O'REILLY

Cloud Native Java

DESIGNING RESILIENT SYSTEMS WITH SPRING BOOT, SPRING CLOUD, AND CLOUD FOUNDRY

Josh Long & Kenny Bastani

Cloud Native Java

First Edition

Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry

Josh Long & Kenny Bastani

Cloud Native Java

by Josh Long, Kenny Bastani

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safaribooksonline.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Editor: Brian Foster
- Developmental Editor: Nan Barber

- Month Year: First Edition

Revision History for the First Edition

- 2015-11-15: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781449370787 for release details.

978-1-4493-7464-8

Table of Contents

Chapter 1. The Cloud Native Application

The patterns for how we develop software, both in teams and as individuals, are rapidly evolving. Software teams are striving to continuously deliver software at a faster pace. New patterns for how we develop software are enabling us to think more about the behavior of our applications in production. Both developers and operators are placing more emphasis on understanding how their software applications will behave in production, with fewer assurances of how complexity will unravel in the event of a failure.

Software architectures are beginning to move away from large risky monolithic application deployments. Architectures are now focused on the art of preventing failures, increasing availability, and decentralizing change management in a push to move faster with less risk. We are starting to see development teams enter into the world of distributed systems development, focusing on building smaller more singularly focused services with independent release cycles.

As software applications turn into complex distributed systems, operational failures become an inevitable result. Companies are now utilizing cloud provided services to take advantage of horizontal scaling, and eliminating single points of failure in their architectures.

This chapter is going to focus on the patterns of building and operating cloud native applications. Throughout this chapter and book we will focus on three central concepts that are important to consider when building resilient distributed systems in the cloud. These three concepts are scalability, reliability, and agility.

We will explore these concepts in the context of building resilient systems, utilizing the lessons of Netflix, as it journeyed to become a cloud native company.

Scalability

To develop software faster, we are required to think about scale at all levels. Scale, in a most general sense, is a function of cost that produces value. The level of unpredictability that reduces that value is called risk. We are forced to frame scale in this context because building software is fraught with risks. The

risks that are being created by software developers are not always known to operators. By demanding that developers deliver features to production at a faster pace, we are adding to the risks of its operation without having a sense of empathy for its operators.

The result of this is that operators grow distrustful of the software that developers produce. The lack of trust between developers and operators creates a blame culture that tends to confuse the causes of failures that impact the creation of value for the business.

To alleviate the strain that is being put on traditional structures of an IT organization, we need to rethink how software delivery and operations teams communicate. Communication between operations and developers can affect our ability to scale, as the goals of each party tends to become misaligned over time. To succeed at this requires a shift towards a more reliable kind of software development, one that puts emphasis on the experience of an operations team inside the software development process.

Reliability

The expectations that are created between teams, be it operations, development, or user experience design, we can think of these expectations as contracts. The contracts that are created between teams imply some level of service is provided or consumed. By looking at how teams provide services to one another in the process of developing software, we can better understand how failures in communication can introduce risk that lead to failures down the road.

Service agreements between teams are created in order to reduce the risk of unexpected behavior in the overall functions of scale that produce value for a business. A service agreement between teams is made explicit in order to guarantee that behaviors are consistent with the expected cost of operations. In this way, services enable units of a business to maximize its total output. The goal here for a software business is to reliably predict the creation of value through cost. The result of this goal is what we call *reliability*.

The service model for a business is the same model that we use when we build software. This is how we guarantee the reliability of a system, whether it be in the software that we produce to automate a business function or in the people that we train to perform a manual operation.

Agility

We are beginning to find that there is no longer only one way to develop and operate software. Driven by the adoption of agile methodologies and a move towards *Software as a Service* business models, the enterprise application stack is becoming increasingly distributed. Developing distributed systems is a complex undertaking. The move towards a more distributed application architecture for companies is being fueled by the need to deliver software faster and with less risk of failure.

The modern day software-defined business is seeking to restructure their development processes to enable faster delivery of software projects and continuous deployment of applications into production. Not only are companies wanting to increase the rate in which they develop software applications, but they also want to increase the number of software applications that are created and operated to serve the various business units of an organization.

Software is increasingly becoming a competitive advantage for companies. Better and better tools are enabling business experts to open up new sources of revenue, or to optimize business functions in ways that lead to rapid innovation.

At the heart of this movement is *the cloud*. When we talk about the cloud, we are talking about a very specific set of technologies that enable developers and operators to take advantage of web services that exist to provision and manage virtualized computing infrastructure.

Companies are starting to move out of the data center and into public clouds. One such company is the popular subscription-based streaming media company Netflix.

Netflix's Cloud Native Journey

Today, Netflix is one of the world's largest on-demand streaming media services, operating their online services in the cloud. Netflix was founded in 1997 in Scotts Valley, California by Reed Hastings and Marc Randolph. Originally, Netflix provided an online DVD rental service that would allow customers to pay a flat-fee subscription each month for unlimited movie rentals without late fees. Customers would be shipped DVDs by mail after selecting from a list of movie titles and placing them into a queue using the Netflix website.

In 2008, Netflix had experienced a major database corruption that prevented the company from shipping any DVDs to its customers. At the time, Netflix was just starting to deploy its streaming video services to customers. The streaming team at Netflix realized that a similar kind of outage in streaming would be devastating to the future of its business. Netflix made a critical decision as a result of the database corruption, that they would move to a different way of developing and operating their software, one that ensured that their services would always be available to their customers.

As a part of Netflix's decision to prevent failures in their online services, they decided that they must move away from vertically scaled infrastructure and single points of failure. The realization stemmed from a result of the database corruption, which was a result of using a vertically scaled relational database. Netflix would eventually migrate their customer data to a distributed NoSQL database, an open source database project named Apache Cassandra. This was the beginning of the move to become a "cloud native" company, a decision to run all of their software applications as highly distributed and resilient services in the cloud. They settled on increasing the robustness of their online services by adding redundancy to their applications and databases in a scale out infrastructure model.

As a part of Netflix's decision to move to the cloud, they would need to migrate their large application deployments to highly reliable distributed systems. They faced a major challenge. The teams at Netflix would have to re-architect their applications while moving away from an on-premise data center to a public cloud. In 2009, Netflix would begin its move to Amazon Web Services (AWS), and they focused on three main goals: scalability, performance, and availability.

By the start of 2009, the subscriptions to Netflix's streaming services had increased by nearly 100 times. Yuri Izrailevsky, VP Cloud Platform at Netflix, gave a presentation in 2013 at the AWS reinvent conference. "We would not be able to scale our services using an on-premise solution," said Izrailevsky.

Furthermore, Izrailevsky stated that the benefits of scalability in the cloud became more evident when looking at its rapid global expansion. "In order to give our European customers a better low-latency experience, we launched a second cloud region in Ireland. Spinning up a new data center in a different territory would take many months and millions of dollars. It would be a huge investment." said Izrailevsky.

As Netflix began its move to hosting their applications on Amazon Web Services, employees of the company would chronicle their learnings on Netflix's company blog. Many of Netflix's employees were advocating a move to a new kind of architecture that focused on horizontal scalability at all layers

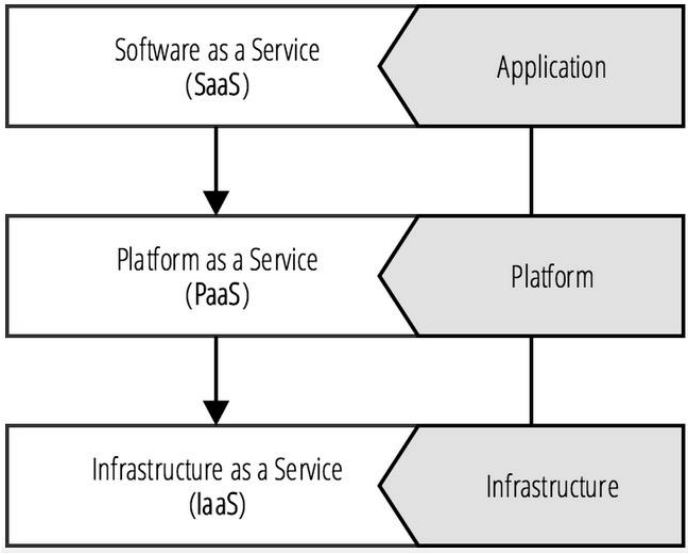
of the software stack.

John Ciancutti, who was then the Vice President of Personalization Technologies at Netflix, said on the company's blog in late 2010 that, "cloud environments are ideal for horizontally scaling architectures. We don't have to guess months ahead what our hardware, storage, and networking needs are going to be. We can programmatically access more of these resources from shared pools within Amazon Web Services almost instantly."

What Ciancutti meant by being able to "programmatically access" resources, was that developers and operators could programmatically access certain management APIs that are exposed by Amazon Web Services in order to give customers a controller for provisioning their virtualized computing infrastructure. The interface for these APIs are in the form of RESTful web services, and they give developers a way to build applications that manage and provision virtual infrastructure for their applications.

Note

Providing management services to control virtualized computing infrastructure is one of the primary concepts of cloud computing, called *Infrastructure as a Service*, commonly referred to as IaaS.



Ciancutti admitted in the same blog post that Netflix was not very good at predicting customer growth or device engagement. This is a central theme behind cloud native companies. Cloud native is a mindset that admits to not being able to reliably predict when and where capacity will be needed.

In Yuri Izrailevsky's presentation at the 2013 AWS reinvent conference, he said that "in the cloud, we can spin up capacity in just a few days as traffic

eventually increases. We could go with a tiny footprint in the beginning and gradually spin it up as our traffic increases."

Izrailevsky goes on to say "As we become a global company, we have the benefit of relying on multiple Amazon Web Services regions throughout the world to give our customers a great interactive experience no matter where they are."

The economies of scale that benefited Amazon Web Services's international expansion also benefited Netflix. With AWS expanding availability zones to regions outside of the United States, Netflix expanded its services globally using only the management APIs provided by AWS.

Izrailevsky quoted a general argument of cloud adoption by enterprise IT, "Sure, the cloud is great, but it's too expensive for us." His response to this argument is that "as a result of Netflix's move to the cloud, the cost of operations has decreased by 87%. We're paying 1/8th of what we used to pay in the data center."

Izrailevsky explained further why the cloud provided such large cost savings to Netflix. "It's really helpful to be able to grow without worrying about capacity buffers. We can scale to demand as we grow."

Splitting the Monolith

There are two cited major benefits by Netflix of moving to a distributed systems architecture in the cloud from a monolith: agility and reliability.

Netflix's architecture before going cloud native comprised of a single monolithic JVM application. While there were multiple advantages of having one large application deployment, the major drawback was that development teams were slowed down due to needing to coordinate their changes.

