draft-kelly-json-hal-08), Collection+JSON (https://github.com/collection-json/spec), and Siren (https://github.com/kevinswiber/siren).

communicates with A to collect statistics using the defined request-and-response protocols. They each have their own domain and external resources and are otherwise completely separate from each other. The UI service is able to call both A and B to present information in a human-readable form, a website, or a heavy client, as shown in figure 1.5.

Tests must be designed to cover comprehensively any and all interaction with external services. It's important to get this right, because network interaction will always present its own set of challenges. We'll cover this extensively in chapter 5.

By now it should be clear that a microservice can be large in terms of application size, and that "micro" refers to the public-facing surface area of the application. Cloud space is cheap today, so the physical size of a microservice is less relevant than in the past.

Another concern that we often hear mentioned is, "What about network speed?" Microservices are generally hosted in the same local network, which is typically Gigabit Ethernet or better. So, from a client perspective, and given the ease of scaling microservices, response times are likely to be much better than expected. Again, don't take our word for it; think of Netflix, Google, Amazon/AWS, and eBay.

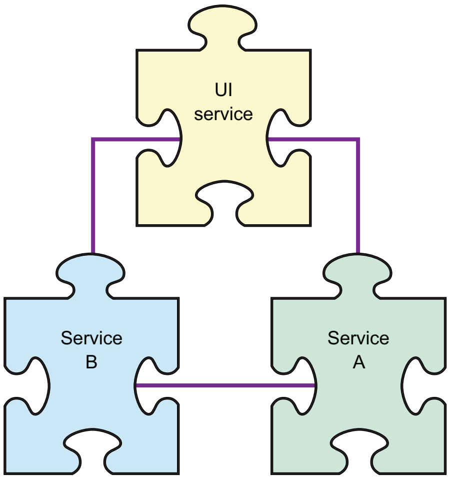

Figure 1.5 Each service communicates by defined protocols.

1.2.2 Microservice features

In our example, both microservices A and B can be developed independently and deployed by two entirely different teams. Each team only needs to understand the resource-component layer of the microservice on which they're working, rather than the entire business-domain component. This is the first big win: development can be much faster and easier to understand in the given context.

JavaScript Object Notation (JSON, www.json.org) and Extensible Markup Language (XML, www.w3.org/XML) are the common resource languages, so it's easy to write clients for such services. Some cases may dictate a different approach, but the basic scenarios remain essentially the same: the endpoints are accessible from a multitude of devices and clients using defined protocols.

Multiple microservices form a network of connected applications, where each individual microservice can be scaled independently. Elastic deployment on the cloud is now commonplace, and this enables an individual service to scale *automatically* up or down—for example, based on load.

Some other interesting benefits of microservices are improved fault isolation and memory management. In a monolithic application, a fault in a single component can bring down an entire server. With resilient microservices, the larger part of the picture will continue to function until the misbehaving service issue is resolved. In figure 1.6, is the statistics service really necessary for the application to function as a whole, or can you live without it for a while?

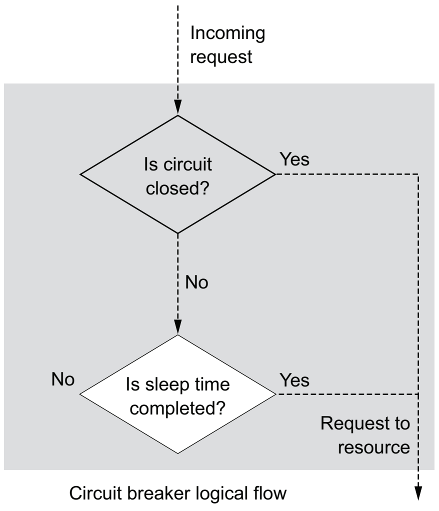

Figure 1.6 Resilient design using circuit breakers

Of course, as is the nature of all good things, microservices have drawbacks. Developers need to learn and understand the complexities of developing a distributed application, including how best to use IDEs, which are often orientated toward monolithic development. Developing use cases spanning multiple services that aren't included in distributed transactions requires more thought and planning than for a monolith. And testing is generally more difficult, at least for the connected elements, which is why we wrote this book.

1.3 Microservice architecture

The anatomy of a microservice can be varied, as shown in figure 1.7, but design similarities are bound to occur. These elements can be grouped together to form the application-component layers. It's important to provide test coverage at each layer, and you'll likely be presented with new challenges along the way; we'll address these challenges and offer solutions throughout the book.

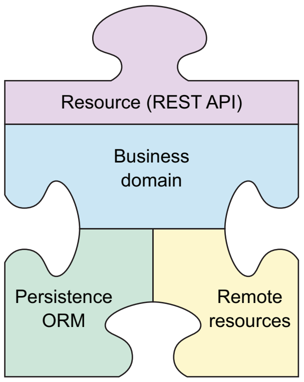

Figure 1.7 The basic microservice components

Let's look at these microservice component layers from the top down.

