Microservices for Java Developers

A Hands-On Introduction to Frameworks & Containers

Christian Posta

Microservices for Java Developers

A Hands-on Introduction to Frameworks and Containers

Christian Posta

Table of Contents

CHAPTER 1
Microservices for Java Developers

What Can You Expect from This Book?

This book is for Java developers and architects interested in developing microservices. We start the book with the high-level understanding and fundamental prerequisites that should be in place to be successful with a microservice architecture. Unfortunately, just using new technology doesn't magically solve distributed systems problems. We take a look at some of the forces involved and what successful companies have done to make microservices work for them, including culture, organizational structure, and market pressures. Then we take a deep dive into a few Java frameworks for implementing microservices. The accompanying source-code repository can be found on GitHub. Once we have our hands dirty, we'll come back up for air and discuss issues around deployment, clustering, failover, and how Docker and Kubernetes deliver solutions in these areas. Then we'll go back into the details with some hands-on examples with Docker, Kubernetes, and NetflixOSS to demonstrate the power they bring for cloud-native, microservice architectures. We finish with thoughts on topics we cannot cover in this small book but are no less important, like configuration, logging, and continuous delivery.

Microservices are not a technology-only discussion. Implementations of microservices have roots in complex-adaptive theory, service design, technology evolution, domain-driven design, dependency thinking, promise theory, and other backgrounds. They all come together to allow the people of an organization to truly

exhibit agile, responsive, learning behaviors to stay competitive in a fast-evolving business world. Let's take a closer look.

You Work for a Software Company

Software really is eating the world. Businesses are slowly starting to realize this, and there are two main drivers for this phenomenon: delivering value through high-quality services and the rapid commoditization of technology. This book is primarily a hands-on, by-example format. But before we dive into the technology, we need to properly set the stage and understand the forces at play. We have been talking ad nauseam in recent years about making businesses *agile*, but we need to fully understand what that means. Otherwise it's just a nice platitude that everyone glosses over.

The Value of Service

For more than 100 years, our business markets have been about creating products and driving consumers to wanting those products: desks, microwaves, cars, shoes, whatever. The idea behind this "producer-led" economy comes from Henry Ford's idea that "if you could produce great volumes of a product at low cost, the market would be virtually unlimited." For that to work, you also need a few one-way channels to directly market toward the masses to convince them they needed these products and their lives would be made substantially better with them. For most of the 20th century, these one-way channels existed in the form of advertisements on TV, in newspapers and magazines, and on highway billboards. However, this producer-led economy has been flipped on its head because markets are fully saturated with product (how many phones/cars/TVs do you need?). Further, the Internet, along with social networks, is changing the dynamics of how companies interact with consumers (or more importantly, how consumers interact with them).

Social networks allow us, as consumers, to more freely share information with one another and the companies with which we do business. We trust our friends, family, and others more than we trust marketing departments. That's why we go to social media outlets to choose restaurants, hotels, and airlines. Our positive feedback in the form of reviews, tweets, shares, etc., can positively favor the brand of a company, and our negative feedback can just as easily and very

swiftly destroy a brand. There is now a powerful bi-directional flow of information with companies and their consumers that previously never existed, and businesses are struggling to keep up with the impact of not owning their brand.
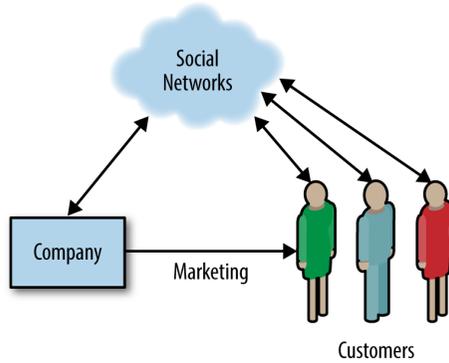
Post-industrial companies are learning they must nurture their relationship (using bi-directional communication) with customers to understand how to bring value to them. Companies do this by providing ongoing conversation through service, customer experience, and feedback. Customers choose which services to consume and for which to pay depending on which ones bring them value and good experience. Take Uber, for example, which doesn't own any inventory or sell products per se. I don't get any value out of sitting in someone else's car, but usually I'm trying to get somewhere (a business meeting, for example) which does bring value. In this way, Uber and I create value by my using its service. Going forward, companies will need to focus on bringing valuable services to customers, and technology will drive these through digital services.

Commoditization of Technology

Technology follows a similar boom-to-bust cycle as economics, biology, and law. It has led to great innovations, like the steam engine, the telephone, and the computer. In our competitive markets, however, game-changing innovations require a lot of investment and build-out to quickly capitalize on a respective market. This brings more competition, greater capacity, and falling prices, eventually making the once-innovative technology a commodity. Upon these commodities, we continue to innovate and differentiate, and the cycle continues. This commoditization has brought us from the

mainframe to the personal computer to what we now call "cloud computing," which is a service bringing us commodity computing with almost no upfront capital expenditure. On top of cloud computing, we're now bringing new innovation in the form of digital services.
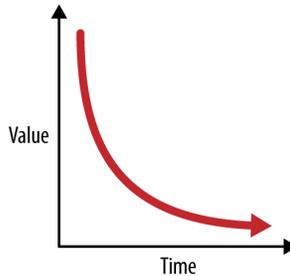
Open source is also leading the charge in the technology space. Following the commoditization curves, open source is a place developers can go to challenge proprietary vendors by building and innovating on software that was once only available (without source no less) with high license costs. This drives communities to build things like operating systems (Linux), programming languages (Go), message queues (Apache ActiveMQ), and web servers (`httpd`). Even companies that originally rejected open source are starting to come around by open sourcing their technologies and contributing to existing communities. As open source and open ecosystems have become the norm, we're starting to see a lot of the innovation in software technology coming directly from open source communities (e.g., Apache Spark, Docker, and Kubernetes).

Disruption

The confluence of these two factors, service design and technology evolution, is lowering the barrier for entry to anyone with a good idea to start experimenting and trying to build new services. You can learn to program, use advanced frameworks, and leverage on-demand computing for next to nothing. You can post to social networks, blog, and carry out bi-directional conversations with potential users of your service for free. With the fluidity of our business markets, any one of the over-the-weekend startups can put a legacy company out of business.

And this fact scares most CIOs and CEOs. As software quickly becomes the mechanism by which companies build digital services, experiences, and differentiation, many are realizing that they must become software companies in their respective verticals. Gone are the days of massive outsourcing and treating IT as a commodity or cost center. For companies to stay truly competitive, they must embrace software as a differentiator and to do that, they must embrace organization agility.

Embrace Organization Agility

Companies in the industrial-era thinking of the 20th century are not built for agility. They are built to maximize efficiencies, reduce variability in processes, eliminate creative thinking in workers, and place workers into boxes the way you would organize an assembly line. They are built like a machine to take inputs, apply a highly tuned process, and create outputs. They are structured with top-down hierarchical management to facilitate this machine-like thinking. Changing the machine requires 18-month planning cycles. Information from the edge goes through many layers of management and translation to get to the top, where decisions are made and handed back down. This organizational approach works great when creating products and trying to squeeze every bit of efficiency out of a process, but does not work for delivering services.

Efficiency   Flexibility

Customers don't fit in neat boxes or processes. They show up whenever they want. They want to talk to a customer service representative, not an automated phone system. They ask for things that aren't on the menu. They need to input something that isn't on the form. Customers want convenience. They want a conversation. And they get mad if they have to wait.

This means our customer-facing services need to account for variability. They need to be able to react to the unexpected. This is at odds with efficiency. Customers want to have a conversation through a service you provide them, and if that service isn't sufficient for solving their needs, you need loud, fast feedback about what's helping solve their needs or getting in their way. This feed-

back can be used by the maintainers of the service to quickly adjust the service and interaction models to better suit users. You cannot wait for decisions to bubble up to the top and through 18-month planning cycles; you need to make decisions quickly with the information you have at the edges of your business. You need autonomous, purpose-driven, self-organizing teams who are responsible for delivering a compelling experience to their customers (paying customers, business partners, peer teams, etc.). Rapid feedback cycles, autonomous teams, shared purpose, and conversation are the prerequisites that organizations must embrace to be able to navigate and live in a post-industrial, unknown, uncharted body of business disruption.

No book on microservices would be complete without quoting Conway's law: "organizations which design systems…are constrained to produce designs which are copies of the communication structures of these organizations."

To build agile software systems, we must start with building agile organizational structures. This structure will facilitate the prerequisites we need for microservices, but what technology do we use? Building distributed systems is hard, and in the subsequent sections, we'll take a look at the problems you must keep in mind when building and designing these services.

What Is a Microservice Architecture?

Microservice architecture (MSA) is an approach to building software systems that decomposes business domain models into smaller, consistent, bounded-contexts implemented by services. These services are isolated and autonomous yet communicate to provide some piece of business functionality. Microservices are typically implemented and operated by small teams with enough autonomy that each team and service can change its internal implementation details (including replacing it outright!) with minimal impact across the rest of the system.

Independence
aids Agility
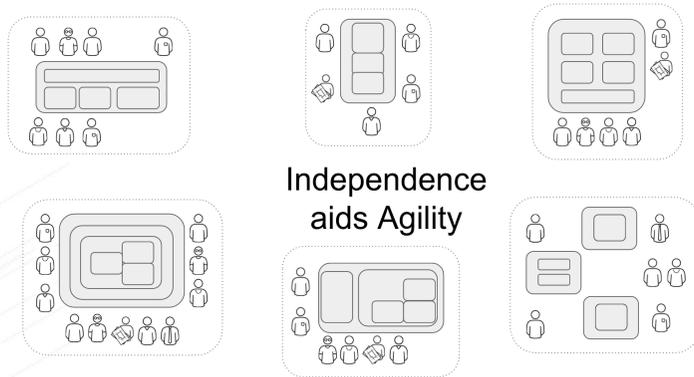
Teams communicate through *promises*, which are a way a service can publish intentions to other components or systems that may wish to use the service. They specify these promises with interfaces of their services and via wikis that document their services. If there isn't enough documentation, or the API isn't clear enough, the service provider hasn't done his job. A little more on promises and promise theory in the next section.

Each team would be responsible for designing the service, picking the right technology for the problem set, and deploying, managing and waking up at 2 a.m. for any issues. For example, at Amazon, there is a single team that owns the tax-calculation functionality that gets called during checkout. The models within this service (Item, Address, Tax, etc.) are all understood to mean "within the context of calculating taxes" for a checkout; there is no confusion about these objects (e.g., is the item a return item or a checkout item?). The team that owns the tax-calculation service designs, develops, and operates this service. Amazon has the luxury of a mature set of self-service tools to automate a lot of the build/deploy/operate steps, but we'll come back to that.

With microservices, we can scope the boundaries of a service, which helps us:

- Understand what the service is doing without being tangled into other concerns in a larger application
- Quickly build the service locally
- Pick the right technology for the problem (lots of writes? lots of queries? low latency? bursty?)

- Test the service
- Build/deploy/release at a cadence necessary for the business, which may be independent of other services
- Identify and horizontally scale parts of the architecture where needed
- Improve resiliency of the system as a whole

Microservices help solve the "how do we decouple our services and teams to move quickly at scale?" problem. It allows teams to focus on providing the service and making changes when necessary and to do so without costly synchronization points. Here are things you won't hear once you've adopted microservices:

- Jira tickets
- Unnecessary meetings
- Shared libraries
- Enterprise-wide canonical models

Is microservice architecture right for you? Microservices have a lot of benefits, but they come with their own set of drawbacks. You can think of microservices as an optimization for problems that require the ability to change things quickly at scale but with a price. It's not efficient. It can be more resource intensive. You may end up with what looks like duplication. Operational complexity is a lot higher. It becomes very difficult to understand the system holistically. It becomes significantly harder to debug problems. In some areas you may have to relax the notion of transaction. Teams may not have been designed to work like this.

Not every part of the business has to be able to change on a dime. A lot of customer-facing applications do. Backend systems may not. But as those two worlds start to blend together we may see the forces that justify microservice architectures push to other parts of the system.

Challenges

Designing cloud-native applications following a microservices approach requires thinking differently about how to build, deploy, and operate them. We can't just build our application thinking we

know all the ways it will fail and then just prevent those. In complex systems like those built with microservices, we must be able to deal with uncertainty. This section will identify five main things to keep in mind when developing microservices.

Design for Faults

In complex systems, things fail. Hard drives crash, network cables get unplugged, we do maintenance on the live database instead of the backups, and VMs disappear. Single faults can be propagated to other parts of the system and result in cascading failures that take an entire system down.

Traditionally, when building applications, we've tried to predict what pieces of our app (e.g., n-tier) might fail and build up a wall big enough to keep things from failing. This mindset is problematic at scale because we cannot always predict what things can go wrong in complex systems. Things *will* fail, so we must develop our applications to be resilient and handle failure, not just prevent it. We should be able to deal with faults gracefully and not let faults propagate to total failure of the system.

Building distributed systems is different from building shared-memory, single process, monolithic applications. One glaring difference is that communication over a network is not the same as a local call with shared memory. Networks are inherently unreliable. Calls over the network can fail for any number of reasons (e.g., signal strength, bad cables/routers/switches, and firewalls), and this can be a major source of bottlenecks. Not only does network unreliability have performance implications on response times to clients of your service, but it can also contribute to upstream systems failure.

Latent network calls can be very difficult to debug; ideally, if your network calls cannot complete successfully, they fail immediately, and your application notices quickly (e.g., through IOException). In this case we can quickly take corrective action, provide degraded functionality, or just respond with a message stating the request could not be completed properly and that users should try again later. But errors in network requests or distributed applications aren't always that easy. What if the downstream application you must call takes longer than normal to respond? This is killer because now your application must take into account this slowness by throttling requests, timing out downstream requests, and potentially

stalling all calls through your service. This backup can cause upstream services to experience slowdown and grind to a halt. And it can cause cascading failures.

Design with Dependencies in Mind

To be able to move fast and be agile from an organization or distributed-systems standpoint, we have to design systems with dependency thinking in mind; we need loose coupling in our teams, in our technology, and our governance. One of the goals with microservices is to take advantage of autonomous teams and autonomous services. This means being able to change things as quickly as the business needs without impacting those services around you or the system at large. This also means we should be able to depend on services, but if they're not available or are degraded, we need to be able to handle this gracefully.

In his book *Dependency Oriented Thinking* (InfoQ Enterprise Software Development Series), Ganesh Prasad hits it on the head when he says, "One of the principles of creativity is to *drop a constraint*. In other words, you can come up with creative solutions to problems if you mentally eliminate one or more dependencies." The problem is our organizations were built with efficiency in mind, and that brings a lot of tangled dependencies along.

For example, when you need to consult with three other teams to make a change to your service (DBA, QA, and Security), this is not very agile; each one of these synchronization points can cause delays. It's a brittle process. If you can shed those dependencies or build them into your team (we definitely can't sacrifice safety or security, so build those components into your team), you're free to be creative and more quickly solve problems that customers face or the business foresees without costly people bottlenecks.

Another angle to the dependency management story is what to do with legacy systems. Exposing details of backend legacy systems (COBOL copybook structures, XML serialization formats used by a specific system, etc.) to downstream systems is a recipe for disaster. Making one small change (customer ID is now 20 numeric characters instead of 16) now ripples across the system and invalidates assumptions made by those downstream systems, potentially breaking them. We need to think carefully about how to insulate the rest of the system from these types of dependencies.

Design with the Domain in Mind

Models have been used for centuries to simplify and understand a problem through a certain lens. For example, the GPS maps on our phones are great models for navigating a city while walking or driving. This model would be completely useless to someone flying a commercial airplane. The models they use are more appropriate to describe way points, landmarks, and jet streams. Different models make more or less sense depending on the context from which they're viewed. Eric Evans's seminal book *Domain-Driven Design* (Addison-Wesley, 2004) helps us build models for complex business processes that can also be implemented in software. Ultimately the real complexity in software is not the technology but rather the ambiguous, circular, contradicting models that business folks sort out in their heads on the fly. Humans can understand models given some context, but computers need a little more help; these models and the context must be baked into the software. If we can achieve this level of modeling that is bound to the implementation (and vice versa), anytime the business changes, we can more clearly understand how that changes in the software. The process we embark upon to build these models and the language surrounding it take time and require fast feedback loops.

One of the tools Evans presents is identifying and explicitly separating the different models and ensuring they're cohesive and unambiguous within their own bounded context.
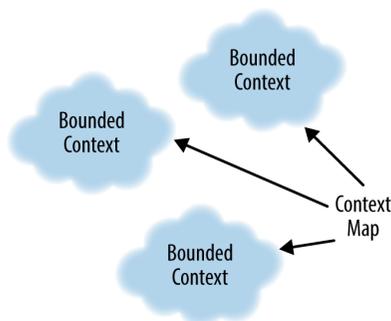
A bounded context is a set of domain objects that implement a model that tries to simplify and communicate a part of the business, code, and organization. For example, we strive for efficiency when designing our systems when we really need flexibility (sound famil-

iar?). In a simple auto-part application, we try to come up with a unified "canonical model" of the entire domain, and we end up with objects like `Part`, `Price`, and `Address`. If the inventory application used the "Part" object it would be referring to a type of part like a type of "brake" or "wheel." In an automotive quality assurance system, `Part` might refer to a very specific part with a serial number and unique identifier to track certain quality tests results and so forth. We tried diligently to efficiently reuse the same canonical model, but the issues of inventory tracking and quality assurance are different business concerns that use the `Part` object, semantically differently. With a bounded context, a `Part` would explicitly be modeled as `PartType` and be understood within that context to represent a "type of part," not a specific instance of a part. With two separate bounded contexts, these `Part` objects can evolve consistently within their own models without depending on one another in weird ways, and thus we've achieved a level of agility or flexibility.

This deep understanding of the domain takes time. It may take a few iterations to fully understand the ambiguities that exist in business models and properly separate them out and allow them to change independently. This is at least one reason starting off building microservices is difficult. Carving up a monolith is no easy task, but a lot of the concepts are already baked into the monolith; your job is to identify and carve it up. With a greenfield project, you cannot carve up anything until you deeply understand it. In fact, all of the microservice success stories we hear about (like Amazon and Netflix) all started out going down the path of the monolith before they successfully made the transition to microservices.

Design with Promises in Mind

In a microservice environment with autonomous teams and services, it's very important to keep in mind the relationship between service provider and service consumer. As an autonomous service team, you cannot place obligations on other teams and services because you do not own them; they're autonomous by definition. All you can do is choose whether or not to accept their promises of functionality or behavior. As a provider of a service to others, all you can do is *promise* them a certain behavior. They are free to trust you or not. Promise theory, a model first proposed by Mark Burgess in 2004 and covered in his book *In Search of Certainty* (O'Reilly, 2015),

is a study of autonomous systems including people, computers, and organizations providing service to each other.
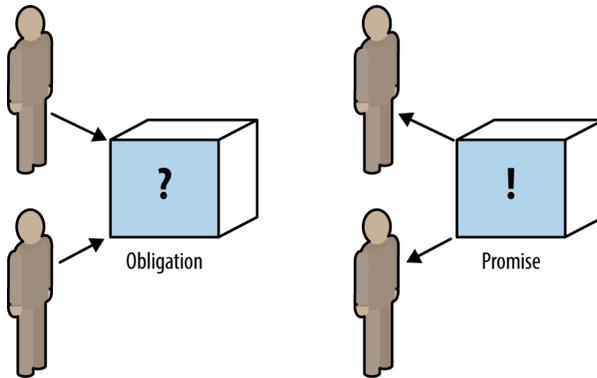
In terms of distributed systems, promises help articulate what a service *may* provide and make clear what assumptions can and cannot be made. For example, our team owns the book-recommendation service, and we promise a personalized set of book recommendations for a specific user you may ask about. What happens when you call our service, and one of our backends (the database that stores that user's current view of recommendations) is unavailable? We could throw exceptions and stack traces back to you, but that would not be a very good experience and could potentially blow up other parts of the system. Because we made a promise, we can try to do everything we can to keep it, including returning a default list of books, or a subset of every book. There are times when promises cannot be kept and identifying the best course of action should be driven by the desired experience or outcome for our users we wish to keep. The key here is the onus on our service to try to keep its promise (return some recommendations), even if our dependent services cannot keep theirs (the database was down). In the course of trying to keep a promise, it helps to have empathy for the rest of the system and the service quality we're trying to uphold.