When building and operating software, increased centralization decreases the risk of a failure at an increased cost of needing to coordinate. Coordination takes time. The more centralized a software architecture is, the more time it will take to coordinate changes to any one piece of it.

Monoliths also tend not to be very reliable. When components share resources on the same virtual machine, a failure in one component can spread to others, causing downtime for users. The risk of making a breaking change in a monolith increases with the amount of effort by teams needing to coordinate their changes. The more changes that occur during a single release cycle also increase the risk of a breaking change that will cause downtime. By splitting

up a monolith into smaller more singularly focused services, deployments can be made with smaller batch sizes on a team's independent release cycle.

DevOps

Netflix not only needed to transform the way they build and operate their software, they needed to transform the culture of their organization. Netflix moved to a new operational model, called DevOps. In this new operational model each team would become a product group, moving away from the traditional project group structure. In a product group, teams were composed vertically, embedding operations and product management into each team. Product teams would have everything they needed to build and operate their software.

Netflix OSS

As Netflix transitioned to become a cloud native company, they also started to participate actively in open source. In late 2010, Kevin McEntee, then the VP of Systems & Ecommerce Engineering at Netflix, announced in a blog post about the company's future role in open source.

McEntee stated that "the great thing about a good open source project that solves a shared challenge is that it develops its own momentum and it is sustained for a long time by a virtuous cycle of continuous improvement."

In the years that followed this announcement, Netflix open sourced over 50 of their internal projects, each of which would become a part of the *Netflix OSS* brand.

Key employees at Netflix would later clarify on the company's aspirations to open source many of their internal tools. In July 2012, Ruslan Meshenberg, Netflix's Director of Cloud Platform Engineering, published a post on the company's technology blog. The blog post, titled *Open Source at Netflix*, explained why Netflix was taking such a bold move to open source so much of its internal tooling.

Meshenberg wrote in the blog post, on the reasoning behind its open source aspirations, that "Netflix was an early cloud adopter, moving all of our streaming services to run on top of AWS infrastructure. We paid the pioneer tax – by encountering and working through many issues, corner cases and limitations."

The cultural motivations at Netflix to contribute back to the open source

community and technology ecosystem are seen to be strongly tied to the principles behind the microeconomics concept known as *Economies of Scale*. Meshenberg then continues, stating that "We've captured the patterns that work in our platform components and automation tools. We benefit from the scale effects of other AWS users adopting similar patterns, and will continue working with the community to develop the ecosystem."

In the advent of what has been referred to as the *era of the cloud*, we have seen that its pioneers are not technology companies such as IBM or Microsoft, but rather they are companies that were born on the back of the internet. Netflix and Amazon are both businesses who started in the late 90s as dot-com companies. Both companies started out by offering online services that aimed to compete with their *brick and mortar* counterparts.

Note

According to the Wikipedia entry for *Economies of Scale*, it states that "economies of scale are the cost advantages that enterprises obtain due to size, output, or scale of operation, with cost per unit of output generally decreasing with increasing scale as fixed costs are spread out over more units of output."

Both Netflix and Amazon would in time surpass the valuation of their *brick and mortar* counterparts. As Amazon had entered itself into the cloud computing market, it did so by turning its collective experience and internal tooling into a set of services. Netflix would then do the same on the back of the services of Amazon. Along the way, Netflix open sourced both its experiences and tooling, transforming themselves into a cloud native company built on virtualized infrastructure services provided by AWS by Amazon. This is how the economies of scale are powering forward a revolution in the cloud computing industry.

In early 2015, on reports of Netflix's first quarterly earnings, the company was reported to be valued at $32.9 billion. As a result of this new valuation for Netflix, the company's value had surpassed the value of the CBS network for the first time. In the same year, Amazon has reportedly surpassed Walmart for the first time, to become the most valuable retailer in the United States by market capitalization.

Building Cloud Native Java Applications

Netflix has provided the software industry with a wealth of knowledge as a result of their move to become a cloud native company. This book is going to focus taking the learnings and open source projects by Netflix and apply them as a set of patterns with two central themes:

- Building resilient distributed systems using Spring and Netflix OSS
- Using continuous delivery to operate cloud native applications with Cloud Foundry

The first stop on our journey will be to understand a set of terms and concepts that we will use throughout this book to describe building and operating cloud native applications.

Twelve Factor Apps

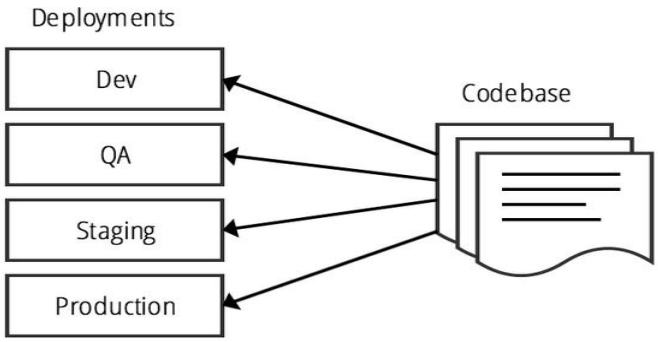
The first of these patterns is the *Twelve Factor App*, a methodology laid out as a set of rules and guidelines that distinguish what it means to be a cloud native application. The *Twelve Factor App* was originally written by Adam Wiggins, co-founder of the popular open platform Heroku, as a manifesto that describes *software-as-a-service* apps that are built to be deployed to cloud providers using a platform.

The twelve factor app breaks down into the following set of concerns.

- Dependency Isolation and Management
- Externalized Configuration
- Backing Services

Dependency Isolation and Management

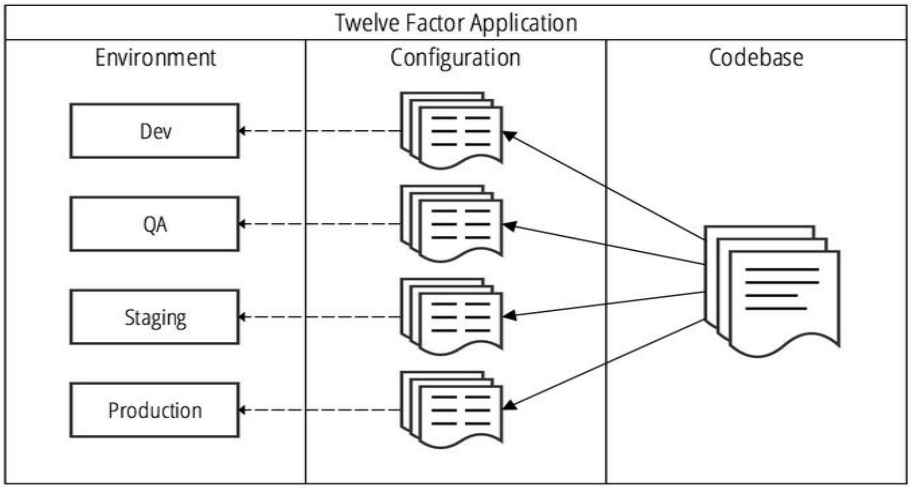
Source code repositories for an application should contain a single application with a manifest to its application dependencies. These application dependencies should be explicitly declared and any and all dependencies should be available from an artifact repository that can be downloaded using a dependency manager, such as Apache Maven.



Externalized Configuration

Application code should be strictly separated from configuration. The configuration of the application should be driven by the environment. Any divergence in your application from environment to environment is considered an environment configuration, and should be stored in the environment and not with the application.

Application settings such as connection strings, credentials, or host names of dependent web services, should be stored as environment variables, making them easy to change without deploying configuration files.



Backing Services

A backing service is any service that the twelve-factor application consumes as a part of its normal operation. Examples of backing services are databases, API-driven RESTful web services, SMTP server, or FTP server.

Backing services are considered to be *resources* of the application. These *resources* are attached to the application for the duration of operation. A deployment of a twelve-factor application should be able to swap out an

embedded SQL database in a testing environment with an external MySQL database hosted in a staging environment without making changes to the application's code.

Containerized Workloads

The next pattern we will look at is containerized workloads. Container deployments are an emerging pattern that have seen rapid adoption in the last few years. During Netflix's transition to become a cloud native company, they admitted to using Amazon Machine Images (AMI).

Note

Amazon Machine Images (AMI) are virtual server instances that have been customized and configured for re-use on EC2 (Elastic Cloud Compute) on AWS. Information from these customized instances are saved and used to launch new instances.

When deploying a new version of an application to AWS, Netflix would bake it into an AMI. The new AMI would then be used to spin up a cluster of the new version of the application. This is a common usage pattern with large cluster deployments on AWS. In this sense, AMIs are a container for our applications to be deployed to a cloud environment, allowing us to bundle our twelve-factor apps as a virtual appliance.

When most people think about containers today, they think Docker. Docker has largely provided us with a standard pattern for understanding what cloud native applications look like. I think it's important though to understand that Docker is not the end result for cloud native applications.

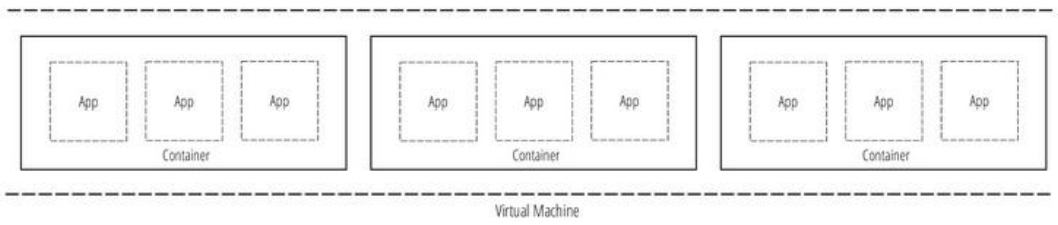


Figure 1-1. A virtual machine hosting multiple containerized applications

Containers give developers the freedom to build applications using the tools that they believe are best suited for solving a specific problem. Containers can take many forms. Core to the idea of containers is the idea of bundling applications with its dependencies and scheduling it to run on a virtual machine where it will only consume an allotted amount of system resources.

Scheduler

Schedulers are services that manage the logistics of container scheduling. Schedulers act as the operator that decide when, where, and how a container is provisioned with a cloud provider, such as Amazon EC2. The responsibility of a scheduler is to manage the logistics of moving containers around in the cloud and can be configured to elastically scale the number of virtual machines that host containers based on usage requirements. Schedulers help to give software operators a dial to control the cost of operating cloud native applications.

Composition and scheduling are important concepts when using containers or developing microservices. Composition and scheduling are how cloud native applications take advantage of elastic scaling, a strategy that maximizes the benefits of using a cloud provider. We are able to package applications into a container that can be composed and scaled to increase operational efficiency.