NOTE A microservice should encapsulate and expose a well-defined area of logic as a service. That doesn't mean that you can't allow interaction from other systems by other means. For example, your service may expose specific documents that are stored in Elasticsearch (ES). In such a case, it's perfectly legitimate for other applications to talk natively to ES in order to seed the documents.

1.3.1 Resource component

Resources are responsible for exposing the service interaction via a chosen protocol. This interaction occurs using mapped objects, usually serialized using JSON or XML. These mapped objects represent the input and/or output of the business domain. Sanitization of the incoming objects and construction of the protocol-specific response usually occur at this layer; see figure 1.8.

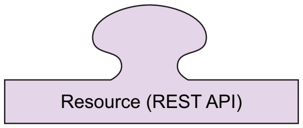

Figure 1.8 The resource component publicly exposes the service.

NOTE Now that we're here, it's worth mentioning that the resource-component layer is the layer that puts the *micro* in *microservice*.

For the rest of this book, and for the sake of simplicity, we'll focus on the most common form of resource providers today: *RESTful endpoints*.[1] If you aren't familiar with RESTful web services, please take the time to research and understand this important topic.

1.3.2 Business-domain component

The business-domain component is the core focus of your service application and is highly specific to the logical task for which the service is being developed. The domain may have to communicate with various other services (including other microservices) in order to calculate a response or process requests to and from the resource component; see figure 1.9.

Business domain

Figure 1.9 The business-domain component is your service's business logic.

A bridge is likely to be required between the domain component and the resource component, and possibly the remote component. Most microservices need to communicate with other microservices at some point.

1.3.3 Remote resources component

This component layer is where your piece of the jigsaw puzzle may need to connect to the next piece, or pieces, of the picture. It consists of a client that understands how to send and receive resource objects to and from other microservice endpoints, which it

[1] See "What Are RESTful Web Services?" in the Java EE 6 tutorial, http://mng.bz/fIa2.

then translates for use in the business component layer; see figure 1.10.

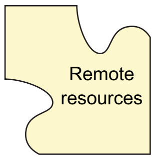

Figure 1.10 The remote resources component is the gateway to other services.

Due to the nature of remote resources, you must pay special attention to creating a resilient design. A resilient framework is designed to provide features such as circuit breakers and timeout fallbacks in the event of a failure. Don't try to reinvent the wheel: several resilient frameworks are available to choose from, including our top pick, Hystrix (https://github.com/Netflix/Hystrix/wiki), which is open source and contributed by Netflix.

A gateway service should act as a bridge between the domain component and the client component. It's responsible for translating request-and-response calls to and from any remote resource via the client. This is the best place to provide a graceful failure if the resource can't be reached.

The client is responsible for speaking the language of your chosen protocol. Nine times out of ten, this will be JAX-RS (https://jax-rs-spec.java.net) over HTTP/S for RESTful web services.

We highly recommend the open source services framework Apache CXF (http://cxf.apache.org) for this layer, because it's fully compliant with JAX-WS, JAX-RS, and others, and it won't tie you down to a specific platform.

1.3.4 Persistence component

More often than not, an application requires some type of persistence or data retrieval (see figure 1.11). This usually comes in the form of an object-relational mapping (ORM)[2] mechanism, such as the Java Persistence API (JPA),[3] but could be something as simple as an embedded database or properties file.

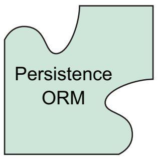

Figure 1.11 The persistence component is for data storage.

1.4 Microservice unit testing

Chapter 3 will take a deep dive into real unit-testing scenarios. The next few paragraphs are an introduction to the terminology we'll use and what to expect as you develop your testing strategies.

A typical unit test is designed to be as small as possible and to test a trivial item: a *unit of work*. In the microservice context, this unit of work may be more difficult to represent, due to the fact that there's often much more underlying complexity to the service than is apparent at first glance.

Unit testing can often lead to the conclusion that you need to refactor your code in order to reduce the complexity of the component under test. This also makes testing useful as a design tool, especially when you're using test-driven development

[2] See "Hibernate ORM: What Is Object/Relational Mapping?" http://hibernate.org/orm/what-is-an-orm.

[3] See "Introduction to the Java Persistence API" in the Java EE 6 tutorial, http://mng.bz/Cy69.

(TDD). A beneficial side effect of unit testing is that it lets you continue developing an application while detecting regressions at the same time.

