With examples in Java

Microservices Patterns

Chris Richardson

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Microservices Patterns
With examples in Java
Version 11

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Microservices Patterns: With examples in Java*. I am very excited to see the first few chapters be released and I am looking forward to completing the book. This is an intermediate level book designed for enterprise developers and architects who want to adopt the microservice architecture.

I don't believe in blindly advocating for any particular technology. The microservice architecture is not a silver bullet. It has both benefits and drawbacks. Moreover, there are numerous issues that you must address when using the microservice architecture. This book captures this philosophy. Most of the content is organized around patterns, which are a great way to describe the trade-offs of using a particular approach.

I am initially releasing the first two chapters. Chapter 1 envisions the state of software development at Food to Go, Inc., which is the fictitious company from my first book *POJOs in Action*. After ten years they are in what I call monolithic hell. All aspects of software development and deployment are slow and painful. Sadly, the odds are high that you are in a similar situation. In this chapter, you will learn how to escape monolithic hell. I describe the microservice architecture, it's benefits and drawbacks. You will learn about the microservices pattern language, which is a collection of patterns that solve the problems that you face when using the microservice architecture.

Chapter 2 describes the key decision that you must make when using the microservice architecture: how to compose an application into a set of services. You will learn about the important of software architecture. I describe how the microservice architecture is what is known as an architectural style. You will learn about two main decomposition strategies.

Chapter 3 looks at how inter-process communication (IPC) plays much more critical role in a microservice architecture than it does in a monolithic application. You will learn about the various IPC options including messaging and REST.I describe why asynchronous messaging is preferred approach. You will learn how to send messages as part of a database transaction and why it's important.

Looking ahead, later chapters dig deeper into the microservice architecture. I'll describe key architectural issues including inter-process communication and transaction management in the microservice architecture. The latter is especially challenging since each service has its own database and traditional distributed transaction are not a viable option for modern applications. After that I will cover numerous other topics including deployment, testing and monitoring patterns. You will also learn how to refactor an existing monolithic application into a microservice architecture.

As you are reading *Microservices patterns*, I hope you'll take advantage of the Author Online forum. I will be reading your comments and responding. I appreciate any feedback, as it will help me write a better book.

Thanks again!

— Chris Richardson

brief contents

1 Escaping monolithic hell

This chapter covers:

- The symptoms of monolithic hell and how to escape it by adopting the microservice architecture
- The essential characteristics of the microservice architecture and its benefits and drawbacks
- How microservices enable the DevOps-style development of large, complex applications.
- The microservice architecture pattern language and why you should use it

It was only Monday lunchtime but Mary, the CTO of Food to Go, Inc (FTGO) was already feeling frustrated. Her day had started off really well. She had spent the previous week with other software architects and developers at an excellent conference learning about the latest software development techniques including continuous deployment and the microservice architecture. Mary had also met up with her former Computer Science classmates from North Carolina A&T State and shared technology leadership war stories. The conference had left her feeling empowered and eager to improve how FTGO develops software.

Unfortunately, that feeling had quickly evaporated. She had just spent the first morning back in the office in yet another painful meeting with senior engineering and business people. They had spent two hours discussing why the development team was going to miss another critical release date. Sadly, this kind of meeting had becoming increasingly common over the past few years. Despite adopting agile, the pace of development was slowing down, making it next to impossible to meet the business's goals. And, to make matters worse there didn't seem to be a simple solution.

The conference had made Mary realize that FTGO was suffering from a case of monolithic hell and that the cure was to adopt the microservice architecture. However, the microservice architecture and the associated state of the art software development practices described at the conference felt like an elusive dream. It was not clear to Mary how she could fight today's fires while simultaneously improving how software was developed at FTGO. Fortunately, as you will learn by reading this book there is a way. But first, let's look at the problems that FTGO is facing and how they got there.

1.1 The slow march towards monolithic hell

Since its launch in late 2005, FTGO had grown by leaps and bounds. Today, it was one of the leading online food delivery companies in the US. The business even plans to expand overseas although those plans are in jeopardy because of delays in implementing the necessary features.

At its core, the FTGO application is quite simple. Consumers use the FTGO website or mobile application to place food orders at local restaurants. FTGO coordinates a network of couriers who deliver the orders. It is also responsible for paying couriers and restaurants. Restaurants use the FTGO website to edit their menus and manage orders. The application uses various web services including Stripe for payments, Twilio for messaging, and Amazon Simple Email Service (SES) for email.

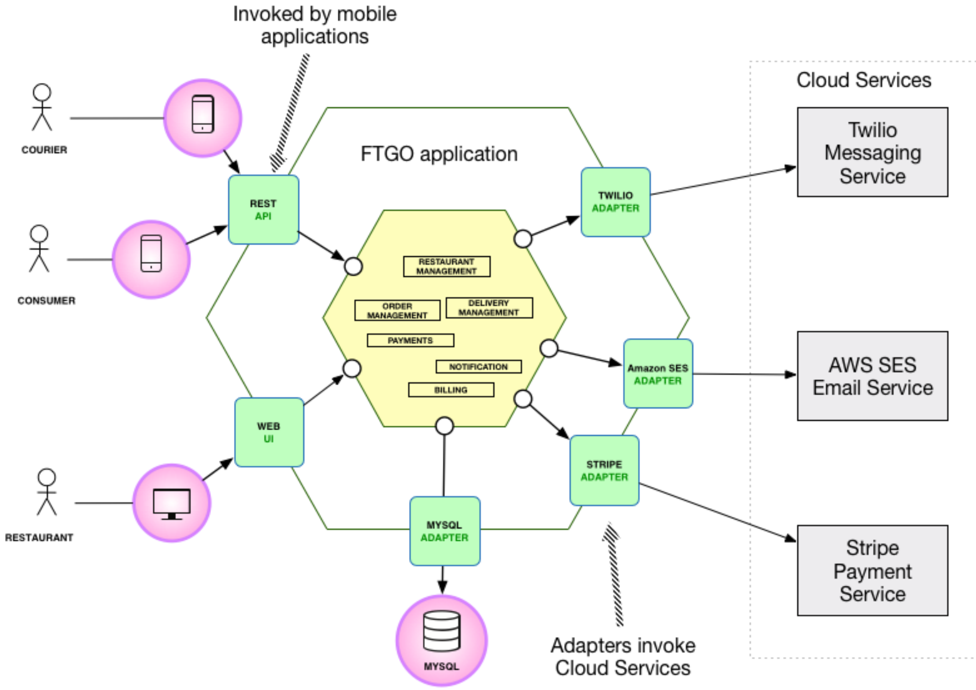
Like many other aging enterprise applications, the FTGO application is a monolith, consisting of a single Java Web Application Archive (WAR) file. Over the years, it has become a very large, very complex application. But despite the best efforts of the FTGO development team, it's become an example of the Big ball of mud pattern [1]. To quote Foote and Yoder, the authors of that pattern, it's a "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle". The pace of software delivery has got slower and slower. And, to make matters worse it's written using some increasingly obsolete frameworks. The FTGO application is exhibiting all the symptoms of monolithic hell.

I begin this section by describing the architecture of the FTGO application. Next, I describe why the monolithic architecture worked well initially. I then describe how the FTGO application has outgrown its architecture and how that's resulted in monolithic hell. Let's take a look at the architecture of the FTGO application.

1.1.1 The architecture of the FTGO application

FTGO is a typical enterprise Java application. Figure 1.1 shows its architecture. The FTGO application has an hexagonal architecture, which is an architectural style that I describe in more detail in chapter 2. In a hexagonal architecture, the core of the application consists of the business logic. Surrounding the business logic are various adapters that implement UIs and integrate with external systems.

Figure 1.1. The FTGO application has a hexagonal architecture. It consists of business logic surrounded by adapters, which implement UIs and interface with external systems, such as mobile applications, and cloud services for payments, messaging and email.



The business logic consists of modules, each of which is a collection of domain objects. Examples of the modules include `Order Management`, `Delivery Management`, `Billing` and `Payments`. There are several adapters that interface with the external systems. Some are inbound adapters, which handle requests by invoking the business logic, include the `REST API` and `Web UI` adapters. Others are outbound adapters, which enable the business logic to access the MySQL database and invoke cloud services, such as Twilio and Stripe.

Despite having a logically modular architecture, the FTGO application is packaged as a single WAR file. The application is an example of the widely used **monolithic** style of software architecture, which structures a system as a single executable or deployable component. If the FTGO application was written in GoLang, it would be a single executable. A Ruby or NodeJS version of the application would be a single directory hierarchy of source code. The monolithic architecture is not inherently bad. The FTGO developers made a good decision when they picked monolithic architecture for their application.

1.1.2 The benefits of the monolithic architecture

In the early days of FTGO, when the application was relatively small, the application's monolithic architecture had lots of benefits:

- simple to develop - IDEs and other developer tools are focused on building a single application.
- easy to make radical changes to the application - change the code and the database schema, build and deploy.
- straightforward to test - the developers wrote end-to-end tests that simply launched the application, invoked the REST API and tested the UI with Selenium.
- straightforward to deploy - all a developer had to do was copy the WAR file to a server that had Tomcat installed.
- easy to scale - FTGO ran multiple instances of the application behind a load balancer.

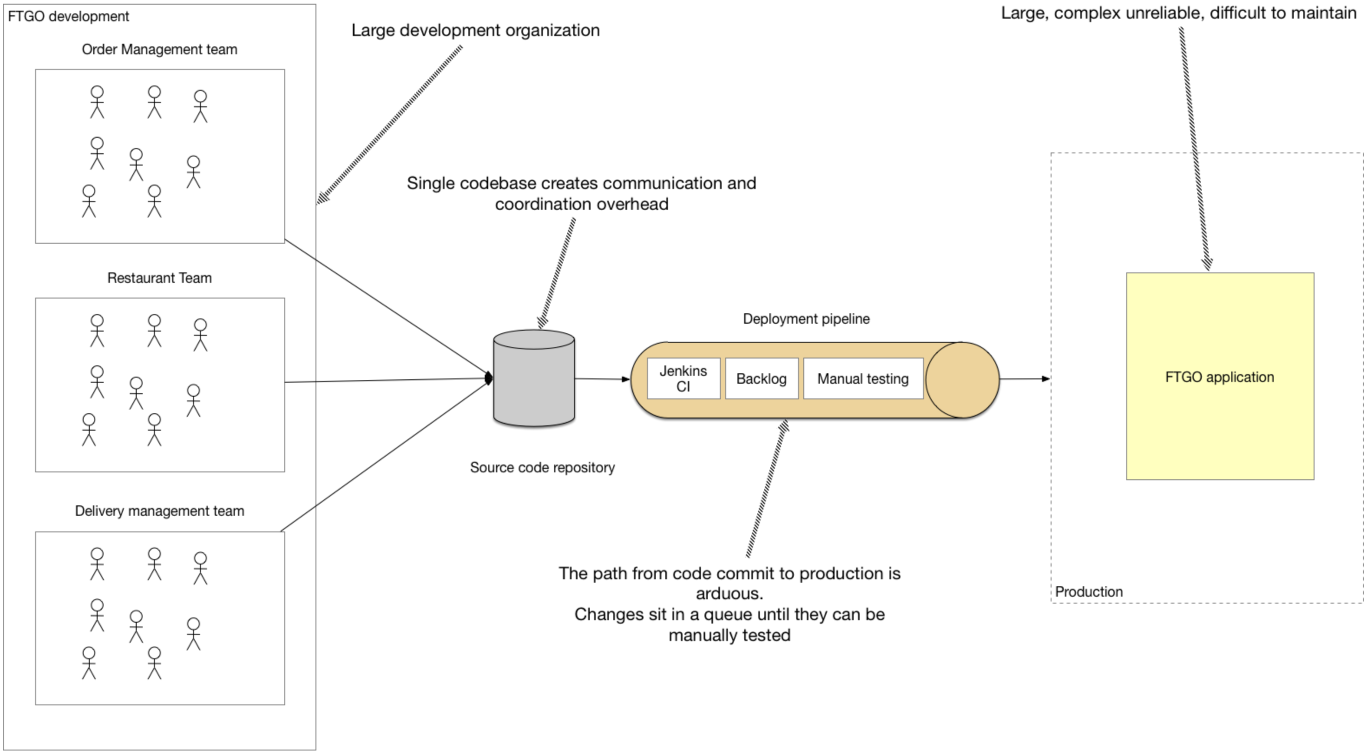
Over time, however, development, testing, deployment and scaling became much more difficult. Let's look at why.