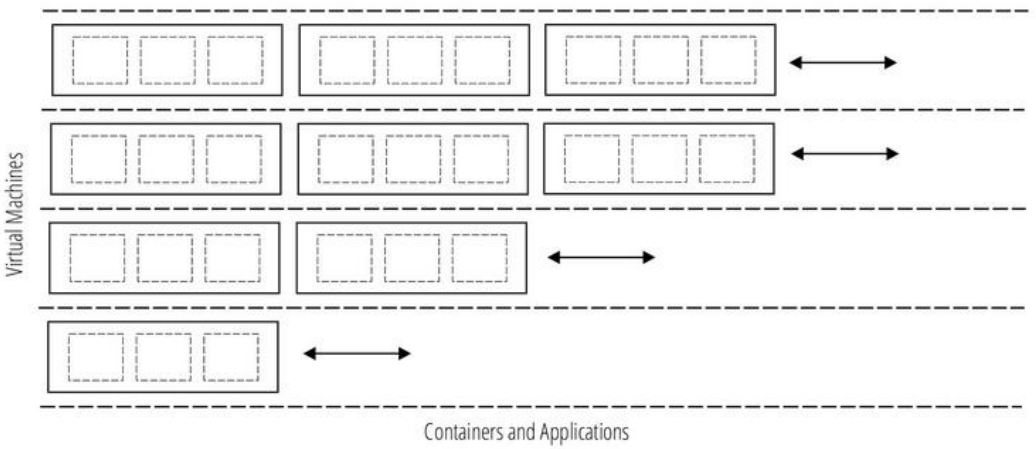


Figure 1-2. Containers are scheduled on virtual machines

The graphic above illustrates rows of 4 virtual machines that host a set of containers. Each container hosts a set of unique applications, isolated from other containers on the machine. The resources available to containers can be described by demarcating the resource priority for the container. Assigning a priority to a container prevents resource contention between applications in separate containers on the same VM. One container may host a long-living web application in production, this container will be given higher priority over a non-production short-living application in another container.

One of the greatest benefits of cloud native application development is that it helps mitigate the cost of operating applications. With a cloud native platform,

developers and operators shouldn't need to be concerned with manually addressing capacity concerns for applications. A cloud native application platform will optimize the cost of operation against a set of policies that a business can define.

Service Discovery

The language of how containers talk to each other on a cloud platform is called composition. The composition of cloud native applications that are scheduled on a cloud provider can be thought of as an automatic process for applications inside containers integrating with applications inside other containers. Composition focuses on how containers find each other after a scheduler has provisioned a new virtual machine and where it will run.

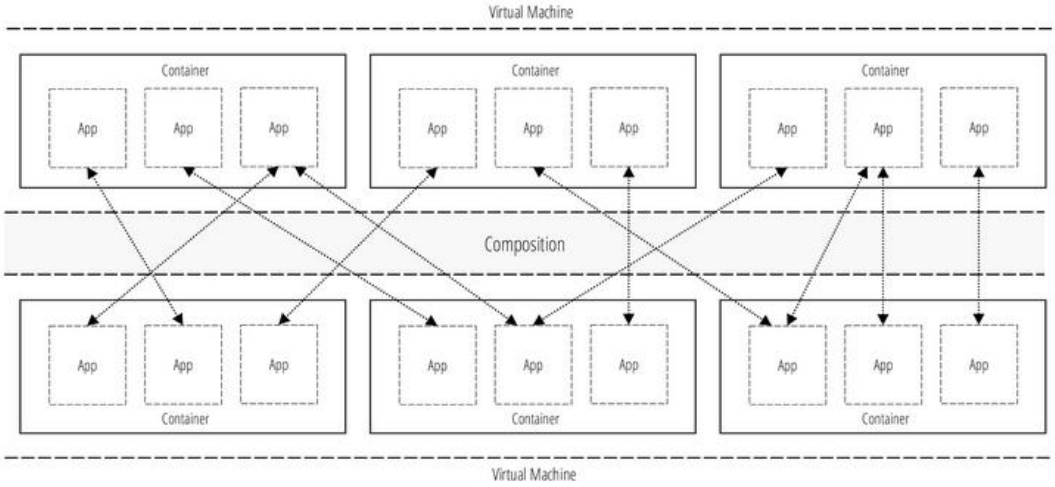


Figure 1-3. Composition of Cloud Native Applications

Container schedulers make it hard for applications to know in advance where to look for an application that is running in another container. Due to the logistics of the scheduler needing to provision virtual machines on-demand, when an application inside a container needs to talk to an application in another container, composition is the mechanism that enables it.

Tip

In the context of microservices, management APIs, such as a discovery service, provides applications a way to automatically compose themselves for integration.

A monolithic application's performance is dependent upon the computer it runs on. Because of this, the capacity of workloads by a monolithic application

will depend on the chip-design of the computer. When scaling an application on a virtual machine, the point of scale depends on a single factor: the time it takes for an instruction cycle to be processed by the machine. This is largely dependent on the hardware of the computer. There is only so much a programmer can do to optimize the speed of a workload. Eventually you will run out of capacity and have to scale up the machine's virtualized hardware resources.

When we talk about availability for web applications, we are talking about the percentage of time that an application is available to its users. Increasing availability depends on the number machines that can balance the load of users actively requesting resources from a web application. The cost to operate a web application is then multiplied by the number of machines that are available to serve traffic to users.

If we were to decompose a monolithic web application into a set of smaller distributed web services, we can schedule and scale these applications much more efficiently on a set of virtual machines. In order to do this effectively, we need to address a set of concerns about the degree of complexity added by the communication between our distributed services.

When we scale out applications, the unit of scale is predicated on two factors: the speed of the network and the number of virtual machines that an application instance runs on. This approach provides opportunities to scale computing resources elastically using a scheduler, delivering cost savings for resource-hungry computing workflows that would otherwise be too expensive to be scaled up.

The adoption of cloud native application development is providing us opportunities to improve application performance, resiliency, availability, and many other factors when it comes to operating web applications at scale.

In this chapter we've talked about some of the common patterns that are requisite for optimizing how we operate cloud native applications. Now we will talk about the abstractions in today's popular open source tooling that we will use to build cloud native JVM applications in this book. At the center of today's Java ecosystem we have a set of popular tools from Spring that companies like Netflix are using to build their cloud native applications.

When it comes to building cloud native applications using Java, the open source projects *Spring Boot* and *Spring Cloud* are leading the way.

Spring Boot

Spring Boot is a JVM microframework within the Spring ecosystem of projects. Spring Boot aims to provide developers with the quickest way to get started with building production-ready cloud native applications. Spring Boot applications can be customized to automatically configure desired components of the Spring Framework as a set of starter projects defined in your application's dependencies.

For those of you who are new to Spring Boot, the name of the project means exactly what it says. You get all of the best things of the Spring Framework and ecosystem of projects, tuned to perfection, with minimal configuration, all ready for production.

As we start building multiple cloud native Spring Boot applications that are dependent on one another, we quickly find that we will have entered into the realm of building *microservices*. We will take a deeper look at building microservices as Spring Boot applications later on in this book.

Tip

Martin Fowler popularized the term microservice on his website as *"an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."*

As we start building many connected microservices, we enter into the world of building and operating distributed systems. On Netflix's journey to become a cloud native company, they too found that they were building and operating distributed systems. Netflix would have to break apart its monolith into microservices. To be successful at this, they needed to develop a set of tools to help prevent failures in this new kind of architecture. Some of the tools Netflix created to build resilient distributed systems would eventually find their way into a new Spring project called *Spring Cloud*.

Spring Cloud

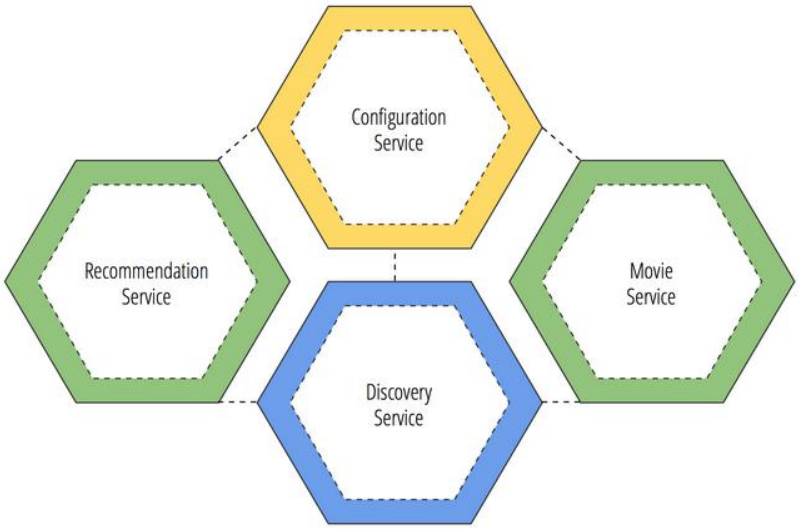
Spring Cloud is a collection of tools that provides solutions to some of the commonly encountered patterns when building distributed systems. If you're familiar with building applications with Spring Framework, Spring Cloud builds upon some of its common building blocks.

Among the solutions provided by Spring Cloud, you will find tools for the

following problems:

- Configuration management
- Service discovery
- Circuit breakers
- Distributed tracing

Each Spring Cloud application in an architecture has a dedicated purpose and serves a specific role. When building Spring Cloud applications using Spring Boot, there are a few primary concerns to deal with first. The two Spring Cloud applications you will likely want to create first are a *Configuration Service* and a *Discovery Service*.



The graphic above illustrates four Spring Boot applications depicted as microservices. Each Spring Boot application plays a specific role in this architecture, with the connections between them indicating a service dependency.

Note

A Spring Cloud application extends a Spring Boot application and will provide a set of minimal features for applications to find each other, as well as preventing cascading failures using resiliency patterns such as circuit breakers.

The configuration service sits at the top, in yellow, and is depended on by the other Spring Boot microservices. The discovery service sits at the bottom, in blue, and also is depended upon by the other Spring Boot microservices.

In green, we have two Spring Boot microservices that deal with a part of the

domain data of the distributed cloud native application.