Although you're likely to encounter more-detailed scenarios along the way, there are basically two styles of unit testing: *sociable* and *solitary*. These styles are loosely based on whether the unit test is isolated from its underlying collaborators. Both styles are nonexclusive, and they complement each other nicely. You should count on using both, depending on the nature of the testing challenge. We'll expand on these concepts throughout the book.

1.4.1 Solitary unit tests

Solitary unit testing should focus on the interaction around a single object class. The test should encompass only the class's own dependents or dependencies on the class. You'll usually test resource, persistence, and remote components using solitary tests, because those components rarely need to collaborate with each other; see figure 1.12.

You need to isolate individual classes for testing by stubbing or mocking all collaborators within that class. You should test all the methods of the class, but not cross any boundaries to other concrete classes. Basically, this means all injected fields should receive either a mock or stubbed implementation that only returns canned responses. The primary aim is for the code coverage of the class under test to be as high as possible.

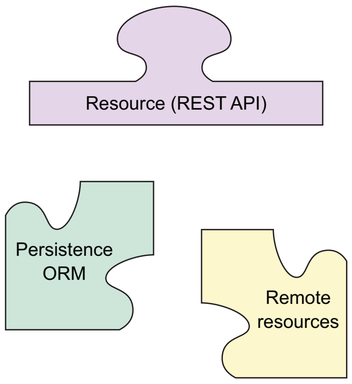

Figure 1.12 Predominantly solitary unit-test components

1.4.2 Sociable unit tests

Sociable unit testing focuses on testing the behavior of modules by observing changes in their state. This approach treats the unit under test as a black box tested entirely through its interface. The domain component is nearly always a candidate for sociable testing, because it needs to collaborate in order to process a request and return a response; see figure 1.13.

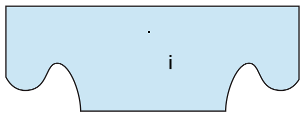

Figure 1.13 Predominantly sociable unit-test component

You may still need to stub or mock some complex collaborators of the class under test, but this should be as far along as possible within the hierarchy of collaborating objects. You shouldn't only be testing that a specific class sends and receives correct payloads, but also that the class collaborators are operating as expected *within* the class. The test coverage should ideally include all models, variables and fields as well as the class collaborators. It's also important to test that the class can correctly handle any response, including invalid responses (negative testing).

Summary

- A microservice is a part of a monolithic application that has been dissected into a smaller logical element.
- Microservices benefit your application by allowing targeted scaling and focused development.
- Microservices offer a logical way to meet scalability requirements by providing the ability to scale not only *where* performance is required, but also *when*.
- You can break monolithic applications into smaller elements that can be used as microservices.
- Microservices allow several teams to focus on individual, nonconflicting tasks that make up the bigger picture.
- Solitary unit tests are used for components that don't store state or don't need to collaborate in order to be tested.
- Sociable unit tests are used for components that must collaborate or store state in order to be tested.

SOFTWARE DEVELOPMENT/JAVA

Testing Java Microservices

Soto Bueno • Gumbrecht • Porter

Free eBook
See first page

Microservice applications present special testing challenges. Even simple services need to handle unpredictable loads, and distributed message-based designs pose unique security and performance concerns. These challenges increase when you throw in asynchronous communication and containers.

Testing Java Microservices teaches you to implement unit and integration tests for microservice systems running on the JVM. You'll work with a microservice environment built using Java EE, WildFly Swarm, and Docker. You'll advance from writing simple unit tests for individual services to more-advanced practices like chaos or integration tests. As you move towards a continuous-delivery pipeline, you'll also master live system testing using technologies like the Arquillian, Wiremock, and Mockito frameworks, along with techniques like contract testing and over-the-wire service virtualization. Master these microservice-specific practices and tools and you'll greatly increase your test coverage and productivity, and gain confidence that your system will work as you expect.

What's Inside

- Test automation
- Integration testing microservice systems
- Testing container-centric systems
- Service virtualization

Written for Java developers familiar with Java EE, EE4J, Spring, or Spring Boot.

Alex Soto Bueno and **Jason Porter** are Arquillian team members. **Andy Gumbrecht** is an Apache TomEE developer and PMC. They all have extensive enterprise-testing experience.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/testing-java-microservices

"A great and invaluable gallery of test solutions, descriptions, and examples."
—Gualtiero Testa, Factor-y

"Covers the full spectrum of microservice-testing techniques. An indispensable book."
—Piotr Gliniewicz
netPR.pl

"Thorough explanations. Concrete examples. Real-world technology."
—Ethan Rivett, Powerley

"A beacon of light for Java developers."
—Yagiz Erkan, Motive Retail

ISBN-13: 978-1-61729-289-7
ISBN-10: 1-61729-289-3
54499
9 781617 292897

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]