O'REILLY®

Modern Java EE Design Patterns

Building Scalable Architecture for Sustainable Enterprise Development

Markus Eisele

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Modern Java EE Design Patterns

Building Scalable Architecture for Sustainable Enterprise Development

Markus Eisele

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Modern Java EE Design Patterns
by Markus Eisele

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
Production Editor: Shiny Kalapurakkel
Copyeditor: Charles Roumeliotis
Proofreader: Jasmine Kwityn

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2015: First Edition

Revision History for the First Edition
2015-10-05: First Release
2016-01-15: Second Release

978-1-491-93982-6

[LSI]

Table of Contents

Acknowledgments

Writing books takes a lot more time than reading them—and it requires a lot more people to be successful at it. I am thankful to have had the technical support and creativity of Mark Little, Arun Gupta, and Daniel Bryant throughout the writing process and beyond.

I cannot forget my girls here. Thank you! I love you!™

CHAPTER 1
Enterprise Development Today

Enterprise is a noun. One of its meanings refers to a project or undertaking, especially a bold or complex one. But it also refers more generally to businesses or corporations. Used in the context of software technology, the term encapsulates a mixture of these meanings, which is underlined by the inability to adopt new technologies at a reasonable speed due to a large organization's inability to move quickly. Nevertheless, all those attributes and descriptions are very personal based on specific work environments. And not everything about this negative introduction is bad. The reasons behind this are obvious: those complex undertakings or large organizations need a much higher level of standardization than startups. Changing a small thing for one out of a hundred projects might lead to unanticipated problems.

One major technology that has become a standard platform across most enterprises to build complex—and stable—applications is Java Enterprise Edition (Java EE). And while this technology stack has come a long way since its inception in 1998, it is still not meant to be used for innovation and the adoption of more cutting-edge technologies and development paradigms.

Nevertheless, innovation and constant improvement are the drivers behind enterprises and enterprise-grade projects. Without innovation, there will be outdated and expensive infrastructure components (e.g., host systems) that are kept alive way longer than the software they are running was designed for. Without constant validation of the status quo, there will be implicit or explicit vendor

lock-in. Aging middleware runs into extended support and only a few suppliers will still be able to provide know-how to develop for it. Platform stacks that stay behind the latest standards attempt to introduce quick and dirty solutions that ultimately produce technical debt.

And typically every 5 to 10 years, the whole software industry, especially in the enterprise integration or enterprise application space, spits out a new methodology or architectural style that promises to make everything 10 times more productive, agile, flexible, and responsive. As a result, we've seen everything from the creation of enterprise application integration, web services, and service-oriented architecture (SOA) to component-based architectures and enterprise service buses (ESBs).

Enterprise Goals and Objectives

As technology has evolved, the decision makers in enterprise IT departments have implemented new capabilities and processes across their organizations. Thus, IT has changed operations and turnaround for the better. But besides this technical standardization and forward march of progress in internal operations and cost cutting, these departments are still accused of not understanding the needs of the business. Operations and buying decisions are still focused on generating quick results from investments and long-term cost savings. These results ignore the need for new business requirements or market developments, such as the still growing mobile market or the new communication style of a whole generation.

Resistant to Change and Economically Efficient

Speaking of this mismatch, operations and business have always followed completely distinct goals while working on the greater good. Operations and sourcing have won out mostly. It's an easier business case to calculate how much a corporation-wide standardization for a Java EE application server can produce in savings than to examine the individual lines of source code and maintenance that must be dealt with for each individual project. And it's not only the difference in mindset behind this. It's also about long-term support and license agreements. Instead of changing the foundation and every-

thing attached to it a couple of times a year, decisions need to guarantee a decent support level over many years. Following this, the gap between what latest technology is state-of-the-art and what enterprises allow developers to work with grows larger each year.

Developers Left Alone

Even if the preceding analysis barely scratches the surface, it reveals why developers are feeling left alone in those enterprise settings. Having to fight the same stack day in and day out might have advantages for generating knowledge about common pitfalls and shortcomings, but it also puts a strong block on everything that promises to solve problems more elegantly, in shorter timeframes, and with a lot less code. And we haven't even talked about the other problem that results from this.

Technology-Centric Versus Business-Centric

Many traditional enterprises have become strongly business-centric and mostly treat IT and operations as cost centers. The goal of providing homogenous IT services was mostly reached by overly focusing on IT architectures, information formats, and technology selection processes to produce a standard platform for application operations. This produced a dangerous comfort zone that siphons attention away from the real value of business software: the business domains and relevant processes whose standardization and optimization promise a much higher payback than just operational optimizations.

The good news is that many organizations have started to take notice and are undertaking changes toward easier and more efficient architecture management. But change is something that doesn't necessarily have to come from above; it is also the responsibility of every developer and architect. As a result, today's buzzwords have to be incorporated in a manageable way by all parties responsible for creating software.

Aims and Scope

So, there's a lot to reflect on here. This report focuses on how enterprises work and how the situation can be improved by understanding how—and when—to adopt the latest technologies in such an

environment. The main emphasis is on understanding Java EE design patterns, as well as how to work with new development paradigms, such as microservices, DevOps, and cloud-based operations.

This report also introduces different angles to the discussion surrounding the use of microservices with well-known technologies, and shows how to migrate existing monoliths into more fine-grained and service-oriented systems by respecting the enterprise environment. As you'll come to find out, Java EE is only a very small—yet crucial—component of today's enterprise platform stacks.

CHAPTER 2
History of Java EE

"Develop once, run everywhere!" This developer promise was the key driving force behind much of the success enjoyed by the Java programming language. And as interest in Java continued to rise through broad adoption, the need for robust enterprise-grade applications soared. The advent of the Internet and the first browser versions led to the implementation of the first web server in Java as well as the introduction of the Servlet and JSP specifications. These two specifications became the foundation for the Java 2 Enterprise Edition (J2EE) platform, and from 1999 to 2003, the number of contained Java Specification Requests (JSRs) grew from 10 to 18. The platform was renamed Java EE in 2006 (and now carries its version number on the tail end). As of the writing of this report, the most recent version is Java EE 7 (JSR 342), with Java EE 8 (JSR 366) due to release at the end of 2016.

Enterprises adopted Java EE early—and often—because of the many advantages it promised, such as centralized infrastructures, a scalable, transactional, and standardized programming model, high throughput, and reliable operations. However, every single promise came with a drawback, and it took a while until the platform as a specification embraced operational and developer performance. Given the slow uptake of new versions by both vendors and customers, we still see a lot of Java EE 5-based applications out in the wild (this particular release dates back to mid-2006).

Mistakes We Made

Traditionally, Java EE applications followed the core pattern defined in the book *Core J2EE Patterns* and were separated into three main layers: presentation, business, and integration. The presentation layer was packaged in Web Application Archives (WARs) while business and integration logic went into separate Java Archives (JARs). Bundled together as one deployment unit, a so-called Enterprise Archive (EAR) was created.

The technology and best practices around Java EE have always been sufficient to build a well-designed monolith application. But most enterprise-grade projects tend to lose a close focus on architecture. The aspects of early Java EE applications outlined in Figure 2-1 don't make assumptions about their technical capabilities, and are derived from experience in the field.

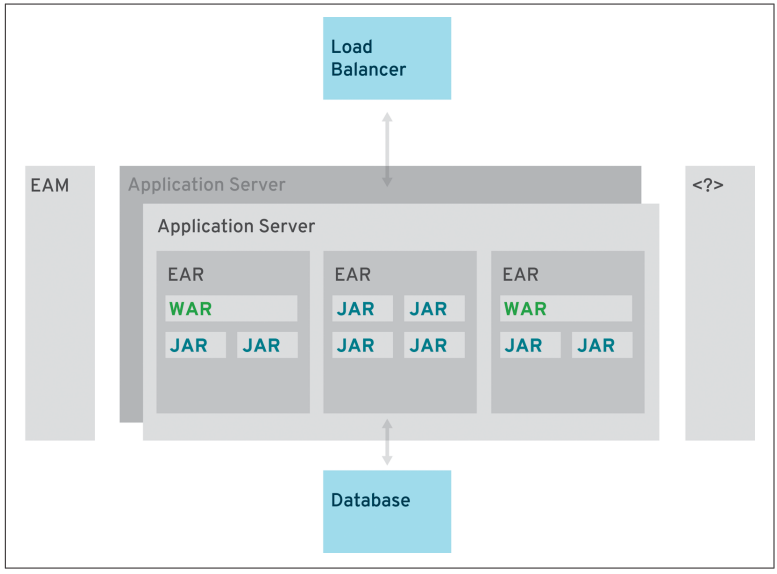

Figure 2-1. Typical enterprise Java application

Those applications could be scaled with the help of more instances of the application server and a load balancer. If the responsible architect thought about reuse here, he most likely considered implementing a common JAR or library that became part of all the applications in the enterprise. Crosscutting concerns, such as single sign-on (SSO), were taken care of by enterprise access management

(EAM) solutions, and there are even more centralized enterprise-level infrastructures (e.g., logging, monitoring, and databases).

Because everything was too coupled and integrated to make small changes, applications also had to be tested with great care and from beginning to end. A new release saw the light of day once or twice a year. The whole application was a lot more than just programmed artifacts: it also consisted of uncountable deployment descriptors and server configuration files, in addition to properties for relevant third-party environments.

Even the teams were heavily influenced by these monolithic software architectures. The multimonth test cycle might have been the most visible proof. But besides that, projects with lifespans longer than five years tended to have huge bugs and feature databases. And if this wasn't hard enough, the testing was barely qualified—no acceptance tests, and hardly any written business requirements or identifiable domains in design and usability.

Handling these kinds of enterprise projects was a multiple team effort and required a lot of people to oversee the entire project. From a software design perspective, the resulting applications had a very technical layering. Business components or domains were mostly driven by existing database designs or dated business object definitions. Our industry had to learn those lessons and we managed not only to keep these enterprise monoliths under control, but also invented new paradigms and methodologies to manage them even better.