Configuration Service

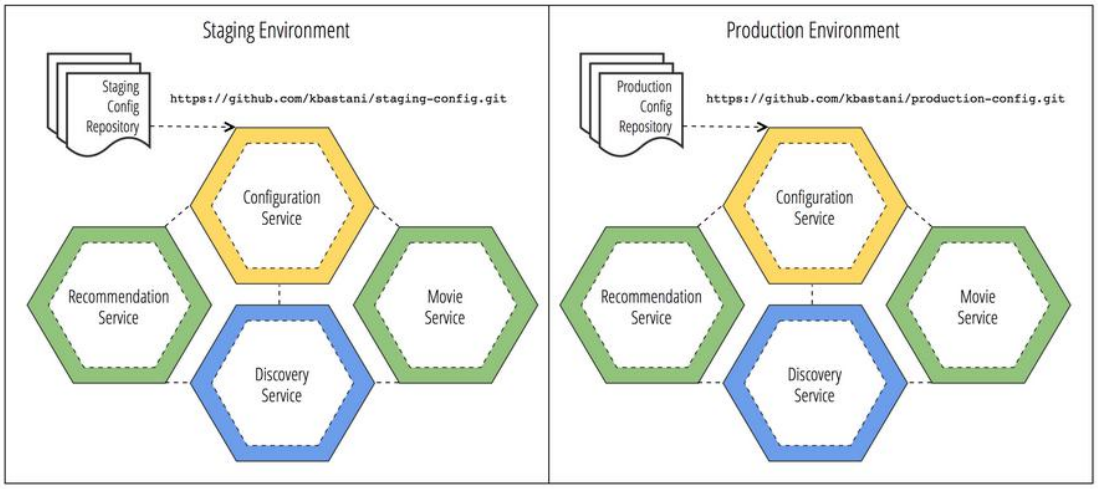
The configuration service is a vital component of any microservices architecture. Based on the principles of twelve-factor applications, the configurations for your microservice should be stored in the environment and not in the project.

Tip

When we use the word *microservice* in this book, we are talking about a type of Spring Boot application that has been customized with a web starter project. The web starter project gives a Spring Boot application the minimal functionality to be considered a microservice.

The configuration service is essential because it is responsible for handling the customized application properties for each microservice in an environment. The configuration service will provide these properties to other services through a simple point-to-point service call. The advantages of this are multi-purpose.

Let's assume that we have multiple deployment environments. If we have a staging environment and a production environment, configurations for those environments will be different. A configuration service might have a dedicated Git repository for the configurations of that environment. None of the other environments will be able to access this configuration, it is available only to the configuration service running in that environment.



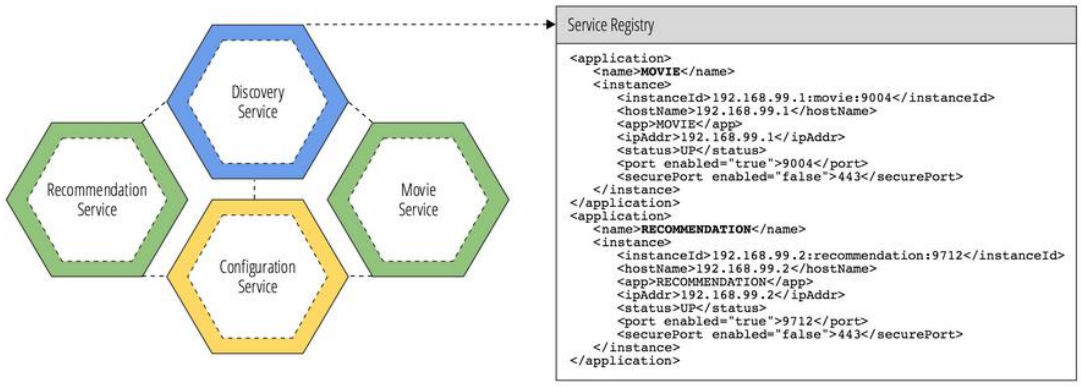
When the configuration service starts up, it will reference the path to those

configuration files and begin to serve them up to the microservices that request those configurations. Each microservice can have their configuration file customized to the specifics of the environment that it is running in. In doing this, the configuration is both externalized and centralized, existing in one place that is version-controlled. By doing this, changes to configurations can be made without having to restart the microservice that a change is intended for.

With management endpoints available from Spring Cloud, you can make a configuration change in the environment and signal a refresh to the discovery service. This refresh signal will force all consumers to fetch the new configurations and to start using them.

Discovery Service

Each Spring Boot application has the ability to take on a certain role in a microservices architecture. One of the central themes behind cloud native applications and microservices is *service discovery*. This takes on the form of a service registry where applications are able to *check-in* and register information about how they can be contacted.



Spring Boot applications that register with a discovery service are able to download a registry that contains information about other service instances. This service registry will contain details of each healthy application instance that has previously registered with the discovery service.

Note

A registry that is managed by a discovery service is often compared to a *phone book*. Long ago, before paper phone books were made obsolete by smart phones and the internet, people needed to find each other's telephone numbers using a telephone directory. Subscribers would provide their full name, location, and telephone numbers, so that they could be reached by

other subscribers in their neighborhood.

The information in the service registry can not only be used to communicate between services, but the registry can also be used to perform client-side load balancing. Client-side load balancing is a technique where consuming services can load balance their requests to a set of service instances within a cluster.

The service instances are grouped together in a service registry, indicating that they are members of a cluster of identical services. The consuming service can then use a load balancing strategy to choose which service in the cluster that it should contact for each request.

Circuit Breakers

The *Circuit Breaker* is a resiliency pattern popularized by Michael Nygard in the book *Release It!*. Nygard introduced this pattern as a systems metaphor into an electric circuit breaker that is installed in a home to prevent electrical fires from an overloaded circuit.

Nygard stated in his book that "the circuit breaker exists to allow one subsystem (an electrical circuit) to fail (excessive current draw, possibly from a short-circuit) without destroying the entire system (the house)".

Figure 1-4 shows each possible state of a circuit breaker, depicted as a rounded box. The directed relationships between the rounded boxes indicate the result of a transition from one state to another state.

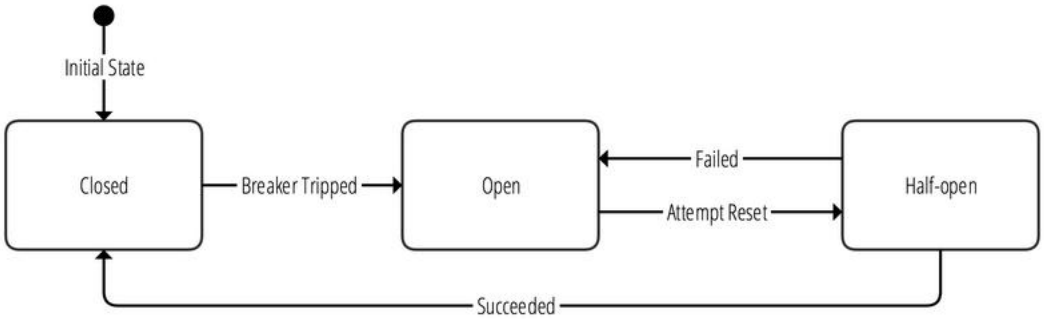


Figure 1-4. state machine diagram of a circuit breaker

Starting from the initial state of the system, we can see that the circuit breaker starts in a *closed* state. When a circuit breaker is in a closed state, traffic to a web service is functioning as normal, with responses being returned at a manageable rate by the service.

Let's now assume that the web service has started to return errors to consumers, indicating that the service is either overloaded or a downstream dependency of the service has failed. The web service will then transition the state of the circuit breaker from a *closed* state to an *open* state. When the breaker has tripped into an *open* state, it will fall back to the designated functionality for this condition. In most cases when using a circuit breaker pattern, the fallback response will closely resemble the response that a client is expecting, with perhaps a default set of properties or an empty list of results.

The circuit breaker will then attempt to try some of the incoming requests while in an *open* state. These sample requests will check to see whether or not functionality has been restored to the offending service call and that responses are returning without errors. In this case, the service will transition from an *open* state to a *half open* state. If the result of the sample requests succeed, without error, then the service's circuit breaker will be reset and transition back to the initial state. The state transition will move from the *half open* state to the *closed* state, and the behavior of the service will settle back into serving traffic normally to any service that requests its resources.

Each Spring Cloud application that has been configured as a *Hystrix client*, will automatically take advantage of the circuit breaker pattern, among other resiliency patterns that come with Hystrix.

Note

Hystrix is a Netflix OSS project that focuses on preventing cascading failures, monitoring latency, and enabling resiliency patterns in distributed systems. Hystrix integrates with Spring Boot applications as a part of Spring Cloud, and also provides a dashboard for monitoring clients.

Later in this book we take a deeper look into resiliency patterns that are used by *Hystrix* and how to take advantage of using them in your Spring Cloud enabled applications.

Summary

In this chapter we've discussed some of the common patterns that will be used throughout this book. We are going to focus a lot on implementation, using the common patterns from twelve-factor application architectures as they are applied using three tools: Spring Boot, Spring Cloud, and Cloud Foundry.

Chapter 2. 12-Factor Application Style Configuration

The Confusing Conflation of "Configuration"

Let's establish some vocabulary. When we talk about *configuration* in Spring, we've *usually* talked about the inputs into the Spring framework's various ApplicationContext implementations that help the container understand how you want beans wired together. This might be an XML file to be fed into a ClassPathXmlApplicationContext, or Java classes annotated a certain way to be fed into an AnnotationConfigApplicationContext. Indeed, when we talk about the latter, we refer to it as *Java configuration*.

In this chapter, however, we're going to look at configuration as it is defined in 12-Factor app style configuration page. Such configuration avoids constants embedded in the code. The page provides a great litmus test for whether configuration has been done correctly: could the codebase of an application be open-sourced at any moment without exposing and compromising important credentials? This sort of configuration refers only to the values that change from one environment to another, not - for example - to Spring bean wiring or Ruby route configuration.

Support in Spring framework

Spring has supported Twelve-Factor-style configuration since the PropertyPlaceholderConfigurer class was introduced. Once an instance is defined, it replaces literals in the XML configuration with values that it resolved in a .properties file. Spring's offered the PropertyPlaceholderConfigurer since 2003. Spring 2.5 introduced XML namespace support and with it XML namespace support for property placeholder resolution. This lets us substitute bean definition literal values in the XML configuration for values assigned to keys in a (external) property file (in this case simple.properties which may be on the class path or external to the application).

Twelve-Factor-style configuration aims to eliminate the fragility of having *magic strings* - values like database locators and credentials, ports, etc. -

hard-coded in the compiled application. If configuration is externalized, then it can be replaced without requiring a rebuild of the code.

The PropertyPlaceholderConfigurer

Let's look at an example using the PropertyPlaceholderConfigurer, Spring XML bean definitions, and an externalized .properties file. We simply want to print out the value in the property file, which looks like this:

This is a Spring ClassPathXmlApplicationContext so we use the Spring context XML namespace and point it to our some.properties file. Then, in the bean definitions, use literals of the form ${configuration.projectName} and Spring will replace them at runtime with the values from our property file.

❶

A classpath: location refers to a file in the current compiled code unit (.jar, .war, etc.). Spring supports many alternatives, including file: and url:, that would let the file live external to the compiled unit.