Another way to look at a promise is as an agreed-upon exchange that provides value for both parties (like a producer and a consumer). But how do we go about deciding between two parties what is valuable and what promises we'd like to agree upon? If nobody calls our service or gets value from our promises, how useful is the service? One way of articulating the promise between consumers

and providers is driving promises with consumer-driven contracts. With consumer-driven contracts, we are able to capture the value of our promises with code or assertions and as a provider, we can use this knowledge to test whether we're upholding our promises.

Distributed Systems Management

At the end of the day, managing a single system is easier than a distributed one. If there's just one machine, and one application server, and there are problems with the system, we know where to look. If you need to make a configuration change, upgrade to a specific version, or secure it, it's still all in one physical and logical location. Managing, debugging, and changing it is easier. A single system may work for some use cases; but for ones where scale is required, we may look to leverage microservices. As we discussed earlier, however, microservices are not free; the trade-off for having flexibility and scalability is having to manage a complicated system.

Some quick questions about the manageability of a microservices deployment:

- How do we start and stop a fleet of services?
- How do we aggregate logs/metrics/SLAs across microservices?
- How do we discover services in an elastic environment where they can be coming, going, moving, etc.?
- How do we do load balancing?
- How do we learn about the health of our cluster or individual services?
- How do we restart services that have fallen over?
- How do we do fine-grained API routing?
- How do we secure our services?
- How do we throttle or disconnect parts of a cluster if it starts to crash or act unexpectedly?
- How do we deploy multiple versions of a service and route to them appropriately?
- How do we make configuration changes across a large fleet of services?

- How do we make changes to our application code and configuration in a safe, auditable, repeatable manner?

These are not easy problems to solve. The rest of the book will be devoted to getting Java developers up and running with microservices and able to solve some of the problems listed. The full, complete list of how-to for the preceding questions (and many others) should be addressed in a second edition of this book.

Technology Solutions

Throughout the rest of the book, we'll introduce you to some popular technology components and how they help solve some of the problems of developing and delivering software using a microservices architecture. As touched upon earlier, microservices is not just a technological problem, and getting the right organizational structure and teams in place to facilitate microservices is paramount. Switching from SOAP to REST doesn't make a microservices architecture.

The first step for a Java development team creating microservices is to get something working locally on their machine! This book will introduce you to three opinionated Java frameworks for working with microservices: Spring Boot, Dropwizard, and WildFly Swarm. Each framework has upsides for different teams, organizations, and approaches to microservices. Just as is the norm with technology, some tools are a better fit for the job or the team using them. These are not the only frameworks to use. There are a couple that take a reactive approach to microservices like Vert.x and Lagom. The mindshift for developing with an event-based model is a bit different and requires a different learning curve so for this book we'll stick with a model that most enterprise Java developers will find comfortable.

The goal of this book is to get you up and running with the basics for each framework. We'll dive into a couple advanced concepts in the last chapter, but for the first steps with each framework, we'll assume a hello-world microservice application. This book is not an all-encompassing reference for developing microservices; each section will leave you with links to reference material to explore more as needed. We will iterate on the hello-world application by creating multiple services and show some simple interaction patterns.

The final iteration for each framework will look at concepts like bulkheading and promise theory to make our services resilient in the face of faults. We will dig into parts of the NetflixOSS stack like Hystrix that can make our lives easier for implementing this functionality. We will discuss the pros and cons of this approach and explore what other options exist.

As we go through the examples, we'll also discuss the value that Linux containers bring to the microservices story for deployment, management, and isolation as well as local development. Docker and Kubernetes bring a wealth of simplifications for dealing with distributed systems at scale, so we'll discuss some good practices around containers and microservices.

In the last section of the book, we'll leave you with a few thoughts on distributed configuration, logging, metrics, and continuous delivery.

Preparing Your Environment

We will be using Java 1.8 for these examples and building them with Maven. Please make sure for your environment you have the following prerequisites installed:

- JDK 1.8
- Maven 3.2+
- Access to a command-line shell (bash, PowerShell, cmd, Cygwin, etc.)

The Spring ecosystem has some great tools you may wish to use either at the command line or in an IDE. Most of the examples will stick to the command line to stay IDE neutral and because each IDE has its own way of working with projects. For Spring Boot, we'll use the Spring Boot CLI 1.3.3.

Alternative IDEs and tooling for Spring:

- Eclipse based IDE: Spring Tool Suite
- Spring Initializr web interface

For both Dropwizard and WildFly Swarm, we'll use JBoss Forge CLI and some addons to create and interact with our projects:

- JBoss Forge 3.0+

Alternative IDEs and tooling for Spring, Dropwizard, or WildFly Swarm projects (and works great with JBoss Forge):

- Eclipse based IDE: JBoss Developer Studio
- Netbeans
- IntelliJ IDEA

Finally, when we build and deploy our microservices as Docker containers running inside of Kubernetes, we'll want the following tools to bootstrap a container environment on our machines:

- Vagrant 1.8.1
- VirtualBox 5.0.x
- Container Development Kit 2.x
- Kubernetes/Openshift CLI
- Docker CLI (optional)

Spring Boot for Microservices

Spring Boot is an opinionated Java framework for building microservices based on the Spring dependency injection framework. Spring Boot allows developers to create microservices through reduced boilerplate, configuration, and developer friction. This is a similar approach to the two other frameworks we'll look at. Spring Boot does this by:

- Favoring automatic, conventional configuration by default
- Curating sets of popular starter dependencies for easier consumption
- Simplifying application packaging
- Baking in application insight (e.g., metrics and environment info)

Simplified Configuration

Spring historically was a nightmare to configure. Although the framework improved upon other high-ceremony component models (EJB 1.x, 2.x, etc.), it did come along with its own set of heavyweight usage patterns. Namely, Spring required a lot of XML configuration and a deep understanding of the individual beans needed to construct JdbcTemplates, JmsTemplates, BeanFactory lifecycle hooks, servlet listeners, and many other components. In fact, writing a simple "hello world" with Spring MVC required understanding of DispatcherServlet and a whole host of Model-View-

Controller classes. Spring Boot aims to eliminate all of this boilerplate configuration with some implied conventions and simplified annotations—although, you can still finely tune the underlying beans if you need to.

Starter Dependencies

Spring was used in large enterprise applications that typically leveraged lots of different technology to do the heavy lifting: JDBC databases, message queues, file systems, application-level caching, etc. A developer would have to stop what she's doing, switch cognitive contexts, figure out what dependencies belonged to which piece of functionality ("Oh, I need the JPA dependencies!") and spend lots of time sorting out versioning mismatches or issues that would arise when trying to use these various pieces together. Spring Boot offers a large collection of curated sets of libraries for adding these pieces of functionality. These starter modules allow you to add things like:

- JPA persistence
- NoSQL databases like MongoDB, Cassandra, and Couchbase
- Redis caching
- Tomcat/Jetty/Undertow servlet engine
- JTA transactions

Adding a submodule to your application brings in the curated set of transitive dependencies and versions that are known to work together saving developers from having to sort out dependencies themselves.

Application Packaging

Spring Boot really is a set of bootstrap libraries with some convention for configurations, but there's no reason why you couldn't run a Spring Boot application inside your existing application servers (as a WAR). The idiom that most developers who use Spring Boot prefer is the self-contained JAR packaging for their application. This means Spring Boot packages all dependencies and application code into a self-contained JAR with a flat class loader. This makes it easier to understand application startup, dependency ordering, and log statements; but more importantly, it helps reduce the number of moving pieces required to take an app safely to production. This

means you don't take an app and chuck it into an app server; the app, once it's built, is ready to run as is—standalone—including embedding its own servlet container if it uses servlets. That's right, a simple `java -jar <name.jar>` is enough to start your application now! Spring Boot, Dropwizard, and WildFly Swarm all follow this pattern of packaging everything into an executable uber JAR.

But what about management things we typically expect out of an application server?

Production Ready

Spring Boot ships with a module called actuator which enables things like metrics and statistics about your application. For example, we can collect logs, view metrics, perform thread dumps, show environment variables, understand garbage collection, and show what beans are configured in the BeanFactory. You can expose this information via HTTP, JMX, or you can even log in directly to the process via SSH.

With Spring Boot, we can leverage the power of the Spring Framework and reduce boilerplate configuration and code to more quickly build powerful, production-ready microservices. Let's see how.

Getting Started

We're going to use the Spring Boot command-line interface (CLI) to bootstrap our first Spring Boot application (the CLI uses Spring Initializr under the covers). You are free to explore the different ways to do this if you're not comfortable with the CLI. Alternatives include using Spring Initializr plug-ins for your favorite IDE or visiting the web version of Spring Initializr. The Spring Boot CLI can be installed a few different ways, including through package managers and by downloading it straight from the website. Check for instructions on installing the CLI most appropriate for your development environment.

Once you've installed the CLI tools, you should be able to check the version of Spring you have:

```
$ spring --version

Spring CLI v1.3.3.RELEASE
```

If you can see a version for your installation of the CLI, congrats! Now navigate to a directory where you want to host your examples from the book and run the following command:

```
spring init --build maven --groupId com.redhat.examples \
--version 1.0 --java-version 1.8 --dependencies web \
--name hola-springboot  hola-springboot
```

After running this command, you should have a directory named *hola-springboot* with a complete Spring Boot application. If you run the command and end up with a *demo.zip*, then just unzip it and continue. Let's take a quick look at what those command-line options are.

`--build`
> The build-management tool we want to use. `maven` or `gradle` are the two valid options at this time.

`--groupId`
> The `groupId` to use in our maven coordinates for our *pom.xml*; unfortunately this does not properly extend to the Java package names that get created. These need to be modified by hand.

`--version`
> The version of our application; will be used in later iterations, so set to 1.0.

`--java-version`
> Allows us to specify the build compiler version for the JDK.

`--dependencies`
> This is an interesting parameter; we can specify fully baked sets of dependencies for doing common types of development. For example, `web` will set up Spring MVC and embed an internal servlet engine (Tomcat by default; Jetty and Undertow as options). Other convenient dependency bundles/starters include `jpa`, `security`, and `cassandra`).

Now if you navigate to the *hola-springboot* directory, try running the following command:

```
$ mvn spring-boot:run
```

If everything boots up without any errors, you should see some logging similar to this:

```
2016-03-25 10:57:08.920  [main] AnnotationMBeanExporter
: Registering beans for JMX exposure on startup
2016-03-25 10:57:08.982  [main] TomcatEmbeddedServletContainer
: Tomcat started on port(s): 8080 (http)
2016-03-25 10:57:08.987  [main] HolaSpringbootApplication
: Started HolaSpringbootApplication in 1.0 seconds
(JVM running for 4.7)
```

Congrats! You have quickly gotten a Spring Boot application up and running! You can even navigate to *http://localhost:8080* in your browser and should see the following output:
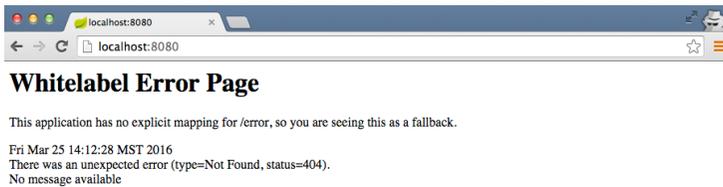
This default error page is expected since our application doesn't do anything yet! Let's move on to the next section to add a REST endpoint to put together a hello-world use case!

Hello World

Now that we have a Spring Boot application that can run, let's add some simple functionality. We want to expose an HTTP/REST endpoint at */api/hola* that will return "Hola Spring Boot from X" where X is the IP address where the service is running. To do this, navigate to *src/main/java/com/example*. This location should have been created for you if you followed the preceding steps; remember, the `groupId` we passed to the `spring init` program did not apply groupId to the Java package hierarchy, and we've left it as it is which should be "com.example". Then create a new Java class called `HolaRestController`, as shown in Example 2-1. We'll add a method named `hola()` that returns a string along with the IP address of where the service is running. You'll see in Chapter 5, in our load balancing and service discovery sections, how the host IPs can be used to demonstrate proper failover, loadbalancing, etc.

Example 2-1. src/main/java/com/example/HolaRestController.java

```java
public class HolaRestController {

    public String hola() throws UnknownHostException {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
        }
        return "Hola Spring Boot de " + hostname;
    }

}
```

Add the HTTP Endpoints

At this point, this piece of code is just a POJO (plain old Java object) and you could (and should) write a unit test that verifies its behavior. To expose this as a REST endpoint, we're going to make use of the following annotations in Example 2-2:

@RestController
Tell Spring this is an HTTP controller capable of exposing HTTP endpoints (GET, PUT, POST, etc.).

@RequestMapping
Map specific parts of the HTTP URI path to classes, methods, and parameters in the Java code.

Note, import statements are omitted.

Example 2-2. src/main/java/com/example/HolaRestController.java

```java
@RestController
@RequestMapping("/api")
public class HolaRestController {

    @RequestMapping(method = RequestMethod.GET, value = "/hola",
        produces = "text/plain")
    public String hola() throws UnknownHostException {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
```

```
        }
        return "Hola Spring Boot de " + hostname;
    }

}
```

In this code, all we've done is add the aforementioned annotations. For example, `@RequestMapping("/api")` at the Class level says "map any method-level HTTP endpoints under this root URI path." When we add `@RequestMapping(method = RequestMethod.GET, value = "/hola", produces = "text/plain")`, we are telling Spring to expose an HTTP GET endpoint at `/hola` (which will really be */api/hola*) and map requests with media type of `Accept: text/plain` to this method. Spring Boot defaults to using an embedded Tomcat servlet container, but this can be switched to other options like Undertow or Jetty.

If we build our application and run `spring-boot:run` again, we should be able to reach our HTTP endpoint:

```
$ mvn clean package spring-boot:run
```

Now if we point our browser to *http://localhost:8080/api/hola*, we should see a response similar to:

What if we want to add some environment-aware configuration to our application? For example, instead of saying "Hola," maybe we want to say "Guten Tag" if we deploy our app in production for German users? We need a way to inject properties to our app.

Externalize Configuration

Spring Boot makes it easy to use external property sources like properties files, command-line arguments, the OS environment, or Java System properties. We can even bind entire "classes" of properties to objects in our Spring context. For example, if I want to bind all `helloapp.*` properties to my `HolaRestController`, I can add `@ConfigurationProperties(prefix="helloapp")`, and Spring Boot will automatically try to bind `helloapp.foo` and `helloapp.bar`

to Java Bean properties in the `HolaRestController` class. Let's define a new property in *src/main/resources/application.properties* called `helloapp.saying`. The *application.properties* file was automatically created for us when we created our project. Note we could change the file name to *application.yml* and Spring would still recognize it as a YAML file as the source of properties.

Let's add a new property to our *src/main/resources/application.properties* file:

```
helloapp.saying=Guten Tag aus
```

In the `HolaRestController` in Example 2-3, let's add the `@ConfigurationProperties` annotation and our new `saying` field. Note we also need setters and getters.

Example 2-3. src/main/java/com/example/HolaRestController.java

```java
@RestController
@RequestMapping("/api")
@ConfigurationProperties(prefix="helloapp")
public class HolaRestController {

    private String saying;

    @RequestMapping(method = RequestMethod.GET, value = "/hola",
        produces = "text/plain")
    public String hola() throws UnknownHostException {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
        }
        return saying + " " + hostname;
    }

    public String getSaying() {
        return saying;
    }

    public void setSaying(String saying) {
        this.saying = saying;
    }
}
```

```
$ mvn clean package spring-boot:run
```

Now if we navigate to *http://localhost:8080/api/hola*, we should see the German version of the saying:

We can now externalize properties that would change depending on the environment in which we are running. Things like service URIs, database URIs and passwords, and message queue configurations would all be great candidates for external configuration. Don't overdo it though; not everything needs to change depending on the environment in which it runs! Ideally an application would be configured exactly the same in all environments including timeouts, thread pools, retry thresholds, etc.

Expose Application Metrics and Information

If we want to put this microservice into production, how will we monitor it? How can we get any insight about how things are running? Often our microservices are black boxes unless we explicitly think through how we want to expose metrics to the outside world. Spring Boot comes with a prepackaged starter called actuator that makes doing this a breeze.

Let's see what it takes to enable the actuator. Open up the *pom.xml* file for your `hola-springboot` microservice and add the following Maven dependency within the `<dependencies>...</dependencies>` section:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Now restart your microservice by stopping it and running:

```
$ mvn clean package spring-boot:run
```

Just by adding the actuator dependency, our application now has a lot of information exposed that would be very handy for debugging

or general microservice insight. Try hitting the following URLs and examine what gets returned:

- *http://localhost:8080/beans*
- *http://localhost:8080/env*
- *http://localhost:8080/health*
- *http://localhost:8080/metrics*
- *http://localhost:8080/trace*
- *http://localhost:8080/mappings*

Here's an example of what the *http://localhost:8080/env* endpoint looks like:

Exposing runtime insight like this relieves the developer to just focus on writing code for the microservice that delivers business value. Delegating to frameworks to do heavy lifting and boilerplate is definitely a good idea.

How to Run This Outside of Maven?

Up to this point we've been thinking through development and building our hello-world microservice from the perspective of a developer's laptop using Maven. But what if you want to distribute

your microservice to others or run it in a live environment (development, QA, production)?

Luckily, with Spring Boot it only takes a few steps to get us ready for shipment and production. Spring Boot prefers atomic, executable JARs with all dependencies packed into a flat classpath. This means the JAR that we create as part of a call to `mvn clean package` is executable and contains all we need to run our microservice in a Java environment! To test this out, go to the root of our `hola-springboot` microservice project and run the following commands:

```
$ mvn clean package
$ java -jar target/hola-springboot-1.0.jar
```

If your project was named `demo` instead of `hola-springboot`, then substitute the properly named JAR file (*demo-1.0.jar*).

That's it!

We'll notice this sort of idiom when we explore Dropwizard and WildFly Swarm.

Calling Another Service

In a microservice environment, each service is responsible for providing the functionality or service to other collaborators. As we've discussed in the first chapter, building distributed systems is hard, and we cannot abstract away the network or the potential for failures. We will cover how to build resilient interactions with our dependencies in Chapter 5. In this section, however, we will just focus on getting a service to talk to a dependent service.

If we wish to extend the hello-world microservice, we will need to create a service to which we can call using Spring's REST client functionality. For this example and the rest of the examples, we'll use a backend service and modify our service to reach out to the backend to generate the greetings we want to be able to use.

If you look in the source code for this book, we'll see a Maven module called `backend` which contains a very simple HTTP servlet that

can be invoked with a GET request and query parameters. The code for this backend is very simple, and does not use any of the microservice frameworks (Spring Boot, Dropwizard, or WildFly Swarm). We have created a `ResponseDTO` object that encapsulates `time`, `ip`, and `greeting` fields. We also leverage the awesome Jackson library for JSON data binding, as seen here:

```java
public class BackendHttpServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException {

        resp.setContentType("application/json");

        ObjectMapper mapper = new ObjectMapper();
        String greeting = req.getParameter("greeting");

        ResponseDTO response = new ResponseDTO();
        response.setGreeting(greeting +
            " from cluster Backend");
        response.setTime(System.currentTimeMillis());
        response.setIp(getIp());

        PrintWriter out = resp.getWriter();
        mapper.writerWithDefaultPrettyPrinter()
                .writeValue(out, response);
    }

    private String getIp() {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                  .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
        }
        return hostname;
    }
}
```

To start up the backend service on port `8080`, navigate to the `back end` directory and run the following:

```
$ mvn clean install jetty:run
```

The `backend` project uses the Maven Jetty plug-in, which allows us to quickly boot up our app using `mvn jetty:run`.

This service is exposed at */api/backend* and takes a query parameter
`greeting`. For example, when we call this service with this path */api/
backend?greeting=Hello*, then the backend service will respond with
a JSON object like this (can also visit this URL with your browser):

```
$ curl -X GET http://localhost:8080/api/backend?greeting=Hello
```

We get something like this:

```
{
  "greeting" : "Hello from cluster Backend",
  "time" : 1459189860895,
  "ip" : "172.20.10.3"
}
```

We will create a new HTTP endpoint, */api/greeting*, in our Spring
Boot `hola-springboot` example and use Spring to call this backend!

Create a new class in *src/main/java/com/example* called `GreeterRest
Controller` and fill it in similarly to how we filled it in for the `HolaR
estController` (see Example 2-4).