Evolution Continues with ESBs

Another technology was created in the advent of business-centered designs and the broader shift into more service-oriented organizations. The enterprise service bus (ESB) promised to deliver reusability and exchangeability by still standing up as a centralized and managed infrastructure component. Evangelized by many vendors, this technology was poised to be the number one solution to all of the monolithic applications in existence.

All of the applications needed to be sliced and rebuilt to support exchangeable services. Service-oriented architectures (SOA) were the new paradigm behind this. Unfortunately, the interface technology of choice tended to be web services (WS). Web services trans-

port data between systems by encoding it into XML and transporting it via the Simple Object Access Protocol (SOAP). This introduced a significant amount of additional code and descriptors into most projects (see Figure 2-2).

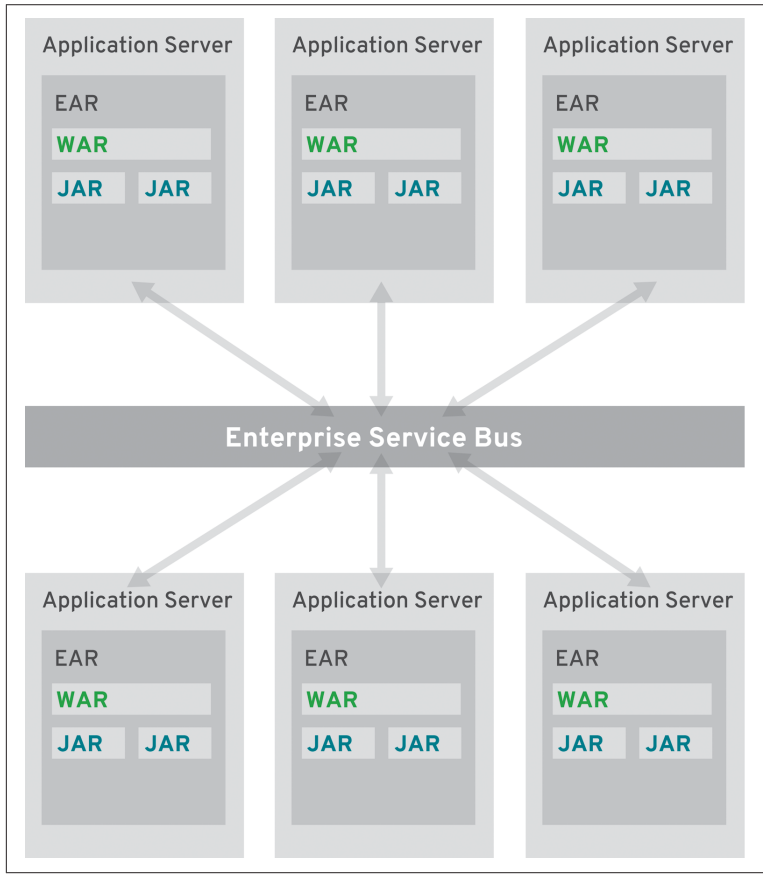

Figure 2-2. ESB-based architectures

What's more, every ESB came with its own tooling and barely two of them could be used without it. As a result, codebases grew even further and developers now had to learn to work with overloaded IDEs to wire all the parts of an application together. Rewiring was now the new coding for the first projects under development.

Instead of switching protocols or serialization methods for the relevant endpoints, every route in the ESB ended up being a transformation. These new possibilities introduced a significant amount of

additional complexity by just offering new methods of transformation and routing inside the ESB. What was monolithic and hard to test until now just became distributed and even harder to test. Although the distribution aspects were not critical, the complex dependencies created serious issues. Small changes in the ESB's routing and transformation logic had a large impact and for a long time there's been no way to even stage those centralized ESB configurations through the different environments—and that's before we even think about versioning them.

While the technology evolved and best practices started to mature, the biggest impact fell on operations. The distributed application had to be monitored, and most importantly, scaled to fit the available resources and required demand. This simple sentence covers a complete set of individual components, starting with the operating system and its resources, which host a single application server or cluster, which itself is hosting any number of web services, all the way up to the scaling of the ESB.

Only completely integrated platforms and expensive, specialized monitoring solutions could help control these deployments. The early days of ESBs were also the days of initial experiments with corporate data models. With business processes becoming the new first-level application design approach and spanning many different attributes, the relevant domain objects needed to be aligned as well. The customer entity turned out to have many more attributes when it was used to complete a process, instead of just one application serving a bunch of related use cases. We learned that data segmentation was a critical success factor.

Challenges and Lessons Learned

From what you've read so far, it's pretty clear that enterprise projects contain challenges on varying levels. And going through the last 5 to 10 years of evolution in this field, it is obvious that the technical challenges haven't been the only ones. It took some time, but the first experts mastered the complexity and refactored the first pattern and best practices.

Our industry also had to learn to handle project management differently, and new project management methodologies became broadly accepted. Iterative and agile approaches required a more fine-grained cut of requirements and teams, and this led to even more

frequent changes, which had to be pushed to test and production. Automation and reliability in the software delivery process were soon accepted practices and made it possible to quickly deliver new features even in the more inflexible setting of an enterprise-grade project.

DevOps: Highly Effective Teams

Another very important part of successful software delivery was a tighter coupling between operations and development. With more frequent changes and a very high automation rate, the number of deployments to individual environments spiked. This was something straight deployment processes could hardly handle. This is why DevOps was born. At its core, the DevOps movement is about team culture. It aims at improving communication and collaboration between developers, operations, and other IT professionals. Based on automation and tooling, it helps organizations to rapidly put high-quality software into production. Even more than that, it manifested itself in a new team communication methodology embracing frequent changes. Development teams not only wanted to just produce code, but also were responsible for pushing complete changes down the DevOps chain into production.

Microservices: Lightweight and Fast

Centralized components no longer fit into this picture, and even heavyweight application servers were revisited alongside wordy protocols and interface technologies. The technical design went back to more handy artifacts and services with the proven impracticality of most of the service implementation in SOA- and ESB-based projects. Instead of intelligent routing and transformations, microservices use simple routes and encapsulate logic in the endpoint itself. And even if the name implies a defined size, there isn't one.

Microservices are about having a single business purpose. And even more vexing for enterprise settings, the most effective runtime for microservices isn't necessarily a full-blown application server. It might just be a servlet engine or that the JVM is already sufficient as an execution environment. With the growing runtime variations and the broader variety of programming language choices, this development turned into yet another operations nightmare. Where Platform as a Service (PaaS) offerings used to be the number one

solution, a new technology found its place in the stack for the next generation of enterprise applications.

Containers: Fully Contained Applications

If operations can't provide full support for all of the available languages and runtimes out there, there has to be something else filling the gap. Instead of hiring an army of specialists for a multitude of runtime environments, containers became an obvious choice. Containers are an approach to virtualization in which the virtualization layer runs as an application within the operating system (OS). The OS's kernel runs on the hardware node with several isolated guest virtual machines (VMs) installed on top of it. The isolated guests are called containers.

They finally gave application developers the opportunity and tooling to not only build, test, and stage applications, but also the complete middleware infrastructure, including the relevant configurations and dependencies. The good news here was that projects no longer depended on centralized platform decisions, and operations were still able to ensure a smooth production.

Public, Private, Hybrid: Scalable Infrastructures

The number of environments needed for those projects spiked. And none of the changes just discussed effectively saved money in operations. Even worse, the additional time spent on making DevOps and containers functional needed to be compensated. This might be the most compelling explanation for the still-growing demand for cloud infrastructures.

Although virtualization has proven to be cost-efficient by running any number of the same instances, it was never easy to manage and was tightly coupled to the hardware underneath. As a matter of fact, it still had to scale alongside the demand of the projects, and there was a cost assigned to every single instance. It literally had to be bought and owned by the project in most cases.

Cloud infrastructures changed this quickly—pay for what you use with rapid provisioning. Just recently, cloud platforms received an upgrade to their capabilities with the emerging container technologies. Instead of spinning up instances of application servers or databases, today's most relevant products rely on containers to define the

software stack to run and provide enough flexibility to projects while maintaining manageability alongside cost-effective operations.

We can see the result of the earlier-mentioned methodologies and technologies in Figure 2-3. This image is a pyramid of modern enterprise application development and reflects the cornerstones of future development. The following chapters will dive deeper into the details of each part of the pyramid.

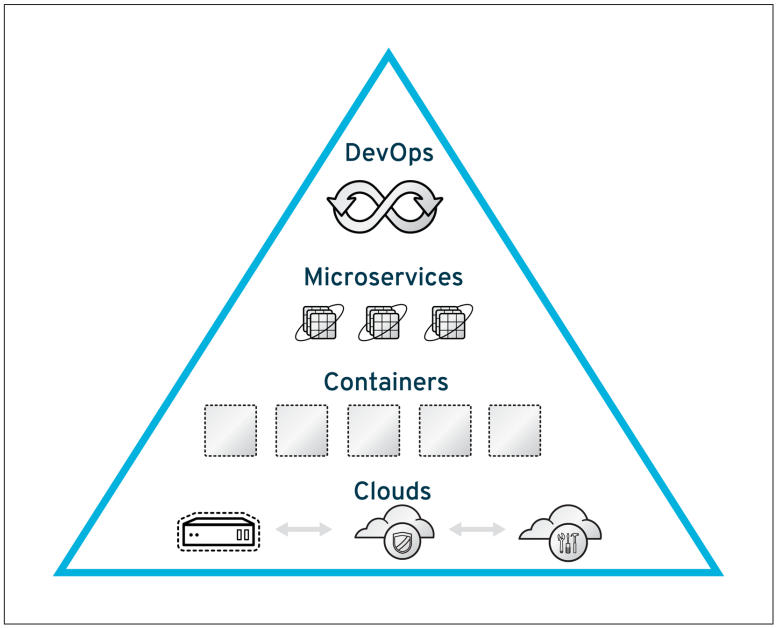

Figure 2-3. The pyramid of modern enterprise application development

CHAPTER 3

Designing Software for a Scalable Enterprise

Looking back at the lessons learned alongside the latest developments in software, the most pressing question becomes: how do developers and architects design software for enterprises that need to scale more quickly? With clouds and containers serving as the new foundation, and more and more applications adopting microservices architecture, everything we knew about software design seems to be turned on its head.

