Existing “Big Ball of Mud” monolith

From <https://microservices.io/book>

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.

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 become 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.

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.

***The slow march toward monolithic hell***

Since its launch in late 2005, FTGO had grown by leaps and bounds. Today, it’s one of the leading online food delivery companies in the United States. 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’s 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 large, complex application. Despite the best efforts of the FTGO development team, it’s become an example of the Big Ball of Mud pattern ([www.laputan](http://www.laputan) .org/mud/).

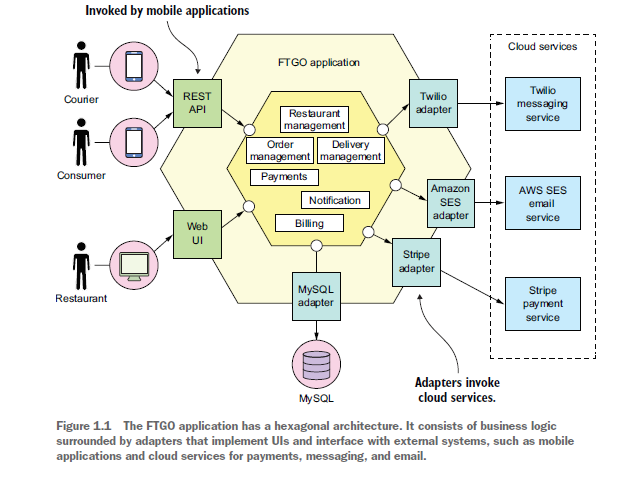
To quote Foote and Yoder, the authors of that pattern, it’s a “haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.” The pace of software delivery has slowed. To make matters worse, the FTGO application has been written using some increasingly obsolete frameworks. The FTGO application

is exhibiting all the symptoms of monolithic hell.

***The architecture of the FTGO application***

FTGO is a typical enterprise Java application. Figure 1.1 shows its architecture. The FTGO application has a hexagonal architecture.. 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.



***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*—You can change the code and the database schema, build, and deploy.

 *Straightforward to test*—The developers wrote end-to-end tests that 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, though, development, testing, deployment, and scaling became much more difficult.

***Living in monolithic hell***

Unfortunately, as the FTGO developers have discovered, the monolithic architecture has a huge limitation.

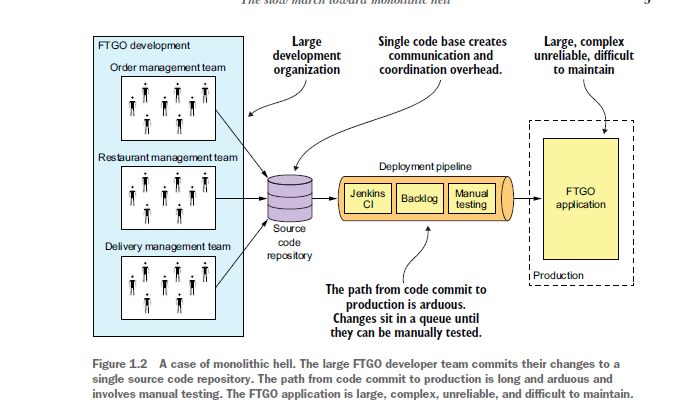
Successful applications like the FTGO application have a habit of outgrowing the monolithic architecture. Each sprint, the FTGO development team implemented a few more stories, which made the code base larger.

Moreover, as the company became more successful, the size of the development team steadily grew.

Not only did this increase the growth rate of the code base, it also increased the management overhead.

As figure 1.2 below 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 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.



To make matters worse, this overwhelming complexity tends to be a downward spiral. If the code base is difficult to understand, a developer won’t make changes correctly. Each change makes the code base incrementally more complex and harder 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.

DEVELOPMENT IS SLOW

As well as 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.

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.

Apparently, as of 2011, Amazon.com deployed a change into production every 11.6 seconds without ever impacting the user!

.

Another reason it takes so long to get changes into production is that testing takes a long time.

SCALING IS DIFFICULT

The FTGO team also has problems scaling its application. That’s 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. Because these modules are part of the same application, FTGO must compromise on the server configuration.

DELIVERING A 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 that testing the application thoroughly is difficult, due to its large size. This lack of testability means bugs make their way into production.