Example 2-4. src/main/java/com/example/GreeterRestController.java

```java
@RestController
@RequestMapping("/api")
@ConfigurationProperties(prefix="greeting")
public class GreeterRestController {

    private String saying;

    private String backendServiceHost;

    private int backendServicePort;

    @RequestMapping(value = "/greeting",
    method = RequestMethod.GET, produces = "text/plain")
    public String greeting(){
        String backendServiceUrl =
            String.format(
            "http://%s:%d/hello?greeting={greeting}",
            backendServiceHost, backendServicePort);
        System.out.println("Sending to: " + backendServiceUrl);
        return backendServiceUrl;
    }
}
```

I've left out the getters/setters for the properties in this class, but
make sure to have them in your source code! Note we are using the

`@ConfigureProperties` annotation again to configure a block of configuration for our REST controller here, although this time we are using the `greeting` prefix. We also create a GET endpoint like we did with the `hola` service, and all it returns at the moment is a string with the values of the backend service `host` and `port` concatenated (and these values are injected in via the `@ConfigurePro perties` annotation). Let's add the `backendServiceHost` and `backendServicePort` to our *application.properties* file:

```
greeting.saying=Hola Spring Boot
greeting.backendServiceHost=localhost
greeting.backendServicePort=8080
```

Next, we're going to use Spring's `RestTemplate` to do the invocation of the remote service. Following a long-lived Spring convention with its template patterns, the `RestTemplate` wraps common HTTP/REST idioms inside of this convenient wrapper abstraction which then handles all the connections and marshalling/unmarshalling the results of an invocation. `RestTemplate` uses the native JDK for HTTP/network access, but you can swap that out for Apache HttpComponents, OkHttp, Netty, or others.

Here's what the source looks like when using the `RestTemplate` (again, the getters/setters omitted, but required). We are communicating with the backend service by constructing a URL based on the host and port that have been injected and we add a GET query parameter called `greeting`. The value we send to the backend service for the `greeting` parameter is from the `saying` field of the `GreeterRestController` object, which gets injected as part of the configuration when we added the `@ConfigurationProperties` annotation (Example 2-5).

Example 2-5. src/main/java/com/example/GreeterRestController.java

```java
@RestController
@RequestMapping("/api")
@ConfigurationProperties(prefix="greeting")
public class GreeterRestController {

    private RestTemplate template = new RestTemplate();

    private String saying;

    private String backendServiceHost;
```

```java
    private int backendServicePort;

    @RequestMapping(value = "/greeting",
    method = RequestMethod.GET, produces = "text/plain")
    public String greeting(){

        String backendServiceUrl =
            String.format(
            "http://%s:%d/api/backend?greeting={greeting}",
            backendServiceHost, backendServicePort);

        BackendDTO response = template.getForObject(
        backendServiceUrl, BackendDTO.class, saying);

        return response.getGreeting() + " at host: " +
        response.getIp();
    }
}
```

Let's add the `BackendDTO` class, which is used to encapsulate responses from the backend (Example 2-6).

Example 2-6. src/main/java/com/example/BackendDTO.java

```java
public class BackendDTO {

    private String greeting;
    private long time;
    private String ip;

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    public long getTime() {
        return time;
    }

    public void setTime(long time) {
        this.time = time;
    }

    public String getIp() {
        return ip;
    }
```

```
    public void setIp(String ip) {
        this.ip = ip;
    }
}
```

Now let's build the microservice and verify that we can call this new Greeting endpoint and that it properly calls the backend. First, let's start the backend if it's not already running. Navigate to the `backend` directory of the source code that comes with this application and run it:

```
$ mvn clean install jetty:run
```

Next let's build and run our Spring Boot microservice. Let's also configure this service to run on a different port than it's default port (8080) so that it doesn't collide with the backend service which is already running on port `8080`.

```
$ mvn clean install spring-boot:run -Dserver.port=9090
```

Later on in the book we can see how running these microservices in their own Linux container removes the restriction of port swizzling at runtime. Now, let's navigate our browser to *http://localhost: 9090/api/greeting* to see if our microservice properly calls the back-end and displays what we're expecting:


```
Hola Spring Boot from cluster Backend at host: 192.168.16.153
```

Where to Look Next

In this chapter, we learned what Spring Boot was, how it's different from traditional WAR/EAR deployments, and some simple use cases, including exposing an HTTP/REST endpoint, externalizing configuration, metrics, and how to call another service. This is just scratching the surface, and if you're interested in learning more about Spring Boot, please take a look at the following links and book:

- Spring Boot
- Spring Boot Reference Guide
- *Spring Boot in Action*
- Spring Boot on GitHub
- Spring Boot Samples on GitHub

Dropwizard for Microservices

Dropwizard was created well before either Spring Boot or WildFly Swarm (the other two microservices frameworks we're looking at in this book). Its first release, v0.1.0, came out December 2011. At the time of this writing, v0.9.2 is the latest, and 1.0 is just around the corner. Dropwizard was created by Coda Hale at Yammer to power the company's distributed-systems architectures (now called microservices!), heavily leveraging the JVM. It started out as a set of glue code to combine some powerful libraries for writing REST web services and has evolved since then, although it still maintains its identity as a minimalist, production-ready, easy-to-use web framework.

Dropwizard is an opinionated framework like Spring Boot; however, it's a little more prescriptive than Spring Boot. There are some components that are just part of the framework and cannot be easily changed. The sweet-spot use case is writing REST-based web applications/microservices without too many fancy frills. For example, Dropwizard has chosen the Servlet container (Jetty), REST library (Jersey), and serialization and deserialization (Jackson) formats for you. Changing them out if you want to switch (i.e., changing the servlet container to Undertow) isn't very straightforward.

Dropwizard also doesn't come with a dependency-injection container (like Spring or CDI). You can add one, but Dropwizard favors keeping development of microservices simple, with no magic. Spring Boot hides a lot of the underlying complexity from you, since Spring under the covers is pretty complex (i.e., spinning up all the

beans actually needed to make Spring run is not trivial) and hides a lot of bean wiring with Java Annotations. Although annotations can be handy and save a lot of boilerplate in some areas, when debugging production applications, the more magic there is, the more difficult it is. Dropwizard prefers to keep everything out in the open and very explicit about what's wired up and how things fit together. If you need to drop into a debugger, line numbers and stack traces should match up very nicely with the source code.

Just like Spring Boot, Dropwizard prefers to bundle your entire project into one, executable uber JAR. This way, developers don't worry about which application server it needs to run in and how to deploy and configure the app server. Applications are not built as WARs and subject to complicated class loaders. The class loader in a Dropwizard application is flat, which is a stark difference from trying to run your application in an application server where there may be many hierarchies or graphs of class loaders. Figuring out class load ordering, which can vary between servers, often leads to a complex deployment environment with dependency collisions and runtime issues (e.g., NoSuchMethodError). Running your microservices in their own process gives isolation between applications so you can tune each JVM individually as needed and monitor them using operating system tools very familiar to operations folks. Gone are the GC or OutOfMemoryExceptions which allow one application to take down an entire set of applications just because they share the same process space.

The Dropwizard Stack

Dropwizard provides some very intuitive abstractions on top of these powerful libraries to make it very easy to write production-ready microservices:

- Jetty for the servlet container
- Jersey for the REST/JAX-RS implementation
- Jackson for JSON serialization/deserialization
- Hibernate Validator
- Guava
- Metrics
- Logback + SLF4J

- JDBI for dealing with databases

Dropwizard is very opinionated in favor of "just get to writing code." The trade-off is if you want to tinker with the underlying stack, it's not very easy. On the other hand, getting up and running quickly so you can delivery business value is much easier than configuring the pieces yourself. Jetty, Jersey, and Jackson are well-known, production-grade libraries for writing REST-based services. Google's Guava library is around to provide utilities and immutable programming. The Dropwizard Metrics library is a very powerful metrics library that exposes more than enough insight to manage your services in production. In fact, the Metrics library is so powerful and popular it was spun out into its own project and can be used with Spring Boot or WildFly Swarm.

Dropwizard exposes the following abstractions with which a developer should be familiar. If you can understand these simple abstractions, you can be very productive with Dropwizard:

Application
Contains our `public void main()`

Environment
Where we put our servlets, resources, filters, health checks, tasks, commands, etc.

Configuration
How we inject environment- and application-specific configuration

Commands
Tells Dropwizard what action to take when starting our microservice (e.g., start a `server`)

Resources
REST/JAX-RS resources

Tasks
Admin tasks to be performed when managing the application (like change the log level or pause database connections)

When you run your Dropwizard application out of the box, one Jetty server gets created with two handlers: one for your application (`8080` by default) and one for the administration interface (`8081` by default). Dropwizard does this so you can expose your microservice

without exposing administration details over the same port (i.e., can keep 8081 behind a firewall so it's inaccessible). Things like metrics and health checks get exposed over the admin port, so take care to secure it properly.

Getting Started

Dropwizard doesn't have any fancy project-initialization helpers or Maven plug-ins. Getting started with Dropwizard follows a similar pattern to any plain-old Java project: use a Maven archetype, or add it to an existing application with whatever tools you currently use. You could also use JBoss Forge, which is a technology-agnostic Java project coordination and management tool that allows you to quickly create projects, add dependencies, add classes, etc. For this section, we'll just use Maven archetypes.

Choose a directory where you want to create your new Dropwizard project. Also verify you have Maven installed. You can run the following command from your operating system's command prompt, or you can use the following information in the following command to populate a dialog box or wizard for your favorite IDE:

```
$ mvn -B archetype:generate \
-DarchetypeGroupId=io.dropwizard.archetypes \
-DarchetypeArtifactId=java-simple -DarchetypeVersion=0.9.2 \
-DgroupId=com.redhat.examples.dropwizard
-DartifactId=hola-dropwizard -Dversion=1.0 -Dname=HolaDropwizard
```

Navigate to the directory that the Maven archetype generator created for us in hola-dropwizard and run the following command to build our project:

```
$ mvn clean install
```

You should have a successful build!

This uses the Dropwizard archetype *java-simple* to create our microservice. If you go into the *hola-dropwizard* directory, you should see this structure:

```
./src/main/java/com/redhat/examples/dropwizard/api
./src/main/java/com/redhat/examples/dropwizard/cli
./src/main/java/com/redhat/examples/dropwizard/client
./src/main/java/com/redhat/examples/dropwizard/core
./src/main/java/com/redhat/examples/dropwizard/db
./src/main/java/com/redhat/examples/dropwizard/health
./src/main/java/com/redhat/examples/dropwizard
```

```
        /HolaDropwizardApplication.java
./src/main/java/com/redhat/examples/dropwizard
    /HolaDropwizardConfiguration.java
./src/main/java/com/redhat/examples/dropwizard/resources
./src/main/resources
./src/main/resources/assets
./src/main/resources/banner.txt
./pom.xml
```

Note that Dropwizard creates a package structure for you that fol-
lows their convention:

api
> POJOs that define the objects used in your REST resources
> (some people call these objects *domain* objects or *DTOs*).

cli
> This is where your Dropwizard commands go (additional com-
> mands you wish to add to startup of your application).

client
> Client helper classes go here.

db
> Any DB related code or configuration goes here.

health
> Microservice-specific health checking that can be exposed at
> runtime in the admin interface goes here.

resources
> Our REST resource classes go here.

We also have the files *HolaDropwizardApplication.java* and *Hola-
DropwizardConfiguration.java*, which is where our configuration
and bootstrapping code goes. Take a look at the `HolaDropwizardAp
plication` class in Example 3-1, for example.

*Example 3-1. src/main/java/com/redhat/examples/dropwizard/
HolaDropwizardApplication.java*

```java
public class HolaDropwizardApplication extends
                Application<HolaDropwizardConfiguration> {

    public static void main(final String[] args)
        throws Exception {
        new HolaDropwizardApplication().run(args);
    }
```

```java
    @Override
    public String getName() {
        return "HolaDropwizard";
    }

    @Override
    public void initialize(
        Bootstrap<HolaDropwizardConfiguration> bootstrap) {

        // TODO: application initialization
    }

    @Override
    public void run(HolaDropwizardConfiguration configuration,
                    final Environment environment) {
        // TODO: implement application
    }

}
```

This class contains our `public static void main()` method, which doesn't do too much except call our microservice's `run()` method. It also has a `getName()` method, which is shown at startup. The `initialize()` and `run()` methods are the key places where we can bootstrap our application as we'll show in the next section.

The configuration class that was generated, `HolaDropwizardConfiguration`, is empty for now (Example 3-2).

Example 3-2. src/main/java/com/redhat/examples/dropwizard/
HolaDropwizardConfiguration.java

```java
public class HolaDropwizardConfiguration
    extends Configuration {
    // TODO: implement service configuration
}
```

Although Dropwizard doesn't have any special Maven plug-ins on its own, take a look at the *pom.xml* that was generated. We see that the Dropwizard dependencies are on the classpath and that we'll be using the `maven-shade-plugin` to package up our JAR as an uber JAR. This means all of our project's dependencies will be unpacked (i.e., all dependency JARs unpacked) and combined into a single JAR that our build will create. For that JAR, we use the `maven-jar-plugin` to make it executable.

One plug-in we do want to add is the `exec-maven-plugin`. With Spring Boot we were able to just run our microservice with `mvn spring-boot:run`. We want to be able to do the same thing with our Dropwizard application, so let's add the following plug-in within the `<build>` section of the *pom.xml*, shown in Example 3-3.

Example 3-3. pom.xml

```xml
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <configuration>
        <mainClass>
        com.redhat.examples.dropwizard.HolaDropwizardApplication
        </mainClass>
        <arguments>
            <argument>server</argument>
        </arguments>
    </configuration>
</plugin>
```

Now we can execute our application from the command line like this:

```
$ mvn exec:java
```

We should see something like what's in Example 3-4.

Example 3-4. HolaDropwizard

```
========================================================================


                    HolaDropwizard


========================================================================


INFO  [2016-03-27 21:54:22,279] io.dropwizard.server.DefaultServer...
: Registering jersey handler with root path prefix: /
INFO  [2016-03-27 21:54:22,291] io.dropwizard.server.DefaultServer...
: Registering admin handler with root path prefix: /
INFO  [2016-03-27 21:54:22,326] org.eclipse.jetty.setuid.SetUIDL...
: Opened application@5dee571c{HTTP/1.1}{0.0.0.0:8080}
INFO  [2016-03-27 21:54:22,327] org.eclipse.jetty.setuid.SetUIDL...
: Opened admin@f8bd099{HTTP/1.1}{0.0.0.0:8081}
INFO  [2016-03-27 21:54:22,329] org.eclipse.jetty.server.Server
: jetty-9.2.13.v20150730
INFO  [2016-03-27 21:54:22] io.dropwizard.jersey.DropwizardResou...
: The following paths were found for the configured resources:
```

```
NONE

INFO  [2016-03-27 21:54] org.eclipse.jetty.server.handler.Context...
: Started i.d.j.MutableServletContextHandler@1dfb9685{/,null,AVAI...
INFO  [2016-03-27 21:54:22] io.dropwizard.setup.AdminEnvironment:...

POST    /tasks/log-level (dropwizard.servlets.tasks.LogConfigurat...
POST    /tasks/gc (io.dropwizard.servlets.tasks.GarbageCollection...

WARN  [2016-03-27 21:54:22,695] io.dropwizard.setup.AdminEnvironm...
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!THIS APPLICATION HAS NO HEALTHCHECKS. THIS MEANS YOU WILL NEVER...
! IF IT DIES IN PRODUCTION, WHICH MEANS YOU WILL NEVER KNOW IF...
!LETTING YOUR USERS DOWN. YOU SHOULD ADD A HEALTHCHECK FOR EACH...
!     APPLICATION'S DEPENDENCIES WHICH FULLY (BUT LIGHTLY) TESTS...
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
INFO  [2016-03-27 21:54] org.eclipse.jetty.server.handler.ContextH...
: Started i.d.j.MutableServletContextHandler@4969dc6{/,null,AVA...
INFO  [2016-03-27 21:54:22,704] org.eclipse.jetty.server.ServerCo...
: Started application@5dee571c{HTTP/1.1}{0.0.0.0:8080}
INFO  [2016-03-27 21:54:22,705] org.eclipse.jetty.server.ServerCo...
: Started admin@f8bd099{HTTP/1.1}{0.0.0.0:8081}
INFO  [2016-03-27 21:54:22,705] org.eclipse.jetty.server.Server
: Started @2914ms
```

If you see the application start up, you can try to navigate in your browser to the default location for RESTful endpoints: *http://local host:8080*. You probably won't see too much:

```
{"code":404,"message":"HTTP 404 Not Found"}
```

If you try going to the admin endpoint *http://localhost:8081*, you should see a simple page with a few links. Try clicking around to see what kind of value is already provided out of the box for managing your microservice!
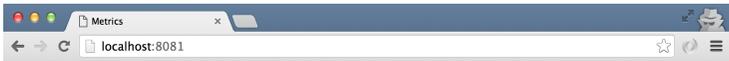
Hello World

Now that we have our Dropwizard microservice template ready to go, let's add a REST endpoint. We want to expose an HTTP/REST endpoint at *api/hola* that will return "Hola Dropwizard from X" where X is the IP address where the service is running. To do this, navigate to *src/main/java/com/redhat/examples/dropwizard/resources* (remember, this is the convention that Dropwizard follows for where to put REST resources) and create a new Java class called `HolaRestResource`. We'll add a method named `hola()` that returns a string along with the IP address of where the service is running, as shown in Example 3-5.

Example 3-5. src/main/java/com/redhat/examples/dropwizard/resources/HolaRestResource.java

```java
public class HolaRestResource {

    public String hola() throws UnknownHostException {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                  .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
        }
        return "Hola Dropwizard de " + hostname;
    }

}
```

Add the HTTP Endpoints

If this is familiar to what we did with Spring Boot, it's for a reason. We want to be able to create REST endpoints and services with POJO code where possible, and a Hello World application is the perfect place to do that. To expose this as a REST service, we're going to leverage good old JAX-RS annotations (see Example 3-6):

@Path
> Tell JAX-RS what the context path for this service should be.

@GET
> Add a GET HTTP service.

Example 3-6. src/main/java/com/redhat/examples/dropwizard/
HolaRestResource.java

```java
@Path("/api")
public class HolaRestResource {

    @Path("/hola")
    @GET
    public String hola() throws UnknownHostException {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                  .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
        }
        return "Hola Dropwizard de " + hostname;
    }
}
```

Now, in our `HolaDropwizardApplication` class, let's give the `run()` method an implementation to add our new REST resource to our microservice (Example 3-7).

Example 3-7. src/main/java/com/redhat/examples/dropwizard/
HolaDropwizardApplication.java

```java
@Override
public void run(HolaDropwizardConfiguration configuration,
                Environment environment) {
    environment.jersey().register(new HolaRestResource());
}
```

Now we should be able to build and run our Dropwizard microservice:

```
$ mvn clean package exec:java
```

When we go to the endpoint at *http://localhost:8080/api/hola* we should see the following:

Externalize Configuration

Dropwizard has many options for configuring the built-in components (like the servlet engine or data sources for databases) as well as creating entirely new commands that you can run and configure with a configurations file. We can also inject environment variables and system properties for those types of configurations that expect to change based on the environment in which they're running. Just like with Spring Boot, we can bind entire classes of properties to specific objects. In this example, let's bind all of the `helloapp.*` properties to our `HolaRestResource` class. With Spring Boot we had the options to write our configs in property files with key-value tuples. We could have also used YAML. With Dropwizard, we only have the YAML option.

So let's create a file in the root of our project called *conf/application.yml* (note, you'll need to create the *conf* directory if it doesn't exist). We put our configuration files in the *conf* folder to help us organize our project's different configuration files (the naming of the directory is not significant (i.e., it does not have any conventional meaning to Dropwizard). Let's add some configuration to our *conf/application.yml* file:

```
# configurations for our sayingFactory
helloapp:

    saying: Hola Dropwizard de
```

In this case, we're setting the property to a specific value. What if we wanted to be able to override it based on some environment conditions? We could override it by passing in a Java system variable like this `-Ddw.helloapp.saying=Guten Tag`. Note that the `dw.*` part of the system property name is significant; it tells Dropwizard to apply the value to one of the configuration settings of our application. What if we wanted to override the property based on the value of an OS environment variable? That would look like Example 3-8.

Example 3-8. conf/application.yml

```
# configurations for our sayingFactory
helloapp:

  saying:${HELLOAPP_SAYING:-Guten Tag aus}
```

The pattern for the value of the property is to first look at an environment variable if it exists. If the environment variable is unset, then use the default value provided. We also need to tell our application specifically that we want to use environment-variable substitution. In the `HolaDropwizardApplication` class, edit the `initialize()` method to look like Example 3-9.