Finally, here's a Java class to pull it all together:

The first examples used Spring's XML bean configuration format. Spring 3.0 and 3.1 improved things considerably for developers using Java configuration. These releases saw the introduction of the @Value annotation and the Environment abstraction.

The Environment Abstraction and @Value

The Environment abstraction provides a bit of runtime indirection between the running application and the environment in which it is running and lets the application ask questions ("what's the current platform's line.separator?") about the environment. The Environment acts as a map of keys and values. You can configure where those values are read from by contributing a PropertySource. By default Spring loads up system environment keys and values, like line.separator. You can tell Spring to load up configuration keys from a file, specifically, using the @PropertySource annotation.

The @Value annotation provides a way to inject values into fields. These values can be computed using the Spring Expression Language or using property placeholder syntax, assuming one registers a PropertySourcesPlaceholderConfigurer.

❶

the @PropertySource annotation is a shortcut, like property-placeholder, that configures a PropertySource from a .properties file.

❷

you need to register the PropertySourcesPlaceholderConfigurer as a static bean because it is a BeanFactoryPostProcessor and must be invoked earlier in the Spring bean initialization lifecycle. This nuance is invisible when you're using Spring's XML bean configuration format.

❸

you can decorate fields with the @Value annotation..

❹

..or you can decorate fields with the @Value annotation.

❺

@Value annotations can be declared on Spring Java configuration @Bean provider method arguments, as well.

This example loads up the values from a file, simple.properties, and then has one value, configuration.projectName, injected using the @Value annotation and then read again from Spring's Environment abstraction in various ways. To be able to inject the values with the @Value annotation, we need to register a PropertySourcesPlaceholderConfigurer.

Profiles

The Environment also brings the idea of *profiles*. It lets you ascribe labels (profiles) to groupings of beans. Use profiles to describe beans and bean graphs that change from one environment to another. You can activate one or more profiles at a time. Beans that do not have a profile assigned to them are always activated. Beans that have the profile default are activated only when there are no other profiles are active. You can specify the profile attribute in

bean definitions in XML or alternatively tag classes configuration classes, individual beans, or @Bean-provider methods with @Profile.

Profiles let you describe sets of beans that need to be created differently in one environment versus another. You might, for example, use an embedded H2 javax.sql.DataSource in your local dev profile, but then switch to a javax.sql.DataSource for PostgreSQL that's resolved through a JNDI lookup or by reading the properties from an environment variable in Cloud Foundry when the prod profile is active. In both cases, your code works: you get a javax.sql.DataSource, but the decision about *which* specialized instance is used is decided by the active profile or profiles.

1

this configuration class and all the @Bean definitions therein will only be evaluated if the prod profile is active.

2

this configuration class and all the @Bean definitions therein will only be evaluated if the dev profile *or* **no** profile - including dev - is active.

3

this InitializingBean simply records the currently active profile and injects the value that was ultimately contributed by the property file.

4

it's easy to programmatically activate a profile (or profiles).

Spring responds to a few other methods for activating profiles using the token spring_profiles_active or spring.profiles.active. You can set the profile using an environment variable (e.g.: SPRING_PROFILES_ACTIVE), a JVM property (-Dspring.profiles.active=..), a Servlet application initialization parameter, or programmatically.

Bootiful Configuration

Spring Boot improves things considerably. Spring Boot will automatically load properties in a hierarchy of well-known places by default. The command-line arguments override property values contributed from JNDI, which override properties contributed from System.getProperties(), etc.

- Command line arguments
- JNDI attributes from java:comp/env
- System.getProperties()` properties
- OS environment variables
- External property files on filesystem - (config/)?application.(yml.properties)
- Internal property files in archive (config/)?application.(yml.properties)
- @PropertySource annotation on configuration classes
- Default properties from SpringApplication.getDefaultProperties()

If a profile is active, it will also automatically reads in the configuration files based on the profile name, like src/main/resources/application-foo.properties where foo is the current profile.

If the Snake YML library is on the classpath, then it will also automatically load YML files following basically the same convention. Yeah, read that part again. YML is so good, and so worth a go! Here's an example YML file:

Spring Boot also makes it much simpler to get the right result in common cases. It makes -D arguments to the java process and environment variables available as properties. It even normalizes them, so an environment variable $CONFIGURATION_PROJECTNAME or a -D argument of the form -Dconfiguration.projectname both become accessible with the key configuration.projectName in the same way that the spring_profiles_active token was earlier.

Configuration values are strings, and if you have enough configuration values it can be unwieldy trying to make sure those keys don't themselves become magic strings in the code. Spring Boot introduces a @ConfigurationProperties component type. Annotate a POJO with @ConfigurationProperties and specify a prefix, and Spring will attempt to map all properties that start with that prefix to the POJO's properties. In the example below the value for configuration.projectName will be mapped to an instance of the POJO that all code can then inject and dereference to read the (type-safe) values. In this way, you only have the mapping from a key in one place.

❶

the @EnableConfigurationProperties annotation tells Spring to map properties to POJOs annotated with @ConfigurationProperties.

❷

@ConfigurationProperties tells Spring that this bean is to be used as the root for all properties starting with configuration., with subsequent tokens mapped to properties on the object.

❸

the projectName field would ultimately have the value assigned to the property key configuration.projectName.

Spring Boot uses the @ConfigurationProps mechanism heavily to let users override bits of the system. You can see what property keys can be used to change things, for example, by adding the org.springframework.boot:spring-boot-starter-actuator dependency to a Spring Boot-based web application and then visiting http://127.0.0.1:8080/configprops. This will give you a list of supported configuration properties based on the types present on the classpath at runtime. As you add more Spring Boot types, you'll see more properties. This endpoint will *also* reflect the properties exported by your @ConfigurationProperties-annotated POJO.

Centralized, Journaled Configuration with the Spring Cloud Configuration Server

So far so good, but there are gaps in the approach so far:

- changes to an application's configuration require restarts
- there is no traceability: how do we determine what changes were introduced into production and, if necessary, roll back?
- configuration is de-centralized and it's not immediately apparent where to go to change what.

- sometimes configuration values should be encrypted and decrypted for security. There is no out-of-the-box support for this.

Spring Cloud, which builds upon Spring Boot and integrates various tools and libraries for working with microservices, including the Netflix OSS stack, offers a configuration server and a client for that configuration server. This support, taken together, address these last three concerns.

Let's look at a simple example. First, we'll setup a configuration server. The configuration server is something to be shared among a set of applications or microservices based on Spring Cloud. You have to get it running, somewhere, once. Then, all other services need only know where to find the configuration service. The configuration service acts as a sort of proxy for configuration keys and values that it reads from a Git repository online or on a disk.

Tip

Add the Spring Cloud Config server dependency: org.springframework.cloud : spring-cloud-config-server

❶

@EnableConfigServer installs a configuration service.

Here's the configuration for the configuration service:

❶

This is a normal Spring Boot-ism that configures on which port the embedded web server (in this Apache Tomcat) is to use.

❷

Points to the working Git repository, either local or over the network (like GitHub), that the Spring Cloud Config server is to use.

This tells the Spring Cloud configuration service to look for configuration files for individual client services in the Git repository on my GitHub account. The URI could, of course, just as easily have been a Git repository on my local file system. The value used for the URI could also have been a property reference, of the form, ${SOME_URI}, that references - perhaps - an environment variable called SOME_URI.

Run the application and you'll be able to verify that your configuration service is working by pointing your browser at http://localhost:8888/SERVICE/master where SERVICE is the ID taken from your client service's boostrap.yml. Spring Cloud-based services look for a file called src/main/resources/bootstrap.(properties,yml) that it expects to find to - you guessed it! - bootstrap the service. One of the things it expects to find in the bootstrap.yml file is the ID of the service specified as a property, spring.application.name.

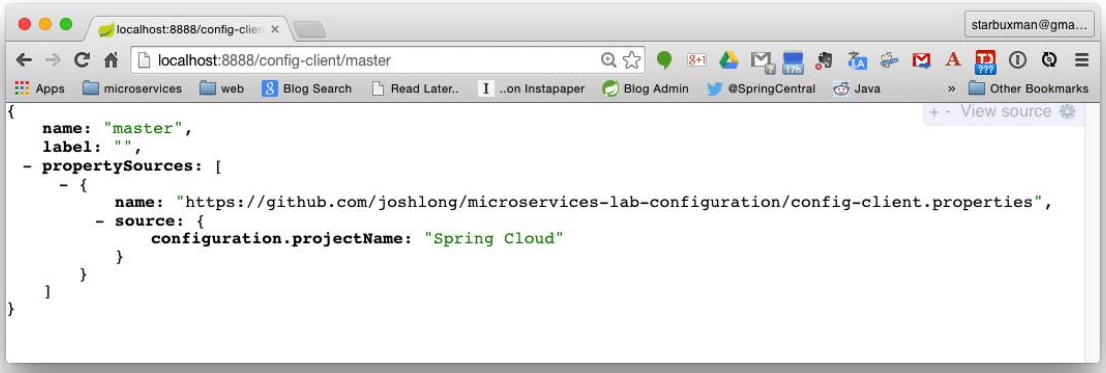


Figure 2-1. The output of the Spring Coud Config Server confirming that it *sees* the configuration in our Git repository

If you manage things correctly, then the only configuration that lives with any of your services should be the configuration that tells the configuration service where to find the Git repository and the configuration that tells the other client services where to find the configuration service, both of which usually live in a file called boostrap.yml in Spring Cloud-based services. This file (or bootstrap.properties) gets loaded than other property files (including application.yml or application.properties). It makes sense: this file tells Spring where it's to find the rest of the application's configuration. Here's our configuration client's bootstrap.yml.

When a Spring Cloud microservice runs, it'll see that its spring.application.name is config-client. It will contact the configuration service (which we've told Spring Cloud is running at http://localhosst:8080, though this too could've been an environment variable) and ask it for any configuration. The configuration service returns back JSON that contains all the configuration values in the application.(properties,yml) file as well as any service-specific configuration in config-client.(yml,properties). It will *also* load any configuration for a given service *and* a specific profile, e.g., config-client-dev.properties.

This all just happens automatically and you can interact with properties exposed via the configuration server like any other configuration property.

Security

Define spring.cloud.config.server.git.username and spring.cloud.config.server.git.password properties for the Spring Cloud Config Server to talk to secured Git repositories.

You can protect the Spring Cloud Configuration Server itself with HTTP BASIC authentication. The easiest is to just include org.springframework.boot: spring-boot-starter-security and then define a security.user.name and a security.user.password property.

The Spring Cloud Config Clients can encode the user and password in the spring.cloud.config.uri value, e.g.: https://user:secret@host.com.

Refreshable Configuration

Centralized configuration is a powerful thing, but changes to configuration aren't immediately visible to the beans that depend on it. Spring Cloud's *refresh* scope offers a solution.