With that said, the basic concepts of software architecture and design that were developed to cater to a multitude of stakeholders, follow separation of concern principles, generate quality, and guarantee the conceptual integrity of our applications remain the primary drivers for creating great software. And yet we do need to pay close attention to some of the principles we already know and choose the correct approach.

Microservices remain an ever-present buzzword and viable design pattern, yet upon closer inspection the name represents a new term for a style of architecture that has been around a while: modular design. Microservices are the right choice if you have a system that is too complex to be handled as a monolith. And this is exactly what makes this architectural style a valid choice for enterprise applications. As Martin Fowler states in his article about "MicroservicePremium":

> The fulcrum of whether or not to use microservices is the complexity of the system you're contemplating.

This quote is perfectly explained in the accompanying graphic in the same article (reproduced here in Figure 3-1). The main point is to not even consider using a microservices architecture unless you have a system that's too large and complex to be built as a classical monolith. As a result, the majority of modern software systems should still be built as a single application that is modular and takes advantage of state-of-the-art software architecture patterns.

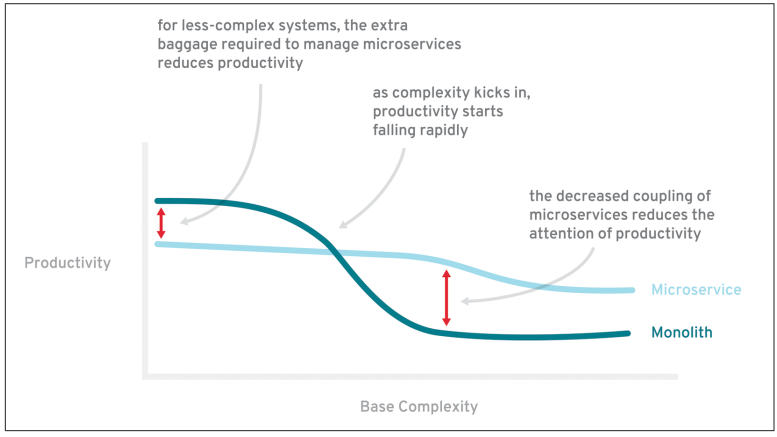

Figure 3-1. Microservices: productivity versus base complexity (source: Martin Fowler; http://martinfowler.com/bliki/MicroservicePremium.html)

As we have been building complex enterprise systems in Java EE for years, it may seem unlikely that we will find one suitable for a microservices architecture. But this is not the complete truth: technical or business complexity should not be the only reason for choosing this kind of architecture.

One of the most important concerns in the current developer environment is team size. With growing developer teams, it seems more reasonable and effective to have completely decoupled services. But there aren't any hard metrics or even estimates about complexity that make a decision easy. The best way to decide which route to pursue will be the overall setting. This starts with the decision about which software system needs to be worked on.

Greenfield Versus Brownfield

Most of today's enterprise software was built years ago and still undergoes regular maintenance to adopt the latest regulations or new business requirements. Unless there is a completely new business case or significant internal restructuring, the need to construct a piece of software from scratch is rarely considered. But let's assume we want to assess the need or even advantage of implementing a new microservices-based architecture. What would be the most successful way to proceed? Start with a new development from scratch (i.e., greenfield), or tear apart an existing application into services (i.e., brownfield)? Both approaches offer some risks and challenges.

> I remain convinced that it is much easier to partition an existing, "brownfield" system than to do so up front with a new, greenfield system.
>
> —Sam Newman (Source: http://bit.ly/1FMXNjs)

As usual, the common ground is small but critical: you need to know the business domain you're working on. And I would like to take this point even further: enterprise projects, especially those considered long-term, tend to be sparse on documentation, and it is even more important to have access to developers who are working in this domain and have firsthand knowledge.

Additionally, I believe any decision will have various shades of complexity. There are a range of options in brownfield developments (i.e., migrations), and this allows for a very selective and risk-free approach that will fit most business requirements (more on this later in "Migration Approaches" on page 35). No matter which avenue you pursue, you'll need to evaluate your own personal toolbox for success. Therefore, in order to help you make the best decision possible, let's get to know the best methodologies and design patterns behind modern enterprise application development.

Domain-Driven Design

The philosophy of domain-driven design (DDD) is about placing the attention at the heart of the application, focusing on the complexity of the core business domain. Alongside the core business features, you'll also find supporting subdomains that are often generic in nature, such as money or time. DDD aims to create models of a problem domain. All the implementation details—like persistence, user interfaces, and messaging—come later. The most crucial thing to understand is the domain, because this is what a majority of software design decisions are going to be based on. DDD defines a set of concepts that are selected to be implemented in software, and then represented in code and any other software artifact used to construct the final system.

Working with a model always happens within a context. It can vary between different requirements or just be derived, for example, from the set of end users of the final system. The chosen context relates to the concepts of the model in a defined way. In DDD, this is called the bounded context (BC). Every domain model lives in precisely one BC, and a BC contains precisely one domain model. A BC helps to model and define interactions between the BC and the model in many different ways. The ultimate mapping for the model is the inside view of the one related BC.

Assuming we already have a layered application approach (e.g., presentation, application, domain, infrastructure), DDD acts on the domain layer. While the application layer mostly acts as a mediator between presentation, domain, and infrastructure (and holds additional crosscutting concerns, such as security and transactions), the domain layer only contains the business objects. This includes the value objects themselves and all related artifacts (e.g., property files, translations) and the module structure, which typically is expressed in packages (e.g., in Java) or namespaces.

Entities, values, and modules are the core building blocks, but DDD also has some additional features that will help you to model your application so that you can build it from domain services. A domain service corresponds to business logic that does not easily live within an entity or it can act as a proxy to another BC. While a domain service can both call or be called by a domain entity, an application service sits above the domain layer, so it cannot be called by entities

within the domain layer, only the other way around. Put another way, the application layer (of a layered architecture) can be thought of as a set of (stateless) application services (Figure 3-2).

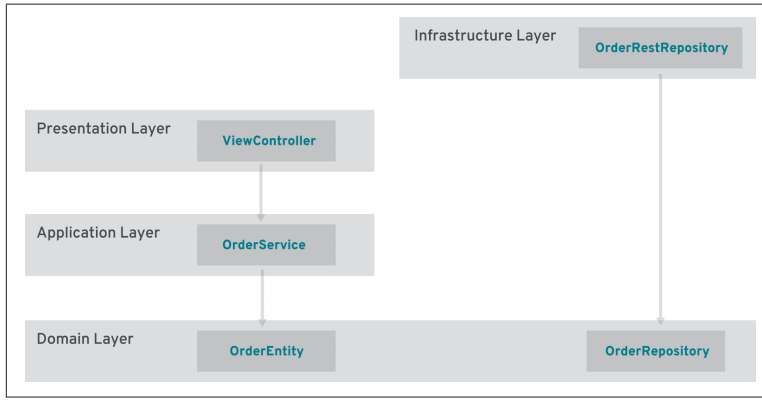

Figure 3-2. Order management BC with layers

What kind of approach should we take if we want to start building a microservices-based application? A greenfield solution isn't the only way to gain the benefits of a microservices-based architecture. Even in existing systems, it might be valuable to identify the critical parts that could perform better or scale easier by refactoring them into separate services. Most mature enterprise solutions actually lack documentation and aren't designed according to DDD. In this case, looking at some very basic characteristics will allow you to make a first assessment of candidates without redesigning everything and starting with a refactoring.

Service Characteristics

For a first migration assessment, it is helpful to identify and separate the services into categories by looking at some key characteristics. It is recommended to only use them in a first round of qualification for a potential microservices migration and not as a design or refactoring methodology. Let's discuss the most important ones in the following subsections.

Core Services

Core services follow the definition of domain services and expose a specific domain entity, including all relevant base operations, directly to a consumer. If you don't have a domain model, you can watch out for entities named after nouns. Another good starting point is a use case or user story. You can even find a lot of examples from common business processes, such as:

- Order
- Shipping
- Catalog
- User
- Customer

Process Services

Process services follow the business services definition and are responsible for performing a single, complex task. They usually represent a business action or process related to and relying on one or more core services. Finding the right partition without a domain model is time consuming and needs to be thought through before implementing. Try to keep the focus on the different business capabilities of a system. Respect the already-known drawbacks from traditional architectures, and keep the network latency and number of hops in mind. It might be easier to verbalize a process service by putting its mission statement up front, such as the following:

- This service lists similar courses for a given course.
- This service places an order for a customer.
- This service reroutes a shipment.
- This service logs an order step for a customer.

If the work of a first assessment is done, you also want to see how much of the existing application already adheres to the basic requirements for building a microservices architecture.

Microservices Best Practices

The following principles have emerged as best practices for developing, and working with, microservices-based architecture. These principles are helpful during the initial assessment and serve as a checklist for your greenfield project.

Design for Automation

Continuous delivery (CD) is a software engineering approach where teams produce usable software in short cycles while ensuring that they can be reliably released at any time. It is used in software development to automate and improve the process of software delivery. CD is a complex and comprehensive enough topic to take up volumes and not just a few paragraphs. However, the idea behind continuous delivery provides the mechanism by which the innovation cycle for microservices-based applications can operate. The principle of continuous delivery that is most relevant here is the ability to deploy rapidly into production, shortening the cycle time between an idea and feedback on the value of the idea.

Achieving rapid deployment requires many continuous delivery techniques, including infrastructure automation, build automation, deployment and rollback automation, data migration automation, and (of course) test automation. Each of these techniques is necessary to support the rapid development of new features, rapid testing of the new system, safe and rapid deployment of the application into production, and safe and rapid rollback in case the system isn't working as expected or if the feature turns out to be a bad idea.