1.1.3 Living in monolithic hell

Unfortunately, as the FTGO developers have discovered, the monolithic architecture has a huge limitation. Successful applications, like the FTGO application, have a habit of outgrowing the monolithic architecture. Each sprint, the FTGO development team implemented a few more stories, which, of course, made the code base larger. Moreover, as the company became more successful, the size of the development team steadily grew. Not only did this increase the growth rate of code base but it also increased the management overhead.

As figure 1.2 shows, the once small, simple FTGO application has grown over the years into a monstrous monolith. Similarly, the small development team has now become multiple scrum teams, each one of which works on a particular functional area. As a result of outgrowing its architecture, FTGO is in monolithic hell. Development is slow and painful. Agile development and deployment is impossible.

Figure 1.2. A case of monolithic hell. The large FTGO developer team commits their changes to a single source code repository. The path from code commit to production is long and arduous and involves manual testing. The FTGO application is large, complex, unreliable, and difficult to maintain.



Lets look at the reasons.

The overwhelming complexity of the monolith intimidates developers

A major problem with the FTGO application is that it is too complex. It is simply too large for any developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. Deadlines are missed.

To make matters worse, this overwhelming complexity tends to be a downwards spiral. If the codebase is difficult to understand then a developer won't make changes correctly. Each change makes the codebase incrementally more complex, and more difficult to understand. The clean, modular architecture shown earlier in figure 1.1 doesn't reflect reality. FTGO is gradually becoming a monstrous, incomprehensible big ball of mud.

Mary remembers recently attending a conference where she met a developer who was writing a tool to analyze the dependencies between the thousands of JARs in their multi-million Lines of Code (LOC) application. At the time, that tool seemed like something FTGO could use. Now she is not so sure. Mary suspects a better approach is to migrate to an architecture that is more suited to a complex application: microservices.

The day to day development of a large, monolithic application is slow

As well having to fight overwhelming complexity, FTGO developers find day to day development tasks slow. The large application overloads and slows down a developer's IDE. Building the FTGO application takes a long time. Moreover, because it is so large, the application takes a long time to startup. As a result, the edit-build-run-test loop takes a long time, which badly impacts productivity.

The path from commit to deployment is long and arduous

Another problem with the FTGO application is that deploying changes into production is a long and painful process. The team typically deploys updates production once a month, usually late Friday or Saturday night. Mary keeps reading that the state of the art for SaaS applications is continuous deployment: deploying changes to production many times a day during business hours. Apparently, as of 2011 Amazon.com deployed a change into production every 11.6 seconds without ever impacting the user! For the FTGO developers, updating production more than once a month seems like a distance dream. And adopting continuous deployment is next to impossible.

FTGO has partially adopted agile. The engineering team is divided up into squads and uses two week sprints. Unfortunately, the journey from code complete to running in production is long and arduous. One problem with so many developers committing to the same code base is that the build is frequently in an unreleasable state. When the FTGO developers attempted to solve this problem by using feature branches, which resulted in lengthy, painful merges. Consequently, once a team completes their sprint, there is a long period of testing and code stabilization.

Another reason it takes so long to get changes into production is that testing takes a long time. Because the code base is so complex and the impact of a change is not well understood, developers and the Continuous Integration (CI) server must run the entire test suite. Some parts of the system even require manual testing. It also takes a while to diagnose and fix the cause of a test failure. As a result, it takes a couple of days to complete a testing cycle.

Scaling a large monolith is challenging because of conflicting resource requirements

The FTGO team also has problems scaling their application. That is because different application modules have conflicting resource requirements. The restaurant data, for example, is stored in a large in-memory database, which is ideally deployed on servers with lots of memory. In contrast, the image processing module is CPU intensive and best deployed on servers with lots of CPU. Since these modules are part of the same application, FTGO must compromise on the server configuration.

Delivering a complex, yet reliable monolith is challenging

Another problem with the FTGO application is the lack of reliability. As a result, there are frequent production outages. One reason it's unreliable is because its difficult to test the application thoroughly due to its size large, This lack of testability means that bugs make their way into production. And, to make matters worse, the application lacks fault isolation because all modules are running within the same process. Every so often, a bug in one module, such as a memory leak, crashes all instances of the application one by one. The FTGO developers don't enjoy being paged in the middle of the night because of a production outage. The business people like the loss of revenue and trust even less.

The monolithic architecture locks you into an increasingly obsolete technology stack

The final aspect of monolithic hell experienced by the FTGO team is that the architecture forces them to use a technology stack that is becoming increasingly obsolete. The monolithic architecture makes it difficult to adopt new frameworks, and languages. It is extremely expensive and risky to rewrite the entire monolithic application to use a new and presumably better technology. Consequently, developers are stuck with the technology choices they made at the start of the project. Quite often, they must maintain an application written using an increasingly obsolete technology stack.

The Spring framework has continued to evolve while being backwards compatible so in theory FTGO might have been able to upgrade. Unfortunately, the FTGO application uses versions of frameworks that are incompatible with newer versions of Spring. The development team has never found the time to upgrade those frameworks. As a result, major parts of the application are written using increasingly out of date frameworks. What's more, the FTGO developers would like to experiment with non-JVM languages such as GoLang and NodeJS. Sadly, this is not possible with a monolithic application.

[1] http://www.laputan.org/mud/

1.2 Why this book is relevant to you

It's likely that you are a developer, architect, CTO, or VP of engineering. You are responsible for an application that has outgrown its monolithic architecture. Like Mary at FTGO, you are struggling struggling with software delivery, and want to know how to escape monolith hell. Or perhaps, you fear that your organization is on the path to monolithic hell. And, you want to know how to change direction before its too late. If you need to escape or avoid monolithic hell, then this is the book for you.

In this book, I spend a lot of time explaining microservice architecture concepts. My goal is for you to find this material accessible regardless of the technology stack that you use. You just need to be familiar with the basics of enterprise application architecture and design. In particular, you need to know:

- 3-tier architecture
- Web application design
- Developing business logic using object-oriented design
- Using an RDBMS: SQL and ACID transactions
- Inter-process communication using a message broker and REST APIs
- Security: authentication and authorization

The code examples, however, in this book are written using Java and the Spring framework. In order to get the most out of the examples, you need to be familiar with the Spring framework.

1.3 What you'll learn in this book

By the time you finish reading this book you will understand:

- The essential characteristics of the microservice architecture, its benefits and drawbacks and when to use it
- Distributed data management patterns
- Effective microservice testing strategies
- Deployment options for microservices
- Strategies for refactoring a monolithic application to a microservice architecture

And, you will be able to:

- Architect an application using the microservice architecture pattern
- Develop the business logic for a service
- Use sagas to maintain data consistency across services
- Implement queries that span services
- Effectively test microservices
- Develop production ready services that are security, configurable and observable
- Refactor an existing monolithic application to services

1.4 The microservice architecture to the rescue

FTGO is in trouble because the application has outgrown its monolithic architecture. The monolithic architecture worked well initially. But as the application grew the development team encountered numerous issues. They have arrived at monolithic hell. All aspects of development are slow and painful. Mary has come to the conclusion that FTGO must migrate to the microservice architecture.

Interestingly, software architecture has very little to do with functional requirements. You can implement a set of use cases - an application's functional requirements - with any architecture. In fact, it's common for a successful applications, such as the FTGO application, to be big ball of mud.

Architecture matters, however, because of how it affects the so-called quality of service requirements, a.k.a. non-functional requirements, quality attributes or "-ilities".

As the FTGO application grew, various quality attributes have suffered, most notably those that impact the velocity of software delivery: maintainability, extensibility, and testability.

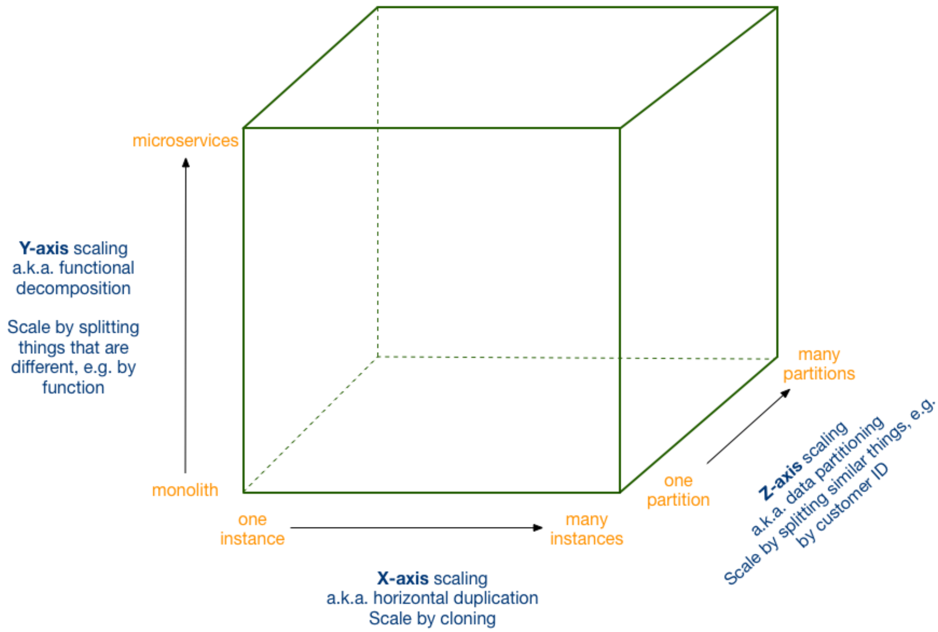
On the one hand, a disciplined team can slow down the pace of their descent towards monolithic hell. They can work hard to maintain the modularity of their application. They can write comprehensive automated tests. But on the other hand, they cannot avoid the issues of a large team working on a single monolithic application. Nor can they solve the problem of an increasingly obsolete technology stack. The best that a team can do is delay the inevitable. In order to escape monolithic hell they must migrate to a new architecture: the Microservice architecture.