Example 3-9. src/main/java/com/redhat/examples/dropwizard/
HolaDropwizardApplication.java

```java
@Override
public void initialize(
            Bootstrap<HolaDropwizardConfiguration> bootstrap) {

    // Enable variable substitution with environment variables
    bootstrap.setConfigurationSourceProvider(
            new SubstitutingSourceProvider(
                    bootstrap.getConfigurationSourceProvider(),
                    new EnvironmentVariableSubstitutor(false)
            )
    );
}
```

Now we've set up our configuration, let's build the backing object for it. We purposefully created a subconfiguration named `helloapp` which allows us to namespace our configs to organize them. We could have left it as a top-level configuration item, but since we didn't, let's see how we bind our *application.yml* file to our Dropwizard configuration objects.

Let's create a new class called `HelloSayingFactory` in *src/main/ java/com/redhat/examples/dropwizard/resources* directory. Fill it out like this:

```java
public class HelloSayingFactory {

    @NotEmpty
    private String saying;

    @JsonProperty
    public String getSaying() {
        return saying;
    }

    @JsonProperty
    public void setSaying(String saying) {
        this.saying = saying;
```

```
        }
    }
```

This is a simple Java Bean with some validation (`@NotEmpty`, from the hibernate validators library) and Jackson (`@JsonProperty`) annotations. This class wraps whatever configurations are under our `helloapp` configuration in our YAML file; at the moment, we only have "saying." When we first created our application, a `HolaDropwi zardConfiguration` class was created. Let's open that class and add our new `HelloSayingFactory`, as shown in Example 3-10.

Example 3-10. src/main/java/com/redhat/examples/dropwizard/ HolaDropwizardConfiguration.java

```java
public class HolaDropwizardConfiguration
        extends Configuration {


    private HelloSayingFactory sayingFactory;

    @JsonProperty("helloapp")
    public HelloSayingFactory getSayingFactory() {
        return sayingFactory;
    }

    @JsonProperty("helloapp")
    public void setSayingFactory(
                HelloSayingFactory sayingFactory) {
        this.sayingFactory = sayingFactory;
    }
}
```

Lastly, we need to inject the configuration into our `HolaRestRe source` (Example 3-11).

Example 3-11. src/main/java/com/redhat/examples/dropwizard/ resources/HolaRestResource.java

```java
@Path("/api")
public class HolaRestResource {

    private String saying;
    public HolaRestResource(final String saying) {
        this.saying = saying;
    }

    @Path("/hola")
    @GET
```

```java
    public String hola() throws UnknownHostException {
        String hostname = null;
        try {
            hostname = InetAddress.getLocalHost()
                                  .getHostAddress();
        } catch (UnknownHostException e) {
            hostname = "unknown";
        }
        return saying + " " + hostname;
    }
}
```

Since there is no magic dependency injection framework that you're forced to use, you'll need to update our `HolaDropwizardApplica tion` to inject the configurations to our REST Resource (Example 3-12).

Example 3-12. src/main/java/com/redhat/examples/dropwizard/ HolaDropwizardApplication.java

```java
    @Override
    public void run(HolaDropwizardConfiguration configuration,
                    Environment environment) {

        environment.jersey().register(
            new HolaRestResource(configuration
                .getSayingFactory()
                .getSaying()));
    }
```

Now we should have a fairly sophisticated configuration injection-capability available to us. In this example, we purposefully made it slightly more complicated to cover what you'd probably do in a real-world use case. Also note, although the mechanics of hooking all of this up is more boilerplate, there's a very clear pattern here: bind our Java objects to a YAML configuration file and keep everything very simple and intuitive without leveraging complex frameworks.

Let's run our application. To do this, let's update our *pom.xml* to pass our new *conf/application.yml* file, as shown in Example 3-13.

Example 3-13. pom.xml

```xml
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <configuration>
```

```xml
        <mainClass>
        com.redhat.examples.dropwizard.HolaDropwizardApplication
        </mainClass>
        <arguments>
            <argument>server</argument>
            <argument>conf/application.yml</argument>
        </arguments>
    </configuration>
</plugin>
```

From the command line, we can now run:

```
$ mvn clean package exec:java
```

When you navigate to *http://localhost:8080/api/hola*, we should see one of the sayings:

Guten Tag aus 10.203.146.114

If we stop the microservice, and export an environment variable, we should see a new saying:

```
$ export HELLOAPP_SAYING='Hello Dropwizard from'
$ mvn clean package exec:java
```


Hello Dropwizard from 10.0.1.20

Expose Application Metrics and Information

The great thing about Dropwizard is metrics are first-class citizens, not an afterthought, and are already there! Metrics are enabled by default on the admin port (`8081`, by default) but how does Dropwizard know anything specific about our application? Let's sprinkle a couple declarative annotations on our `HolaRestResource` (Example 3-14).

Example 3-14. src/main/java/com/redhat/examples/dropwizard/ resources/HolaRestResource.java

```java
@Path("/hola")
@GET
```

```
@Timed
public String hola() throws UnknownHostException {
    String hostname = null;
    try {
        hostname = InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException e) {
        hostname = "unknown";
    }
    return saying + " " + hostname;
}
```

We've added the `@Timed` annotation which tracks how long invocations to our service take. Other annotations for gathering metrics include:

`@Metered`

> The rate at which the service is called

`@ExceptionMetered`

> The rate at which exceptions are thrown

Build and restart your microservice and try hitting your service at *http://localhost:8080/api/hola* a couple times. Then if you navigate to *http://localhost:8081/metrics?pretty=true* and scroll toward the bottom (may be different for yours), you should see the metrics for our service:

```
com.redhat.examples.dropwizard.resources.HolaRestResource.hola{
    count: 3,
    max: 0.0060773830000000004,
    mean: 0.002282724632345539,
    min: 0.000085167,
    p50: 0.0007421190000000001,
    p75: 0.0060773830000000004,
    p95: 0.0060773830000000004,
    p98: 0.0060773830000000004,
    p99: 0.0060773830000000004,
    p999: 0.0060773830000000004,
    stddev: 0.002676717391830948,
    m15_rate: 0,
    m1_rate: 0,
    m5_rate: 0,
    mean_rate: 0.12945390398989548,
    duration_units: "seconds",
    rate_units: "calls/second"
}
```

How to Run This Outside of Maven?

Dropwizard packages our microservice as a single executable uber JAR. All we have to do is build our application and run it like this:

```
$ mvn clean package
$ java -jar target/hola-dropwizard-1.0.jar \
server conf/application.yml
```

Calling Another Service

In a microservice environment, each service is responsible for providing the functionality or service to other collaborators. If we wish to extend the "hello world" microservice, we will need to create a service to which we can call using Dropwizard's REST client functionality. Just like we did for the Spring Boot microservice, we'll leverage the backend service from the source code that accompanies the book. The interaction will look similar to this:

If you look in this book's source code, we'll see a Maven module called `backend` which contains a very simple HTTP servlet that can be invoked with a GET request and query parameters. The code for this backend is very simple, and does not use any of the microservice frameworks (Spring Boot, Dropwizard, or WildFly Swarm).

To start up the backend service on port `8080`, navigate to the `backend` directory and run the following:

```
$ mvn clean install jetty:run
```

This service is exposed at */api/backend* and takes a query parameter `greeting.` For example, when we call this service with this path */api/backend?greeting=Hello,* then the backend service will respond with a JSON object like this:

```
$ curl -X GET http://localhost:8080/api/backend?greeting=Hello
```

Before we get started, let's add the `dropwizard-client` dependency to our *pom.xml*:

```
<dependency>
    <groupId>io.dropwizard</groupId>
```

```
        <artifactId>dropwizard-client</artifactId>
    </dependency>
```

We will create a new HTTP endpoint, */api/greeting* in our Dropwizard `hola-dropwizard` example and use Jersey to call this backend! Let's start off by creating two new classes in the *src/main/java/com/redhat/examples/dropwizard/resources* folder of our hola-dropwizard project named `GreeterRestResource` and `Greeter SayingFactory`. The `GreeterRestResource` will implement our JAX-RS REST endpoints, and the `GreeterSayingFactory` class will encapsulate the configuration options we wish to use for our Greeter service.

The `GreeterRestResource` class is shown in Example 3-15.

Example 3-15. src/main/java/com/redhat/examples/dropwizard/resources/GreeterRestResource.java

```java
@Path("/api")
public class GreeterRestResource {

    private String saying;
    private String backendServiceHost;
    private int backendServicePort;
    private Client client;

    public GreeterRestResource(final String saying, String host,
            int port, Client client) {

        this.saying = saying;
        this.backendServiceHost = host;
        this.backendServicePort = port;
        this.client = client;
    }

    @Path("/greeting")
    @GET
    @Timed
    public String greeting() {
        String backendServiceUrl =
            String.format("http://%s:%d",
                backendServiceHost, backendServicePort);

        System.out.println("Sending to: " + backendServiceUrl);


        return backendServiceUrl;
    }
}
```

The `GreeterSayingFactory` class encapsulates configuration options we'll need, like the service host and service port, which could change in different environments (see Example 3-16). It also declares a `JerseyClientConfiguration`, which provides us a nice DSL for building up our Jersey client. Let's add this as a section to our `HolaDropwizardConfiguration` class, as shown in Example 3-17.

Example 3-16. src/main/java/com/redhat/examples/dropwizard/resources/GreeterSayingFactory.java

```java
public class GreeterSayingFactory {

    @NotEmpty
    private String saying;

    @NotEmpty
    private String host;

    @NotEmpty
    private int port;

    private JerseyClientConfiguration jerseyClientConfig =
        new JerseyClientConfiguration();


    @JsonProperty("jerseyClient")
    public JerseyClientConfiguration getJerseyClientConfig() {
        return jerseyClientConfig;
    }

    public String getSaying() {
        return saying;
    }

    public void setSaying(String saying) {
        this.saying = saying;
    }

    public String getHost() {
        return host;
    }

    public void setHost(String host) {
        this.host = host;
    }

    public int getPort() {
        return port;
```

```
    }

    public void setPort(int port) {
        this.port = port;
    }
}
```

*Example 3-17. src/main/java/com/redhat/examples/dropwizard/
HolaDropwizardConfiguration.java*

```java
public class HolaDropwizardConfiguration extends Configuration {


    private HelloSayingFactory sayingFactory;
    private GreeterSayingFactory greeterSayingFactory;

    @JsonProperty("helloapp")
    public HelloSayingFactory getSayingFactory() {
        return sayingFactory;
    }

    @JsonProperty("helloapp")
    public void setSayingFactory(
                HelloSayingFactory sayingFactory) {
        this.sayingFactory = sayingFactory;
    }

    @JsonProperty("greeter")
    public GreeterSayingFactory getGreeterSayingFactory() {
        return greeterSayingFactory;
    }

    @JsonProperty("greeter")
    public void setGreeterSayingFactory(
            GreeterSayingFactory greeterSayingFactory) {

        this.greeterSayingFactory = greeterSayingFactory;
    }
}
```

We can also update our *conf/application.yml* file to add these values
in:

```yaml
greeter:
  saying:${GREETER_SAYING:-Guten Tag Dropwizard}
  host:${GREETER_BACKEND_HOST:-localhost}
  port:${GREETER_BACKEND_PORT:-8080}
```

Note we specify defaults for the various configuration options if
they're not provided by OS environment variables.

Now let's wire up the client and the greeter resource inside our `Hola DropwizardApplication` class, which is where all of our services and resources are wired into the `Environment` object (Example 3-18).

Example 3-18. src/main/java/com/redhat/examples/dropwizard/ HolaDropwizardApplication.java

```java
@Override
public void run(HolaDropwizardConfiguration configuration,
                Environment environment) {

    environment.jersey().register(
        new HolaRestResource(configuration
            .getSayingFactory()
            .getSaying()));


    // greeter service
    GreeterSayingFactory greeterSayingFactory =
        configuration.getGreeterSayingFactory();

    Client greeterClient =
            new JerseyClientBuilder(environment)
            .using(
            greeterSayingFactory.getJerseyClientConfig())
            .build("greeterClient");

    environment.jersey().register(
        new GreeterRestResource(
            greeterSayingFactory.getSaying(),
            greeterSayingFactory.getHost(),
            greeterSayingFactory.getPort(), greeterClient));
}
```

Lastly, let's implement the client that will make the call to the backend service with all of the appropriate host and port information injected, see Example 3-19.

Example 3-19. src/main/java/com/redhat/examples/dropwizard/ resources/GreeterRestResource.java

```java
@Path("/greeting")
@GET
@Timed
public String greeting() {
    String backendServiceUrl = String.format(
    "http://%s:%d", backendServiceHost,
```

```
        backendServicePort);

    System.out.println("Sending to: " + backendServiceUrl);


    BackendDTO backendDTO = client
            .target(backendServiceUrl)
            .path("api")
            .path("backend")
            .queryParam("greeting", saying)
            .request().accept("application/json")
            .get(BackendDTO.class);

    return backendDTO.getGreeting() +
        " at host: " + backendDTO.getIp();
}
```

Dropwizard offers two convenience wrappers for making REST calls: the HttpComponents library directly (if you need low-level HTTP access) or the *Jersey/JAX-RS* REST client libraries, which have a nice higher-level DSL for making HTTP calls. In the previous example, we're using the Jersey client to make the call.

Now let's build the microservice and verify that we can call this new greeting endpoint and that it properly calls the backend. First, let's start the backend if it's not already running. Navigate to the `backend` directory of the source code that comes with this application and run it:

```
$ mvn clean install jetty:run
```

Next let's build and run our Dropwizard microservice. Let's also configure this service to run on a different port than its default port (`8080`) so that it doesn't collide with the backend service which is already running on port `8080`. Later on in the book we can see how running these microservices in their own container removes the restriction of port swizzling at runtime.

First let's make explicit which port we want to use as default. Open the *conf/application.yml* file and add the server port information:

```
server:
  applicationConnectors:
    - type: http
      port: 8080
```

Now start the microservice by overriding the default port:

```
$ mvn clean install exec:java \
    -Ddw.server.applicationConnectors[0].port=9090
```

Now, let's navigate our browser to *http://localhost:9090/api/greeting* to see if our microservice properly calls the backend and displays what we're expecting:

Guten Tag Dropwizard from cluster Backend at host: 10.203.146.114

Where to Look Next

In this chapter we learned about Dropwizard, saw some differences and similarities with Spring Boot and how to expose REST end-points, configuration, and metrics, and make calls to external services. This was meant as a quick introduction to Dropwizard and is by no means a comprehensive guide. Check out the following links for more information:

- Dropwizard Core
- Dropwizard Getting Started
- Client API
- Dropwizard on GitHub
- Dropwizard examples on GitHub

WildFly Swarm for Microservices

The last Java microservice framework we'll look at is a relative new-comer to the scene yet leverages tried-and-trusted Java EE function-ality found in the JBoss WildFly application server. WildFly Swarm is a complete teardown of the WildFly application server into bite-sized, reusable components called fractions that can be assembled and formed into a microservice application that leverages Java EE APIs. Assembling these fractions is as simple as including a depend-ency in your Java Maven (or Gradle) build file, and WildFly Swarm takes care of the rest.

Application servers and Java EE have been the workhorse of enter-prise Java applications for more than 15 years. WildFly (formerly JBoss Application Server) emerged as an enterprise-capable, open source application server. Many enterprises heavily invested in the Java EE technology (whether open source or proprietary vendors) from how they hire software talent as well as overall training, tool-ing, and management. Java EE has always been very capable at help-ing developers build tiered applications by offering functionality like servlets/JSPs, transactions, component models, messaging, and per-sistence. Deployments of Java EE applications were packaged as EARs, which typically contained many WARs, JARs, and associated configuration. Once you had your Java archive file (*EAR/WAR*), you would need to find a server, verify it was configured the way you expect, and then install your archive. You could even take advantage of dynamic deployment and redeployment (although doing this in production is not recommended, it can be useful in development). This meant your archives could be fairly lean and only include the

business code you needed. Unfortunately, this lead to bloated implementations of Java EE servers that had to account for any functionality that an application *might* need. It also led to over-optimization in terms of which dependencies to share (just put everything in the app server!) and which dependencies needed isolation because they would change at a different rate from other applications.

MyApp				
JAX-RS	EJB3	Transactions	CORBA	Batch
WildFly				
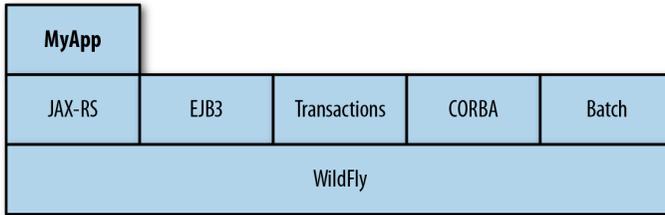
The application server provided a single point of surface area for managing, deploying, and configuring multiple applications within a single instance of the app server. Typically you'd cluster these for high availability by creating exact instances of the app server on different nodes. The problems start to arise when too many applications share a single deployment model, a single process, and a single JVM. The impedance arises when multiple teams who develop the applications running inside the app server have different types of applications, velocities of change, performance or SLA needs, and so on. Insofar as microservices architecture enables rapid change, innovation, and autonomy, Java EE application servers and managing a collection of applications as a single, all-in-one server don't enable rapid change. Additionally, from the operations side of the house, it becomes very complex to accurately manage and monitor the services and applications running within a single application server. In theory a single JVM is easier to manage, since it's just one thing, but the applications within the JVM are all independent deployments and should be treated as such. We can feel this pain when we try to treat the individual applications and services within a single process as "one thing," which is why we have very expensive and complicated tooling to try and accomplish that introspection. One way teams get around some of these issues is by deploying a single application to an application server.

Even though the deployment and management of applications within a Java EE environment may not suit a microservices environ-

ment, the component models, APIs, and libraries that Java EE provides to application developers still provide a lot of value. We still want to be able to use persistence, transactions, security, dependency injection, etc., but we want an à la carte usage of those libraries where needed. So how do we leverage our knowledge of Java EE, the power it brings within the context of microservices? That's where WildFly Swarm fits in.

WildFly Swarm evaluates your *pom.xml* (or Gradle file) and determines what Java EE dependencies your microservice actually uses (e.g., CDI, messaging, and servlet) and then builds an uber JAR (just like Spring Boot and Dropwizard) that includes the minimal Java EE APIs and implementations necessary to run your service. This process is known as "just enough application server," which allows you to continue to use the Java EE APIs you know and love and to deploy them both in a microservices and traditional-application style. You can even just start using your existing WAR projects and WildFly Swarm can introspect them automatically and properly include the requisite Java EE APIs/fractions without having to explicitly specify them. This is a very powerful way to move your existing applications to a microservice-style deployment.

Getting Started

There are three ways to get started with WildFly Swarm. You can start out with a blank Java Maven or Gradle project and manually add dependencies and Maven plug-ins. Another option is to use the WildFly Swarm Generator web console to bootstrap your project (similar to Spring Initializr for Spring Boot). Lastly, you can use the JBoss Forge tool, which is a generic Java project creation and altering tool which makes it easy to add Java classes, dependencies, and entire classes of functionality (e.g., JPA and transactions) to a Java Maven project. We highly recommend JBoss Forge in general, and we will use it in the guide here. For completeness, we'll also include the minimal plug-ins and dependencies you might need for a vanilla Java project. JBoss Forge also has plug-ins for the three most popular Java IDEs (Eclipse, Netbeans, or IntelliJ).

Vanilla Java Project

If you have an existing Java project or you create one from scratch using a Maven archetype or your favorite IDE, then you can add the

following pieces to your *pom.xml* to get up and running with Wild-Fly Swarm. First we want to be able to create uber JARs that know what pieces of the Java EE API functionality should be included. To do this, we'll use the WildFly Swarm Maven plug-in. Let's add the WildFly Swarm Maven plug-in to our *pom.xml*:

```xml
<plugins>
  <plugin>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>wildfly-swarm-plugin</artifactId>
    <version>${version.wildfly.swarm}</version>
    <executions>
      <execution>
        <goals>
          <goal>package</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

We also want to include the WildFly Swarm BOM (bill of materials) as a dependency in our `<dependencyManagement>` section to help sort out the proper versions of all of the APIs and WildFly Swarm fractions that we may depend on:

```xml
<dependencyManagement>
    <dependencies>
        <!-- JBoss distributes a complete set of Java EE 7 APIs
            including a Bill of Materials (BOM). A BOM specifies
            the versions of a "stack" (or a collection) of
            artifacts. We use this here so that we always get
            the correct versions of artifacts. -->
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>bom</artifactId>
            <version>${version.wildfly.swarm}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Now you can add the fractions of the Java EE API you need (or leave the fractions out and let WildFly Swarm autodetect them; useful if migrating an existing WAR application)! Let's take a look at some of the convenience that JBoss Forge brings.