Design for Failure

The premium standard for high availability is five 9s, which stands for a guaranteed uptime of 99.999%. Over the course of a complete year, that means just five and a half minutes of downtime. Traditional approaches often use the words "reliability" and "preventing failure" interchangeably. But cloud-based microservices architectures are completely different.

With applications composed of a large number of individual services, you have to deal with an exponentially growing complexity that touches all relevant parts of an application in order to measure availability and design for failure. And complexity due to more interde-

pendencies is just one way to look at it. Most important is the unknown user behavior that won't let you classify a demand until the complete application is live.

The goal for everything you design around failure tolerance is to minimize human intervention. Implementing automatic failure routines has to be part of every service call that is happening. Looking back at the usability metrics and acceptable response times, it is incredibly beneficial to always fail sooner than later. But what can be done with a failed service? And how do you still produce a meaningful response to the incoming request?

Service load balancing and automatic scaling

A first line of defense is load balancing based on service-level agreements (SLAs). Every microservice needs a defined set of metadata that allows you to find out more information about utilization and average response times. Depending on thresholds, services should be scaled automatically, either horizontally (add more physical machines) or vertically (add more running software instances to one machine).

At the time of writing, this is a commodity feature of most known cloud platforms with respect to applications. Scaling based on individual SLAs and metrics for microservices will be implemented soon enough with orchestration layers like Kubernetes. Until then, you will have to build your own set of metainformation and scaling automations.

The easiest part in all of this is to fail fast and detect those failures early. To mark services as failing, you need to keep track of invocation numbers and invent a way to retry a reasonable number of times until you decide to completely dismiss a service instance for future calls. There are four patterns that will help you to implement the desired behavior of services:

Retry on failure
: This pattern enables the application to handle anticipated, temporary failures when it attempts to connect to a service by transparently retrying an operation that has previously failed in the expectation that the cause of the failure is transient. You may implement the retry pattern with or without a dynamic and configurable number of retries or just stick to a fixed number based on service metadata. The retries can be implemented as

synchronous, blocking, or asynchronous nonblocking, and there are a couple of libraries available to help you with the implementation.

Working with messages and a messaging system makes retry on failure a little easier. The relevant metadata for services can be interpreted by the queues or the event bus and reacted upon accordingly. In the case of a persistent failure, the messages will end up in a compensating service or a dead-letter endpoint. Either way, the messaging or event bus-driven solution will be easier to integrate and handle in most enterprise environments because of the available experience in messaging.

Circuit breaker
: The circuit breaker handles faults that may take a variable time to connect to a remote service. It acts as a proxy for operations that are at risk to fail. The proxy monitors the number of recent failures, and then uses this information to decide whether to allow the operation to proceed or simply return an exception immediately. It was first popularized by Michal Nygard in his 2007 book *Release It!* and you get an excellent overview by Martin Fowler in his "CircuitBreaker" post.

Bulkheads
: As bulkheads prevent a ship from going down in real life, the name stands for partitioning your system and making it failure-proof. If this is done correctly, you can confine errors to one area as opposed to taking the entire system down. Partitions can be completly different things, ranging from hardware redundancy, to processes bound to certain CPUs, to segmentation of dedicated functionality to different server clusters.

Timeouts
: Unlike endlessly waiting for a resource to serve a request, a dedicated timeout leads to signaling a failure early. This is a very simplistic form of the retry or circuit breaker and may be used in situations when talking to more low-level services.

Design for Data Separation

Consider a traditional monolithic application that stores data in a single relational database. Every part of the application accesses the same domain objects, and you don't usually experience problems

around transactions or data separation. Data seperation is different with microservices. If two or more services operate on the same data store, you will run into consistency issues. There are potential ways around this (e.g., transactions), but it is generally considered an antipattern.

So, the first approach is to make all of the systems independent. This is a common approach with microservices because it enables decoupled services. But you will have to implement the code that makes the underlying data consistent. This includes handling of race conditions, failures, and consistency guarantees of the various data stores for each service. This will be easier while you're looking at domain services, and becomes harder and more complex with growing dependencies to other services. You will need to explicitly design for integrity.

Design for Integrity

While data for each service is kept fully separate, services can be kept in a consistent state with compensating transactions. The rule of thumb should be that one service is exactly related to one transaction. This is only a viable solution while all services which persist data are up and running and available. If this isn't the case, you can still completely fail the calling service cascade and rollback earlier calls with compensation transactions, but the end result is eventual consistency without any guarantees. This might not be enough for enterprise systems. The following subsections discuss several different approaches you can use to solve this issue.

Use transactions

It is a common misunderstanding that microservices-based architectures can't have or use transactions at all. There are plenty of ways to use atomic or extended transactions with different technologies that consider themselves part of the modern software stack. Examples of technologies range from server-supported transaction managers, to OMG's Additional Structuring Mechanisms for the OTS and WS-Transactions from OASIS, to even vendor-specific solutions like REST-AT. Implementing equivalent capabilities in your infrastructure or the services themselves (e.g., consistency in the presence of arbitrary failures, opaque recovery for services, modular structuring mechanisms, and spanning different communication patterns) is something you should consider very carefully.

Separate reads from writes

If you don't want to look into transactions first thing, you might want to reduce the complexity by just separating read-only services from write-only services. Given that a significant portion of services will only read the underlying domain objects instead of modifying them, it will be easier to separate services by this attribute to reduce the number of compensation actions you might have to take.

Event-driven design

Another approach to transactions is the event-driven design of services. This requires some logic to record all writes of all services as a sequence of events. By registering and consuming this event series, multiple services can react to the ordered stream of events and do something useful with it. The consuming services must be responsible and able to read the events at their own speed and availability. This includes a tracking of the events to be able to restart consumption after a particular service goes down. With the complete write history as an events database, it would also be possible to add new services at a later stage and let them work through all the recorded events to add their own useful business logic.

Use transaction IDs

Another variant is to correlate combined service calls with transaction IDs. By adding a transaction ID into the payload, the subsequent service calls are able to identify long-running transactional requests. Until all services successfully pass all contained transactions, the data modification is only flagged and a second (asynchronous) service call is needed to let all contributing services know about the successful outcome. As this significantly raises the number of requests in a system, it is only a solution for very rare and complex cases that need full consistency while the majority of services can run without it.

Note: All of the preceding solutions lead to different levels of consistency and there might be even more ways of working around two-phase-commit/XA (eXtended Architecture) transactions (e.g., correlation IDs or a reactive system design), but all of them influence the most critical part of the system, which is overall performance.

Design for Performance

Performance is the most critical part of all enterprise applications. Even if it is the most underspecified, nonfunctional requirement of all, it is still the most complained about.

Microservices-based architectures can significantly impact performance in both directions. First of all, the more coarse-grained services lead to a lot more service calls. Depending on the business logic and service size, this effect is known to fan out a single service call to up to 6 to 10 individual backend-service calls, which only adds the same amount of additional network latency in the case of a synchronous service. The strategies to control this issue are plenty and vary depending on many factors.

Load-test early, load-test often

Performance testing is an essential part of distributed applications. This is even more important with new architectures. You need to make sure that the performance of the complete system is actively tested and individual services perform as they've been tested in development already.

This is equally important as actual runtime monitoring. But the biggest difference is that load testing is a proactive way to verify the initial metainformation of an individual service or group of services. It is also a way to identify and define the initial SLAs. Whereas most articles and books on microservices don't stress this part explicitly, load testing is especially important in enterprises to help with the mind shift needed for this new kind of application architecture and operational model.

Use the right technologies for the job

The usual approach is to base all your endpoints on RESTful calls. As a matter of fact, this might not be the only feasible solution for your requirements. The often-preached, one-to-one relationship between HTTP-based RESTful services and microservices architectures isn't cast in stone. Everything about endpoint technologies, interface architecture, and protocols can be put to the test in enterprise environments.

Some services will be better off communicating via synchronous or asynchronous messaging, but others will be ideally implemented using RESTful endpoints communicating over HTTP. There may

even be some rare instances that require the use of even more low-level service interfaces based on older remoting technologies. The performance of the whole shouldn't be sacrificed just to be buzzword compatible. Further on, it might be valid to test different scenarios and interface technology stacks for optimal performance.

Use API gateways and load balancers

Another important aspect is API versioning and management. As we don't have to control a complete monolithic deployment anymore, it is even more attractive to use explicit versioning on the service APIs and endpoints. There are different API management solutions out there, and these come with all kinds of complexity ranging from simple frameworks and best practices to complete products, which have to be deployed as part of the product.

When you are going to use RESTful services, you have to use an API gateway at minimum. It will help you to keep track of various aspects of your interfaces. Most importantly, they allow you to dispatch based on service versions, and most of them offer load-balancing features.

Besides monitoring, versioning, and load balancing, it is also important to keep track of the individual number of calls per service and version. This is the first step to actually acquiring a complete SLA overview and also tracking down issues with service usage and bottlenecks. Outside performance-relevant topics, API gateways and management solutions offer a broad range of additional features, including increased governance and security.

Use caches at the right layer

Caching is the most important and performance-relevant part of microservices architectures. There are basically two different kinds of data in applications: the type that can be heavily cached, and the type that should never be cached. The latter is represented by constantly refreshing data streams (e.g., stock information) or by secure, personalized, or critical information (e.g., credit card or medical data). Everything else can be heavily cached on different levels.

The UI aspects of a microservice can actually take advantage of the high-performance web technologies already available, such as edge caches, content delivery networks (CDN), or simpler HTTP proxies. All of these solutions rely on the cache expiry settings negotiated

between the server and the client. A different layer of caching technology comes in at the backend. The easiest case is to use a second-level cache with a JPA provider or a dedicated in-memory datastore as a caching layer for your domain entities. The biggest issue is maintaining consistency between cache replicas and between the cache and the backend data source. The best approach here is to use an existing implementation such as JBoss Infinispan.

Independently Deployable and Fully Contained

A microservices architecture will make it easier to scale development. With this technology, there is no large team of developers responsible for a large set of features or individual layers of the monolith. However, with constantly shifting team setups and responsibilities for developers comes another requirement: services need to be independently deployable.