Today, there is growing consensus that if you are building a large, complex application then you should consider using the microservice architecture. But what are microservices exactly? Unfortunately, the name doesn't help since it overemphasizes size. There are numerous definitions of the microservice architecture. Some take the name too literally and claim that a service should be tiny, e.g. 100 LOC. Others claim that a service should only take two weeks develop. Adrian Cockcroft, formerly of Netflix, defines a microservice architecture as a service-oriented architecture composed of loosely coupled elements that have bounded contexts. That is not bad a definition but it is a little dense. Lets see if we can do better.

1.4.1 Scale cube and microservices

My definition of the microservice architecture is inspired by the excellent book **The art of scalability**. This book describes a really useful, three dimension scalability model: the scale cube, which is shown in Figure 1.3.

Figure 1.3. The scale cube defines three separate ways to scale an application: X-axis scaling load balancers requests across multiple, identical instances; Z-axis scaling routes requests based on an attribute of the request; Y-axis functionally decomposes an application into services.

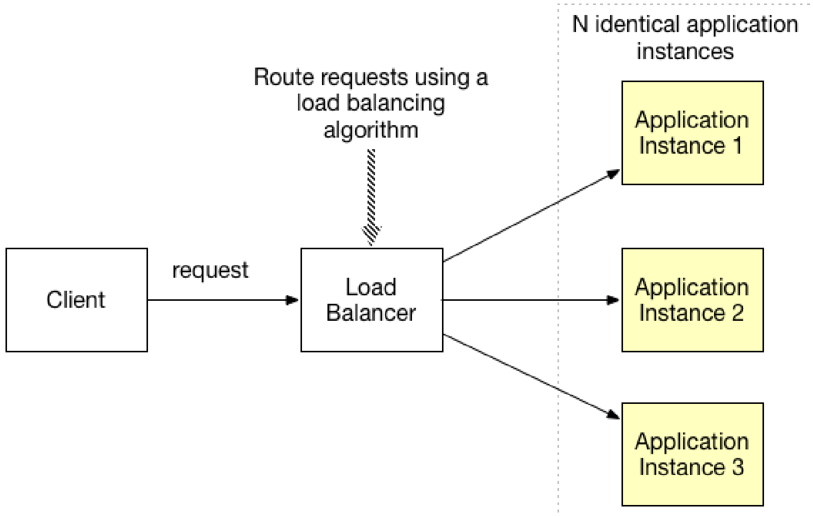


The model defines three ways to scale an application: X, Y and Z.

X-axis scaling load balances requests across multiple instances

X-axis scaling is a commonly used way to scale a monolithic application. Figure 1.4 shows how X-axis scaling works. You simply run multiple instances of the application behind a load balancer.

Figure 1.4. X-axis scaling runs multiple, identical instances of the monolithic application behind a load balancer

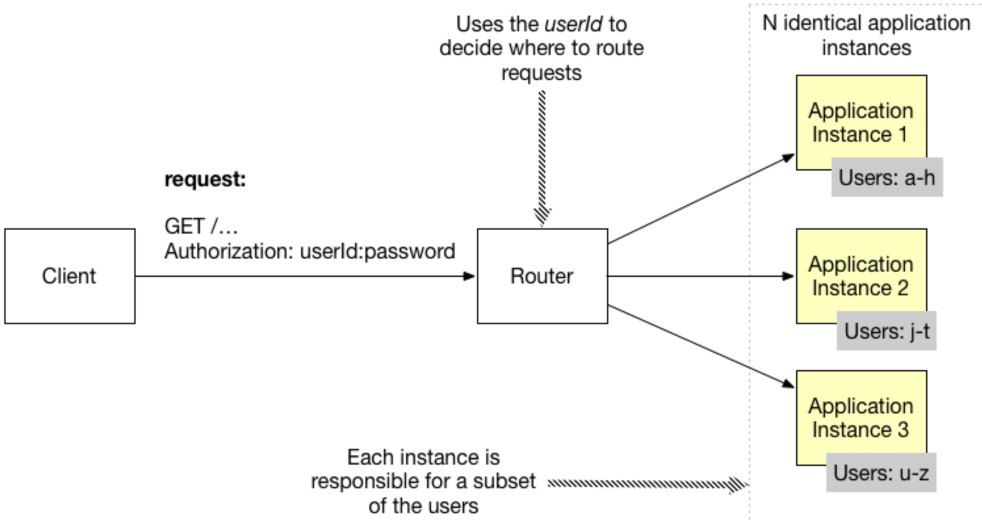


The load balancer distributes requests amongst the N identical instances of the application. This is a great way of improving the capacity and the availability of an application.

Z-axis scaling routes requests based on an attribute of the request

Z-axis scaling also runs multiple instances of the monolith application. However, unlike X-axis scaling, each instance is responsible for only a subset of the data. Figure 1.5 shows how Z-axis scaling works. The router in front of the instances uses an attribute of request to route it to the appropriate instance. An application might, for example, route requests using the *userId*.

Figure 1.5. Z-axis scaling runs multiple identical instances of the monolithic application behind a router, which routes based on a request attribute. Each instance is responsible for a subset of the data.

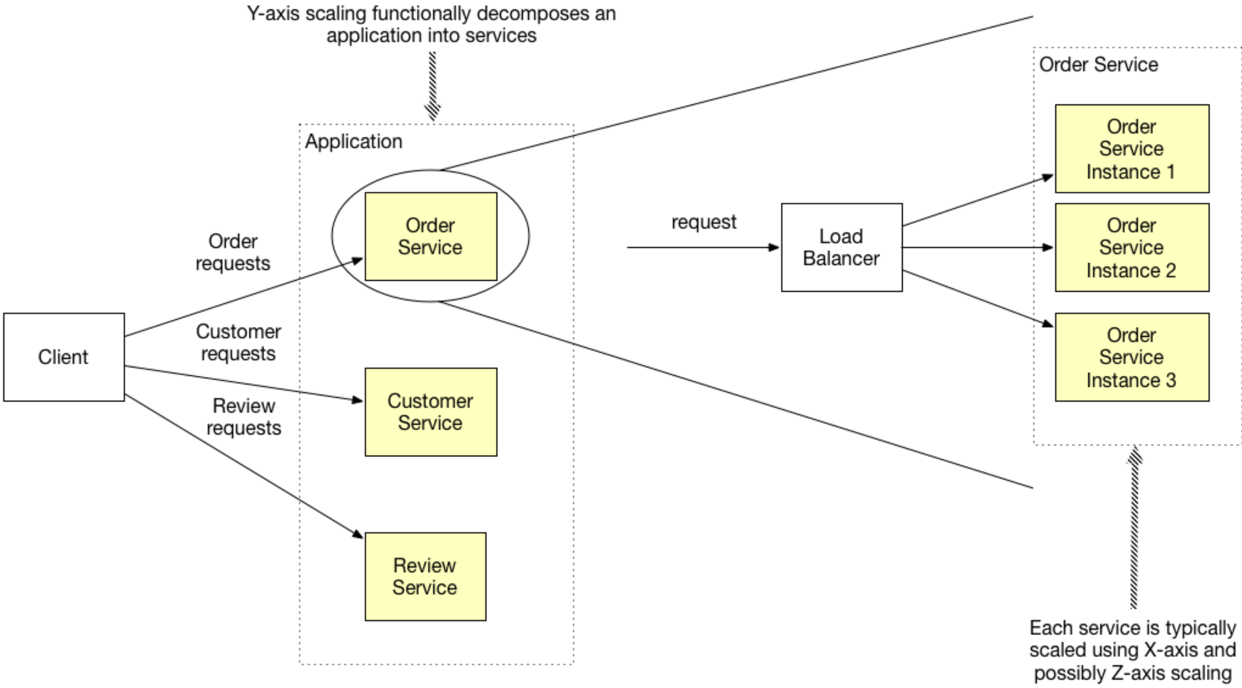


In this example, each application instance is responsible for a subset of the users. The router uses the *userId* specified by the request `Authorization` header to select one of the N identical instances of the application. Z-axis scaling is a great way to scale an application to handle increasing transaction and data volumes.

Y-axis scaling functionally decomposes an application into services

X and Z-axis scaling improve the application's capacity and availability. Neither approach, however, solves the problem of increasing development and application complexity. To solve those problems you need to apply Y-axis scaling a.k.a. functional decomposition. Figure 1.6 shows how this works. Y-axis scaling splits a monolithic application into a set of services.

Figure 1.6. Y-axis scaling splits the application into a set of services. Each service is responsible for a particular function. A service is scaled using X-axis scaling and, possibly, Z-axis scaling.



A service is a mini-application that implements narrowed focussed functionality such as order management, customer management etc. A service is scaled using X-axis scaling. Some services might also use Z-axis scaling. For example, the `Order Service` consists of a set of load-balanced service instances.

The definition of the microservice architecture

The microservice architecture (a.k.a. microservices) is an architectural style that functionally decomposes an application into a set of services.

This is my high-level definition of microservices: an architectural style that functionally decomposes an application into a set of services. Note that this definition does not say anything about size. Instead, what matters is that each service has a focussed, cohesive set of responsibilities. Later in this book I discuss what this really means. But now, let's look at how the microservice architecture is a form of modularity.

1.4.2 Microservices as a form of modularity

Modularity is essential when developing large, complex applications. A modern applications such as FTGO is too large to be developed by an individual. It is also too complex to be understood by a single person. Applications must be decomposed into modules that are developed and understood by different people. In a monolithic application modules are defined using a combination of programming language constructs, such as Java packages, and build artifacts, such as Java JAR files. However, as the FTGO developers have discovered this approach tends not to work well in practice. Long lived, monolithic applications usually degenerate into big balls of mud.

The microservice architecture uses services as the unit of modularity. A service has an API, which is an impermeable boundary that is difficult to violate. You can't simply bypass the API and access an internal class as you can with a Java package. As a result, it's much easier to preserve the modularity of the application over time. There are other benefits of using services as the building blocks including the ability to deploy and scale them independently.

1.4.3 Each service has its own database

A key characteristic of the microservice architecture is that the services are loosely coupled and communicate only via APIs. One way to achieve loose coupling is by each service having its own datastore. In the online store, for example, the `OrderService` has a database that includes the `ORDERS` table and the `CustomerService` has its database, which includes the `CUSTOMERS` table. At development time, a developer can change their service's schema without having to coordinate with developers working on other service. At runtime, the services are isolated from each other. One service will never be blocked because another service holds a database lock, for example.