Using JBoss Forge

JBoss Forge is a set of IDE plug-ins and CLI for quickly creating and working on Java projects. It has plug-ins for Netbeans, Eclipse, and IntelliJ to help you create Java projects, add CDI beans, add JPA entities, add and configure servlets, etc. Let's look at a quick example. First verify you have JDK/Java 1.8 installed then install JBoss Forge.

Once you have Forge installed, you should be able to start up the CLI (all of these commands available in the IDE plug-in as well):

```
$ forge
```

Feel free to explore what commands are available by pressing Tab, which also gives auto-completion for any command. JBoss Forge is built on a modular, plug-in–based architecture which allows others to write plug-ins to take advantage of the built-in tooling for the CLI and your favorite IDE. Take a look at some of the addons contributed by the community, including AsciiDoctor, Twitter, Arquillian, and AssertJ. Let's also install the WildFly Swarm addon for JBoss Forge:

```
[temp]$ addon-install \
--coordinate org.jboss.forge.addon:wildfly-swarm,1.0.0.Beta2

***SUCCESS*** Addon org.jboss.forge.addon:wildfly-swarm,
1.0.0.Beta2 was installed successfully.
```

Let's try a `project-new` command to build a new Java EE project that will be built and packaged with WildFly Swarm. Follow the interactive command prompt with the following inputs:

```
[swarm]$ project-new
***INFO*** Required inputs not satisfied, interactive mode
* Project name:  hola-wildflyswarm
? Package [org.hola.wildflyswarm]: com.redhat.examples.wfswarm
? Version [1.0.0-SNAPSHOT]:  1.0
? Final name:  hola-wildflyswarm
? Project location [/Users/ceposta/temp/swarm]:

[0] (x) war
[1] ( ) jar
[2] ( ) parent
[3] ( ) forge-addon
[4] ( ) resource-jar
[5] ( ) ear
[6] ( ) from-archetype
[7] ( ) generic
```

```
Press <ENTER> to confirm, or <CTRL>+C to cancel.
* Project type: [0-7]

[0] (x) Maven

Press <ENTER> to confirm, or <CTRL>+C to cancel.
* Build system: [0]

[0] ( ) JAVA_EE_7
[1] ( ) JAVA_EE_6
[2] ( ) NONE

Press <ENTER> to confirm, or <CTRL>+C to cancel.
? Stack (The technology stack to be used in project): [0-2] 2
***SUCCESS*** Project named 'hola-wildflyswarm'
has been created.
```

So what we have right now is an empty Java project that doesn't do too much. That's OK, though; we're just getting started. Let's set it up for a JAX-RS application:

```
[hola-wildflyswarm]$ rest-setup --application-path=/
***SUCCESS*** JAX-RS has been installed.
```

Now, let's add in the WildFly Swarm configurations like the Maven plug-in and the BOM dependency management section:

```
[hola-wildflyswarm]$ wildfly-swarm-setup --context-path=/
***SUCCESS*** Wildfly Swarm is now set up! Enjoy!
```

That's it! Now let's build and try to run our new WildFly Swarm microservice:

```
[HelloResource.java]$ cd ~~
[hola-wildflyswarm]$ wildfly-swarm-run
```

You should see it successfully start, but it doesn't do anything or expose any REST services. But what did JBoss Forge create for us here? If you look at the directory structure, you should see something similar:

```
./pom.xml
./src
./src/main
./src/main/java
./src/main/java/com/redhat/examples/wfswarm/rest
    /RestApplication.java
./src/main/resources
./src/main/webapp
./src/test
```

```
./src/test/java
./src/test/resources
```

Pretty bare bones! If we look at the *pom.xml,* we see some relevant
Java EE APIs and the WildFly Swarm plug-in/BOM:

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>bom</artifactId>
      <version>${version.wildfly-swarm}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>com.redhat.spec</groupId>
      <artifactId>jboss-javaee-6.0</artifactId>
      <version>3.0.3.Final</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.redhat.spec.javax.servlet</groupId>
      <artifactId>jboss-servlet-api_3.0_spec</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.redhat.spec.javax.ws.rs</groupId>
      <artifactId>jboss-jaxrs-api_1.1_spec</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>hola-wildflyswarm</finalName>
    <plugins>
      <plugin>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>wildfly-swarm-plugin</artifactId>
        <version>${version.wildfly-swarm}</version>
        <executions>
          <execution>
            <goals>
              <goal>package</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
```

```
        <properties>
          <swarm.context.path>/</swarm.context.path>
        </properties>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Remember, however, WildFly Swarm will only package the pieces of the Java EE framework that you need to run your application. In this case, we've already set up JAX-RS APIs, so WildFly Swarm will automatically include the JAX-RS and servlet fractions of an application server and embed them in your application.

Let's see how we can add some more functionality.

Hello World

Just like with the other frameworks in the preceding chapters, we want to add some basic hello-world functionality and then incrementally add more functionality on top of it. Let's start by creating a `HolaResource` in our project. You can do this with your IDE, or however you'd like; but again we can leverage JBoss Forge to do any of the heavy lifting for us here.

Navigate to the directory where you have your project, and fire up `forge` if it's not already running:

```
$ forge
```

Add the HTTP Endpoints

Now let's create a new JAX-RS endpoint with the `rest-new-endpoint` command and the interactive wizard, filling in the prompts using the following example as guidance:

```
[hola-wildflyswarm]$ rest-new-endpoint
***INFO*** Required inputs not satisfied, interactive mode
? Package Name (The package name where type will be created) \
[com.redhat.examples.wfswarm.rest]:

* Type Name (The type name):  HolaResource

[0] (x) GET
[1] ( ) POST
[2] ( ) PUT
[3] ( ) DELETE
```

```
Press <ENTER> to confirm, or <CTRL>+C to cancel.
? Methods (REST methods to be defined): [0-3]
? Path (The root path of the endpoint):  api/hola
***SUCCESS***
REST com.redhat.examples.wfswarm.rest.HolaResource created
```

That's it! Forge has created the *./src/main/java/com/redhat/examples/ wfswarm/rest/HolaResource.java* JAX-RS resource for us, and it looks similar to this:

```java
package com.redhat.examples.wfswarm.rest;

import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

@Path("/api/hola")
public class HolaResource {

        @GET
        @Produces("text/plain")
        public Response doGet() {
                return Response.ok("method doGet invoked")
                                .build();
        }
}
```

Let's go to the root of the project, build it, and try to fire it up again:

```
[HelloResource.java]$ cd ~~
[hola-wildflyswarm]$ wildfly-swarm-run
```

And navigate in a web browser to *http://localhost:8080/api/hola* (if an endpoint is not correctly displayed at this endpoint, please go back and check the preceding steps):

What did we just do? We built a JAX-RS web application using native Java EE with the JBoss Forge tooling and then ran it as a microservice inside WildFly Swarm!

Externalize Configuration

At the time of this writing, WildFly Swarm does not have an opinionated way of doing configuration and folks can choose to use well-established configuration, frameworks like Apache Commons Configuration or Apache DeltaSpike Configuration. Feel free to pay attention to this JIRA thread for more. In this section, we'll take a look at quickly adding Apache DeltaSpike Configuration for our configuration needs.

Apache DeltaSpike is a collection of CDI extensions that can be used to simplify things like configuration, data access, and security. Take a look at the DeltaSpike documentation. We're going to use a CDI extension that lets us easily inject configuration properties that can be sourced from properties files, the command line, JNDI, and environment variables. To take advantage of CDI, JAX-RS, and DeltaSpike, let's add a dependency on the `jaxrs-cdi` WildFly Swarm Fraction for integrating CDI and JAX-RS:

```xml
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jaxrs-cdi</artifactId>
</dependency>
```

We'll also want to add a dependency on the DeltaSpike libraries:

```xml
<dependency>
  <groupId>org.apache.deltaspike.core</groupId>
  <artifactId>deltaspike-core-api</artifactId>
  <version>1.5.3</version>
</dependency>
<dependency>
  <groupId>org.apache.deltaspike.core</groupId>
  <artifactId>deltaspike-core-impl</artifactId>
  <version>1.5.3</version>
</dependency>
```

We can create a new file called *META-INF/apache-deltaspike.properties* as well to store our application-specific properties. In this example, we'll try to grab our environment-specific properties from OS environment variables like we've been doing with the other frameworks and then default to values that may not exist. Edit your `HolaResource` class to add the `@ConfigProperty` annotation:

```java
@Path("/api/hola")
public class HolaResource {
```

```
    @Inject
    @ConfigProperty(name = "WF_SWARM_SAYING",
        defaultValue = "Hola")
    private String saying;

    @GET
    @Produces("text/plain")
    public Response doGet() {
        return Response.ok(saying + " from WF Swarm").build();
    }
}
```

With this simple annotation, we're able to quickly inject our properties from either the *META-INF/apache-deltaspike.properties* file, from the command line, from environment variables, or from JNDI. We will default to "Hola" if there is no environmental variable set. Take a look at the Apache DeltaSpike documentation for more ways to group or customize the functionality.

Now we can run our service with either `java -jar target/hola-wildflyswarm-swarm.jar` or with `mvn clean install wildfly-swarm:run`. We should see the default response "Hola from WF Swarm" and if we set the environment variable `WF_SWARM_SAYING`, then we should be able to alter the saying:

```
$ mvn clean install wildfly-swarm:run
```



```
Hola from WF Swarm
```

```
$ export WF_SWARM_SAYING=Yo
$ mvn clean install wildfly-swarm:run
```



```
Yo from WF Swarm
```

Expose Application Metrics and Information

To expose useful information about our microserivce, all we need to do is add the `monitor` fraction to our *pom.xml*:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>monitor</artifactId>
</dependency>
```

This will enable the WildFly management and monitoring functionality. From the monitoring perspective, WildFly Swarm exposes some basic metrics:

- Information about the node on which WildFly Swarm is running at */node*
- JVM heap usage at */heap*
- JVM/process thread information at */threads*

We can also add our own health endpoints that can invoke some actions or query parts of our microservice to easily expose how our service is doing. You can leverage the built-in health checking probes of most modern clusters to call your microservice health endpoints to determine whether or not the microservice is healthy, and in some cases, just kill it and restart. See the WildFly Swarm documentation for more on adding health probes.

How to Run This Outside of Maven

We've seen a couple ways to run our WildFly Swarm microservice. For development, you'll probably run with the Maven plug-in like this:

```
$ mvn clean install wildfly-swarm:run
```

When you build the binary, you can run it like this:

```
$ mvn clean package
```

This will take our project, whether it's packaged as a JAR or a WAR (as specified by the `<packaging>` definition in your *pom.xml*) and turn it into an executable uber JAR. Then you can run it like this:

```
$ java -jar target/hola-wildfly-swarm.jar
```

Note, whatever your final build-artifact is named, the WildFly Swarm Maven plug-in will add the *-swarm.jar* extension to the name.

Calling Another Service

In a microservice environment, each service is responsible for providing the functionality or service to other collaborators. If we wish to extend the "hello world" microservice, we will need to create a service to which we can call using JAX-RS client functionality. Just like we did for the Spring Boot microservice, we'll leverage the backend service from the source code that accompanies the book. The interaction will look similar to this:

If you look in this book's source code, we'll see a Maven module called `backend` which contains a very simple HTTP servlet that can be invoked with a GET request and query parameters. The code for this backend is very simple, and does not use any of the microservice frameworks (Spring Boot, Dropwizard, or WildFly Swarm).

To start up the backend service on port `8080`, navigate to the *backend* directory and run the following:

```
$ mvn clean install jetty:run
```

This service is exposed at */api/backend* and takes a query parameter `greeting`. For example, when we call this service with this path */api/backend?greeting=Hello*, then the backend service will respond with a JSON object like this:

```
$ curl -X GET http://localhost:8080/api/backend?greeting=Hello
```

We get something like this:

```
{
    "greeting" : "Hello from cluster Backend",
    "time" : 1459189860895,
    "ip" : "172.20.10.3"
}
```

We will create a new HTTP endpoint, */api/greeting* in our WildFly Swarm `hola-wildflyswarm` example and use JAX-RS client to call this backend!

Create a new class in *src/main/java/com/redhat/examples/wfswarm/rest* called `GreeterResource`, and fill it in similar to what we did for the `HolaResource` like in Example 4-1.

Example 4-1. src/main/java/com/redhat/examples/wfswarm/rest/ GreeterResource.java

```java
@Path("/api")
public class GreeterResource {

    @Inject
    @ConfigProperty(name = "WF_SWARM_SAYING",
        defaultValue = "Hola")
    private String saying;

    @Inject
    @ConfigProperty(name = "GREETING_BACKEND_SERVICE_HOST",
        defaultValue = "localhost")
    private String backendServiceHost;

    @Inject
    @ConfigProperty(name = "GREETING_BACKEND_SERVICE_PORT",
        defaultValue = "8080")
    private int backendServicePort;

    @Path("/greeting")
    @GET
    public String greeting() {
        String backendServiceUrl = String.format(
            "http://%s:%d",
            backendServiceHost,backendServicePort);

        System.out.println("Sending to: " + backendServiceUrl);

        return backendServiceUrl;
    }
}
```

We've created a simple JAX-RS resource here that exposes an */api/ greeting* endpoint that just returns the value of the `backendServi ceUrl` field. Also note, we're injecting the backend host and port as environment variables that have default values if no environment variables are set. Again, we're just using DeltaSpike `@ConfigProp erty` to accomplish this.

Let's also add the `BackendDTO` class as shown in Example 4-2, which is used to encapsulate responses from the backend.

*Example 4-2. src/main/java/com/redhat/examples/wfswarm/rest/
BackendDTO.java*

```java
public class BackendDTO {

    private String greeting;
    private long time;
    private String ip;

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    public long getTime() {
        return time;
    }

    public void setTime(long time) {
        this.time = time;
    }

    public String getIp() {
        return ip;
    }

    public void setIp(String ip) {
        this.ip = ip;
    }
}
```

Next, let's add our JAX-RS client implementation to communicate
with the backend service. It should look like Example 4-3.

*Example 4-3. src/main/java/com/redhat/examples/wfswarm/rest/
GreeterResource.java*

```java
@Path("/api")
public class GreeterResource {

    @Inject
    @ConfigProperty(name = "WF_SWARM_SAYING",
        defaultValue = "Hola")
    private String saying;

    @Inject
    @ConfigProperty(name = "GREETING_BACKEND_SERVICE_HOST",
```

```
        defaultValue = "localhost")
    private String backendServiceHost;

    @Inject
    @ConfigProperty(name = "GREETING_BACKEND_SERVICE_PORT",
        defaultValue = "8080")
    private int backendServicePort;

    @Path("/greeting")
    @GET
    public String greeting() {
        String backendServiceUrl = String.format("http://%s:%d",
            backendServiceHost,backendServicePort);

        System.out.println("Sending to: " + backendServiceUrl);

        Client client = ClientBuilder.newClient();
        BackendDTO backendDTO = client.target(backendServiceUrl)
                .path("api")
                .path("backend")
                .queryParam("greeting", saying)
                .request(MediaType.APPLICATION_JSON_TYPE)
                .get(BackendDTO.class);

        return backendDTO.getGreeting()
            + " at host: " + backendDTO.getIp();
    }
}
```

Now we can build our microservice either using Maven at the command line; or if you're still in JBoss Forge, you can run the build command:

```
$ mvn clean install
```

or:

```
[hola-wildflyswarm]$ build
```

When we start up our WildFly Swarm microservice, we will need to specify a new HTTP port (since the backend service is already running on port 8080), or we can just specify a port offset. If we specify a port offset, WildFly Swarm will try to deploy under its default port of 8080; but if that port is already in use, then it will increment the port by the swarm.port.offset amount and try again. If we use an offset of 1, and there is a collision on port 8080, then port 8081 will be what WildFly Swarm tries next. Let's run our microservice with a port offset:

```
$ mvn clean install wildfly-swarm:run -Dswarm.port.offset=1
```

Now, let's navigate our browser to *http://localhost:8081/api/greeting* to see if our microservice properly calls the backend and displays what we're expecting:

Yo from cluster Backend at host: 10.0.1.18

Where to Look Next

In this chapter, we learned about WildFly Swarm and saw some differences and similarities with Dropwizard and Spring Boot. We also learned how to expose REST endpoints, configuration, and metrics and make calls to external services. This was meant as a quick introduction to WildFly Swarm and is by no means a comprehensive guide. Check out the following links for more information:

- WildFly Swarm
- WildFly Swarm documentation
- WildFly Swarm examples on GitHub
- WildFly Swarm Core examples on GitHub
- WildFly Swarm blog
- *http://wildfly-swarm.io/community/*

Deploy Microservices at Scale with Docker and Kubernetes

Up to now, we've talked about microservices at a higher level, covering organizational agility, designing with dependency thinking, domain-driven design, and promise theory. Then we took a deep dive into the weeds with three popular Java frameworks for developing microservices: Spring Boot, Dropwizard, and WildFly Swarm. We can leverage powerful out-of-the-box capabilities easily by exposing and consuming REST endpoints, utilizing environment configuration options, packaging as all-in-one executable JAR files, and exposing metrics. These concepts all revolve around a single instance of a microservice. But what happens when you need to manage dependencies, get consistent startup or shutdown, do health checks, and load balance your microservices at scale? In this chapter, we're going to discuss those high-level concepts to understand more about the challenges of deploying microservices, regardless of language, at scale.

When we start to break out applications and services into microservices, we end up with more moving pieces by definition: we have more services, more binaries, more configuration, more interaction points, etc. We've traditionally dealt with deploying Java applications by building binary artifacts (JARs, WARs, and EARs), staging them somewhere (shared disks, JIRAs, and artifact repositories), opening a ticket, and hoping the operations team deploys them into an application server as we intended, with the correct permissions and environment variables and configurations. We also deploy our

application servers in clusters with redundant hardware, load balancers, and shared disks and try to keep things from failing as much as possible. We may have built some automation around the infrastructure that supports this with great tools like Chef or Ansible, but somehow deploying applications still tends to be fraught with mistakes, configuration drift, and unexpected behaviors.

With this model, we do a lot of hoping, which tends to break down quickly in current environments, nevermind at scale. Is the application server configured in Dev/QA/Prod like it is on our machine? If it's not, have we completely captured the changes that need to be made and expressed to the operations folks? Do any of our changes impact other applications also running in the same application server(s)? Are the runtime components like the operating system, JVM, and associated dependencies exactly the same as on our development machine? The JVM on which you run your application is very much a highly coupled implementation detail of our application in terms of how we configure, tune, and run, so these variations across environments can wreak havoc. When you start to deliver microservices, do you run them in separate processes on traditional servers? Is process isolation enough? What happens if one JVM goes berserk and takes over 100% of the CPU? Or the network IO? Or a shared disk? What if all of the services running on that host crash? Are your applications designed to accommodate that? As we split our applications into smaller pieces, these issues become magnified.

Immutable Delivery

Immutable delivery concepts help us reason about these problems. With immutable delivery, we try to reduce the number of moving pieces into prebaked images as part of the build process. For example, imagine in your build process you could output a fully baked image with the operating system, the intended version of the JVM, any sidecar applications, and all configuration? You could then deploy this in one environment, test it, and migrate it along a delivery pipeline toward production without worrying about "whether the environment or application is configured consistently." If you needed to make a change to your application, you rerun this pipeline, which produces a new immutable image of your application, and then do a rolling upgrade to deliver it. If it doesn't work, you can rollback by deploying the previous image. No more worrying

about configuration or environment drift or whether things were properly restored on a rollback.

This sounds great, but how do we do this? Executable JARs is one facet to get us part of the way there, but still falls short. The JVM is an implementation detail of our microservice, so how do we bundle the JVM? JVMs are written in native code and have native OS-level dependencies that we'll need to also package. We will also need configuration, environment variables, permissions, file directories, and other things that must be packaged. All of these details cannot be captured within a single executable JAR. Other binary formats like virtual machine (VM) images can properly encapsulate these details. However, for each microservice that may have different packaging requirements (JVM? NodeJS? Golang? properties files? private keys?), we could easily see an explosion of VM images and combinations of language runtimes. If you automate this with infrastructure as code and have access to infrastructure as a service with properly exposed APIs, you can certainly accomplish this. In fact, building up VMs as part of an automated delivery pipeline is exactly what Netflix did to achieve this level of immutable delivery. But VMs become hard to manage, patch, and change. Each VM virtualizes an entire machine with required device drivers, operating systems, and management tooling.

What other lightweight packaging and image formats can we explore?

Docker, Docker, Docker

Docker came along a few years ago with an elegant solution to immutable delivery. Docker allows us to package our applications with all of the dependencies it needs (OS, JVM, other application dependencies, etc.) in a lightweight, layered, image format. Additionally, Docker uses these images to run instances which run our applications inside Linux containers with isolated CPU, memory, network, and disk usage. In a way, these containers are a form of *application virtualization* or *process virtualization*. They allow a process to execute thinking it's the only thing running (e.g., list processes with `ps` and you see only your application's process there), and that it has full access to the CPUs, memory, disk, network, and other resources, when in reality, it doesn't. It can only use resources it's allocated. For example, I can start a Docker container with a slice

of CPU, a segment of memory, and limits on how much network IO can be used. From outside the Linux container, on the host, the application just looks like another process. No virtualization of device drivers, operating systems, or network stacks, and special hypervisors. It's just a process. This fact also means we can get even more applications running on a single set of hardware for higher density without the overhead of additional operating systems and other pieces of a VM which would be required to achieve similar isolation qualities.

What's happening under the covers is nothing revolutionary either. Features called `cgroups`, `namespaces`, and `chroot`, which have been built into the Linux kernel (and have for some time), are used to create the appearance of this application virtualization. Linux containers have been around for over 10 years, and process virtualization existed in Solaris and FreeBSD even before that. Traditionally, using these underlying Linux primitives, or even higher-level abstractions like `lxc`, have been complicated at best. Docker came along and simplified the API and user experience around Linux containers. Docker brought a single client CLI that can easily spin up these Linux containers based on the Docker image format, which has now been opened to the larger community in the Open Container Initiative (OCI). This ease of use and image format is changing the way we package and deliver software.

Once you have an image, spinning up many of these Linux containers becomes trivial. The layers are built as deltas between a base image (e.g., RHEL, Debian, or some other Linux operating system) and the application files. Distributing new applications just distributes the new layers on top of existing base layers. This makes distributing images much easier than shuttling around bloated cloud machine images. Also, if a vulnerability (e.g., shell shock, heartbleed, etc.) is found in the base image, the base images can be rebuilt without having to try patch each and every VM. This makes it easy to run a container anywhere: they can then be moved from a developer's desktop to dev, QA, or production in a portable way without having to manually hope that all of the correct dependencies are in the right place (does this application use JVM 1.6, 1.7, 1.8?). If we need to redeploy with a change (new app) or fix a base image, doing so just changes the layers in the image that require changes.

When we have standard APIs and open formats, we can build tooling that doesn't have to know or care what's running in the con-

tainer. How do we start an application? How do we stop? How do we do health checking? How do we aggregate logs, metrics, and insight? We can build or leverage tooling that does these things in a technology-agnostic way. Powerful clustering mechanics like service discovery, load balancing, fault tolerance, and configuration also can be pushed to lower layers in the application stack so that application developers don't have to try and hand cobble this together and complicate their application code.

Kubernetes

Google is known for running Linux containers at scale. In fact, "everything" running at Google runs in Linux containers and is managed by their Borg cluster management platform. Former Google engineer Joe Beda said the company starts over two billion containers per week. Google even had a hand in creating the underlying Linux technology that makes containers possible. In 2006 they started working on "process containers," which eventually became `cgroups` and was merged into the Linux kernel code base and released in 2008. With its breadth and background of operating containers at scale, it's not a surprise Google has had such an influence on platforms built around containers. For example, some popular container management projects that preceded Kubernetes were influenced by Google:

- The original Cloud Foundry creators (Derek Collison and Vadim Spivak) worked at Google and spent several years using Google's Borg cluster management solution.
- Apache Mesos was created for a PhD thesis, and its creator (Ben Hindman) interned at Google and had many conversations with Google engineers around container clustering, scheduling, and management.
- Kubernetes, an open source container cluster management platform and community, was originally created by the same engineers who built Borg at Google.

Back in 2013 when Docker rocked the technology industry, Google decided it was time to open source their next-generation successor to Borg, which they named Kubernetes. Today, Kubernetes is a large, open, and rapidly growing community with contributions from Google, Red Hat, CoreOS, and many others (including lots of inde-

pendent individuals!). Kubernetes brings a lot of functionality for running clusters of microservices inside Linux containers at scale. Google has packaged over a decade of experience into Kubernetes, so being able to leverage this knowledge and functionality for our own microservices deployments is game changing. The web-scale companies have been doing this for years, and a lot of them (Netflix, Amazon, etc.) had to hand build a lot of the primitives that Kubernetes now has baked in. Kubernetes has a handful of simple primitives that you should understand before we dig into examples. In this chapter, we'll introduce you to these concepts; and in the following chapter, we'll make use of them for managing a cluster of microservices.

Pods

A pod is a grouping of one or more Docker containers (like a pod of whales?). A typical deployment of a pod, however, will often be one-to-one with a Docker container. If you have sidecar, ambassador, or adapter deployments that must always be co-located with the application, a pod is the way to group them. This abstraction is also a way to guarantee container affinity (i.e., Docker container A will always be deployed alongside Docker container B on the same host).

Kubernetes orchestrates, schedules, and manages pods. When we refer to an application running inside of Kubernetes, it's running within a Docker container inside of a pod. A pod is given its own IP address, and all containers within the pod share this IP (which is different from plain Docker, where each container gets an IP address). When volumes are mounted to the pod, they are also shared between the individual Docker containers running in the pod.

One last thing to know about pods: they are fungible. This means they can disappear at any time (either because the service crashed or the cluster killed it). They are not like a VM, which you care for and nurture. Pods can be destroyed at any point. This falls within our expectation in a microservice world that things will (and do) fail, so we are strongly encouraged to write our microservices with this premise in mind. This is an important distinction as we talk about some of the other concepts in the following sections.

Labels

Labels are simple key/value pairs that we can assign to pods like `release=stable` or `tier=backend`. Pods (and other resources, but we'll focus on pods) can have multiple labels that group and categorize in a loosely coupled fashion, which becomes quite apparent the more you use Kubernetes. It's not a surprise that Google has identified such simple constructs from which we can build powerful clusters at scale. After we've labeled our pods, we can use label selectors to find which pods belong in which group. For example, if we had some pods labeled `tier=backend` and others labeled `tier=fron tend`, using a label selector expression of `tier != frontend`. We can select all of the pods that are not labeled "frontend." Label selectors are used under the covers for the next two concepts: replication controllers and services.

Replication Controllers

When talking about running microservices at scale, we will probably be talking about multiple instances of any given microservice. Kubernetes has a concept called `ReplicationController` that manages the number of `replicas` for a given set of microservices. For example, let's say we wanted to manage the number of pods labeled with `tier=backend` and `release=stable`. We could create a replication controller with the appropriate label selector and then be able to control the number of those pods in the cluster with the value of `replica` on our `ReplicationController`. If we set the `replica` count equal to 10, then Kubernetes will reconcile its current state to reflect 10 pods running for a given `ReplicationController`. If there are only five running at the moment, Kubernetes will spin up five more. If there are 20 running, Kubernetes will kill 10 (which 10 it kills is nondeterministic, as far as your app is concerned). Kubernetes will do whatever it needs to converge with the desired state of 10 replicas. You can imagine controlling the size of a cluster very easily with a `ReplicationController`. We will see examples of `Rep licationController` in action in the next chapter.

Services

The last Kubernetes concept we should understand is the Kubernetes Service. ReplicationControllers can control the number of replicas of a service we have. We also saw that pods can be killed (either

crash on their own, or be killed, maybe as part of a ReplicationController scale-down event). Therefore, when we try to communicate with a group of pods, we should not rely on their direct IP addresses (each pod will have its own IP address) as pods can come and go. What we need is a way to group pods to discover where they are, how to communicate with them, and possibly load balance against them. That's exactly what the Kubernetes Service does. It allows us to use a label selector to group our pods and abstract them with a single virtual (cluster) IP that we can then use to discover them and interact with them. We'll show concrete examples in the next chapter.

With these simple concepts, pods, labels, `ReplicationControllers`, and services, we can manage and scale our microservices the way Google has learned to (or learned not to). It takes many years and many failures to identify simple solutions to complex problems, so we highly encourage you to learn these concepts and experience the power of managing containers with Kubernetes for your microservices.

Getting Started with Kubernetes

Docker and Kubernetes are both Linux-native technologies; therefore, they must run in a Linux host operating system. We assume most Java developers work with either a Windows or Mac developer machine, so in order for us to take advantage of the great features Docker and Kubernetes bring, we'll need to use a guest Linux VM on our host operating system. You could download Docker machine and toolbox for your environment but then you'd need to go about manually installing Kubernetes (which can be a little tricky). You could use the upstream Kubernetes vagrant images, but like any fast-moving, open source project, those can change swiftly and be unstable at times. Additionally, to take full advantage of Docker's portability, it's best to use at least the same kernel of Linux between environments but optimally the same Linux distribution and version. What other options do we have?

Microservices and Linux Containers

To get started developing microservices with Docker and Kubernetes, we're going to leverage a set of developer tools called the Red Hat Container Development Kit (CDK). The CDK is free and is a

small, self-contained VM that runs on a developer's machine that contains Docker, Kubernetes, and a web console (actually, it's Red Hat OpenShift, which is basically an enterprise-ready version of Kubernetes with other developer self-service and application lifecycle management features; but for this book we'll just be using the Kubernetes APIs).

OpenShift?

Red Hat OpenShift 3.x is an Apache v2, licensed, open source developer self-service platform OpenShift Origin that has been revamped to use Docker and Kubernetes. OpenShift at one point had its own cluster management and orchestration engine, but with the knowledge, simplicity, and power that Kubernetes brings to the world of container cluster management, it would have been silly to try and re-create yet another one. The broader community is converging around Kubernetes and Red Hat is all in with Kubernetes.

OpenShift has many features, but one of the most important is that it's still native Kubernetes under the covers and supports role-based access control, out-of-the-box software-defined networking, security, logins, developer builds, and many other things. We mention it here because the flavor of Kubernetes that we'll use for the rest of this book is based on OpenShift. We'll also use the oc OpenShift command-line tools, which give us a better user experience and allow us to easily log in to our Kubernetes cluster and control which project into which we're deploying. The CDK that we mentioned has both vanilla Kubernetes and OpenShift. For the rest of this book, we'll be referring to OpenShift and Kubernetes interchangeably but using OpenShift.

Getting Started with the CDK

With the CDK, you can build, deploy, and run your microservices as Docker containers right on your laptop and then opt to deliver your microservice in a delivery pipeline through other application lifecycle management features inside of OpenShift or with your own tooling. The best part of the CDK is that it runs in a RHEL VM, which should match your development, QA, and production environments.

Instructions for installing the CDK can be found at *http://red.ht/ 1SVUp6g*. There are multiple flavors of virtualization (e.g., Virtual-

Box, VMWare, KVM) that you can use with the CDK. The installation documents for the CDK contain all of the details you need for getting up and running. To continue with the examples and idioms in the rest of this book, please install the CDK (or any other Docker/Kubernetes local VM) following the instructions.

To start up the CDK, navigate to the installation directory and to the *./components/rhel/rhel-ose* subdirectory and type the following:

```
$ vagrant up
```

This should take you through the provisioning process and boot the VM. The VM will expose a Docker daemon at *tcp://10.1.2.2:2376* and the Kubernetes API at *https://10.1.2.2:8443*. We will next need to install the OpenShift `oc` command-line tools for your environment. This will allow us to log in to OpenShift/Kubernetes and manage our projects/namespaces. You could use the `kubectl` commands yourself, but logging in is easier with the `oc login` command, so for these examples, we'll use `oc`. Download and install the oc client tools.

Once you've downloaded and installed both the CDK and the `oc` command-line tools, the last thing we want to do is set a couple of environment variables so our tooling will be able to find the Open-Shift installation and Docker daemon. To do this, navigate to the *./components/rhel/rhel-ose* directory and run the following command:

```
$ eval "$(vagrant service-manager env docker)"
```

This will set up your environment variables. You can output the environment variables and manually configure them if you wish:

```
$ vagrant service-manager env docker
export DOCKER_HOST=tcp://192.168.121.195:2376
export DOCKER_CERT_PATH=/home/john/cdk/components/rhel/rhel-ose/
.vagrant/machines/default/libvirt/.docker
export DOCKER_TLS_VERIFY=1
export DOCKER_MACHINE_NAME=081d3cd
```

We should now be able to log in to the OpenShift running in the CDK:

```
$ oc login 10.1.2.2:8443
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to
the server could be intercepted by others.
Use insecure connections? (y/n): y

Authentication required for https://10.1.2.2:8443 (openshift)
```

```
Username: admin
Password:
Login successful.

You have access to the following projects and can switch between
them with 'oc project <projectname>':

  * default

Using project "default".
Welcome! See 'oc help' to get started.
```

Let's create a new project/namespace into which we'll deploy our microservices:

```
$ oc new-project microservice-book
Now using project "microservice-book" on server
"https://10.1.2.2:8443".
```

You should be ready to go to the next steps!

Although not required to run these examples, installing the Docker CLI for your native developer laptop is useful as well. This will allow you to list Docker images and Docker containers right from your developer laptop as opposed to having to log in to the vagrant VM. Once you have the Docker CLI installed, you should be able to run Docker directly from your command-line shell (note, the environment variables previously discussed should be set up):

```
$ docker ps
$ docker images
```

Where to Look Next

In this chapter, we learned a little about the pains of deploying and managing microservices at scale and how Linux containers can help. We can leverage immutable delivery to reduce configuration drift, enable repeatable deployments, and help us reason about our applications regardless of whether they're running. We can use Linux containers to enable service isolation, rapid delivery, and portability. We can leverage scalable container management systems like Kubernetes and take advantage of a lot of distributed-system features like service discovery, failover, health-checking (and more!) that are built in. You don't need complicated port swizzling or complex service discovery systems when deploying on Kubernetes because these are problems that have been solved within the infrastructure itself. To learn more, please review the following links:

- Kubernetes Reference Documentation
- "An Introduction to Immutable Infrastructure" by Josha Stella
- "The Decline of Java Application Servers when Using Docker Containers" by James Strachan
- Docker docs
- OpenShift Enterprise 3.1 Documentation
- Kubernetes
- Kubernetes Reference Documentation: Pods
- Kubernetes Reference Documentation: Labels and Selectors
- Kubernetes Reference Documentation: Replication Controller
- Kubernetes Reference Documentation: Services

Hands-on Cluster Management, Failover, and Load Balancing

In Chapter 5, we just had a quick introduction to cluster management, Linux containers, and cluster management. Let's jump into using these things to solve issues with running microservices at scale. For reference, we'll be using the microservice projects we developed in Chapters 2, 3, and 4 (Spring Boot, Dropwizard, and WildFly Swarm, respectively). The following steps can be accomplished with any of the three Java frameworks.

Getting Started

To package our microservice as a Docker image and eventually deploy it to Kubernetes, let's navigate to our project (Spring Boot example in this case) and return to JBoss Forge. JBoss Forge has some plug-ins for making it easy to quickly add the Maven plug-ins we need to use:

```
$ cd hola-springboot
$ forge
```

Now let's install a JBoss Forge addon:

```
hola-springboot]$ addon-install \
--coordinate io.fabric8.forge:devops,2.2.148

***SUCCESS*** Addon io.fabric8.forge:devops,2.2.148 was
installed successfully.
```

Now let's add the Maven plug-ins:

```
[hola-springboot]$ fabric8-setup
***SUCCESS*** Added Fabric8 Maven support with base Docker
image: fabric8/java-jboss-openjdk8-jdk:1.0.10. Added the
following Maven profiles [f8-build, f8-deploy,
f8-local-deploy] to make building the project easier,
e.g., mvn -Pf8-local-deploy
```

Let's take a look at what the tooling did. If we open the *pom.xml* file,
we see it added some properties:

```xml
<docker.assemblyDescriptorRef>
    artifact
</docker.assemblyDescriptorRef>
<docker.from>
    docker.io/fabric8/java-jboss-openjdk8-jdk:1.0.10
</docker.from>
<docker.image>
    fabric8/${project.artifactId}:${project.version}
</docker.image>
<docker.port.container.http>8080</docker.port.container.http>
<docker.port.container.jolokia>
    8778
</docker.port.container.jolokia>
<fabric8.iconRef>icons/spring-boot</fabric8.iconRef>
<fabric8.service.containerPort>
    8080
</fabric8.service.containerPort>
<fabric8.service.name>hola-springboot</fabric8.service.name>
<fabric8.service.port>80</fabric8.service.port>
<fabric8.service.type>LoadBalancer</fabric8.service.type>
```

It also added two Maven plug-ins: docker-maven-plugin and
fabric8-maven-plugin:

```xml
<plugin>
 <groupId>io.fabric8</groupId>
 <artifactId>docker-maven-plugin</artifactId>
 <version>0.14.2</version>
 <configuration>
   <images>
     <image>
       <name>${docker.image}</name>
       <build>
         <from>${docker.from}</from>
         <assembly>
           <basedir>/app</basedir>
           <descriptorRef>
             ${docker.assemblyDescriptorRef}
           </descriptorRef >
         </assembly>
         <env>
           <JAR>
```

```
                ${project.artifactId}-${project.version}.war
              </JAR>
              <JAVA_OPTIONS>
                -Djava.security.egd=/dev/./urandom<
              /JAVA_OPTIONS>
            </env>
          </build>
        </image>
      </images>
    </configuration>
  </plugin>
      <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>2.2.100</version>
    <executions>
      <execution>
        <id>json</id>
        <phase>generate-resources</phase>
        <goals>
          <goal>json</goal>
        </goals>
      </execution>
      <execution>
        <id>attach</id>
        <phase>package</phase>
        <goals>
          <goal>attach</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

Lastly, the tooling added some convenience Maven profiles:

f8-build
> Build the docker image and Kubernetes manifest YML.

f8-deploy
> Build the docker image and deploy to a remote docker registry; then deploy the application to Kubernetes.

f8-local-deploy
> Build the docker image, generate the Kubernetes *manifest.yml*, and deploy to a locally running Kubernetes.

The JBoss Forge addon is part of the Fabric8 open source project. Fabric8 builds developer tooling for interacting with Docker, Kubernetes, and OpenShift, including Maven plug-ins, variable injection libraries for Spring/CDI, and clients for accessing the Kubernetes/

OpenShift API. Fabric8 also builds API management, CI/CD, chaos monkey and Kubernetes-based NetflixOSS functionality on top of Kubernetes.