Teams are fully responsible for everything from implementation to commissioning and this requires that they are in full control of the individual services they are touching. Another advantage is that this design pattern supports fault isolation. If every service ideally comes with its own runtime, there is no chance a memory leak in one service can affect other services.

Crosscutting Concerns

Crosscutting concerns typically represent key areas of your software design that do not relate to a specific layer in your application. In a domain-driven design approach, this shouldn't happen. But you really want crosscutting concerns to be reusable non-domain-related concerns that aren't scattered around the whole project. This is where design concepts like dependency injection (DI) and aspect-oriented programming (AOP) can be used to complement object-oriented design principles to minimize tight coupling, enhance modularity, and better manage the crosscutting concerns.

Security

Security in microservices applications breaks down into three different levels (Figure 3-3).

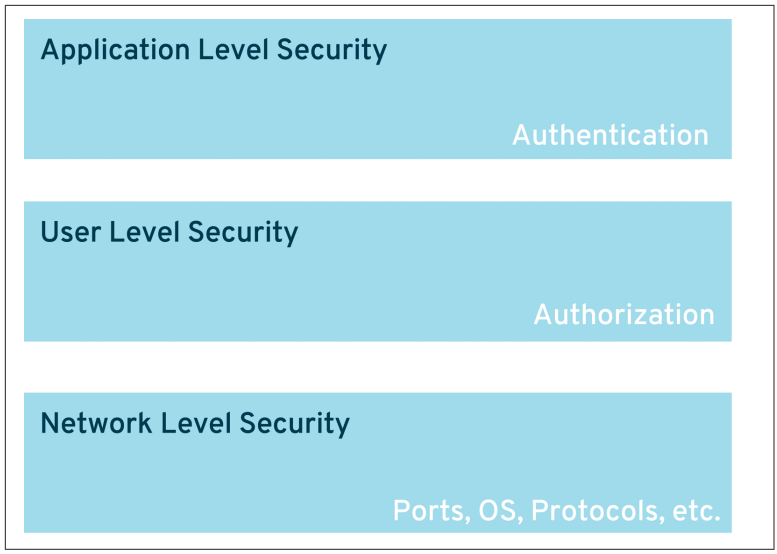

Figure 3-3. Security layers for microservices

First, is *application-level security*. Think about an authorized user who belongs to a role and has to access the entry point of the system. The application will present a login service that a user has to access. Based on the outcome of this service, if it is a username/password, a client-cert, a two-factor authentication, or a social media login, a "credential" has to be issued, which will be used further downstream and passed to all of the involved components in this user session.

The described mechanism is well known and works perfectly fine in monolithic applications. Keeping a credential accessible application-wide makes it easy to resolve application-level security authorizations when needed. This is completely different in a heavily distributed application that is composed of individual services, eventually running on different containers and implemented in different programming languages.

Although there is still a lack of suitable security standards to overcome this problem, there are different options to implement. Possible solutions range from custom tokens that get passed on with every request to developing industry standards like OAuth2 to infrastructure-based enterprise access management (EAM) solutions. A distributed identity needs to be made available to all services.

The next level is the *user-level security*. It maps the credential or token in a downstream service to a user (and/or his personal information) and gathers the relevant service-specific roles. There are basically two different approaches here: pack the needed downstream information into the credential/token, or gather it when needed.

Both methods have pros and cons. For example, packing the information into the request may lead to additional payload and/or marshalling/unmarshalling times. On the other hand, gathering the required information with every service invocation will add additional loading times to every service because security information can't be cached. Finding the correct approach is highly specific to the business requirement of the individual services and depends on a number of factors. These include the time needed to retrieve the user information, the additional payload size, the complete number of downstream (chained) service requests, and potentially more.

Last but not least, there is *network-level security*. Network-level security is typically the most important layer in enterprise scenarios. With penetration tests and other related security scans of applications, you have to implement a solution that doesn't allow malicious requests to even reach your service endpoints without prior accreditation. Additionally, it might be feasible to also train an application security manager (ASM) solution with the allowed and wanted requests for your externally available services.

Logging

Although logging in typical enterprise environments only has to fulfill a few basic needs (such as developer support, debugging in production, and business transaction logging), the nature of a distributed system requires a lot more.

Because one service request can be split out to many different subsequent requests and produce an error somewhere downstream, logging should be able to follow the complete request path down to the error. This might be done with unique service request IDs or even with the help of an HttpSession or SSL session ID (captured at the entry service). And all the distributed logging sources need to be collected in a single application-wide log.

Depending on the existing environment, this can be done with logging frameworks that support syslog or other existing centralized

logging solutions but also built using the ELK (Elasticsearch, Logstash, and Kibana) stack.

Health Checks

Health checks are an important part of DevOps. Every part needs to be controlled and monitored from the very beginning. Besides just having a simple "is-alive" servlet, the need for more sophisticated health checks on a service level arises when using a microservices architecture.

However, there are different ways of approaching this requirement. A simple approach is to select an API management solution that not only deals with governance and load balancing but also handles the SLA and implicit health monitoring of every service. Although this is strongly recommended, there are plenty of other solutions starting from custom implementations up to more complex monitoring approaches.

Integration Testing

While integration testing for Java EE applications has always been important but complex, it is even harder for microservices-based, distributed systems. As usual, testing begins with the so-called module or developer tests. Typically running on a single developer machine, integration tests for distributed systems require the presence of all downstream services. With everything completely automated, this also includes controlling the relevant containers with the dependent services. Although mocking and tests based on intelligent assumptions were best practices a couple of years back, today's systems have to be tested with all the involved services at the correct version.

First and foremost, this requires a decent infrastructure, including complete test and integration systems for the various teams. If you're coming from a traditional enterprise development environment, it might feel odd to exceed the existing five different test stages and corresponding physical machines. But working with microservices and being successful in enterprise settings will require having a PaaS offering, which can spin up needed instances easily and still be cost-effective.

Depending on the delivery model that has been chosen, this might involve building container images and spinning up new instances as

needed. There are very few integration testing frameworks available today that can handle these kind of requirements. The most complete one is Arquillian together with the Cube extension. It can run a complete integration test including image build and spinning up containers as needed as developer local instances or even using remote PaaS offerings. The traditional test concepts and plans as executed by enterprise-grade developments have to step back a bit at this point in favor of more agile and DevOps-related approaches.

CHAPTER 4
Java EE and Microservices

Java EE began with less than 10 individual specifications, but it has grown over time through subsequent updates and releases to encompass 34. Compared to microservices-based architectures, Java EE and its included specifications were originally designed for a different development and deployment model. Only one monolithic server runtime or cluster hosted many different applications packaged according to standards. Such a model runs opposite to the goal of microservices.

Most Java EE APIs are synchronous, and scaling these resources is done through thread pools. Of course, this has its limits and is not meant to be adjusted for quickly changing requirements or excessive load situations. Given these requirements, it appears as if Java EE isn't the best choice for developing microservices-based architectures.

But the latest versions of Java EE added a ton of developer productivity to the platform alongside a streamlined package. With the sloping modularity of Java and the JVM, an established platform, and skilled developers alongside individual implementations, Java EE is considered to be a reasonable solution for microservices development.

Matching the Real World

The latest available Java EE specification as of the writing of this report is Java EE 7. It contains 34 individual specifications, as shown in Figure 4-1.

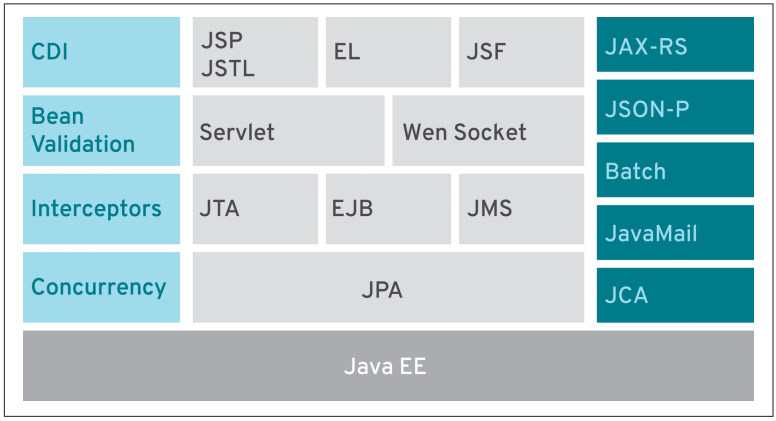

Figure 4-1. Java EE 7 at a glance

There are a lot of technologies that barely offer any advantages to microservices-based architectures, such as the Java Connector Architecture or the Batch Processing API. If you are starting to build microservices architectures on top of Java EE, make sure to look at the asynchronous features and try to use the best available parts. One important item to keep in mind: Java EE was never built to work with distributed applications or microservices. So every decision and line of code should be carefully inspected and validated to maximize asynchronicity.

JAX-RS 2.0

To execute asynchronous requests in JAX-RS, inject a reference to a `javax.ws.rs.container.AsyncResponse` interface in the JAX-RS resource method as a parameter. The `resume` method on the `AsyncResponse` object needs to be called from within a separate thread after the business logic execution is complete, as illustrated here:

```
@Inject
private Executor executor;

        @GET
        public void asyncGet(@Suspended final AsyncResponse
                                asyncResponse) {
            executor.execute(() -> {
            String result = service.timeConsumingOperation();
            asyncResponse.resume(result);
                });
        }
```

WebSocket 1.0

To send asynchronous messages with the WebSocket API, use the `getAsyncRemote` method on the `javax.websocket.Session` interface. This is an instance of the nested interface of the `javax.websocket.RemoteEndpoint`:

```
public void sendAsync(String message, Session session){
    Future<Void> future = session.getAsyncRemote()
                                    .sendText(message);
}
```

Concurrency utilities 1.0

The concurrency utilities for Java EE provide a framework of high-performance threading utilities and thus offer a standard way of accessing low-level asynchronous processing.