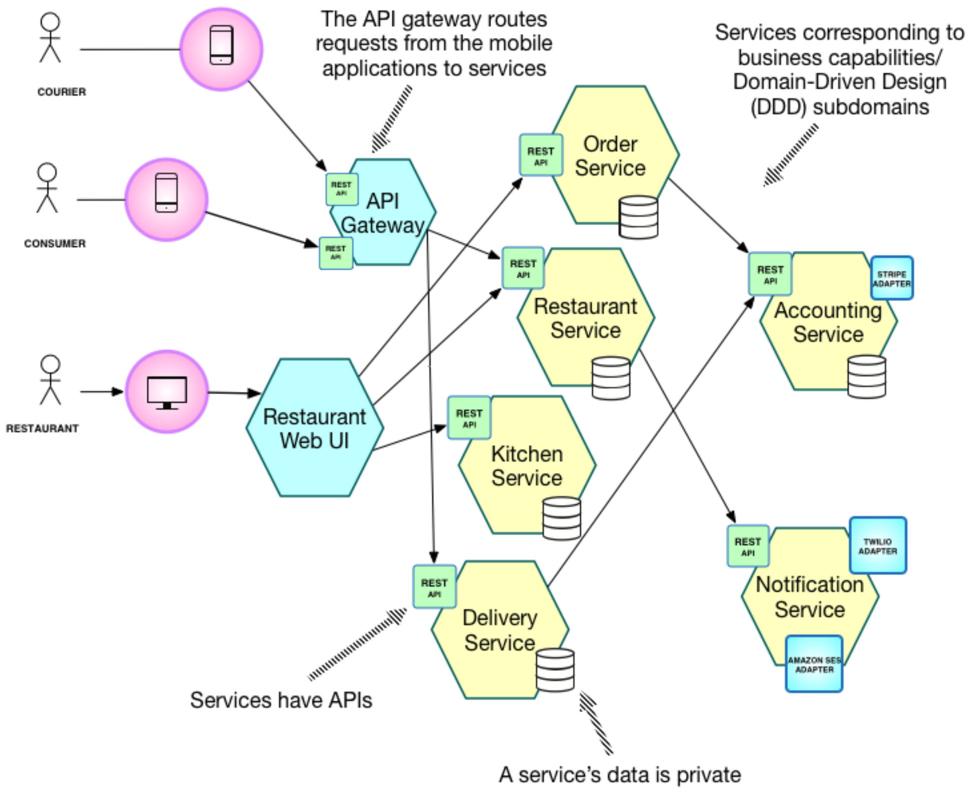
> ***Don't panic: this doesn't make Larry Ellison even richer!***
> The requirement for each service to have its own database does not mean that it has its own database server. You do not, for example, have to spend 10x more on Oracle RDBMS licenses. In chapter 2, we will explore this topic in depth.

Now that we have defined the microservice architecture and described some of its essential characteristics, lets look how this applies to the FTGO application.

1.4.4 The FTGO microservice architecture

The rest of this book discusses in-depth the FTGO application's microservices architecture. But let's quickly look at what it means to apply Y-axis scaling to this application. If we apply Y-axis decomposition to the FTGO application we get the architecture shown in figure 1.7. The decomposed application consists of numerous front-end and backend services. We would also apply X-axis and, possibly Z-axis scaling, so that at runtime, there would be multiple instances of each service.

Figure 1.7. Some of the services of the microservice architecture-based version of the FTGO application. An API gateway routes requests from the mobile applications to services. The services collaborate via APIs.



The front-end services include an `API gateway` and the `Restaurant Web UI`. The API gateway, which plays the role of a facade and is described in detail in chapter 8, provides the REST APIs that are used by the consumers' and couriers' mobile applications. The `Restaurant Web UI` implements the web interface that is used by the restaurants to manage menus and process orders.

The FTGO application's business logic consists of numerous backend services. Each backend service has a REST API and its own private datastore. The backend services include:

- `Order service` - manages orders
- `Delivery service` - manages delivery of orders from restaurants to consumers
- `Restaurant service` - maintains information about restaurants
- `Restaurant order service` - manages the preparation of orders
- `Accounting service` - handles billing and payments

Many services correspond to the modules that I described earlier this chapter. What's different is that each service and its API is very clearly defined. Each one can be independently developed, tested, deployed and scaled. Also, this architecture does a better job of preserving modularity. A developer cannot bypass a service’s API and access its internal components. In chapter 11, I describe how to transform an existing monolithic application into microservices.

1.4.5 Isn’t the microservice architecture the same as SOA?

Some critics of the microservice architecture claim that it is nothing new and that it is just SOA. At a very high-level, there are some similarities. Service-orriented architecture (SOA) and the microservice architecture are architectural styles that structure a system as a set of services. But, as table 1.1 shows, once you dig deep you encounter significant differences.

Table 1.1. Comparing SOA with microservices

	SOA	Microservices
Inter-service communication	Smart pipes, e.g. Enterprise Service Bus, using heavyweight protocols, e.g. SOAP and the other WS-* standards.	Dumb pipes, such as a message broker, or direct service-to-service communication using lightweight protocols such as REST or gRPC, etc.
Data	Global data model and shared databases	Data model and database per service
Typical service	Larger monolithic application	Smaller service

SOA and the microservice architecture usually use different technology stacks. SOA applications typically use heavyweight technologies such as SOAP and other WS* standards. They often use a ESB, which is a 'smart pipe' that contain business and messaging processing logic, to integrate the services. Applications built using the microservice architecture tend to use lightweight, open-source technologies. The services communicate via 'dumb pipes', such as a message broker or lightweight protocols such as REST or gRPC.

SOA and the microservice architecture also differ in how they treat data. SOA applications typically have a global data model and share databases. In contrast, as mentioned earlier, in the microservice architecture each service has its own database. Moreover, as I describe in chapter 2, each service is usually considered to have its own domain model.

Another key difference between SOA and the microservice architecture is the size of the services. SOA is typically used to integrate large, complex monolithic applications. While services in a microservice architecture are not always tiny they are almost always much smaller. As a result, a SOA application usually consists of a few large services where as a microservices-based application typically consists of 10s or 100s of smaller services.

1.5 Benefits and drawbacks of the microservice architecture

The microservice architecture has both benefits and drawbacks. Let’s first consider the benefits and then drawbacks.

1.5.1 Benefits of the microservice architecture

The microservice architecture has the following benefits:

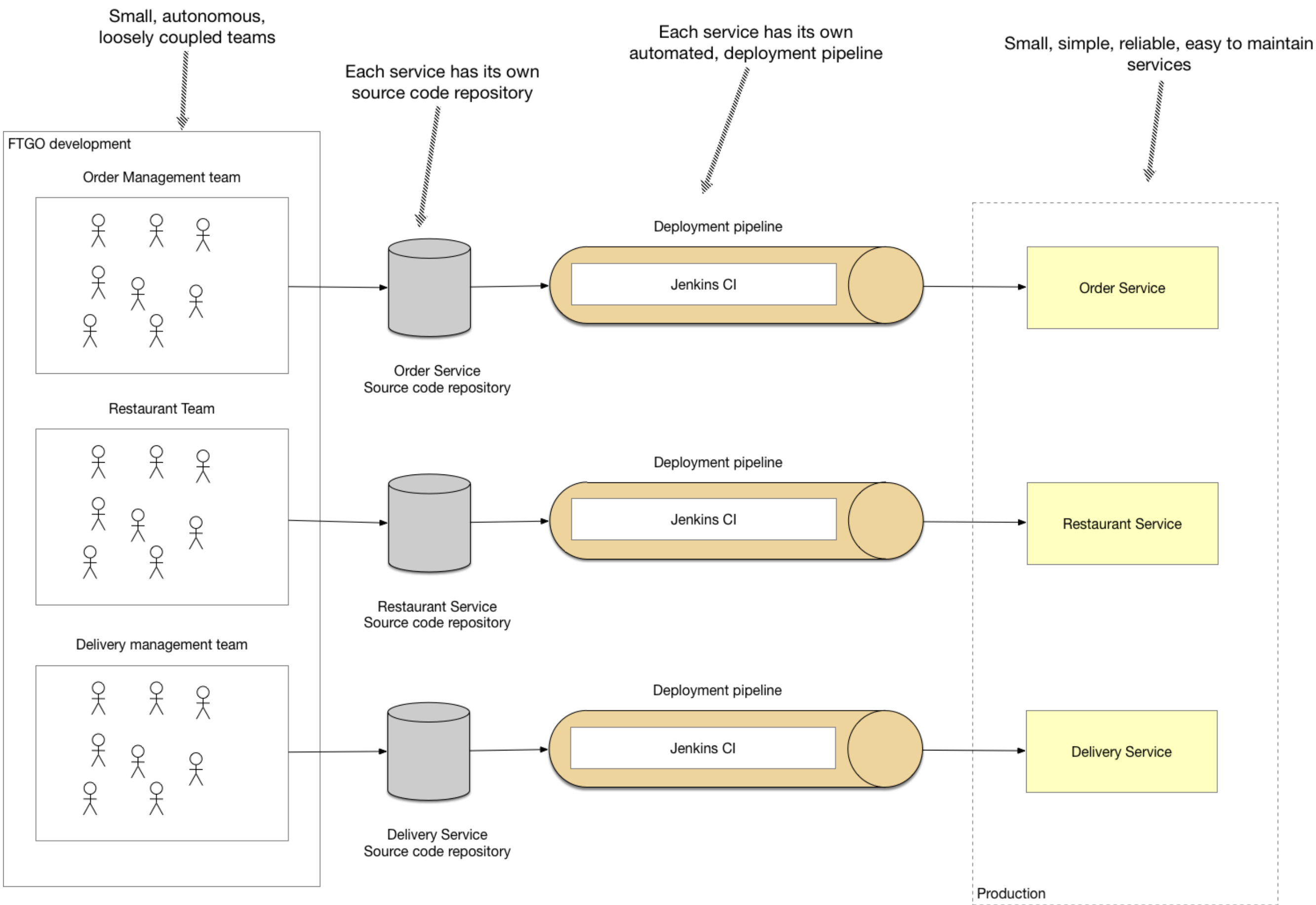
- Enables the continuous delivery and deployment of large, complex applications.
- Services are small and easily maintained
- Services are independently deployable
- Services are independently scalable
- The microservice architecture enables teams to be autonomous
- Easily experiment with and adopt new technologies
- Better fault isolation

Let's look at each benefit.

Enables the continuous delivery and deployment of large, complex applications

The most important benefit of the microservice architecture is that it enables continuous delivery and deployment of large, complex applications. As I describe later in section "Beyond microservices: process and organization", continuous delivery/deployment is part of DevOps, which is a set of practices for the rapid, frequent, and reliable delivery of software. High-performing DevOps organizations typically deploy changes into production with very few production issues.

Figure 1.8. The microservices-based FTGO application consists of a set of loosely coupled services. Each team develops, tests and deploys their services independently.