The ProjectNameRestController is annotated with @RefreshScope, a Spring Cloud scope that lets any bean recreate itself (and re-read configuration values from the configuration service) in-place. In this case, the ProjectNameRestController will be recreated - its lifecycle callbacks honored and @Value and @Autowired injects re-established - whenever a *refresh* is triggered.

Fundamentlaly, all *refresh*-scoped beans will refresh themselves when they receive a Spring ApplicationContext-event of the type RefreshScopeRefreshedEvent. There are various ways to trigger the refresh.

You can trigger the refresh by sending an empty POST request to http://127.0.0.1:8080/refresh, which is a Spring Boot Actuator endpoint that is exposed automatically. Here's how to do that using curl:

curl -d{} http://127.0.0.1:8080/refresh`

Alternatively, you can use the auto-exposed Spring Boot Actuator JMX refresh endpoint.

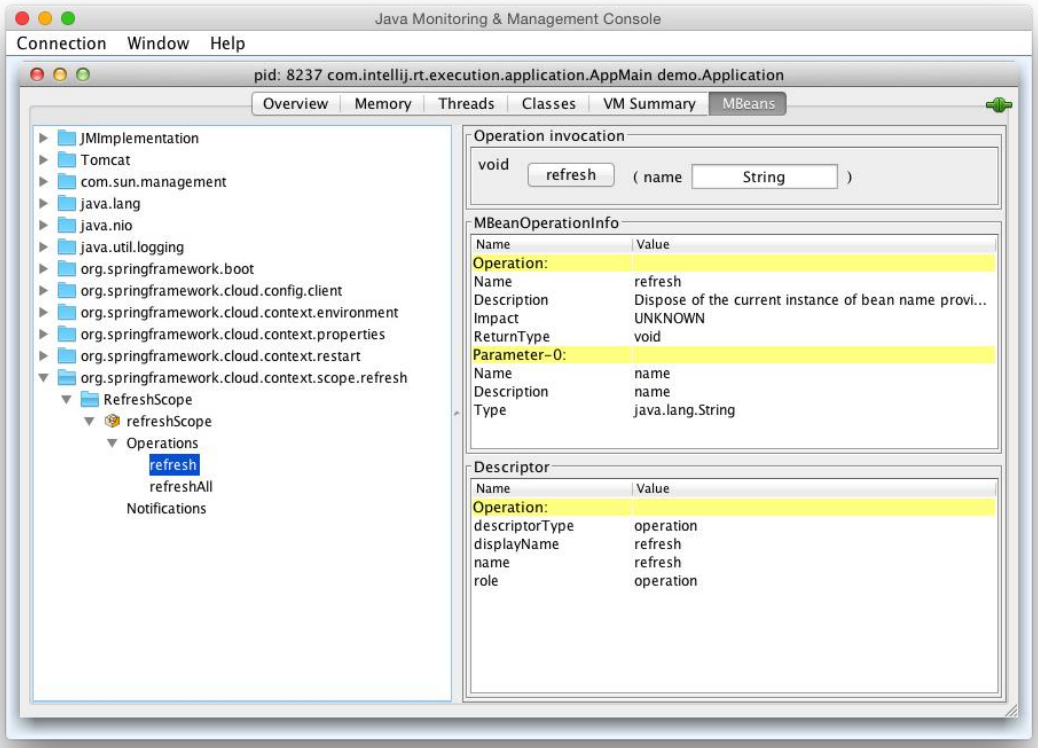


Figure 2-2. Using jconsole to activate the refresh (or refreshAll) Actuator endpoints

To see all this in action, make changes to the configuration file in Git, and at the very least git commit them. Then invoke the REST endpoint or the JMX endpoint on the node you want to see the configuration changed in, *not* the configuration server.

Both of those Spring Boot Actuator endpoints work on an ApplicationContext-by-ApplicationContext basis. The Spring Cloud Bus, on the other hand, provides a simple way to refresh multiple ApplicationContext's (e.g.: many nodes) in one go.

The Spring Cloud Bus links all services through a RabbitMQ powered-bus. This is particularly powerful. You can tell one (or thousands!) of microservices to refresh themselves by sending a single message to a message bus. This prevents downtime and is *much* more friendly than having to systematically restart individual services or nodes.

Tip

Add the Spring Cloud Event Bus dependency org.springframework.cloud : spring-cloud-starter-bus-amqp.

By default Spring Boot's auto-configuration for RabbitMQ will attempt to connect to a local RabbitMQ instance. You can configure the specific host and port in the usual way in your application's application.yml. These specifics tend to apply to multiple services, so you might consider putting them in the application.yml in your configuration server's Git repository. This way, *all* services that connect to the configuration service will *also* talk to the right RabbitMQ instance.

```
spring:
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: user
    password: secret
```

These configuration values create a Spring AMQP ConnectionFactory bean that will be used to listen for event bus messages. If you have multiple ConnectionFactory instances in the Spring application context, you need to *qualify* which instance is to be used with the @BusConnectionFactory annotation. Qualify any other instance to be used for regular, non-bus business processing with the usual Spring qualifier annotation, @Primary.

The Spring Cloud Event Bus exposes a *different* Actuator endpoint, /bus/refresh, that will publish a message to the connected RabbitMQ broker that will trigger *all* connected nodes to refresh themselves. You can send the following message to *any* node with the spring-cloud-starter-bus-amqp auto-configuration, and it'll trigger a refresh in *all* the connected nodes.

```
curl -d{} http://127.0.0.1:8080/bus/refresh`
```

Next Steps

We've covered a *lot* here! Armed with all of this, it should be easy to package one artifact and then move that artifact from one environment to another without changes to the artifact itself. If you're going to start an application today, I'd recommend starting on Spring Boot and Spring Cloud, especially now that we've looked at all the good stuff it brings you by default. Don't forget to check out the code behind all of these examples.

Chapter 3. Using Spring Boot with Java EE

In this chapter we'll look at how to integrate Spring Boot applications with Java EE. Java EE, for our purposes, is an umbrella name for a set of APIs and, sometimes, runtimes - *Java EE application servers*. Java EE application servers, like RedHat's WildFly AS - the application server formerly named JBoss Application Server - provide implementations of these APIs. We'll look at how to build applications that leverage Java EE APIs outside of a Java EE application server. If you're building a brand new application today, you don't need this chapter. This chapter is more useful for those with existing functionality trapped in an application server that want to move to a microservices architecture. For a broader discussion of moving ("forklifting") legacy applications to a cloud platform like Cloud Foundry with minimal refactoring, see the chapter on forklifting applications. [Chapter to Come]

Spring acts as a consumer of Java EE APIs, where practical. It doesn't *require* any of them, or any of the myriad XML configuration files that go along with them. Wherever possible, Spring supports consuming Java EE APIs à la carte, independent of a full Java EE application server. Spring applications should ideally be portable across environments, including embedded web applications, application servers, and virtually any platform-as-a-service (PaaS) offering. Spring also works in any application server, as well.

Compatibility and Stability

Spring 4.2 (the baseline release for Spring Boot 1.3 and later) supports Java SE 6 or later (specifically, the minimum API level is JDK 6u18, which was released in early 2010). Oracle has stopped publishing updates - save for security fixes - for Java SE 6 and Java SE 7. Consider moving to Java SE 8.

Spring also supports Java EE 6+ (which came out in 2009). In practical terms, Spring 4 is Servlet 3.0+ focused but Servlet 2.5 compatible. Spring Boot, which builds on Spring, works best with Servlet 3.1 or later. Spring Boot's approach to dependencies ensure that you get the latest and greatest, as soon as possible, though you can revert to older Servlet 3.0 generation web servers if you like.

Java EE isn't a large market. The market consistently favors, by a wide margin, Apache's Tomcat (a project that Pivotal contributes heavily to), Eclipse's Jetty,

and RedHat's Wildfly. What's up for debate is whether WildFly is in the number two position or whether Jetty is. My friend Arun Gupta (who works at RedHat as their head of evangelism) put together a nice blog on WildFly's penetration in the last 5 years. In all the charts, Tomcat is #1, and WildFly's in the top three. WildFly, let's say, is the most popular *full* Java EE application server implementation, and it represents the sole Java EE implementation with any significant marketshare and less than a third of the market.

Put another way: most developers don't deploy to Java EE-compatible containers. Spring brings useful Java EE APIs to the growing majority that aren't running Java EE application servers.

Java EE offers some very compelling APIs. The Servlet API, JSR 330 (javax.inject) and JSR 303 (javax.validation), JCache, JDBC, JPA, JMS, etc., to name a few, are very useful and - where practical for users - Spring supports them. Indeed, sometimes Spring offers support months or years before certified Java EE implementations, as it did with JSR 107 (the JCache API).

As developers move to the cloud, and as a result, to a cloud-native architecture, we see decreasing value in relying on Java EE application servers; the all-or-nothing approach goes against the grain of building cloud-native systems composed of small, singly focused, scalable services.

Some Java EE APIs are very useful, but they're notoriously slow to evolve. This suggests that the benefits of slow-to-evolve, standardized Java EE APIs are best reaped at layers where there is little volatility. JDBC, which provides the interface for decades-old SQL-based databases, is an example of a perfectly reasonable API.

Java EE 8, as a standard, is due to be released (*very* tentatively) in the early part of 2017. At the time of this writing, the Java EE 8 timeline is a few months off-track. A release in 2017 implies a production timeline of 2019 or longer (because the first major, commercially supported Java EE 7 application server came from IBM more than two years after the release of the standard). Java EE 8 offers no support for NoSQL, or cloud-based applications, but it *does*, very importantly, finally offer a standard API for binding JSON to Java objects! This support is very similar to what Spring supports through the *non-standard* Jackson and GSON libraries in one form or another since 2009. Your choice is simple: prefer business value, or prefer standards. If you want standards, then choose them where they are not differentiators, where they may as well be commoditized: HTTP, REST, JDBC, etc., are good examples. Another great reason to choose a standard is for ubiquity: JPA is a *very* ubiquitous technology and is used in *many* Spring applications, outside of Java EE application servers. Community and the associated hiring pool, in this case, are the features.

Software serves a purpose, usually a business purposes, and businesses don't compete by relying on sometimes decades-old, inferior solutions. If we, on the JVM, are to thrive then we must be able to develop and deploy solutions competitively. Imagine suggesting to a Node.js or Rails developer that they *may* get JSON support in 2019!

Compatibility is a feature, too. Spring tries to retain backwards-compatibility. An application written using Spring 1.0 can in 99% of situations be upgraded with a drop-in replacement of the newer release libraries of Spring, even on older generations of Java EE. Replace the dependencies for an application written against J2EE (what Java EE was called before it was rebranded ten years or so ago from J2EE and J2SE and J2ME to *Java EE*, *Java SE*, and *Java ME*) with modern Java EE dependencies and chunks of the code - based on EJB 1.0-2.1, for example - would not compile, let alone run.