Servlet 3.1

The servlet specification also allows the use of asynchronous request processing. The parts that need to be implemented are thread pools (using the `ExecutorService`), `AsyncContext`, the runnable instance of work, and a `Filter` to mark the complete processing chain as asynchronous.

Enterprise JavaBeans (EJB) 3.2

Java EE 6 already introduced the `javax.ejb.Asynchronous` annotation. This is placed on a Stateless, Stateful, or Singleton EJB to make all the contained methods asynchronous or placed at the method level itself. A method with the `@Asynchronous` annotation can either return `void` (fire and forget) or an instance of `java.util.concurrent.Future` if the asynchronous method result

needs to be tracked. This is done by calling the `Future.get()` method.

The Missing Pieces

To build a complete and reliable microservices application, you need something more than what typical Java EE servers provide today. Those missing pieces are relevant, and you only have the chance to build them individually or put an infrastructure in place to deliver them. Those pieces are also called "NoOps" or the outer architecture; see Figure 4-2.

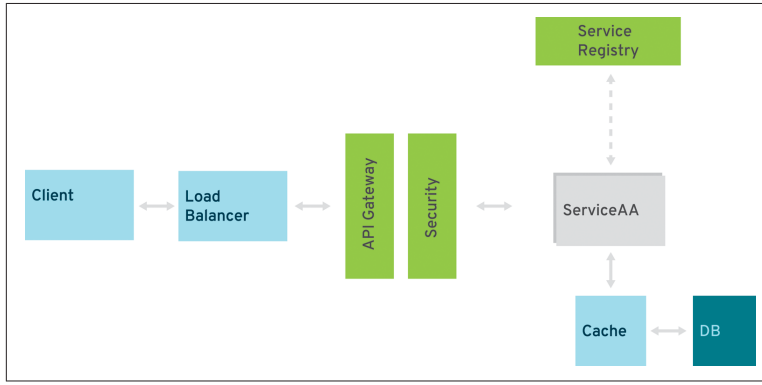

Figure 4-2. Outer architecture for microservices

API Gateway/Management Solution

See "Microservices Best Practices" on page 19 in Chapter 3 for further information about this solution. API gateway/management is an integral part of any microservices infrastructure.

Service Registry

Multiple microservices are composed to create an application, and each microservice can scale independently. The endpoint of the service may not be known until it's deployed, especially if it's deployed in a PaaS. Service registration allows each microservice to register itself with a registry using a logical name. This name is bound to a physical URI and additional metainformation.

By using the logical name, a consumer can locate and invoke the microservice after a simple registry query. If the microservice goes

down, then the consumers are notified accordingly or alternative services get returned. The registry should work closely together with the API gateway. There are multiple tools used for service registry and discovery, such as Apache ZooKeper, Consult, etcd, or JBoss APIMan.

Security

In a traditional multitiered server architecture, a server-side web tier deals with authenticating the user by calling out to a relational database or a Lightweight Directory Access Protocol (LDAP) server. An HTTP session is then created containing the required authentication and user details. The security context is propagated between the tiers within the application server so there's no need to reauthenticate the user.

This is different with microservices because you don't want to let this expensive operation occur in every single microservices request over and over again. Having a central component that authenticates a user and propagates a token containing the relevant information downstream is unavoidable. Enterprise access management (EAM) systems mostly provide the needed features in an enterprise environment. In addition, some API management solutions also contain security features on top of their government engine. And last but not least, there are dedicated products, like JBoss Keycloak.

Migration Approaches

Putting the discussion in Chapter 3 about greenfield versus brownfield development into practice, there are three different approaches to migrating existing applications to microservices.

Selective Improvements

The most risk-free approach is using selective improvements (Figure 4-3). After the initial assessment, you know exactly which parts of the existing application can take advantage of a microservices architecture. By scraping out those parts into one or more services and adding the necessary glue to the original application, you're able to scale out the microservices in multiple steps:

- First, as a separate deployment in the same application server cluster or instance

- Second, on a separately scaled instance
- And finally, using a new deployment and scaling approach by switching to a "fat JAR" container

There are many advantages to this approach. While doing archaeology on the existing system, you'll receive a very good overview about the parts that would make for ideal candidates. And while moving out individual services one at a time, the team has a fair chance to adapt to the new development methodology and make its first experience with the technology stack a positive one.

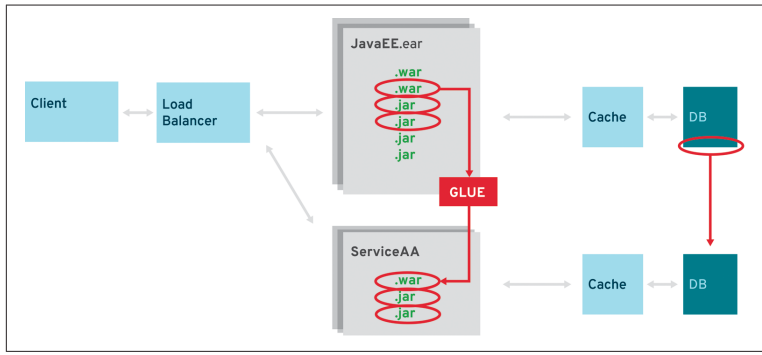

Figure 4-3. Selective improvements

The Strangler Pattern

Comparable but not equal is the second approach where you run two different systems in parallel (Figure 4-4). First coined by Martin Fowler as the StranglerApplication, the refactor/extraction candidates move into a complete new technology stack, and the existing parts of the applications remain untouched. A load balancer or proxy decides which requests need to reach the original application and which go to the new parts. There are some synchronization issues between the two stacks. Most importantly, the existing application can't be allowed to change the microservices' databases.

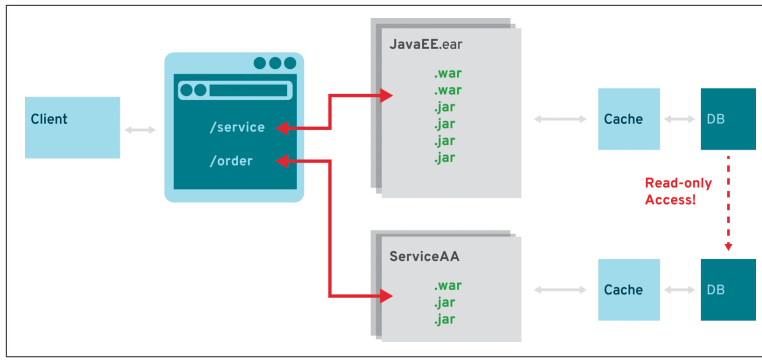

Figure 4-4. Parallel operations: strangler pattern

Big Bang: Refactor an Existing System

In very rare cases, complete refactoring of the original application might be the right way to go. It's rare because enterprise applications will need ongoing maintenance during the complete refactoring. What's more, there won't be enough time to make a complete stop for a couple of weeks—or even months, depending on the size of the application—to rebuild it on a new stack. This is the least recommended approach because it carries a comparably high risk of failure.

CHAPTER 5

Microservices Design Pattern

Functional decomposition of an application with the help of DDD is a prerequisite for building a microservices architecture. Only this approach allows you to effectively design for loose coupling and high cohesion. Even if you go with the much simpler service characteristics, you'll still be able to decompose already existing applications. However, unlike with applications, which are tied together by the frontend, microservices can interact with each other and span a network of service calls. To keep the variety of interactions comprehensible and maintainable, a first set of patterns have emerged that will help you to model the service interaction. These patterns were first published by Arun Gupta, but have been revised for this report.

Common Principles

Every microservice has some common basic principles that need to be taken into account. They are derived from a quick recap of "Service Characteristics" on page 17 and "Microservices Best Practices" on page 19.

To Avoid Trunk Conflict, Each Microservice Is Its Own Build

Conduct a separate build for each microservice. One reason for this is that teams can be fully responsible for putting new versions into production. It also enables the team to use the needed downstream services at the correct revision by querying the repository. Compare "Independently Deployable and Fully Contained" on page 26.

The Business Logic Is Stateless

Treat the logic in your services as stateless. Needing to replicate state across various services is a strong indicator of a bad design. Services are fully contained and independent and should be able to work without any prepopulated state. Compare Chapter 3.

The Data Access Layer Is Cached

In order to keep service response times to a minimum, you should consider data caching in every service you build. And keep in mind "Design for Performance" on page 24.

Create a Separate Data Store for Each Microservice

Compare "Design for Integrity" on page 22 and "Design for Data Separation" on page 21.

Aggregator Pattern

The most simplistic pattern used with microservices is the aggregator pattern (Figure 5-1). It is already well known from the Enterprise Integration pattern catalog and has proven to be useful outside microservices architecture. The primary goal of this pattern is to act as a special filter that receives a stream of responses from service calls and identifies or recognizes the responses that are correlated. Once all the responses have been been collected, the aggregator correlates them and publishes a single response to the client for further processing.

In its most basic form, aggregator is a simple, single-page application (e.g., JavaScript, AngularJS) that invokes multiple services to achieve the functionality required by a certain use case. Assuming

all three services in this example are exposing a REST interface, the application simply consumes the data and exposes it to the user. The services in this example should be application services (compare above) and do not require any additional business logic in the frontend. If they represent domain services, they should be called by an application service first and brought into a representable state.

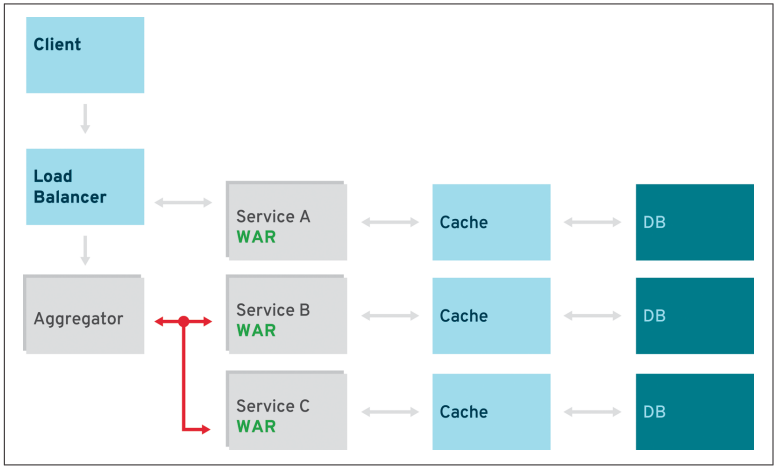

Figure 5-1. Aggregator pattern

The endpoints don't necessarily have to be REST based. It is totally valid to use different protocols. Because the aggregator is another business service heavily accessing asynchronous domain services, it uses a message-driven approach with the relevant protocols on top (e.g., JMS).