There are three ways that the microservice architecture enables continuous delivery/deployment.

First, microservice architecture has the testability required by continuous delivery/deployment. Automated testing is a key practice of continuous delivery/deployment. Since each service in a microservice architecture is relatively small, automated tests are much easier to write and faster to execute. As a result, the application will have fewer bugs.

Second, the microservices architecture has the deployability required by continuous delivery/deployment. Each service can be deployed independently of other services. If the developers responsible for a service need to deploy a change that's local to that service they do not need to coordinate with other developers. They can simply deploy their changes. As a result, it's much easier to deploy changes frequently into production.

Finally, the microservice architecture enables development teams to be autonomous and loosely coupled. You can structure the engineering organization as a collection of small (e.g. two pizza) teams. Each team is solely responsible for the development and deployment of one or more related services. Each team can develop, deploy and scale their services independently of all of the other teams. As a result, the development velocity is much higher.

The ability to do continuous delivery and deployment has several business benefits. It reduces the time to market, which enables the business to rapidly react to feedback from customers. It enables the business to provide the kind of reliable service, which today's customers have come to expect. Also, employee satisfaction is higher because more time is spent delivering valuable features instead of fighting fires. As a result, the microservice architecture has become the table stakes of any business that is depends upon software technology.

Each service a small and easily maintained

Another benefit of the microservice architecture is that each service is relatively small. The code is easier for a developer to understand. The small code base doesn't slow down the IDE making developers more productive. Also, each service typically starts a lot faster than a large monolith, which again makes developers more productive, and speeds up deployments

Services are independently scalable

Each service in a microservice architecture can be scaled independently of other services using X-axis cloning and Z-axis partitioning. Moreover, each service can be deployed on hardware that is best suited to its resource requirements. This is quite different than when using a monolithic architecture where components with wildly different resource requirements – e.g. CPU intensive vs. memory intensive – must be deployed together.

Better fault isolation

The microservice architecture also has better fault isolation. For example, a memory leak in one service only affects that service. Other services will continue to handle requests normally. In comparison, one misbehaving component of a monolithic architecture will bring down the entire system.

Easily experiment with and adopt new technologies

Last but not least, the microservice architecture eliminates any long-term commitment to a technology stack. In principle, when developing a new service the developers are free to pick whatever language and frameworks are best suited for that service. Of course, in many organizations it makes sense to restrict the choices but the key point is that you aren't constrained by past decisions.

Moreover, because the services are small, it becomes practical to rewrite them using better languages and technologies. It also means that if the trial of a new technology fails you can throw away that work without risking the entire project. This is quite different than when using a monolithic architecture, where your initial technology choices severely constrain your ability to use different languages and frameworks in the future.

1.5.2 The drawbacks of the microservice architecture

Of course, no technology is a silver bullet, and the microservice architecture has a number of significant drawbacks and issues. Indeed most of this book is about how to address these drawbacks and issues. As you read about the challenges don't worry - later in this book, I describe ways to address them.

The drawbacks and issues of the microservice architecture are:

- Finding the right set of services is challenging
- Distributed systems are complex and so development, testing and deployment is challenging
- Deploying features that span multiple services requires careful coordination
- Deciding when to adopt the microservice architecture is difficult

Let's look at each one.

Finding the right set of services is challenging

One challenge with using the microservice architecture is that there isn't a concrete, well-defined algorithm for decomposing a system into services. Like much of software development, it is somewhat of an art. To make matters worse, if you decompose a system incorrectly you will build a distributed monolith, a system consisting of coupled services that must be deployed together. A distributed monolith has the drawbacks of both the monolithic architecture and the microservice architecture.

Distributed systems are complex

Another challenge with using the microservice architecture is that developers must deal with the additional complexity of creating a distributed system. Services must use an inter-process communication mechanism. This is more complex than a simple method call Moreover, a service must be designed to handle partial failure and the remote service is either unavailable or exhibiting high latency.

Implementing use cases that span multiple services requires the use of unfamiliar techniques. Each service has its own database, which makes it challenging to implement transactions and queries that span services. As I describe in chapter 4, a microservices-based application must use what are known as sagas to maintain data consistency across services. Also, as I describe in chapter 7, a microservices-based application can't retrieve data from multiple services using simple queries. Instead, it must implement queries using either API composition or CQRS views.

IDEs and other development tools are focused on building monolithic applications and don't provide explicit support for developing distributed applications. Writing automated tests that involve multiple services is challenging. These are all issues that are specific to the microservice architecture. Consequently, your organization's developers must have sophisticated software development and delivery skills in order to successfully use microservices.

The microservice architecture also introduces significant operational complexity. There are many more moving parts – multiple instances of different types of service – that must be managed in production. To successfully deploy microservices you need a high-level of automation. You must use technologies such as:

- Automated deployment tooling such as Netflix Spinnaker
- An off the shelf PaaS such as Pivotal Cloud Foundry or Redhat Openshift
- A Docker orchestration platform such as Docker Swarm or Kubernetes

I describe the deployment options more detail in chapter 10.

Deploying features that span multiple services requires careful coordination

Another challenge with using the microservice architecture is that deploying features that span multiple services requires careful coordination between the various development teams. You have to create a rollout plan that orders service deployments based on the dependencies between services. That's quite different than when using a monolithic architecture where you can easily deploy updates to multiple components atomically.

Deciding when to adopt the microservice architecture is difficult

Another challenge with using the microservice architecture is deciding at what point during the lifecycle of the application you should use this architecture. When developing the first version of an application, you often do not have the problems that this architecture solves. Moreover, using an elaborate, distributed architecture will slow down development. This can be a major dilemma for startups whose biggest challenge is usually how to rapidly evolve the business model and accompanying application. Using the microservice architecture makes it much more difficult to iterate rapidly. A startup should almost certainly begin with a monolithic application.

Later on, however, when the challenge is how to handle complexity then it makes sense to functionally decompose the application a set of microservices. However, you might find refactoring difficult because of tangled dependencies. Later in chapter 11, I describe strategies for refactoring a monolithic application into microservices.

As you can see, the microservice architecture has many benefits but it is also has some significant drawbacks. Because of these issues, adopting a microservice architecture should not be undertaken lightly. However, for complex applications, such as a consumer-facing web application or SaaS application, it is usually the right choice. Well known sites such as eBay [PDF], Amazon.com, Groupon, and Gilt have all evolved from a monolithic architecture to a microservice architecture.

You must address numerous design and architectural issues when using the microservice architecture. What's more, many of these issues have multiple solutions, each with a different set of trade-offs. There is no one single perfect solution. In order to guide your decision making I've created the Microservice Architecture pattern language. I reference this pattern language throughout the rest of the book as I teach you about the microservice architecture. Let's look at what is a pattern language and why it is helpful.

1.6 The microservice architecture pattern language

Architecture and design is all about making decisions. You need to decide whether the monolithic or microservice architecture is the best fit for your application. And then if you pick the microservice architecture, there are lots of issues that you need to address. When making these decisions there are lots of tradeoffs to consider.

A good way to describe the various architectural and design options and improve decision making is to use a pattern language. Let's first look at why we need patterns and a pattern language. After that we will take a tour of the microservice architecture pattern language.

1.6.1 Microservices are not a silver bullet

Back in 1986, Fred Brooks, author of The Mythical Man-Month, said that in software engineering, there are no silver bullets. In other words, there are no techniques or technologies that if you adopted would give you a 10X boost in productivity. Yet 30 years later, developers are still arguing passionately about their favorite silver bullets, absolutely convinced that their favorite technology will give them a massive boost in productivity.

A lot of arguments follow the Suck/Rock dichotomy [2], which is a term coined by Neal Ford that describes how everything in the software world either sucks or rocks, with no middle ground. These arguments have this structure: If you do X then a puppy will die so, therefore, you must do Y. For example, synchronous vs. reactive programming, object-oriented vs. functional, Java vs. JavaScript, REST vs messaging. Of course, reality is much more nuanced. No technology is a silver bullet. Every technology has drawbacks and limitations, which are often overlooked by its advocates. As a result, the adoption of a technology usually follows the Gartner hype cycle [3], which describes how an emerging technology goes through five phases including the *peak of inflated expectations* (i.e. it rocks), followed by the *trough of disillusionment* (i.e. it sucks) and ending with the *plateau of productivity* (ie. we now understand the trade-offs and when to use it).

Microservices are not immune to the silver bullet phenomenon. Whether this architecture is appropriate for your application depends on a lot of factors. Consequently, it is bad advice to say to always use the microservice architecture. But conversely, it is equally bad advice to say never use them. Like many things, it depends!

The underlying reason for these polarized and hyped arguments about technology is that humans are primarily driven by their emotions. Jonathan Haidt in his excellent book The Righteous Mind: Why Good People Are Divided by Politics and Religion uses the metaphor of an elephant and its rider to describe how the human mind works. T he elephant represents the emotion part of the human brain. It makes most of the decisions. The rider represents the rational part of the brain. It can sometimes influence the elephant but it mostly provides justifications for the elephant's decisions.

We—as in the software development community—need to overcome our emotional nature and find a better way of discussing and applying technology. A great way to discuss and describe technology is to use the pattern format since it's objective. When describing a technology in the pattern format, you must, for example, describe the drawbacks. Let's take a look at the pattern format.

1.6.2 What is a pattern and a pattern language?

A pattern is a reusable solution to a problem that occurs in particular context. It's an idea, which has its origins in real-world architecture, that that has proven to be useful in software architecture and design. The concept of a pattern along was created by Christopher Alexander, a real-world architect. He also created the concept of a pattern language, which is a collection of related patterns that solve problems within a particular domain. His book A Pattern Language: Towns, Buildings, Construction (Oxford University Press, 1977) describes a pattern language for architecture that consists of 253 patterns. The patterns range from solutions to high-level problems, such as where to locate a city ("Access to Water"), to low-level problems, such as how to design a room ("Light on two sides of every room"). Each of these patterns solve a problem by arranging physical objects, which range in scope from cities to windows.

Christopher Alexander's writings inspired the software community to adopt the concept of patterns and pattern langauges. The book Gamma et al, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional, 1994), which is a collection of object-oriented design patterns, popularized patterns amongst software developers. Since the mid-nineties, software developers have documented numerous software patterns. A software pattern solves a software architecture or design problem by defining a set of collaborating software elements.

Let's imagine, for example, that you are building a banking application that must support a variety of overdraft policies. Each policy defines limits on the balance of an account and the fees charged for an overdrawn account. You can solve this problem using the Strategy pattern, which is one well known pattern from the Gang of Four book. The solution defined by this pattern consists of three parts:

- A strategy interface called `Overdraft`, which encapsulates the overdraft algorithm
- One or more concrete strategy classes, one for each particular context
- The `Account` class, which is the class that uses the algorithm

The Strategy pattern is an object-oriented design pattern so the elements of the solution are classes. Later in this section, I describe high-level design patterns where the solution consists of collaborating services.