With that quick discussion of the wisdom of Java EE in modern applications behind us, let's look at a few commonplace APIs that you'll find work marvelously in Spring Boot applications.

Dependency Injection with JSR 330 (and JSR 250)

Spring has long offered various approaches to providing configuration. Spring doesn't care, really, where it learns about your objects and how you wire them together. In the beginning there was the XML configuration format. Later, Spring introduced component scanning to discovery and register components with stereotype annotations like @Component. In 2006 the Spring Java Configuration project was born. This approach describes beans to Spring in terms of objects returned from methods that are annotated with the @Bean annotation. You might call these bean definition *provider methods*.

Meanwhile, at Google, the amazing Bob Lee introduced Guice, which also provides the ability to define bean definition provider methods, like the Spring Java configuration project. These two options, along with a few others, evolved independently and each garnered a large community.

Let's skip ahead to 2007, 2008, and 2009 and formative days of Java EE 6. The team behind JBoss' Seam who had developed their own dependency injection technology attempted to define a standard, JSR 299 (CDI), to define what it means to be a dependency injection technology. Naturally, neither Seam nor JSR 299 looked anything like Spring or Guice, the two most entrenched and popular technologies, *by any stretch*, so Spring founder Rod Johnson and Guice founder Bob Lee proposed JSR 330. JSR 330 defines a

common set of annotations for the surface area of dependency injection frameworks *that impact business component code*. This left each dependency injection container to differentiate in the way configuration itself is handled.

Tip

JSR 330 was not offered as a large, tedious specification with hundreds of pages, but instead as a tiny set of JavaDocs for the handful of annotations and a single interface in the proposed API. Many (particularly those behind JSR 299) felt it was an affront to the JCP process, and it was even suggested that it could never make it through the JCP process in time for inclusion in Java EE 6, but it did! At the time, it was the fastest JCP standard ever.

JSR 330 is natively supported, of course, by Spring and Guice and - because cooler heads eventually prevailed - by JSR 299 implementations. It's so common in fact other DI technologies like Dagger, which is optimized for compile-time code-generation and mobile environments like Android, also support it. If you need to have portable dependency injection, use JSR 330.

You'll commonly use a few annotations with JSR 330, assuming you have javax.inject : javax-inject : 1 on the classpath, to define, resolve and inject bean references.

- @Inject is equivalent to Spring's @Autowired annotation. It identifies injectable constructors, methods and fields.
- @Named is more or less equivalent to Spring's various stereotype annotations like @Component. They mark a bean to be registered with the container and it can be used to give the bean a String-based ID by which it may be registered.
- @Qualifier can be used to *qualify* beans by type (or String ID). This is, naturally, almost identical to Spring's @Qualifier annotation.
- @Scope is more or less analogous to Spring's @Scope annotation and is used to tell the container what the lifecycle of a bean is. You might, for example, specify that a bean is session scoped, that is - it should live and die along the lines of an HTTP request.
- @Singleton tells the container that the bean should be instantiated only once. This is Spring's default, but it's such a common concept that it was worth making sure all implementations support it.

JSR 330 also defines one interface, javax.inject.Provider<T>. Business components can inject instances of a given bean, Foo, directly or using a Provider<Foo>. This is more or less analogous to Spring's ObjectFactory<T> type. Compared to injecting an instance directly, the Provider<T> instance can be used to retrieve multiple instances of a given bean, handle *lazy* semantics, break circular dependencies, and abstract containing scope.

Tip

JSR 250, which comes from Java EE 5, is *also* supported natively in Spring if the types are on the classpath (they are in newer versions of the JDK). You may have seen these annotations if you've ever used @javax.annotation.Resource, for example. These annotations are commonly used in EJB 3 code, but I've never really seen them used elsewhere. They can make it easy for developers moving code over from EJB 3 environments where references resolution is signalled with @Resource.

Building REST APIs with JAX-RS (Jersey)

Spring Boot makes it dead simple to stand up REST APIs using JAX-RS. JAX-RS is a standard and requires an implementation. Our example demonstrates Boot's JAX-RS auto-configuration for Jersey 2.x in the GreetingEndpoint. The example uses the Spring Boot starter org.springframework.boot : spring-boot-starter-jersey.

Example 3-1. The JAX-RS GreetingEndpoint

1. JSR 330's @Inject annotation

2. JAX-RS's @Path annotation is functionally equivalent to Spring MVC's @RequestMapping. It tells the container under what route this endpoint should be exposed.

3. Spring MVC's @RequestMapping provides a produces and consumes attribute that lets you specify what content-types a given endpoint can consume, or produce. In JAX-RS, this mapping is done with standalone annotations.

4.

The HTTP verb, also otherwise specified in Spring MVC's @RequestMapping annotation, is specified here as a standalone annotation.

❺

The @QueryParam annotation tells JAX-RS to inject any incoming request parameters (?name==..) as method arguments. In Spring MVC, you'd use @RequestParam-annotated method arguments, instead.

Jersey requires a ResourceConfig subclass to enable basic features and register components.

Example 3-2. The ResourceConfig subclass that configures Jersey

❶

we register the endpoint using JSR 330, though we could just as easily have used any of Spring's stereotype annotations here. They're interchangeable.

❷

we need to explicitly register our JAX-RS endpoint.

❸

we need to tell JAX-RS that we want to handle JSON marshalling. Java EE doesn't have a built-in API for marshalling JSON (as we discussed above, it'll tentatively be available in Java EE 8), but you can use Jersey-specific *feature* implementations to plugin popular JSON-marshalling implementations like Jackson.

Spring Boot auto-configures the Jersey org.glassfish.jersey.servlet.ServletContainer as both a javax.servlet.Servlet and a javax.servlet.Filter that listens for all requests relative to the application root.

In JAX-RS, message encoding and decoding is done through the javax.ws.rs.ext.MessageBodyWriter and javax.ws.rs.ext.MessageBodyReader SPIs, somewhat akin to Spring MVC's org.springframework.http.converter.HttpMessageConverter hierarchy. By default, JAX-RS does not have many useful message body readers and writers enabled. The example registers a JsonFeature in the ResourceConfig subclass to support JSON encoding and decoding.

JAX-RS is a capable API, but it's forever constrained to evolve slower, and be supported by a smaller community that is itself fractured across a handful of implementations. If you have code using JAX-RS, it's comforting to know it works, but consider using a more mature REST toolkit, otherwise. Spring offers a richer, integrated Spring MVC-based stack complete with MVC, REST, HATEOAS, OAuth, SSE, and websocket support.

JTA and XA Transaction Management

Resource-Local Transactons with Spring's PlatformTransactionManager

Fundamentally, transactions work more or less the same way: a client begins work, does some work, commits the work and - if something goes wrong - restores (*rolls back*) the state of a resource to how it was before the work began. Implementations across the numerous resources vary considerably. JMS clients create a *transacted* Session that is then committed. JDBC Connection can be made to *not* auto-commmit work, which is the same as saying it batches work, and then commit when explicitly instructed to do so.

Resource-local transactions should be your default approach for transaction management. You'll use them with various transactional Java EE APIs like JMS, JPA, JMS, CCI, and JDBC. Spring supports numerous other transactional resources like AMQP-brokers such as RabbitMQ, the Neo4j graph database, and Pivotal Gemfire. Unfortunately, each of these transactional resources offers a different API for initiating, committing, or rolling back work. To simplify things, Spring provides the PlatformTransactionManager hierarchy. There are numerous, pluggable implementations of PlatformTransactionManager that adapt the various notions of transactions to a common API. Spring is able to manage transactions through implementations of this hierarchy. Spring provides tiered support for transactions.

At the lowest level, Spring provides the TransactionTemplate. The TransactionTemplate wraps a PlatformTransactionManager bean and uses it to manage a transaction given a unit of work which the client provides as an implementation of TransactionCallback. Let's first wire everyhing up. We have some things that will be common to the following two examples - a database DataSource, a DataSourceInitializer, a JdbcTemplate, a PlatformTransactionManager, etc. - so we'll define them in a shared Java configuration class. This Java configuratino also defines a RowMapper, which is a JdbcTemplate callback interface that maps rows of results to Java objects.

1. define a DataSource that just talks to an in-memory embedded H2 database

2. define a DataSourceInitializer that runs schema and data initialization DDL

3. a Spring JdbcTemplate which reduces common JDBC calls to one liners

4. a RowMapper is a callback interface that JdbcTemplate uses to map SQL ResultSet objects into objects, in this case of type Customer.

5. the DataSourceTransactionManager bean adapts the DataSource transactions to Spring's PlatformTransactionManager hierarchy.

We can use Spring's low-level TransactionTemplate to demarcate transaction boundaries explicitly.

1. the TransactionTemplate requires only a PlatformTransactionManager

❷

a TransactionCallback defined with Java 8 lambas. The body of this method will be run in a valid transaction and comitted and closed upon completion.

❸

the return value could be whatever you want, but I think this is in itself an interesting clue: transactions should ultimately be towards obtaining one, valid, biproduct.

The TransactionTemplate should be familiar if you've ever used any of Spring's other template implementations. The TransactionTemplate defines transaction boundaries explicitly. It's very useful when you want to cordon off sections of logic in a transaction, and to control when that transaction starts and stops.

Use annotations to declaratively demarcate transactional boundaries. Spring has long supported declarative transaction rules, both external to the business components to which the rules apply, and inline. The most popular way to define transaction rules as of Spring 1.2, released in May 2005 just after Java SE 5, is to use Java annotations.

So, what's all this to do with Java EE? Spring supports declarative transaction management with its @Transactional annotation. It can be placed on a type or on individual methods. The annotation can be used to specify transactional qualities such as propagation, the exceptions to rollback for, etc.

A year later, Java EE 5 debuted and included EJB 3 which defined an annotation-based way to define transaction boundaries with its javax.ejb.TransactionAttribute annotation. Spring *also* honors this annotation if it's discovered in Spring beans. The TransactionAttribute works well for EJB based business components, but the approach falls apart fast outside of EJB based components. JTA 1.2, which was included in Java EE 7 in 2013 (but which, as we discussed above, isn't readily available and not at all supported in production as of April 2015), defined javax.transaction.Transactional as a general purpose transaction boundary annotation, more or less like Spring's @Transactional from 8 years earlier. Spring *also* honors this annotation, if present.

The configuration to make this work is basically the same as with the TransactionTemplate, except that we turn on annotation-based transaction

boundaries with the @EnableTransactionManagement annotation and don't need the TransactionTemplate bean anymore.

❶

@EnableTransactionManagement turns on the transaction processing. It requires that a valid PlatformTransactionManager be defined somewhere.