Proxy Pattern

The proxy pattern allows you to provide additional interfaces to services by creating a wrapper service as the proxy (Figure 5-2). The wrapper service can add additional functionality to the service of interest without changing its code.

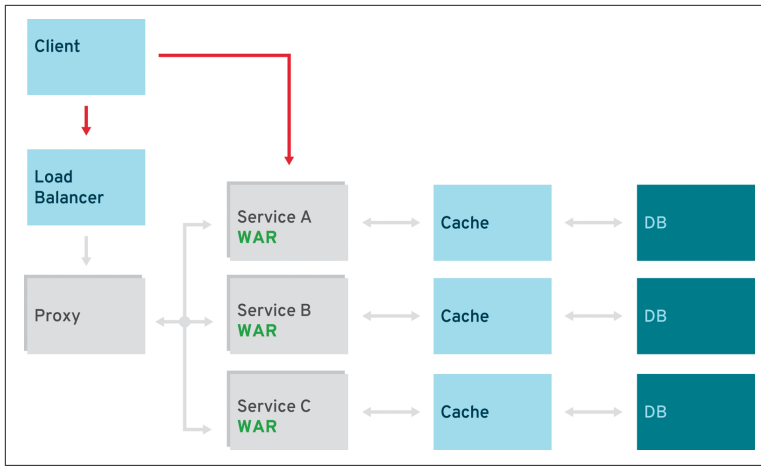

Figure 5-2. Proxy pattern

The proxy may be a simple pass-through proxy, in which case it just delegates the request to one of the proxied services. It is usually called a smart proxy when additional logic is happening inside the proxy service. The applicable logic varies in complexity and can range from simple logging to adding a transaction. If used as a router, it can also proxy requests to different services by parameter or client request.

Pipeline Pattern

In more complex scenarios, a single request triggers a complete series of steps to be executed. In this case, the number of services that have to be called for a single response is larger than one. Using a pipeline of services allows the execution of different operations on the incoming request (Figure 5-3). A pipeline can be triggered synchronously or asynchronously, although the processing steps are most likely synchronous and rely on each other. But if the services are using synchronous requests, the client will have to wait for the last step in the pipeline to be finished.

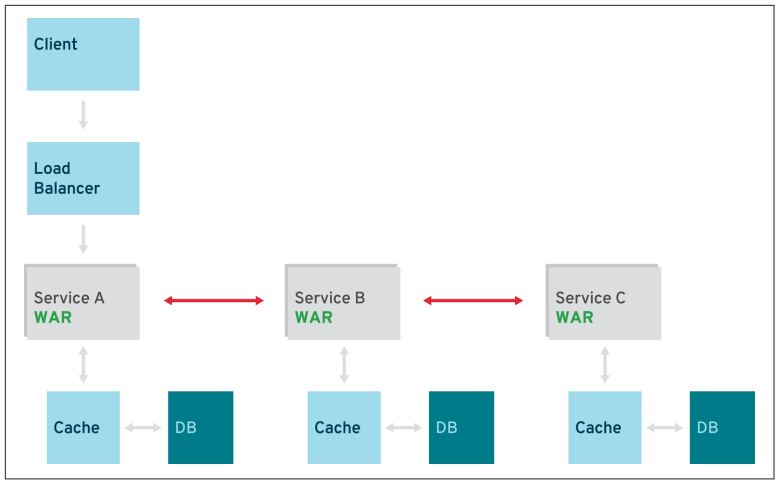

Figure 5-3. Pipeline pattern

Chains shouldn't exceed a certain amount of time if called synchronously. As a general rule of thumb, according to usability studies, one-tenth of a second is about the limit for having the user feel that the system is reacting instantaneously. One second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data. Ten seconds is about the limit for keeping the user's attention focused on the dialogue.

Shared Resources

One of the critical design principles of microservices is autonomy. Especially in migration scenarios (see "Migration Approaches" on page 43), it might be hard to correct design mistakes made a couple of years ago. And instead of reaching for the big bang, there might be a more reasonable way to handle those special cases.

Running into a situation where microservices have to share a common data source isn't ideal. However, it can be worked around with the "shared resources" pattern (Figure 5-4). The key here is to keep the business domain closely related and not to treat this exception as a rule; it may be considered an antipattern but business needs might

require it. With that said, it is certainly an antipattern for greenfield applications.

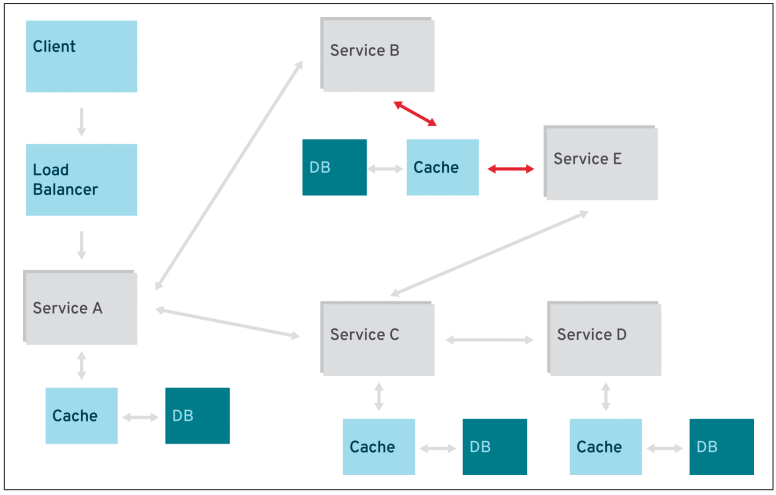

Figure 5-4. Shared resources pattern

Asynchronous Messaging

Typical RESTful design patterns are common in the microservices world. Most likely, they are implemented in a synchronous and therefore blocking manner. Even if this can be changed in Java EE, and the implementations support asynchronous calls, it might still be considered a second-class citizen in the enterprise systems you are trying to build. Message-oriented middleware (MOM) is a more reasonable solution to integration and messaging problems in this field, especially when it comes to microservices that are exposed by host systems and connected via MOMs. A combination of REST request/response and pub/sub messaging may be used to accomplish the business need (Figure 5-5).

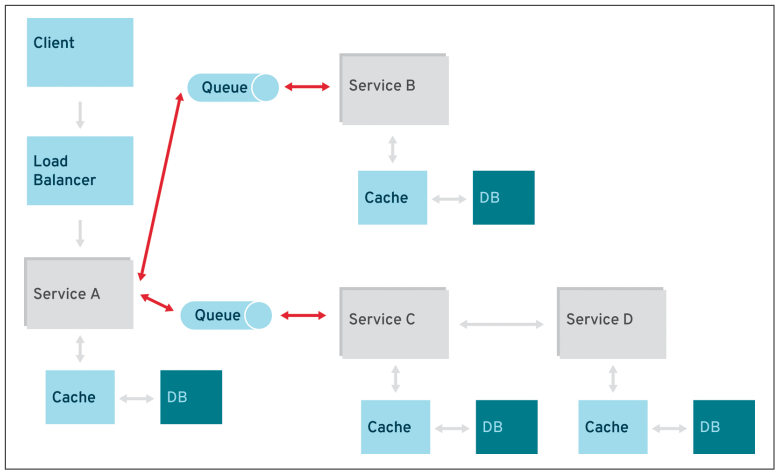

Figure 5-5. Asynchronous messaging

CHAPTER 6
Conclusion

The world of IT as we know it is changing dramatically. Just over five years ago, developers would spend months or even years developing infrastructures and working on the integration of various applications. Huge projects with multiple participants were required to implement the desired specific features.

With the advent of DevOps and various Platform as a Service (PaaS) environments, many complex requirements must now be met within a much shorter timeframe. The Internet of Things (IoT) is also anticipated to change established applications and infrastructures. As a result of these converging trends, the way in which developers work is set to undergo a fundamental shift in the coming years.

As these trends unfold, the industry is already mapping the way forward, anticipating how all the components—from technologies to processes—will come together in this new development paradigm. And all of this will find its way into today's enterprises. While the adoption speed will vary and the pure doctrine of the early adopters will have to be tweaked, there are strong signs that the recent uptake in microservices architectures will not fade. Knowing this, we need to be aware of the challenges to come and figure out how to adapt to these paradigms in practice.

It is a core responsibility for enterprise developers to help further shape this future and keep on learning how to best adopt the new technologies in the field. Appendix B contains a long list of references and recommended readings for getting started with this future. Another excellent publication for learning more about

changing market conditions, customer needs, and emerging technologies as well as how to successfully build software products is the book *Lean Enterprise* (O'Reilly).

APPENDIX A

Additional Technologies and Team Considerations

As already mentioned, software architecture does not adhere to a strict process for creation. However, what it does involve is a lot of teamwork, creativity, and flexibility in adopting changing requirements. This not only covers the design of the system or individual services, but also reaches out to the technologies used and various team dynamics. Unlike with traditional Java EE applications, where the infrastructure is well defined by the application server in use, the solution space for microservices-based systems is open ended and requires a different perspective on teams.

This appendix is designed to point you to alternative microservices solutions outside of the traditional Java EE ecosystem. It also provides greater insight into aligning teams to work with highly scalable architectures.

Architecture != Implementation

Approaches to architectural design do not contain an implicit method for implementation. This is also true for microservices, although the service contracts in a microservices-based architecture allow for a flexible decision about the underlying implementation. It doesn't even have to be on one platform or language.