One reason patterns are valuable is because a pattern must describe the context within which applies. The idea that a solution is specific to a particular context and might not work well in other contexts, is an improvement over how technology is typically discussed. For example, a solution that solves the problem at the scale of Netflix might not be the best approach for an application with fewer users. The value of a pattern, however, goes far beyond requiring you to consider the context of a problem. It forces you to describe other critical yet frequently overlooked aspects of a solution. A commonly used pattern structure includes three especially valuable sections:

- Forces
- Resulting context
- Related patterns

Let's look at each section starting with the forces section.

Forces: the issues that you must address when solving a problem

The forces section of a pattern describes the forces (a.k.a. issues) that you must address when solving a problem in a given context. Forces can conflict so it might not be possible to solve all them. Which forces are more important depends on the context. You have to prioritize solving some forces over others. For example, code must be easy to understand and have good performance. Code written in a reactive style has better performance than synchronous code, yet is often more difficult to understand. Explicitly listing the forces is useful because it makes it clear what issues need to be solved.

Resulting context: the consequences of applying a pattern

The resulting context section of a pattern describes the consequences of applying the pattern. It consists of three parts:

- benefits - the benefits of the pattern including the forces that have been resolved
- drawbacks - the drawbacks of the pattern including the unresolved forces
- issues - the new problems that have been introduced by applying the pattern.

The resulting context provides a more complete and less biased view of the solution, which enables better design decisions.

Related patterns: the five different types of relationships

The related patterns section of a pattern describes the relationship between the pattern and other patterns. There are five types of relationships between patterns:

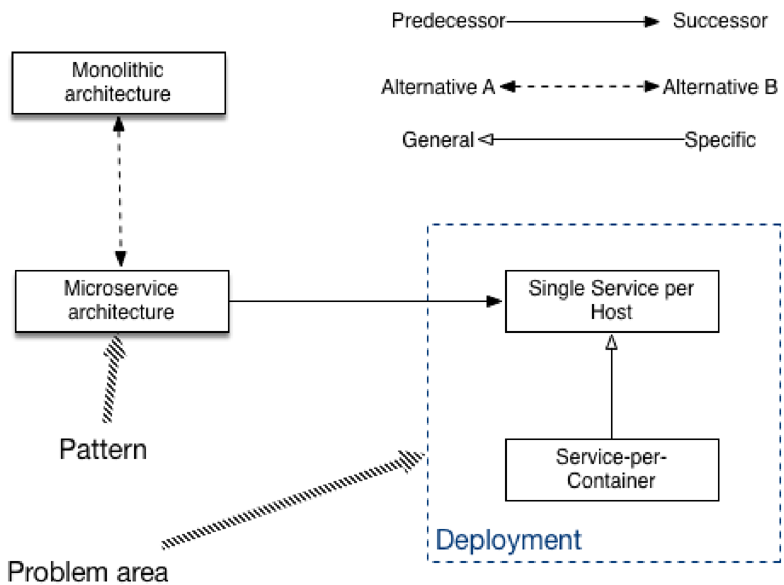
- Predecessor - a predecessor pattern is a pattern that motivates the need for this pattern. For example, the Microservice Architecture pattern is the predecessor to the rest of the patterns in the pattern language except the monolithic architecture pattern.
- Successor - a pattern that solves an issue that is introduced by this pattern. For example, if you apply the Microservice Architecture pattern you must then apply

numerous successor patterns including service discovery patterns and the Circuit Breaker pattern.

- Alternative - a pattern that provides an alternative solution to this pattern. For example, the Monolithic Architecture pattern and the Microservice Architecture pattern are alternative ways of architecting an application. You pick one or the other.
- Generalization - a pattern that is general solution to a problem. For example, in chapter 10 you will learn about the One service per host pattern, which has a few different implementation.
- Specialization - a specialized form of a particular pattern For example, in chapter 10 you will learn that the Container per service pattern is a specialization of the One service per host

In addition, you can organize patterns that tackle issues in a particular problem area into groups. The explicit description of related patterns provides valuable guidance on how to effectively solve a particular problem. Figure 1.9 shows how the relationships between patterns is visually represented.

Figure 1.9. The visual representation of different types of relationships between the patterns: a *successor* pattern solves a problem created by applying the *predecessor* pattern; two or more patterns can be *alternative* solutions to the same problem; one pattern can be a specialization of another pattern; and patterns that solve problems in the same area can be grouped.



The different kinds of relationships between patterns shown in figure 1.9 are represented as follows:

- 

 - represents the predecessor-successor

relationship.

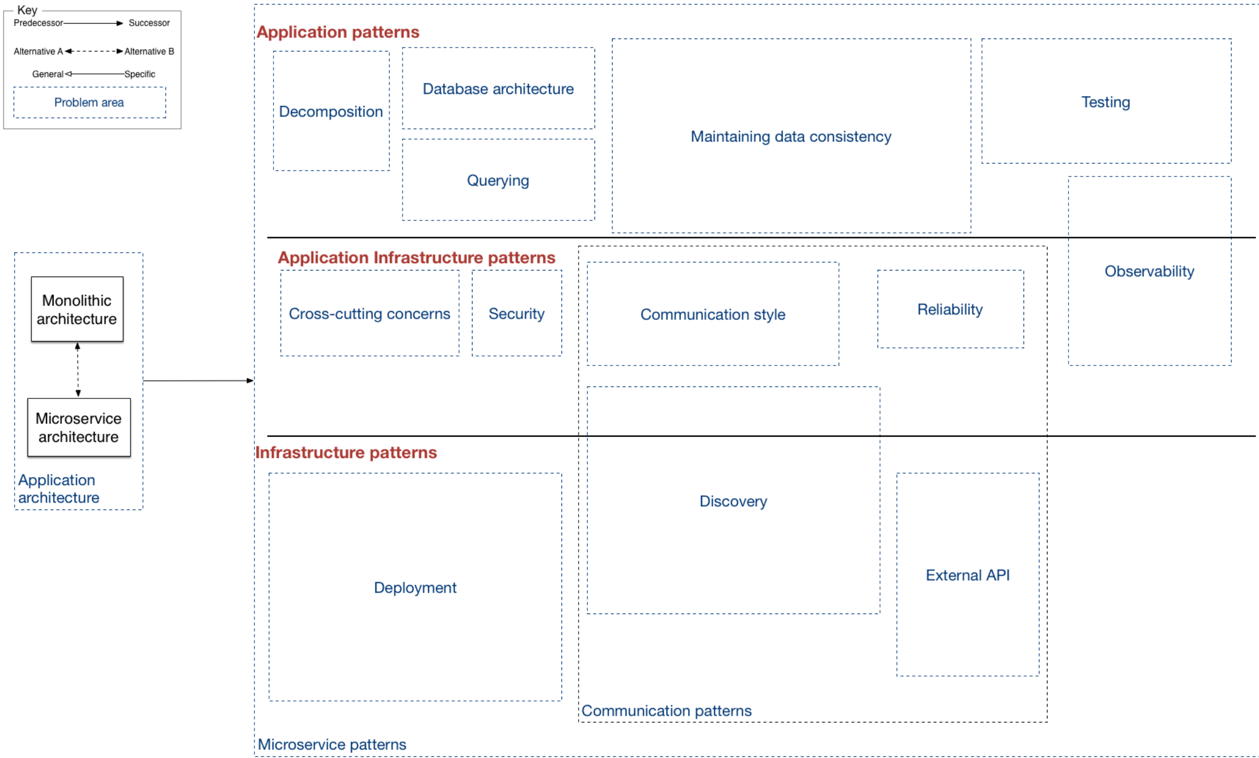
- Alternative A ◄- - - - - - - - -► Alternative B - patterns that are alternative solutions to the same problem.
- General ◁———————— Specific - indicates that one pattern is a specialization of another pattern.
- - patterns that apply to a particular problem area

A collection of patterns related through these relationship sometimes form what is known as a pattern language. The patterns in a pattern language work together to solve problems in a particular domain. In particular, I've created the microservice architecture pattern language. It's a collection of inter-related software architecture and design patterns for microservices. Let's take a look pattern language.

1.6.3 Overview of the microservice architecture pattern language

The microservice architecture pattern language is a collection of patterns that help you architect an application using the microservice architecture. Figure 1.10 shows the high-level structure of the pattern language. The pattern language first helps you to decide whether to use the microservice architecture. It describes the monolithic architecture and the microservice architecture and their benefits and drawbacks. Then, if the microservice architecture is a good fit for your application, the pattern language helps you use it effectively by solving various architecture and design issues.

Figure 1.10. A high-level view of the Microservice architecture pattern language showing the different problem areas that the patterns solve. On the left are the application architecture patterns: Monolithic architecture and Microservice architecture. All of the other groups of patterns solve problems that result from choosing the microservice architecture.



The pattern language consists of several groups of patterns. On the left in figure 1.10 , is the application architecture patterns group, the Monolithic Architecture pattern and the Microservice Architecture pattern. Those are the patterns that we have discussed in this chapter. The rest of the pattern language consists of groups of patterns that are solutions to issues that are introduced by using the microservice architecture pattern.

The patterns are also divided into three layers:

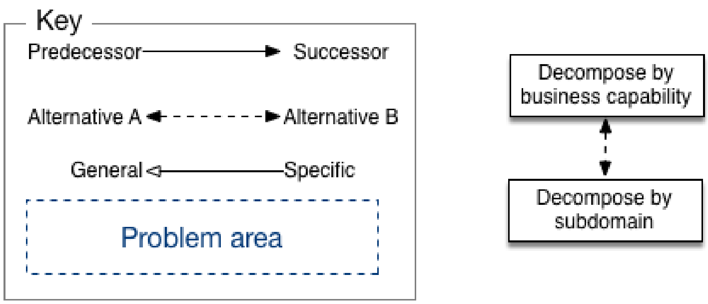
- infrastructure patterns - these are patterns that solve problems that are mostly infrastructure issues that are outside of development
- application infrastructure - these are patterns for infrastructure issues that also impact development
- application patterns - these are patterns that solve problems faced by developers

These patterns are grouped together based on the kind of problem they solve. Let's look at the main groups of patterns.

Patterns for decomposing an application into services

Deciding how to decompose a system into a set of services is very much an art but there are number of strategies that can help. The two decomposition patterns, which are shown figure 1.11, are different strategies that you can use to define your application's architecture.

Figure 1.11. There are two decomposition patterns: Decompose by business capability, which organizes services around business capabilities; and Decompose by Subdomain, which organizes services around Domain Driven Design subdomains.



I describe these patterns in detail in chapter 2.