Packaging Our Microservice as a Docker Image

With the Maven plug-ins added from the previous step, all we have to do to build the docker image is run the following Maven command. This step, and all others related to building Docker images or deploying to Kubernetes, assume the CDK (earlier in this chapter) is up and running:

```
$ mvn -Pf8-build

[INFO] DOCKER> ... d3f157b39583 Pull complete
        ============ 10% =========== 20% ============
        30% ============ 40% =========== 50% =============
        60% =========== 70% =========== 80% ============
        90% =========== 100% =
[INFO] DOCKER> ... f5a6e0d26670 Pull complete
        = 100% ==
[INFO] DOCKER> ... 6d1f91fc8ac8 Pull complete
        = 100% ==
[INFO] DOCKER> ... 77c58da5314d Pull complete
        = 100% ==
[INFO] DOCKER> ... 1416b43aef4d Pull complete
        = 100% ==
[INFO] DOCKER> ... fcc736051e6e Pull complete
[INFO] DOCKER> ... Digest: sha256:e77380a4924bb599162e3382e6443e
8aa50c0
[INFO] DOCKER> ... Downloaded image for java-jboss-openjdk8-jdk:
1.0.10
[INFO] DOCKER> [fabric8/hola-springboot:1.0] : Built image 13e72
5c3c771
[INFO]
[INFO] fabric8-maven-plugin:2.2.100:json (default-cli) @ hola-
springboot
[INFO] Configured with file: /Users/ceposta/dev/sandbox/micro
services-by-example/source/spring-boot/hola-springboot/target
/classes/kubernetes.json
[INFO] Generated env mappings: {}
[INFO] Generated port mappings: {http=ContainerPort(container
Port=8080, hostIP=null, hostPort=null, name=http, protocol=
null, additionalProperties={}), jolokia=ContainerPort(
containerPort=8778, hostIP=null, hostPort=null, name=jolokia,
protocol=null, additionalProperties={})}
[INFO] Removed 'version' label from service selector for
service ``
[INFO] Generated ports: [ServicePort(name=null, nodePort=null,
```

```
      port=80, protocol=TCP, targetPort=IntOrString(IntVal=8080,
      Kind=null, StrVal=null, additionalProperties={}),
      additionalProperties={})]
[INFO] Icon URL: img/icons/spring-boot.svg
[INFO] Looking at repo with directory /microservices-by-example
/.git
[INFO] Added environment annotations:
[INFO]     Service hola-springboot selector: {project=hola-
springboot,
[INFO]          provider=fabric8, group=com.redhat.examples}
ports: 80
[INFO]     ReplicationController hola-springboot replicas: 1,
[INFO]          image: fabric8/hola-springboot:1.0
[INFO] Template is now:
[INFO]     Service hola-springboot selector: {project=hola-
springboot,
[INFO]          provider=fabric8, group=com.redhat.examples}
ports: 80
[INFO]     ReplicationController hola-springboot replicas: 1,
[INFO]          image: fabric8/hola-springboot:1.0
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
[INFO] Total time: 04:22 min
[INFO] Finished at: 2016-03-31T15:59:58-07:00
[INFO] Final Memory: 47M/560M
[INFO] ------------------------------------------------------
```

Deploying to Kubernetes

If we have the Docker tooling installed, we should see that our microservice has been packaged in a Docker container:

```
$ docker images
langswif01(cdk-v2 (master))$ docker images
REPOSITORY              TAG  IMAGE ID      CREATED   SIZE
fabric8/hola-springboot 1.0  13e725c3c771  3d ago    439.7 MB
```

We could start up the Docker container using `docker run`, but we want to deploy this into a cluster and leave the management of the microservice to Kubernetes. Let's deploy it with the following Maven command:

```
$ mvn -Pf8-local-deploy
```

If your environment is configured correctly (i.e., you've started the CDK, installed the `oc` tooling, logged in with the `oc login`, and created a new project with `oc new-project microservices-book`), you should see a successful build similar to this:

```
[INFO] --- fabric8-maven-plugin:apply (default-cli) @ hola-
springboot ---
[INFO] Using https://10.1.2.2:8443/ in namespace microservice-
book
[INFO] Kubernetes JSON: /Users/ceposta/dev/sandbox
[INFO]      /microservices-by-example/source/spring-boot/hola-
springboot
[INFO]      /target/classes/kubernetes.json
[INFO] OpenShift platform detected
[INFO] Using namespace: microservice-book
[INFO] Creating a Template from kubernetes.json namespace
[INFO]      microservice-book name hola-springboot
[INFO] Created Template: target/fabric8/applyJson/microservice-
book/
[INFO]      template-hola-springboot.json
[INFO] Looking at repo with directory /Users/ceposta/dev/
sandbox/
[INFO]      microservices-by-example/.git
[INFO] Creating a Service from kubernetes.json namespace
[INFO]      microservice-book name hola-springboot
[INFO] Created Service: target/fabric8/applyJson/microservice-
book
[INFO]      /service-hola-springboot.json
[INFO] Creating a ReplicationController from kubernetes.json
namespace
[INFO]      microservice-book name hola-springboot
[INFO] Created ReplicationController: target/fabric8/applyJson
[INFO]      /microservice-book/replicationcontroller-hola-
springboot.json
[INFO] Creating Route microservice-book:hola-springboot host:
[INFO] -------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------
[INFO] Total time: 19.101 s
[INFO] Finished at: 2016-04-04T09:05:02-07:00
[INFO] Final Memory: 52M/726M
[INFO] -------------------------------------------------------
```

Let's take a quick look at what the fabric8-maven-plugin plug-in did for us.

First, Kubernetes exposes a REST API that allows us to manipulate the cluster (what's deployed, how many, etc.). Kubernetes follows a "reconciliation of end state" model where you describe what you want your deployment to look like and Kubernetes makes it happen. This is similar to how some configuration management systems work where you declaratively express what should be deployed and not how it should be accomplished. When we post data to the Kubernetes REST API, Kubernetes will reconcile what needs to hap-

pen inside the cluster. For example, if "we want a pod running `hola-springboot`" we can make an HTTP POST to the REST API with a JSON/YAML manifest file, and Kubernetes will create the pod, create the Docker containers running inside that pod, and schedule the pod to one of the hosts in the cluster. A Kubernetes pod is an atomic unit that can be scheduled within a Kubernetes cluster. It typically consists of a single Docker container, but it can contain many Docker containers. In our code samples, we will treat our hello-world deployments as a single Docker container per pod.

The `fabric8-maven-plugin` we used in the preceding code example will automatically generate the REST objects inside a JSON/YAML manifest file for us and POST this data to the Kubernetes API. After running the `mvn -Pf8-local-deploy` command successfully, we should be able to navigate to the webconsole (*https://10.1.2.2:8443*) or using the CLI tooling to see our new pod running our hello-springboot application:

```
$ oc get pod
NAME                    READY    STATUS    RESTARTS   AGE
hola-springboot-8xtdm   1/1      Running   0          3d
```

At this point we have a single pod running in our cluster. What advantage does Kubernetes bring as a cluster manager? Let's start by exploring the first of many. Let's kill the pod and see what happens:

```
$ oc delete pod/hola-springboot-8xtdm
pod "hola-springboot-8xtdm" deleted
```

Now let's list our pods again:

```
$ oc get pod
NAME                    READY    STATUS    RESTARTS   AGE
hola-springboot-42p89   0/1      Running   0          3d
```

Wow! It's still there! Or, more correctly, another pod has been created after we deleted the previous one. Kubernetes can start/stop/auto-restart your microservices for you. Can you imagine what a headache it would be to determine whether a service is started/stopped at any kind of scale? Let's continue exploring some of the other valuable cluster management features Kubernetes brings to the table for managing microservices.

Scaling

One of the advantages of deploying in a microservices architecture is independent scalability. We should be able to replicate the number

of services in our cluster easily without having to worry about port conflicts, JVM or dependency mismatches, or what else is running on the same machine. With Kubernetes, these types of scaling concerns can be accomplished with the ReplicationController. Let's see what replication controllers exist in our deployment:

```
 $ oc get replicationcontroller
CONTROLLER        CONTAINER(S)      IMAGE(S)
hola-springboot   hola-springboot   fabric8/hola-springboot:1.0

SELECTOR
group=com.redhat.examples,project=hola-springboot,
provider=fabric8,version=1.0

REPLICAS   AGE
1          3d
```

We can also abbreviate the command:

```
$ oc get rc
```

One of the objects that the `fabric8-maven-plugin` created for us is the `ReplicationController` with a `replica` value of 1. This means we want to always have one pod/instance of our microservice at all times. If a pod dies (or gets deleted), then Kubernetes is charged with reconciling the desired state for us, which is `replicas=1`. If the cluster is not in the desired state, Kubernetes will take action to make sure the desired configuration is satisfied. What happens if we want to change the desired number of replicas and scale up our service?

```
$ oc scale rc hola-springboot --replicas=3
replicationcontroller "hola-springboot" scaled
```

Now if we list the pods, we should see three pods of our hello-springboot application:

```
$$ oc get pod
NAME                    READY     STATUS    RESTARTS   AGE
hola-springboot-42p89   1/1       Running   0          3d
hola-springboot-9s6a6   1/1       Running   0          3d
hola-springboot-np2l1   1/1       Running   0          3d
```

Now if any of our pods dies or gets deleted, Kubernetes will do what it needs to do to make sure the replica count is 3. Notice, also, that we didn't have to change ports on these services or do any unnatural port remapping. Each one of the services is listening on port `8080` and does not collide with the others.

Let's go ahead and scale down to 0 to get ready for the next section; we can just run the same command:

```
$ oc scale rc hola-springboot --replicas=0
```

Kubernetes also has the ability to do autoscaling by watching metrics like CPU, memory usage, or user-defined triggers, to scale the number of replicas up or down. Autoscaling is outside the scope of this book but is a very valuable piece of the cluster-management puzzle.

Service discovery

One last concept in Kubernetes that we should understand is `Service`. In Kubernetes, a `Service` is a simple abstraction that provides a level of indirection between a group of pods and an application using the service represented by that group of pods. We've seen how pods are managed by Kubernetes and can come and go. We've also seen how Kubernetes can easily scale up the number of instances of a particular service. In our example, we're going to start our `backend` service from the previous chapters to play the role of service provider. How will our `hola-springboot` communicate with the `backend`?

Let's run the `backend` service by navigating to our source code to the folder `backend` and deploy it locally to our locally running Kubernetes cluster running in the CDK:

```
$ mvn -Pf8-local-deploy
```

Let's take a look at what Kubernetes services exist:

```
$ oc get service
NAME            CLUSTER_IP      EXTERNAL_IP   PORT(S)
backend         172.30.231.63                 80/TCP
hola-springboot 172.30.202.59                 80/TCP

SELECTOR                                       AGE
component=backend,provider=fabric8             3d
group=com.redhat.examples,project=hola-springboot,
provider=fabric8                               3d
```

Note the `Service` objects get automatically created by the `fabric8-maven-plugin` just like the `ReplicationController` objects in the previous section. There are two interesting attributes of a service that appear in the preceding code example. One is the `CLUSTER_IP.` This is a virtual IP that is assigned when a `Service` object is created

and never goes away. It's a single, fixed IP that is available to any applications running within the Kubernetes cluster and can be used to talk to `backend` pods. The pods are "selected" with the `SELECTOR` field. Pods in Kubernetes can be "labeled" with whatever metadata you want to apply (like "version" or "component" or "team") and can subsequently be used in the selector for a `Service`. In this example, we're selecting all the pods with label `component=backend` *and* `provider=fabric8`. This means any pods that are "selected" by the selector can be reached just by using the cluster IP. No need for complicated distributed registries (e.g., Zookeeper, Consul, or Eureka) or anything like that. It's all built right into Kubernetes. Cluster-level DNS is also built into Kubernetes. Using DNS in general for microservice service discovery can be very challenging and downright painful. In Kubernetes, the cluster DNS points to the cluster IP; and since the cluster IP is a fixed IP and doesn't go away, there are no issues with DNS caching and other gremlins that can pop up with traditional DNS.

Let's add a couple environment variables to our `hola-springboot` project to use our `backend` service when running inside a Kubernetes cluster:

```
<fabric8.env.GREETING_BACKENDSERVICEHOST>
    backend<
/fabric8.env.GREETING_BACKENDSERVICEHOST>
<fabric8.env.GREETING_BACKENDSERVICEPORT>
    80
</fabric8.env.GREETING_BACKENDSERVICEPORT>
```

Let's build the Kubernetes manifest and verify we're passing in these environment variables to our pod. Note that Spring Boot will resolve configuration from `application.properties` but can be overridden with system properties and environment variables at runtime:

```
$ mvn fabric8:json
```

Inspect file *target/classes/kubernetes.json*:

```
"containers" : [ {
  "args" : [ ],
  "command" : [ ],
  "env" : [ {
    "name" : "GREETING_BACKENDSERVICEHOST",
    "value" : "backend"
  }, {
    "name" : "GREETING_BACKENDSERVICEPORT",
    "value" : "80"
```

```
    }, {
      "name" : "KUBERNETES_NAMESPACE",
      "valueFrom" : {
        "fieldRef" : {
          "fieldPath" : "metadata.namespace"
        }
      }
    }
  } ],
```

Let's delete all of the pieces of the `hola-springboot` project and redeploy:

```
$ oc delete all -l project=hola-springboot
$ mvn -Pf8-local-deploy
```

We should now be able to list the pods and see our `hola-springboot` pod running as well as our backend service pod:

```
$ oc get pod
NAME                  READY     STATUS     RESTARTS     AGE
backend-nk224         1/1       Running    5            3d
hola-springboot-r5ykr 1/1       Running    0            2m
```

Now, just to illustrate a handy debugging technique, we're going to set up port-forwarding between our local machine and our `hola-springboot-r5ykr` pod and verify that our service is working correctly and we can call the backend. Let's set up portforward to port `9000` on our local machine:

```
$ oc port-forward -p hola-springboot-r5ykr 9000:8080
```

We should not be able to communicate with the pod over our localhost port `9000`. Note this technique works great even across a remote Kubernetes cluster, not just on our local CDK. So instead of having to try and find which host our pod is running and how to `ssh` into it, we can just use `oc port-forward` to expose it locally.

So now we should be able to navigate locally using our browser or a CLI command line:

```
$ curl http://localhost:9000/api/hola
Hola Spring Boot de 172.17.0.9
```

We can see the *api/hola* endpoint at `http://localhost:9000` using our port-forwarding! We also see that the `/api/hola` endpoint is returning the IP address of the pod in which it's running. Let's call the *api/greeting* API, which is supposed to call our backend:

```
$ curl http://localhost:9000/api/greeting
Hola Spring Boot from cluster Backend at host: 172.17.0.5
```

We can see that the `backend` pod returns its IP address in this call! So our service was discovered correctly, and all it took was a little bit of DNS and the power of Kubernetes service discovery. One big thing to notice about this approach is that we did not specify any extra client libraries or set up any registries or anything. We happen to be using Java in this case, but using Kubernetes cluster DNS provides a technology-agnostic way of doing basic service discovery!

Fault Tolerance

Complex distributed systems like a microservice architecture must be built with an important premise in mind: things will fail. We can spend a lot of energy preventing things from failing, but even then we won't be able to predict every case where and how dependencies in a microservice environment can fail. A corollary to our premise of "things will fail" is that "we design our services for failure." Another way of saying that is "figure out how to survive in an environment where there are failures."

Cluster Self-Healing

If a service begins to misbehave, how will we know about it? Ideally our cluster management solution can detect and alert us about failures and let human intervention take over. This is the approach we typically take in traditional environments. When running microservices at scale, where we have lots of services that are supposed to be identical, do we really want to stop and troubleshoot every possible thing that can go wrong with a single service? Long-running services may experience unhealthy states. An easier approach is to design our microservices such that they can be terminated at any moment, especially when they appear to be behaving incorrectly.

Kubernetes has a couple of health probes we can use out of the box to allow the cluster to administer and self-heal itself. The first is a *readiness* probe, which allows Kubernetes to determine whether or not a pod should be considered in any service discovery or load-balancing algorithms. For example, some Java apps may take a few seconds to bootstrap the containerized process, even though the pod is technically up and running.

If we start sending traffic to a pod in this state, users may experience failures or inconsistent states. With *readiness* probes, we can let Kubernetes query an HTTP endpoint (for example) and only con-

sider the pod ready if it gets an HTTP 200 or some other response. If Kubernetes determines a pod does not become ready within a specified period of time, the pod will be killed and restarted.

Another health probe we can use is a *liveness* probe. This is similar to the *readiness* probe; however, it's applicable after a pod has been determined to be "ready" and is eligible to receive traffic. Over the course of the life of a pod or service, if the *liveness* probe (which could also be a simple HTTP endpoint) starts to indicate an unhealthy state (e.g., HTTP 500 errors), Kubernetes can automatically kill the pod and restart it.

When we used the JBoss Forge tooling and the `fabric8-setup` command from the Fabric8 addon, a readiness probe was automatically added to our Kubernetes manifest by adding the following Maven properties to the respective *pom.xml*. If it wasn't, you can use the command `fabric8-readiness-probe` or `fabric8-liveness-probe` to add it to an existing project:

```
<fabric8.readinessProbe.httpGet.path>
    /health
</fabric8.readinessProbe.httpGet.path>
<fabric8.readinessProbe.httpGet.port>
    8080
</fabric8.readinessProbe.httpGet.port>
<fabric8.readinessProbe.initialDelaySeconds>
    5
</fabric8.readinessProbe.initialDelaySeconds>
<fabric8.readinessProbe.timeoutSeconds>
    30
</fabric8.readinessProbe.timeoutSeconds>
```

The Kubernetes JSON that gets generated for including these Maven properties includes:

```
"readinessProbe" : {
  "httpGet" : {
    "path" : "/health",
    "port" : 8080
  },
```

This means the "readiness" quality of the `hola-springboot` pod will be determined by periodically polling the */health* endpoint of our pod. When we added the actuator to our Spring Boot microservice earlier, a */health* endpoint was added which returns:

```
{
  "diskSpace": {
    "free": 106880393216,
```

```
    "status": "UP",
    "threshold": 10485760,
    "total": 107313364992
  },
  "status": "UP"
}
```

The same thing can be done with Dropwizard and WildFly Swarm!

Circuit Breaker

As a service provider, your responsibility is to your consumers to provide the functionality you've promised. Following promise theory, a service provider may depend on other services or downstream systems but cannot and should not impose requirements upon them. A service provider is wholly responsible for its promise to consumers. Because distributed systems can and do fail, there will be times when service promises can't be met or can be only partly met. In our previous examples, we showed our Hola apps reaching out to a backend service to form a greeting at the */api/greeting* endpoint. What happens if the backend service is not available? How do we hold up our end of the promise?

We need to be able to deal with these kinds of distributed systems faults. A service may not be available; a network may be experiencing intermittent connectivity; the backend service may be experiencing enough load to slow it down and introduce latency; a bug in the backend service may be causing application-level exceptions. If we don't deal with these situations explicitly, we run the risk of degrading our own service, holding up threads, database locks, and resources, and contributing to rolling, cascading failures that can take an entire distributed network down. To help us account for these failures, we're going to leverage a library from the NetflixOSS stack named Hystrix.

Hystrix is a fault-tolerant Java library that allows microservices to hold up their end of a promise by:

- Providing protection against dependencies that are unavailable
- Monitoring and providing timeouts to guard against unexpected dependency latency
- Load shedding and self-healing
- Degrading gracefully

- Monitoring failure states in real time
- Injecting business-logic and other stateful handling of faults

With Hystrix, you wrap any call to your external dependencies with a `HystrixCommand` and implement the possibly faulty calls inside the `run()` method. To help you get started, let's look at implementing a `HystrixCommand` for the `hola-wildflyswarm` project. Note for this example, we're going to follow the Netflix best practices of making everything explicit, even if that introduces some boilerplate code. Debugging distributed systems is difficult and having exact stack traces for your code without too much magic is more important than hiding everything behind complicated magic that becomes impossible to debug at runtime. Even though the Hystrix library has annotations for convenience, we'll stick with implementing the Java objects directly for this book and leave it to the reader to explore the more mystical ways to use Hystrix.

First let's add the `hystrix-core` dependency to our Maven *pom.xml*:

```xml
<dependency>
  <groupId>com.netflix.hystrix</groupId>
  <artifactId>hystrix-core</artifactId>
  <version>${hystrix.version}</version>
</dependency>
```

Let's create a new Java class called `BackendCommand` that extends from `HystrixCommand` in our `hola-wildflyswarm` project shown in Example 6-1.

Example 6-1. src/main/java/com/redhat/examples/wfswarm/rest/ BackendCommand

```java
public class BackendCommand extends HystrixCommand<BackendDTO> {

    private String host;
    private int port;
    private String saying;

    public BackendCommand(String host, int port) {
        super(HystrixCommandGroupKey.Factory
            .asKey("wfswarm.backend"));
        this.host = host;
        this.port = port;
    }

    public BackendCommand withSaying(String saying) {
        this.saying = saying;
```

```java
        return this;
    }

    @Override
    protected BackendDTO run() throws Exception {
        String backendServiceUrl =
            String.format("http://%s:%d", host, port);

        System.out.println("Sending to: " + backendServiceUrl);

        Client client = ClientBuilder.newClient();
        return client.target(backendServiceUrl)
                .path("api")
                .path("backend")
                .queryParam("greeting", saying)
                .request(MediaType.APPLICATION_JSON_TYPE)
                .get(BackendDTO.class);

    }

}
```

You can see here we've extended `HystrixCommand` and provided our `BackendDTO` class as the type of response our command object will return. We've also added some constructor and builder methods for configuring the command. Lastly, and most importantly, we've added a `run()` method here that actually implements the logic for making an external call to the `backend` service. Hystrix will add thread timeouts and fault behavior around this `run()` method.

What happens, though, if the `backend` service is not available or becomes latent? You can configure thread timeouts and rate of failures which would trigger circuit-breaker behavior. A circuit breaker in this case will simply open a circuit to the backend service by not allowing any calls to go through (failing fast) for a period of time. The idea with this circuit-breaker behavior is to allow any backend remote resources time to recover or heal without continuing to take load and possibly further cause it to persist or degrade into unhealthy states.