❷

You could use Spring's @org.springframework.transaction.annotation.Transactional

❸

or EJB's @javax.ejb.TransactionAttribute

❹

or JTA 1.2's @javax.transaction.Transactional

Prefer Spring's @Transactional variant. It exposes more power than the EJB3 alternative in that it is able to expose transactional semantics (things like transaction suspension and resumption) that the EJB-specific @TransactionAttribute (and indeed EJB itself) don't expose, and doesn't require an extra JTA or EJB dependency. It is nice to know that it will work, either way, however.

These examples require only a DataSource driver and org.springframework.boot : spring-boot-starter-jdbc.

Global Transactions with the Java Transaction API (JTA)

Spring makes working with *one* transactional resource easy. But what is a developer to do when trying to transactionally manipulate more than one transactional resource, e.g., a *global transaction*? For example, how should a developer transactionally write to a database *and* acknowledge the receipt of a JMS message? *Global* transactions are the alternative to *resource-local*

transactions; they involve *multiple* resources in a transaction. To ensure the integrity of a global transaction, the coordinator must be an agent independent of the resources enlisted in a transaction. It must be able to guarantee that it can replay a failed transaction, and that it is itself immune to failures. JTA (necessarily) adds complexity and state to the process that a resource-local transaction avoids. Most global transaction managers speak the X/Open XA protocol which lets transactional resources (like a database or a message broker) enlist and participate in global transactions. Java EE provides a very handy API on top of this protocol called JTA.

Integration is, at least in theory, simple. For our purposes, it suffices to know that JTA exposes two key interfaces - the javax.transaction.UserTransaction and javax.transaction.TransactionManager. The UserTransaction supports the usual suspects: begin(), commit(), rollback(), etc. Clients use this object to begin a global transaction. While the global transaction is open, JTA-aware resources like a JTA-aware JDBC javax.sql.XADataSource, or a JTA-aware JMS javax.jms.XAConnectionFactory may *enlist* in the JTA transaction. They communicate with the global transaction coordinator and follow a protocol to either atomically commit the work in all enlisted resources or rollback. It is up to each atomic resource to support, or not support, the JTA protocol, which itself speaks the XA protocol.

In a Java EE container, the UserTransaction object is *required* to exist in a JNDI context under the binding java:comp/UserTransaction, so it's easy for Spring's JtaTransactionManager to locate it. This simple interface is enough to handle *basic* transaction management chores. Notably, it is *not* sophisticated enough to handle sub-transactions, transaction suspension and resumption, or other features typical of quality JTA implementations. Instead, use a TransactionManager instance, if it's available. Java EE application servers are *not* required to expose this interface, and *rarely* expose it under the same JNDI binding. Spring knows the well-known contexts for many popular Java EE application servers, and will attempt to automatically locate it for you.

Tip

Often, the bean that implements javax.transaction.UserTransaction *also* implements javax.transaction.TransactionManager!

JTA can also be run outside of a Java EE container. There are numerous popular third party (and open-source) JTA implementations, like Atomikos and Bitronix. Spring Boot provides auto-configurations for both.

Atomikos is commercially supported and provides an open-source base edition. Let's look at an example - in GreetingService - that uses JMS to send

notifications and JPA to persist records to an RDBMS as part of a global transaction.

1

We tell Spring that all public methods on the component are transactional.

2

This code uses Spring's JmsTemplate JMS client to work with a JMS resource.

3

This code uses the spring-boot-starter-data-jpa support so can inject a JPA EntityManager with JPA's @PersistenceContext annotation as per normal Java EE conventions.

4

This method takes a boolean that triggers an exception if true. This exception triggers a rollback of the JTA transaction. You'll only see evidence that two of the three bodies of work completed after the code is run.

We demonstrate this in GreetingServiceClient by creating three transactions and simulating a rollback on the third one. You should see printed to the console that there are two records that come back from the JDBC javax.sql.DataSource data source and two records that are received from the embedded JMS javax.jms.Destination destination.

1

We haven't seen @PostConstruct yet. It's part of JSR 250, and is semantically the same as Spring's InitializingBean interface. It defines a callback to be invoked *after* the beans dependencies - be they constructor arguments, JavaBean proprties, or fields.

Tip

Spring Boot will automatically setup JPA based on the configured DataSource. This example uses Spring Boot's embedded DataSource support. If an embedded database (like H2, which is what we're using here, or Derby, and HSQL) is on the classpath, and no javax.sql.DataSource is explicitly defined, Spring Boot will create a DataSource bean for use.

Spring Boot makes it dead simple to setup a JMS connection, as well. Just like the embedded DataSource, Spring Boot can also create an embedded JMS ConnectionFactory using RedHat's HornetQ message broker in embedded mode assuming the correct types are on the classpath (org.springframework.boot : spring-boot-starter-hornetq) and a few properties are specified:

With that in place, just add the requisite Spring Boot starters (org.springframework.boot : spring-boot-starter-hornetq) and HornetQ support (org.hornetq: hornetq-jms-server) to activate the right auto-configurations.

If you wanted to connect to traditional, non-embedded instances, it's straightforward to either specify properties in application.yml or application.properties like spring.datasource.url and spring.hornetq.host, or simply define @Bean definitions of the appropriate types.

This example uses the Spring Boot Atomikos starter (org.springframework.boot: spring-boot-starter-atomikos) to configure Atomikos and the XA-aware JMS and JDBC resources.

In today's distributed world, consider global transaction managers an architecture smell. Distributed transactions gate the ability of one service to process transactions at an independent cadence. Distributed transactions imply that state is being maintained across multiple services when they should ideally be in a single microservice. There are other patterns for state synchronization that promote horizontal scalability and temporal decoupling, centered around messaging. Be sure to check out chapter [Link to Come] the section on messaging and integration.

Deployment in a Java EE Environment

This example also uses the Wildfly (from RedHat) application server's **awesome** Undertow embedded HTTP server instead of (the default) Apache Tomcat. It's as easy to use Undertow as it is to use Jetty or Tomcat - just exclude spring-boot-starter-tomcat and add spring-boot-starter-undertow!

This contribution originated as a third-party pull request and I've really enjoyed it because it's very fast. You can as easily use Eclipse's Jetty project if you disable Apache Tomcat. To do this, define a dependency for org.springframework.boot: spring-boot-starter-tomcat and then set its scope to provided. This will have the effect of ensuring that Tomcat is *not* on the classpath. From there, you need only add org.springframework.boot: spring-boot-starter-jetty or org.springframework.boot: spring-boot-starter-undertow!

Though this example uses a lot of fairly familiar Java EE APIs, this is still just typical Spring Boot, so by default you can run this application using java -jar ee.jar or easily deploy it to process-centric platforms-as-a-service offerings like Heroku or Cloud Foundry.

If you want to deploy it to a standalone application server like (like Apache Tomcat, or Websphere, or anything in between), it's straightforward to convert the build into a .war and deploy it accordingly to any Servlet 3 container. In most cases, it's as simple as making provided or otherwise excluding the Spring Boot dependencies that provide Servlet APIs (like spring-boot-starter-tomcat or spring-boot-starter-jetty) that the Java EE container is going to provide. If you use the Spring Initializr, then you can skip this step by simply choosing war from the packaging drop-down.

For Servlet based web applications, you'll need to also add a Servlet initializer class. This is the programmatic equivalent of web.xml. Spring Boot provides a base class, org.springframework.boot.context.web.SpringBootServletInitializer, that will automatically standup the Spring Boot based auto-configuration and machinery in a standard Servlet 3 fashion. Note that any embedded web-container features - like SSL configuration and port management - will not work when applications are deployed like this.

If you deploy the application to a more classic application server, Spring Boot can take advantage of the application server's facilities, instead. It's dead-simple to consume a JNDI-bound JMS ConnectionFactory, JDBC DataSource or JTA UserTransaction.

Final Word

We would question a lot of these APIs. Do you **really** need distributed, multi-resource transactions?. JPA's a nice API for talking to a SQL-based javax.sql.DataSource, but Spring Data repositories (which include support for JPA, of course, go further and simpify straight JPA considerably. They *also* include support for Cassandra, MongoDB, Redis, Riak, Couchbase, Neo4j, and

an increasingly long list of alternative technologies. Repositories are reduced to a single interface definition for the common cases. Do you really need all of this? It might well be that you do, and the choice is yours, but hopefully this is a stopgap on the road to a microservice architecture.

In the context of microservices, Java EE goes against the grain of what a modern microservice architecture needs. Microservices are about small singly focused services with as minimal heft as possible. Some modern application servers can be very tiny, but the principal is the same: why pay for what you don't need? Why go so far to decouple your services from each other only to be satisfied with unneeded coupling of the infrastructure that runs the code?

Application servers were built for a different era (the late 1990s) when RAM was at a premium. The application server promised application isolation and monitoring and consolidated infrastructure, but today it's fairly easy to see it doesn't do a particularly good job of any of that. An application server doesn't protect applications from contention from other applications beyond the class loader. It can't isolate CPU, RAM, filesystems, and even in the JVM itself one component (like the JTA coordinator) can starve the web request processing pool. Instead, these things are provided by the operating system. The operating system cares and knows about processes, not application servers.

It's very common to see applications with the application server configuration (or even the application server itself!) checked into source control systems alongside the application. This (worrisome) approach implies that nobody's trying to squeeze RAM out of their machines by collocating applications on the same application server. Instead, applications need certain application-specific services and with a Java EE container the path of least resistance is to access those from the container. But that's not the easiest way. Spring applications enjoy security, messaging, distributed transactions, centralized configuration, service resolution, and much more without being coupled to an application server.

Many developers use the application server (or web servers like Apache Tomcat, to which Pivotal is a leading contributor) because it presents a consistent operations burden. Operations know how to deploy, scale and manage the application server. Fair enough. Today, however, the container of choice is something like Docker. Docker exposes a consistent management surface area for operations to work with. What runs inside is entirely opaque.

Docker only cares about processes. There's no assumption that your application will even expose an HTTP endpoint. Indeed, there's often no reason that it should. There's nothing about microservices that implies HTTP and REST, common though it is. By moving away from the application server and moving to .jar based deployments, you can use HTTP (and a Servlet

container) if and only if it's appropriate. I like to think of it like this: what if the protocol *du jour* was FTP, and not HTTP? What if the whole world accessed services exposed through an FTP server? Does it feel just as architecturally unassailable to deploy our applications to an FTP server? No? Then why do we deploy them to HTTP, EJB, JMS, JNDI and XA/Open servers?

In this chapter we've tried to show that Spring plays well with Java EE services and APIs, and that those APIs can be handy, even in the world of microservices. What we hope you'll take away is that there's no need for the Java EE application server itself. It's just going to slow you down.