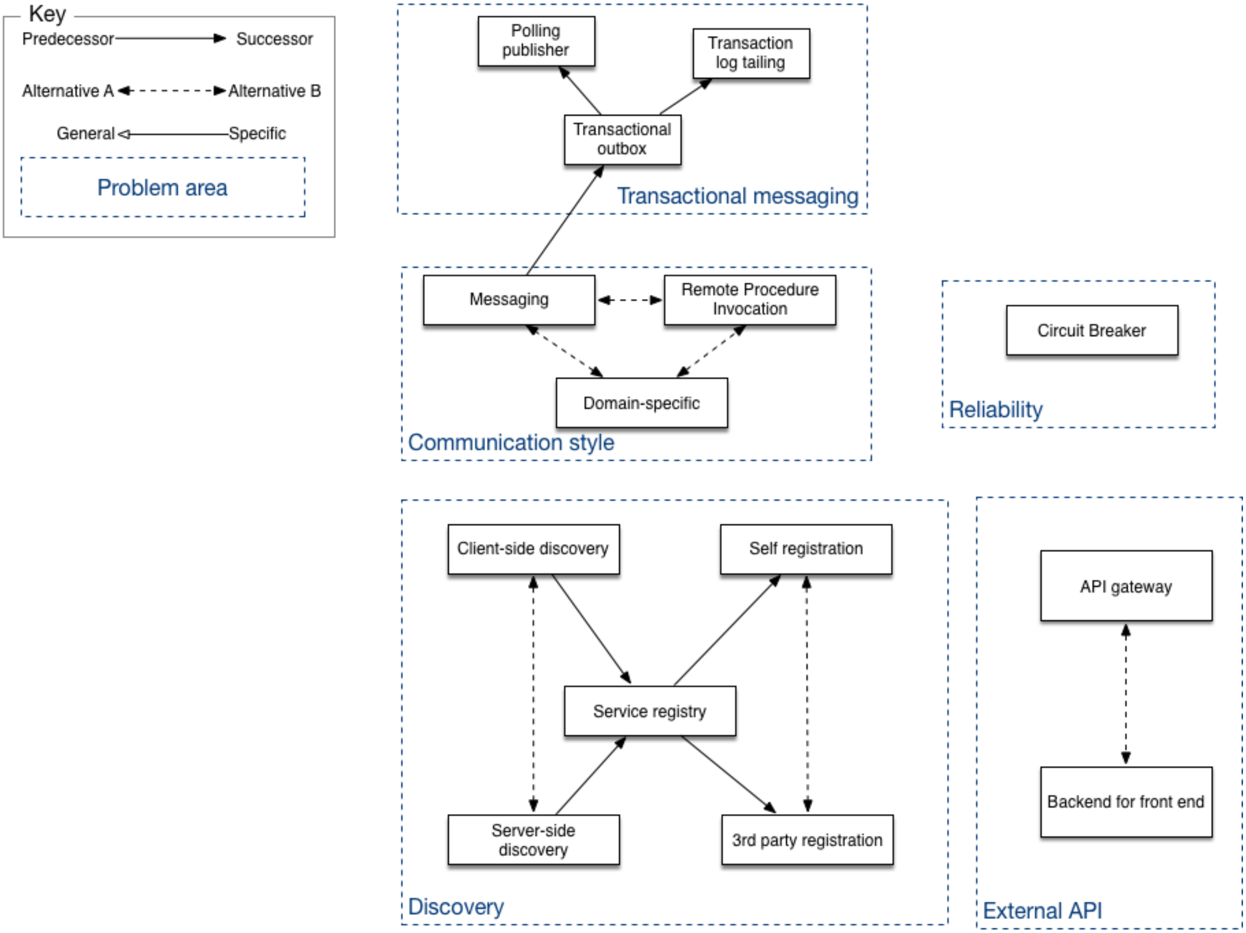
Communication patterns define how services communicate with each other and the external clients

An application that is built using the microservice architecture is a distributed system. Consequently, inter-process communication is an important part of the microservice architecture. You must make a variety of architectural and design decisions about how your services communicate with one another and the outside world. Figure 1.12 shows the communication patterns, which are organized into five groups:

- Communication style - what kind of IPC mechanism to use?
- Discovery - how does a client of service determine the IP address of a service instance so that it make, for example, an HTTP request?
- Reliability - how to ensure that communication between services is reliable even though services can be unavailable?
- Transactional messaging - how to integrate the sending of messages and publishing of events with database transactions that update business data?
- External API - how do clients of your application communicate with the services?

Figure 1.12. There are five groups of communication patterns: the communication style group, which define ways that services can communicate; the transactional messaging group, which integration messaging with database transactions; the discovery group of patterns, which address the issue of how does one service know the network location of a service that it wants to invoke; reliability group, which make inter-process communication robust; and the external

API patterns, which define how requests from outside of the application are handled.

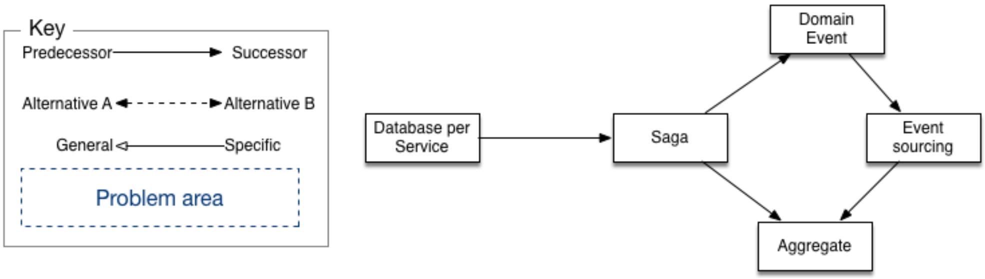


In chapter 3, we look at the first four groups of patterns: communication style, discovery, reliability, and transaction messaging. And, in chapter 8, we look at the external API patterns.

Data consistency patterns for implementing transaction management

As mentioned earlier, in order to ensure loose coupling each service has its own database. Unfortunately, having a database per service introduces some significant issues. I describe in chapter 4 that the traditional approach of using distributed transactions (aka. 2PC) is not a viable option for modern application. Instead, an application needs to maintain data consistency by using the Saga pattern. Figure 1.13 shows data-related patterns.

Figure 1.13. Because each service has its own database you must use the Saga pattern to maintain data consistency across services.

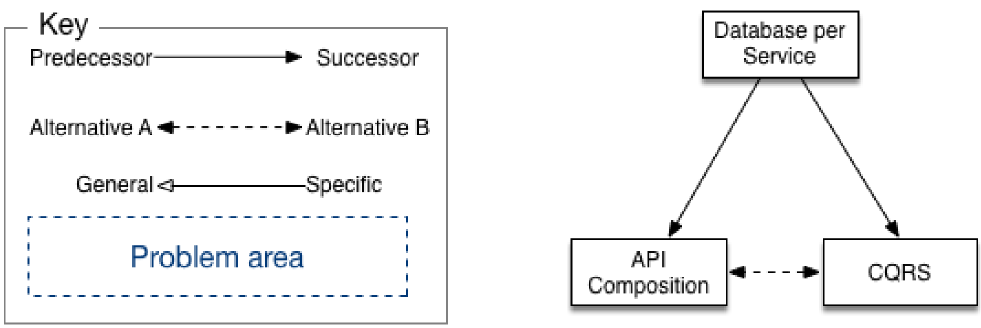


In chapters 4, 5 and 6 I describe these patterns in more detail.

Patterns for querying data in a microservice architecture

The other challenge with using a database per service is that some queries need to join data that is owned by multiple services. A service's data is only accessible via its API and so you cannot use distributed queries against its database. As figure 1.14 shows, there are a couple of patterns that you can use to implement queries.

Figure 1.14. Because each service has its own database, you must use one of the querying patterns to retrieve data that is scattered across multiple services.



Sometimes you can use the API Composition pattern, which invokes the APIs of one or more services and aggregates results. Other times, you must use the Command Query Responsibility Segregation (CQRS) pattern, which maintains one or more easily queried replicas of the data. In chapter 7 we look at the different ways of implementing queries.

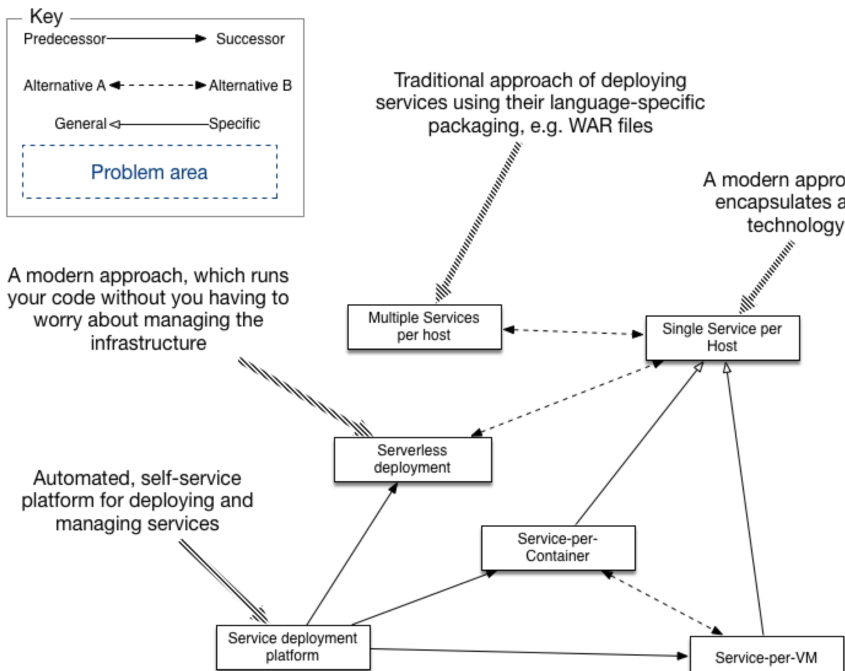
Service deployment patterns

Deploying a monolithic application is not always easy. But it is straightforward in the sense that there is a single application to deploy. You simply have to run multiple instances of the application behind a load balancer.

In comparison, deploying a microservices-based application is much more complex.

There are tens or hundreds of services that are written in a variety of languages and frameworks. There are many more moving parts that need to be managed. Figure 1.15 shows the deployment patterns.

Figure 1.15. There are several patterns for deploying microservices. The traditional approach is to deploy services in a language-specific packaging format. There are two modern approaches to deploying services. The first deploys services as VM or containers. The second is the serverless approach. You simply upload the service's code and the serverless platform runs it. You should use a Service deployment platform, which is an automated, self-service platform for deploying and managing services.



The traditional, and often manual, way of deploying applications in a language-specific packaging format, e.g. WAR files, doesn't scale to support a microservice architecture. You need a highly automated deployment infrastructure. Ideally, you should use a deployment platform that provides the developer with a simple UI (command line or GUI) for deploying and managing their services. The deployment platform will typically be based on VMs, Containers or Serverless technology. In chapter 10 we look at the different deployment options.

Observability patterns provide insight into application behavior

A key part of operating an application is understanding its runtime behavior and troubleshooting problems such as failed requests and high latency. While understanding and troubleshooting a monolithic application is not always easy, it helps that requests are handled in a simple, straightforward way. Each incoming request is load balanced to a particular application instance, which makes a few calls to the database, and returns a response. If, for example, you need to understand how a particular request was handled you simply look at the log file of the application instance that handled the request.