You can configure Hystrix by providing configuration keys, JVM system properties, or by using a type-safe DSL for your command object. For example, if we want to enable the circuit breaker (default true) and open the circuit if we get five or more failed requests (timeout, network error, etc.) within five seconds, we could pass the following into the constructor of our `BackendCommand` object:

```
public BackendCommand(String host, int port) {
    super(Setter.withGroupKey(
      HystrixCommandGroupKey.Factory
        .asKey("wildflyswarm.backend"))
        .andCommandPropertiesDefaults(
            HystrixCommandProperties.Setter()
            .withCircuitBreakerEnabled(true)
            .withCircuitBreakerRequestVolumeThreshold(5)
            .withMetricsRollingStatistical \
            WindowInMilliseconds(5000)
        ))
    ;
    this.host = host;
    this.port = port;
}
```

Please see the Hystrix documentation for more advanced configurations as well as for how to externalize the configurations or even configure them dynamically at runtime.

If a backend dependency becomes latent or unavailable and Hystrix intervenes with a circuit breaker, how does our service keep its promise? The answer to this may be very domain specific. For example, if we consider a team that is part of a personalization service, we want to display custom book recommendations for a user. We may end up calling the book-recommendation service, but what if it isn't available or is too slow to respond? We could degrade to a book list that may not be personalized; maybe we'd send back a book list that's generic for users in a particular region. Or maybe we'd not send back any personalized list and just a very generic "list of the day." To do this, we can use Hystrix's built-in fallback method. In our example, if the backend service is not available, let's add a fallback method to return a generic BackendDTO response:

```
public class BackendCommand extends HystrixCommand<BackendDTO> {

    <rest of class here>

    @Override
    protected BackendDTO getFallback() {
        BackendDTO rc = new BackendDTO();
        rc.setGreeting("Greeting fallback!");
        rc.setIp("127.0.0,1");
        rc.setTime(System.currentTimeMillis());
        return rc;
    }
```

```
    }
```

Our `/api/greeting-hystrix` service should not be able to service a client and hold up part of its promise, even if the backend service is not available.

Note this is a contrived example, but the idea is ubiquitous. However, the application of whether to fallback or gracefully degrade versus breaking a promise is very domain specific. For example, if you're trying to transfer money in a banking application and a backend service is down, you may wish to reject the transfer. Or you may wish to make only a certain part of the transfer available while the backend gets reconciled. Either way, there is no one-size-fits-all fallback method. In general, coming up with the fallback is related to what kind of customer experience gets exposed and how best to gracefully degrade considering the domain.

Bulkhead

Hystrix offers some powerful features out of the box, as we've seen. One more failure mode to consider is when services become latent but not latent enough to trigger a timeout or the circuit breaker. This is one of the worst situations to deal with in distributed systems as latency like this can quickly stall (or appear to stall) all worker threads and cascade the latency all the way back to users. We would like to be able to limit the effect of this latency to just the dependency that's causing the slowness without consuming every available resource. To accomplish this, we'll employ a technique called the bulkhead. A bulkhead is basically a separation of resources such that exhausting one set of resources does not impact others. You often see bulkheads in airplanes or trains dividing passenger classes or in boats used to stem the failure of a section of the boat (e.g., if there's a crack in the hull, allow it to fill up a specific partition but not the entire boat).

Hystrix implements this bulkhead pattern with thread pools. Each downstream dependency can be allocated a thread pool to which it's assigned to handle external communication. Netflix has benchmarked the overhead of these thread pools and has found for these types of use cases, the overhead of the context switching is minimal, but it's always worth benchmarking in your own environment if you have concerns. If a dependency downstream becomes latent, then

the thread pool assigned to that dependency can become fully utilized, but other requests to the dependency will be rejected. This has the effect of containing the resource consumption to just the degraded dependency instead of cascading across all of our resources.

If the thread pools are a concern, Hystrix also can implement the bulkhead on the calling thread with counting semaphores. Refer to the Hystrix documentation for more information.

The bulkhead is enabled by default with a thread pool of 10 worker threads with no BlockingQueue as a backup. This is usually a sufficient configuration, but if you must tweak it, refer to the configuration documentation of the Hystrix component. Configuration would look something like this (external configuration is possible as well):

```java
public BackendCommand(String host, int port) {
    super(Setter.withGroupKey(
      HystrixCommandGroupKey.Factory
        .asKey("wildflyswarm.backend"))
        .andThreadPoolPropertiesDefaults(
            HystrixThreadPoolProperties.Setter()
            .withCoreSize(10)
            .withMaxQueueSize(-1))
        .andCommandPropertiesDefaults(
            HystrixCommandProperties.Setter()
            .withCircuitBreakerEnabled(true)
            .withCircuitBreakerRequestVolumeThreshold(5)
            .withMetricsRollingStatisticalWindow \
            InMilliseconds(5000)
          ))
    ;
    this.host = host;
    this.port = port;
}
```

To test out this configuration, let's build and deploy the `hola-wildflyswarm` project and play around with the environment.

Build Docker image and deploy to Kubernetes:

```
$ mvn -Pf8-local-deploy
```

Let's verify the new */api/greeting-hystrix* endpoint is up and functioning correctly (this assumes you've been following along and still have the `backend` service deployed; refer to previous sections to get that up and running):

```
$ oc get pod
NAME                    READY   STATUS    RESTARTS   AGE
backend-pwawu           1/1     Running   0          18h
```

```
hola-dropwizard-bf5nn     1/1    Running    0         19h
hola-springboot-n87w3     1/1    Running    0         19h
hola-wildflyswarm-z73g3   1/1    Running    0         18h
```

Let's port-forward the `hola-wildflyswarm` pod again so we can reach it locally. Recall this is a great benefit of using Kubernetes that you can run this command regardless of where the pod is actually running in the cluster:

```
$ oc port-forward -p hola-wildflyswarm-z73g3 9000:8080
```

Now let's navigtate to *http://localhost:9000/api/greeting-hystrix*:
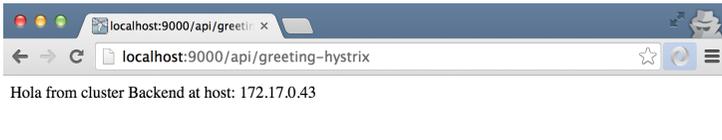
Now let's take down the `backend` service by scaling its `Replication Controller` replica count down to zero:

```
$ oc scale rc/backend --replicas=0
```

By doing this, there should be no `backend` pods running:

```
$ oc get pod
NAME                      READY  STATUS       RESTARTS  AGE
backend-pwawu             1/1    Terminating  0         18h
hola-dropwizard-bf5nn     1/1    Running      0         19h
hola-springboot-n87w3     1/1    Running      0         19h
hola-wildflyswarm-z73g3   1/1    Running      0         18h
```

Now if we refresh our browser pointed at *http://localhost:9000/api/ greeting-hystrix*, we should see the service degrade to using the Hystrix fallback method:

Load Balancing

In a highly scaled distributed system, we need a way to discover and load balance against services in the cluster. As we've seen in previous examples, our microservices must be able to handle failures; there-

fore, we have to be able to load balance against services that exist, services that may be joining or leaving the cluster, or services that exist in an autoscaling group. Rudimentary approaches to load balancing, like round-robin DNS, are not adequate. We may also need sticky sessions, autoscaling, or more complex load-balancing algorithms. Let's take a look at a few different ways of doing load balancing in a microservices environment.

Kubernetes Load Balancing

The great thing about Kubernetes is that it provides a lot of distributed-systems features out of the box; no need to add any extra components (server side) or libraries (client side). Kubernetes `Services` provided a means to discover microservices and they also provide server-side load balancing. If you recall, a Kubernetes `Service` is an abstraction over a group of pods that can be specified with label selectors. For all the pods that can be selected with the specified selector, Kubernetes will load balance any requests across them. The default Kubernetes load-balancing algorithm is round robin, but it can be configured for other algorithms such as session affinity. Note that clients don't have to do anything to add a pod to the `Service`; just adding a label to your pod will enable it for selection and be available. Clients reach the Kubernetes `Service` by using the cluster IP or cluster DNS provided out of the box by Kubernetes. Also recall the cluster DNS is not like traditional DNS and does not fall prey to the DNS caching TTL problems typically encountered with using DNS for discovery/load balancing. Also note, there are no hardware load balancers to configure or maintain; it's all just built in.

To demonstrate load balancing, let's scale up the `backend` services in our cluster:

```
$ oc scale rc/backend --replicas=3
```

Now if we check our pods, we should see three `backend` pods:

```
$ oc get pod
NAME                    READY   STATUS    RESTARTS   AGE
backend-8ywcl           1/1     Running   0          18h
backend-d9wm6           1/1     Running   0          18h
backend-vt61x           1/1     Running   0          18h
hola-dropwizard-bf5nn   1/1     Running   0          20h
hola-springboot-n87w3   1/1     Running   0          20h
hola-wildflyswarm-z73g3 1/1     Running   0          19h
```

If we list the Kubernetes services available, we should see the `back end` service as well as the selector used to select the pods that will be eligible for taking requests. The `Service` will load balance to these pods:

```
$ oc get svc
NAME                CLUSTER_IP         PORT(S)
backend             172.30.231.63      80/TCP
hola-dropwizard     172.30.124.61      80/TCP
hola-springboot     172.30.55.130      80/TCP
hola-wildflyswarm   172.30.198.148     80/TCP
```

We can see here that the `backend` service will select all pods with labels `component=backend` and `provider=fabric8`. Let's take a quick moment to see what labels are on one of the `backend` pods:

```
$ oc describe pod/backend-8ywcl | grep Labels
Labels:                     component=backend,provider=fabric8
```

We can see that the `backend` pods have the labels that match what the service is looking for; so any time we communicate with the service, we will be load balanced over these matching pods.

Let's make a call to our `hola-wildflyswarm` service. We should see the response contain different IP addresses for the `backend` service:

```
$ oc port-forward -p hola-wildflyswarm-z73g3 9000:8080

$ curl http://localhost:9000/api/greeting
Hola from cluster Backend at host: 172.17.0.45

$ curl http://localhost:9000/api/greeting
Hola from cluster Backend at host: 172.17.0.44

$ curl http://localhost:9000/api/greeting
Hola from cluster Backend at host: 172.17.0.46
```

Here we enabled port forwarding so that we can reach our `hola-wildflyswarm` service and tried to access the *http://localhost:9000/api/greeting* endpoint. I used `curl` here, but you can use your favorite HTTP/REST tool, including your web browser. Just refresh your web browser a few times to see that the backend, which gets called is different each time. The Kubernetes `Service` is load balancing over the respective pods as expected.

Do We Need Client-Side Load Balancing?

Client-side load balancers can be used inside Kubernetes if you need more fine-grained control or domain-specific algorithms for determining which service or pod you need to send to. You can even do things like weighted load balancing, skipping pods that seem to be faulty, or some custom-based Java logic to determine which service/pod to call. The downside to client-side load balancing is that it adds complexity to your application and is often language specific. In a majority of cases, you should prefer to use the technology-agnostic, built-in Kubernetes service load balancing. If you find you're in a minority case where more sophisticated load balancing is required, consider a client-side load balancer like SmartStack, `bakerstreet.io`, or NetflixOSS Ribbon.

In this example, we'll use NetflixOSS Ribbon to provide client-side load balancing. There are different ways to use Ribbon and a few options for registering and discovering clients. Service registries like Eureka and Consul may be good options in some cases, but when running within Kubernetes, we can just leverage the built-in Kubernetes API to discover services/pods. To enable this behavior, we'll use `ribbon-discovery` project from Kubeflix. Let's enable the dependencies in our *pom.xml* that we'll need:

```xml
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>ribbon</artifactId>
</dependency>
  <dependency>
    <groupId>io.fabric8.kubeflix</groupId>
    <artifactId>ribbon-discovery</artifactId>
    <version>${kubeflix.version}</version>
</dependency>
```

For Spring Boot we could opt to use Spring Cloud, which provides convenient Ribbon integration, or we could just use the NetflixOSS dependencies directly:

```xml
<dependency>
  <groupId>com.netflix.ribbon</groupId>
  <artifactId>ribbon-core</artifactId>
  <version>${ribbon.version}</version>
</dependency>
<dependency>
  <groupId>com.netflix.ribbon</groupId>
  <artifactId>ribbon-loadbalancer</artifactId>
```

```
      <version>${ribbon.version}</version>
    </dependency>
```

Once we've got the right dependencies, we can configure Ribbon to use Kubernetes discovery:

```
loadBalancer = LoadBalancerBuilder.newBuilder()
        .withDynamicServerList(
            new KubernetesServerList(config))
        .buildDynamicServerListLoadBalancer();
```

Then we can use the load balancer with the Ribbon LoadBalancer Command:

```
@Path("/greeting-ribbon")
@GET
public String greetingRibbon() {
    BackendDTO backendDTO = LoadBalancerCommand.
    <BackendDTO>builder()
    .withLoadBalancer(loadBalancer)
    .build()
    .submit(new ServerOperation<BackendDTO>() {
        @Override
        public Observable<BackendDTO> call(Server server) {
            String backendServiceUrl = String.format(
                "http://%s:%d",
                server.getHost(), server.getPort());

            System.out.println("Sending to: " +
                backendServiceUrl);

            Client client = ClientBuilder.newClient();
            return Observable.just(client
                    .target(backendServiceUrl)
                    .path("api")
                    .path("backend")
                    .queryParam("greeting", saying)
                    .request(MediaType.APPLICATION_JSON_TYPE)
                    .get(BackendDTO.class));
        }
    }).toBlocking().first();
    return backendDTO.getGreeting() + " at host: " +
        backendDTO.getIp();
}
```

See the accompanying source code for the exact details.

Where to Look Next

In this chapter, we learned a little about the pains of deploying and managing microservices at scale and how Linux containers can help. We can leverage true immutable delivery to reduce configuration drift, and we can use Linux containers to enable service isolation, rapid delivery, and portability. We can leverage scalable container management systems like Kubernetes and take advantage of a lot of distributed-system features like service discovery, failover, health-checking (and more!) that are built in. You don't need complicated port swizzling or complex service discovery systems when deploying on Kubernetes because these are problems that have been solved within the infrastructure itself. To learn more, please review the following links:

- "An Introduction to Immutable Infrastructure" by Josha Stella
- "The Decline of Java Applications When Using Docker Containers" by James Strachan
- Docker documentation
- OpenShift Enterprise 3.1 Documentation
- Kubernetes Reference Documentation: Horizontal Pod Autoscaling
- Kubernetes Reference Documentation: Services
- Fabric8 Kubeflix on GitHub
- Hystrix on GitHub
- Netflix Ribbon on GitHub
- Spring Cloud

Where Do We Go from Here?

We have covered a lot in this small book but certainly didn't cover everything! Keep in mind we are just scratching the surface here, and there are many more things to consider in a microservices environment than what we can cover in this book. In this last chapter, we'll very briefly talk about a couple of additional concepts you must consider. We'll leave it as an exercise for the reader to dig into more detail for each section!

Configuration

Configuration is a very important part of any distributed system and becomes even more difficult with microservices. We need to find a good balance between configuration and immutable delivery because we don't want to end up with snowflake services. For example, we'll need to be able to change logging, switch on features for A/B testing, configure database connections, or use secret keys or passwords. We saw in some of our examples how to configure our microservices using each of the three Java frameworks, but each framework does configuration slightly differently. What if we have microservices written in Python, Scala, Golang, NodeJS, etc?

To be able to manage configuration across technologies and within containers, we need to adopt an approach that works regardless of what's actually running in the container. In a Docker environment we can inject environment variables and allow our application to consume those environment variables. Kubernetes allows us to do that as well and is considered a good practice. Kubernetes also adds

APIs for mounting `Secrets` that allow us to safely decouple user-names, passwords, and private keys from our applications and inject them into the Linux container when needed. Kubernetes also recently added `ConfigMaps` which are very similar to `Secrets` in that application-level configuration can be managed and decoupled from the application Docker image but allow us to inject configuration via environment variables and/or files on the container's file system. If an application can consume configuration files from the filesystem (which we saw with all three Java frameworks) or read environment variables, it can leverage Kubernetes configuration functionality. Taking this approach, we don't have to set up additional configuration services and complex clients for consuming it. Configuration for our microservices running inside containers (or even outside), regardless of technology, is now baked into the cluster management infrastructure.

Logging, Metrics, and Tracing

Without a doubt, a lot of the drawbacks to implementing a microservices architecture revolve around management of the services in terms of logging, metrics, and tracing. The more you break a system into individual parts, the more tooling, forethought, and insight you need to invest to see the big picture. When you run services at scale, especially assuming a model where things fail, we need a way to grab information about services and correlate that with other data (like metrics and tracing) regardless of whether the containers are still alive. There are a handful of approaches to consider when devising your logging, metrics, and tracing strategy:

- Developers exposing their logs
- Aggregation/centralization
- Search and correlate
- Visualize and chart

Kubernetes has addons to enable cluster-wide logging and metrics collection for microservices. Typical technology for solving these issues include syslog, Fluentd, or Logstash for getting logs out of services and streamed to a centralized aggregator. Some folks use messaging solutions to provide some reliability for these logs if needed. ElasticSearch is an excellent choice for aggregating logs in a central, scalable, search index; and if you layer Kibana on top, you

can get nice dashboards and search UIs. Other tools like Prometheus, Zipkin, Grafana, Hawkular, Netflix Servo, and many others should be considered as well.

Continuous Delivery

Deploying microservices with immutable images discussed earlier in Chapter 5 is paramount. When we have many more smaller services than before, our existing manual processes will not scale. Moreover, with each team owning and operating its own microservices, we need a way for teams to make immutable delivery a reality without bottlenecks and human error. Once we release our microservices, we need to have insight and feedback about their usage to help drive further change. As the business requests change, and as we get more feedback loops into the system, we will be doing more releases more often. To make this a reality, we need a capable software-delivery pipeline. This pipeline may be composed of multiple subpipelines with gates and promotion steps, but ideally, we want to automate the build, test, and deploy mechanics as much as possible.

Tools like Docker and Kubernetes also give us the built-in capacity to do rolling upgrades, blue-green deployments, canary releases, and other deployment strategies. Obviously these tools are not required to deploy in this manner (places like Amazon and Netflix have done it for years without Linux containers), but the inception of containers does give us the isolation and immutability factors to make this easier. You can use your CI/CD tooling like Jenkins and Jenkins Pipeline in conjunction with Kubernetes and build out flexible yet powerful build and deployment pipelines. Take a look at the Fabric8 and OpenShift projects for more details on an implementation of CI/CD with Kubernetes based on Jenkins Pipeline.

Summary

This book was meant as a hands-on, step-by-step guide for getting started with some popular Java frameworks to build distributed systems following a microservices approach. Microservices is not a technology-only solution as we discussed in the opening chapter. People are the most important part of a complex system (a business) and to scale and stay agile, you must consider scaling the organization structure as well as the technology systems involved.

After building microservices with either of the Java frameworks we discussed, we need to build, deploy, and manage them. Doing this at scale using our current techniques and primitives is overly complex, costly, and does not scale. We can turn to new technology like Docker and Kubernetes that can help us build, deploy, and operate following best practices like immutable delivery.

When getting started with microservices built and deployed in Docker and managed by Kubernetes, it helps to have a local environment used for development purposes. For this we looked at the Red Hat Container Development Kit which is a small, local VM that has Red Hat OpenShift running inside a free edition of Red Hat Enterprise Linux (RHEL). OpenShift provides a production-ready Kubernetes distribution, and RHEL is a popular, secure, supported operating system for running production workloads. This allows us to develop applications using the same technologies that will be running in Production and take advantage of application packaging and portability provided by Linux containers.

Lastly we touched on a few additional important concepts to keep in mind like configuration, logging, metrics, and continuous, automated delivery. We didn't touch on security, self-service, and countless other topics; but make no mistake: they are very much a part of the microservices story.

We hope you've found this book useful. Please follow @openshift, @kubernetesio, @fabric8io, @christianposta, and @RedHatNews for more information, and take a look at the source code repository.

About the Author

Christian Posta (@christianposta) is a principal middleware specialist and architect at Red Hat. He's well known for being an author, blogger, speaker, and open source contributor. He is a committer on Apache ActiveMQ, Apache Camel, Fabric8, and others. Christian has spent time at web-scale companies and now helps enterprise companies creating and deploying large-scale distributed architectures, many of which are now microservice. He enjoys mentoring, training, and leading teams to success through distributed-systems concepts, microservices, DevOps, and cloud-native application design. When not working, he enjoys time with his wife, Jackie, and his two daughters, Madelyn and Claire.