If you are grounded in Java EE, you've already seen some recommendations and platform-specific thoughts for working with micro-

services. The basic metric used to compile this short list was that Java is the most commonly used programming language in today's enterprises. To keep this a little more to the point, the following products and technologies will give you an overview of Java runtimes that aren't Java EE application server-based for your microservices stack.

Vert.x

Vert.x is an asynchronous, nonblocking framework for development of applications of all kinds. Although it has been mainly discussed in the context of web applications, it has far broader appeal than purely the Web.

Unlike traditional stacks, it's been designed from day one to be scalable and compatible with microservices architectures, so it's almost completely nonblocking when it comes to OS threads. This is the most critical component for microservices-based applications, which naturally have to handle a lot of concurrent processing of messages or events while holding up a lot of connections. Vert.x also supports the usage of a variety of different languages (e.g., JavaScript, Ruby, and Groovy).

This type of functionality can be achieved without being a container or an invasive framework. You can use Vert.x inside your applications and integrate with already existing frameworks such as Spring. The nonblocking nature and reactive programing model speeds along the adoption of basic microservices design principles and recommendations, making this framework easier to use than other platforms. It's also minimally invasive and can be integrated with existing applications, in turn offering an interesting migration path for brownfield developments.

WildFly Swarm

WildFly Swarm is a sidecar project of WildFly 9.x to enable deconstructing the WildFly Java EE application server to your needs. WildFly Swarm allows developers to package just enough of its modules back together with their application to create a self-contained executable JAR.

The typical application development model for a Java EE application is to create an EAR or WAR archive and deploy it to an application server. All the required Java EE dependencies are already available to

the application with the application server base installation, and containers provide additional features like transactions and security. Multimodule applications typically are deployed together on the same instance or cluster and share the same server base libraries.

With Swarm, you are able to freely decide which parts of the application server base libraries your application needs. And only those relevant parts get packaged together with your application into a "fat JAR," which is nothing more than an executable JAR file. After the packaging process, the application can be run using the `java -jar` command.

By designing applications constructed out of many "fat JAR" instances, you can independently upgrade, replace, or scale the individual service instances. This reduces the available amount of specifications and containers for the application to the needed minimum. It also improves the footprint, rollout, and scaling in the final infrastructure while still utilizing the Java EE programing model.

On top of that, it supports the NetflixOSS suite of Ribbon and Hystrix. They make it easy to hide a service behind an interface, find instances of services, and load-balance between them. In the default case, Ribbon uses the Netflix Eureka server to register and discover individual services. With WildFly Swarm, the standard clustering subsystem can be used to locate these services and maintain the lists of endpoints.

Spring Boot with Spring Cloud

Spring Boot is part of the larger Spring ecosystem. It has evolved as a framework especially designed for microservices. It is built on top of the Spring framework and uses the maturity of it while adding additional features to aid the development of microservices-based applications.

Developer productivity is a "first class" citizen, and the framework adds some basic assumptions about how microservices applications should be built. This includes the assumption that all services have RESTful endpoints and are embedded into a standalone web application runtime. The overall Spring methodology to adopt the relevant features and leave out the others is also practiced here. This leads to a very lean approach that can produce small units of deployments that can be used as runnable Java archives.

On top of Spring Boot is Spring Cloud, which provides NetflixOSS integrations for Boot apps through auto-configuration, binding to the Spring Environment, and other Spring programming model idioms. You can enable and configure the common patterns inside your application via Java annotations and build distributed systems while transparently using a set of Netflix OSS components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul), and Client-Side Load Balancing (Ribbon).

Dropwizard

Dropwizard is a Java framework for developing ops-friendly, high-performance, RESTful web services. It pulls together well-known, stable, mature libraries from the Java ecosystem (e.g., Jetty, Jersey, and Jackson) into a "fat JAR." Dropwizard has out-of-the-box support for configuration, application metrics, logging, operational tools, and more. The individual technologies are wired together with the help of various interfaces and annotations that can be viewed as the glue in between. This leaves the user with having to know the individual technologies first, plus the wiring in between them. So, there is a learning curve involved, but not a steep one.

Roll Your Own

Another very common alternative is to roll your own Java EE-like platform on the base of Apache Tomcat. By packaging the relevant and needed modules together, it can be a feasible alternative even if it will require a lot more effort in building the initial stack of frameworks and libraries.

Thoughts About Teams and Cultures

While you can read a lot about how early adopters like Netflix structured their teams for speed instead of efficiency, there is another more reasonable approach for enterprise software development teams. Most basically, it is important to keep them focused. Teams should be aligned around business capabilities and responsibilities. This ensures that the business focus is present and can be reused with every new service that falls into one of the business domains. On the other hand, it is also very important to still have a business consultant as part of a team.

As much as we wish for completely responsible teams, it is highly unlikely that only developers will ever work on the complete applications from the requirement gathering stage through to implementation in an enterprise setting. There will always be a business consultant involved to spend time and energy on asking the right questions to the business owners. The structure of those teams shouldn't be a lot different from what the early adopters invented: the so-called "two-pizza team" definition.

At maximum, this definition suggests four people should be responsible for a business capability. You can scale and coordinate those "two-pizza teams" according to the needs of an enterprise project. The bigger pill to swallow is that the basic assumption of the individual teams has to be "freedom and responsibility." However, most enterprises often rely on controlling and reporting project success and progress. This doesn't align very well with the collaborative team culture that supports microservices-based architectures the best. The only practical way to solve this is to find a good balance between the enterprise's needs for controlling and project management and the independence of the individual teams.

There's a good chance that both can be achieved with a little good will from all involved. Scrum and agile project management practices are well known and mostly applicable. Running the "two-pizza team" approach in an agile fashion shouldn't be new at all. The bigger challenge is extracting the right reporting metrics and mapping them to an overall project plan. But using an iterative approach with only broad planning topics should allow for enough flexibility to make it work (see Figure A-1).

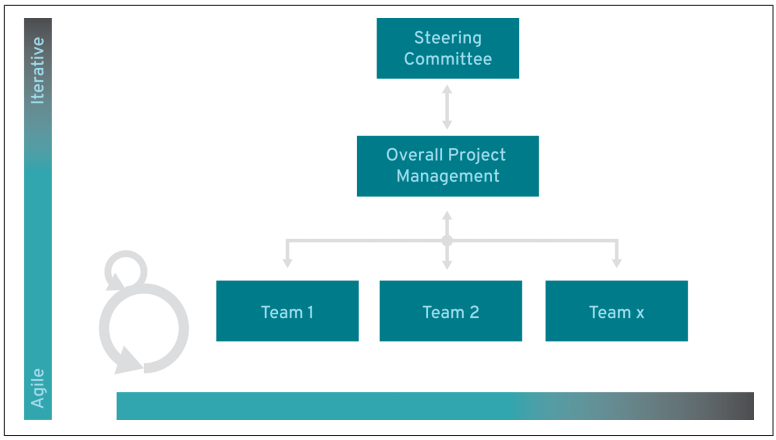

Figure A-1. Mixing agile and iterative

If the teams are in place and management is OK with the reporting structures, you still need to think about all the other silos and departments in a typical enterprise. All those overengineered processes and outdated technologies have to be taken into account when starting to build teams that can work—and act—like the early adopters envisioned. Everyone on the team doesn't have to be a full-stack developer to work with the latest technology. But they do all have to work better with one another, including across teams and within the boundaries of the technologies they use.

APPENDIX B

Further Resources

Each of the following resources will provide additional insight and help you to develop your own perspective on microservices-based architecture:

- Martin Fowler on "Microservices". A gentle introduction and a first approach at a definition.
- Martin Fowler on "MicroservicePremium". Some further explanation on what microservices and enterprises mean.
- "The microservice resource guide" by Martin Fowler. Collecting various articles and discussions along this topic.
- Netflix Open Source Software Center. Contains links and further information on the frameworks and libraries referenced throughout the book.
- Microservices at Netflix: Best Practices and Tools. One of many presentations from the Netflix team that gives a good overview of the company's work.
- "Microservices Design Patterns" by Arun Gupta. The first collection of patterns for the newly emerging architecture style. Arun is also featuring more microservices-related posts on his blog.
- Mark Little, "What is so Special about Microservices?" Deals with how microservices reflect an evolution in our understanding of how to build services.

- Christian Posta, "The Real Success Story of Microservices Architectures".
- Patterns and Best Practices for Enterprise Integration is the bible for system integration and has a complete implementation in Apache Camel. While not primarily focused on microservices, it contains a well-crafted set of patterns to use for integration.
- Microservices in Fabric8. A brief introduction about how to work with microservices in Fabric8.
- Michael Nyguard's *Release It!*.
- Sam Newman's *Building Microservices: Designing Fine-Grained Systems*.
- Mark Little on "Distributed systems theory for the distributed systems engineer". A gentle yet complete introduction to the distributed systems theory for systems engineers.
- Daniel Bryant interviews Randy Shoup on microservices, the reality of Conway's Law, and evolutionary architecture.
- Mark Little, "Transactions and Microservices", plus an updated blog post with additional information.
- Simon Brown asks the question, "If you can't build a monolith, what makes you think microservices are the answer?".
- "Java EE, WildFly and Microservices on Docker". This is a blog post to get you started with Java EE, WildFly, and microservices on Docker.
- Christian Posta, "The Cold Hard Truth About Microservices—vjBug". You've tried all the past hyped technologies and architectures, but those promises have been underdelivering. Can microservices help here?

About the Author

Markus Eisele is a Developer Advocate at Red Hat, and focuses on JBoss Middleware. He has been working with Java EE servers from different vendors for more than 14 years and talks about his favorite topics relating to Java EE at conferences all over the world. He has been a principal consultant and worked with different customers on all kinds of Java EE-related applications and solutions. Outside of this, he is a prolific blogger, writer, and tech editor for Java EE-related books and publications. He is a board member of the German DOAG e.V. and serves as its representative on the iJUG e.V. As a Java Champion and former ACE Director, he is well known in the community. More frequent updates are available on his Twitter feed and blog.