In contrast, understanding and diagnosing problems in a Microservice architecture is much more complicated. A request can bounce around between multiple services before a response is finally returned into a client Consequently, there isn't just one log file to examine. Similarly, problems with latency are more difficult to diagnose since there are multiple suspects.

You can use the following patterns to design observable services:

- Health check API - expose an endpoint that returns the health of the service
- Log aggregation - log service activity and write logs into a centralized logging server, which provides searching and alerting
- Distributed tracing - assign each external request a unique id and trace requests as they flow between services
- Exception tracking - report exceptions to an exception tracking service, which deduplicates exceptions, alerts developers, and tracks the resolution of each exception
- Application metrics - service maintain metrics, such as counters and gauges, and expose them to a metrics server
- Audit logging - log user actions

In chapter {chapter-prod-ready}, I describe these patterns in more detail.

Patterns for the automated testing of services

The microservice architecture makes individual services easier to test since they are much smaller than the monolithic application. At the same time, however, it is important to test that the different services work together, while avoiding using complex, slow and brittle end-to-end tests, which test multiple services together. There are the following patterns for simplifying testing by testing services in isolation:

- Consumer-driven contract test - verify that a service meets the expectations of its clients
- Consumer-side contract test - verify that the client of a service can communicate with the service
- Service component test - test a service in isolation

I describe these testing patterns in more detail in the chapters {chapter-testing-part-1} and {chapter-testing-part-2}.

Patterns for handling cross-cutting concerns

In a microservice architecture there are numerous concerns that every service must implement including the observability patterns, discovery patterns. It must also implement the Externalized configuration pattern, which supplies configuration parameters, such as database credentials, to a service at runtime. When developing a new service, it would be too time consuming to reimplement these concerns from scratch. A much better approach is to apply the Microservice Chassis pattern and build services on top of a framework that handles these concerns. I describe these patterns in more detail in chapter {chapter-prod-ready}.

Security patterns

The final pattern group is the security patterns group. In a microservice architecture, users are typically authenticated by the API Gateway. It must then pass information about the user, such as their identity and roles, to the services that it invokes. A common solution is to apply the Access token pattern. The API gateway passes an access token, such as a JSON Web Tokens (JWT), to the services, which can validate the token and obtain information the user. In chapter {chapter-prod-ready}, I describe the Access token pattern in more detail.

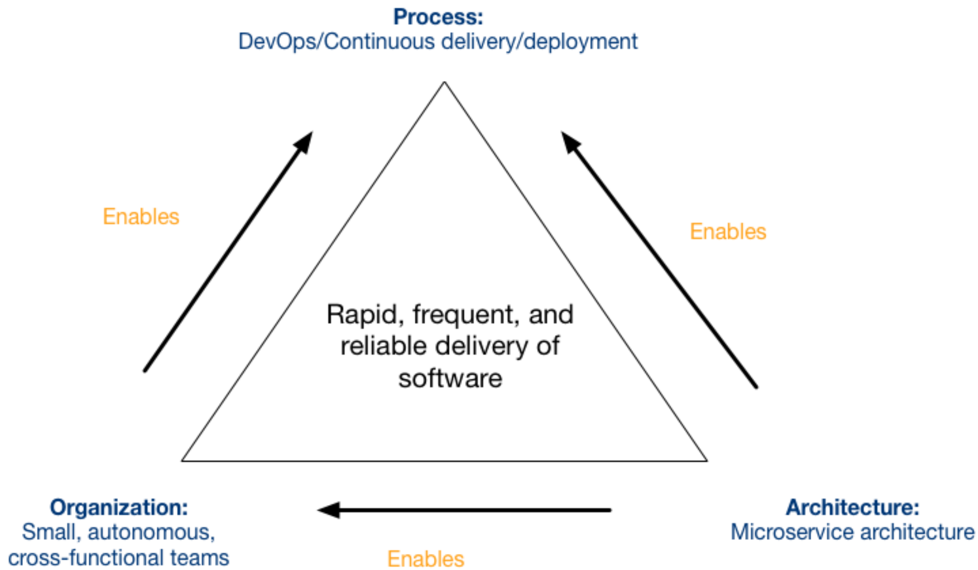
Not surprisingly, the patterns in the microservices pattern language are focussed on solving architect and design problems. You certainly need the right architecture in order to successfully develop software, It is not the only concern, however. You must also consider process and organization.

[2] http://nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html
[3] https://en.wikipedia.org/wiki/Hype_cycle

1.7 Beyond microservices: process and organization

For a large, complex application, the microservice architecture is usually the best choice. However, as well as having the right architecture, successful software development requires you to also have the organization and development and delivery process. Figure 1.16 shows the relationship between process, organization and architecture.

Figure 1.16. The rapid, frequent, and reliable delivery of software of large, complex applications requires a combination of DevOps, which includes continuous delivery/deployment; small, autonomous teams; and the microservice architecture



I have already described the microservice architecture. Let's look at organization and process.

1.7.1 Software development and delivery organization

Success inevitably means that the engineering team will grow. On the one hand, that is a good thing since more developers can get more done. The trouble with large teams is, as Fred Brooks described in **The mythical man month**, the communication overhead of a team of size N is O(N^2). If the team gets too large, then it will become inefficient due to the communication overhead. Imagine, for example, trying to do a daily standup with 20 people.

The solution is to refactor a large single team into a team of teams. Each team is small, consisting of no more than 8-12 people. It has a clearly defined business-oriented mission: developing and possibly operating one or more services, which implement a feature or a business capability. The team is cross functional and can develop, test and deploy its services without having to frequently communicate or coordinate with other teams.

Reverse Conway maneuver

In order to effectively deliver software when using the microservice architecture you need to take into account Conway's law. Conway's Law [4] states *organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations*

-- Melvin Conway 1978

In other words, your application's architecture mirrors the structure of the organization that developed it. It is important, therefore, to apply Conway's law in reverse [5] and design your organization so that its structure mirrors your microservice architecture. By doing so, you ensure that your development teams are as loosely coupled as the services.

The velocity of the team of teams is significant higher than that of a single large team. As I described earlier in section "Enables the continuous delivery and deployment of large, complex applications", the microservice architecture plays a key role in enabling the teams to be autonomous. Each team can develop, deploy and scale their services without coordinating with other teams. Moreover, it is very clear who to contact when a service is not meeting its SLA.

What's more, the development organization is much more scalable. You grow the organization by adding teams. Also, if a single team becomes too large you split it and its associated service or services. Since the teams are loosely coupled, you avoid the communication over of a large team. As a result, you can add people without impacting productivity.

1.7.2 Software development and delivery process

Using the microservice architecture with a waterfall development process is like driving a horse-drawn Ferrari. You would squander most of the benefit of using microservices. If you want to develop an application with the microservice architecture, it is essential that you adopt agile development and deployment practices such as Scrum or Kanban. Better yet, you should practice continuous delivery/deployment, which is a part of DevOps.

Jez Humble [6] defines continuous delivery as follows:

Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way.

A key characteristic of continuous delivery is that software is always releasable. It relies on a high-level of automation including automated testing. Continuous deployment takes continuous delivery one step further is the practice of automatically deploying releasable code into production. High performing organizations that practice continuous deployment [7] deploy multiple times per day into production, have far fewer production outages, and recover quickly from any that do occur. As I described earlier in section "Enables the continuous delivery and deployment of large, complex applications", the microservice architecture directly supports continuous delivery/deployment.

Move fast without breaking things

The goal of continuous delivery/deployment (and, more generally, DevOps) is to rapidly yet reliably deliver software. Four useful metrics for assessing software development are:

- Deployment frequency - how often is software deployed into production
- Lead time - the time from a developer checking in a change to that change being deployed
- Meantime to recover - time to recover from a production problem
- Change failure rate - the percentage of changes that result in a production problem

In a traditional organization the deployment frequency is low, and the lead time is high. Stressed out developers and operations people typically stay up late into the night fixing last minute issues during the maintenance window. In contrast, a DevOps organization releases software frequently, often multiple times per day, with far fewer production issues. Amazon, for example, deployed changes into production every 11.6 seconds in 2014 [8] and Netflix had a lead time of 16 minutes for one software component [9].

1.7.3 The human side of adopting microservices

Adopting the microservice architecture changes your architecture, your organization and your development processes. Ultimately, however, it changes the working environment of people, who are, as mentioned earlier, emotional creatures. Their emotions, if ignored, can make the adoption of microservices a bumpy ride. Mary and the other FTGO leaders will struggle to change how FTGO develops software.

The best selling book William and Susan Bridges, Managing Transitions (Da Capo Lifelong Books, 2017, https, ://wmbridges.com/books) introduces the concept of a transition, which is the process of how people respond emotionally to a change. It describes a three stage Transition Model:

1. Ending, Losing, and Letting Go - the period of emotional upheaval and resistance when people are presented with a change that forces them out of their comfort zone. They often mourn the loss of the old way of doing things. For example, when people reorganize into cross-functional teams they miss their former team-mates. Similarly, a data modeling group, which owns the global data model, will be threatened by the idea of each service having its own data model.
2. The Neutral Zone - the intermediate stage between the old and new ways of doing things where people are often confused. They are, often, struggling to learn the new way of doing things.
3. The New Beginning - the final stage where people have enthusiastically embraced the new way of doing things and are starting to experience the benefits.

The book describes how best to manage each stage of the transition and increase the likelihood of successfully implementing the change. FTGO is certainly suffering from monolithic hell and needs to migrate to a microservice architecture. They must also change their organization and their development processes. In order for FTGO to successfully accomplish this, however, it is essential that they take into account the transition model and consider people's emotions. In the next chapter, you will learn about the goal of software architecture and how to decompose an application into services.

[4] https://en.wikipedia.org/wiki/Conway%27s_law
[5] https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver
[6] https://continuousdelivery.com/
[7] 2017 State of DevOps report. https://puppet.com/resources/whitepaper/state-of-devops-report
[8] Jon Jenkins, Velocity conference 2011, https://www.youtube.com/watch?v=dxk8b9rSKOo
[9] Netflix TechBlog, https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15

1.8 Summary

- The Monolithic architecture pattern structures the application as a single deployable unit
- The Microservice architecture decomposes a system into a set of independently deployable services each with their own database
- The Monolithic architecture is a good choice for simple applications but the Microservice architecture is usually a better choice for large, complex applications
- The Microservice architecture accelerates the velocity of software development by enabling small, autonomous teams to work in parallel.
- The microservice architecture is not a silver bullet since there are significant drawbacks including complexity.
- The microservice architecture pattern language is a collection of patterns that help you architect an application using the microservice architecture. It helps you decide whether to use the microservice architecture. And, if you pick the microservice architecture, the pattern language helps you apply it effectively.
- You need more than just the microservice architecture to accelerate software delivery. Successful software development also requires DevOps and small, autonomous teams
- Don’t forget about the human side of adopting microservices. You need to consider employees' emotions in order to successfully transition to a microservice architecture.