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—for example, a memory leak—crashes all instances of the application, one by one.

LOCKED INTO INCREASINGLY OBSOLETE TECHNOLOGY STACK

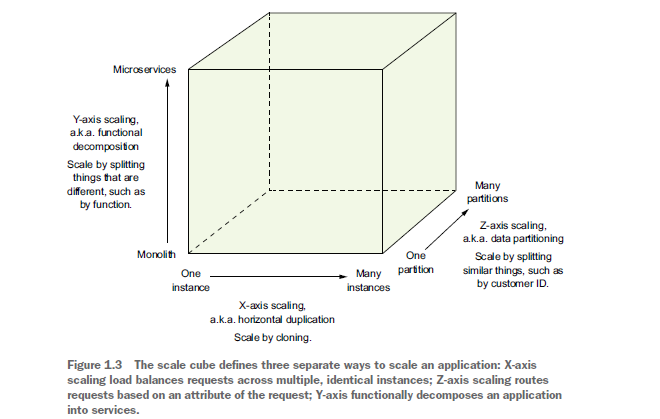
The final aspect of monolithic hell experienced by the FTGO team is that the architecture forces them to use a technology stack that’s becoming increasingly obsolete. The monolithic architecture makes it difficult to adopt new frameworks and languages.

***Microservice architecture to the rescue***

Mary has come to the conclusion that FTGO must migrate to the microservice architecture.

***Scale cube and microservices***

My definition of the microservice architecture is inspired by Martin Abbott and Michael Fisher’s excellent book, *The Art of Scalability* (Addison-Wesley, 2015). This book describes a useful, three-dimensional scalability model: the *scale cube*, shown in figure 1.3.



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

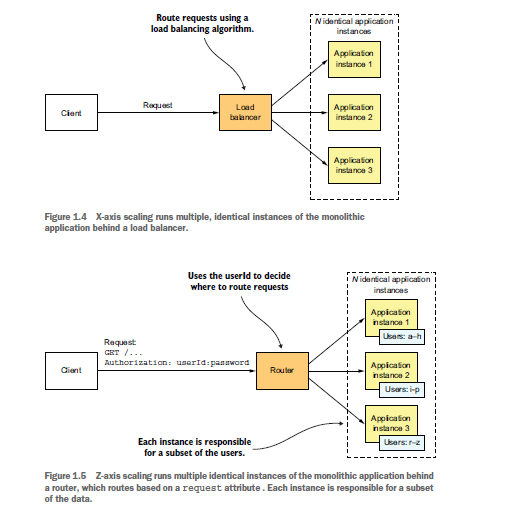
X-AXIS SCALING LOAD BALANCES REQUESTS ACROSS MULTIPLE INSTANCES

*X-axis* scaling is a common way to scale a monolithic application. Figure 1.4 shows how X-axis scaling works. You run multiple instances of the application behind a load balancer. The load balancer distributes requests among the *N* identical instances of the application. This is a great way of improving the capacity and availability of an application.

Z-AXIS SCALING ROUTES REQUESTS BASED ON AN ATTRIBUTE OF THE REQUEST

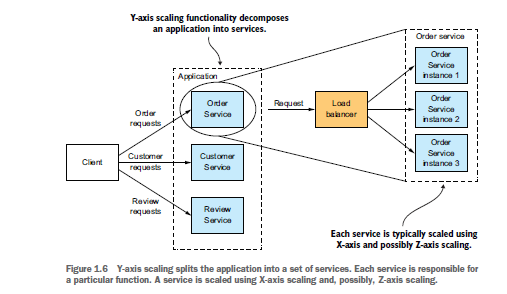
*Z-axis* scaling also runs multiple instances of the monolith application, but unlike X-axis scaling, each instance is responsible for only a subset of the data. Figure 1.5 shows how Z-axis scaling works. The router in front of the instances uses a request attribute to route it to the appropriate instance.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

In this example, each application instance is responsible for a subset of users.



Y-AXIS SCALING FUNCTIONALLY DECOMPOSES AN APPLICATION INTO SERVICES

X- and Z-axis scaling improve the application’s capacity and availability. But neither approach solves the problem of increasing development and application complexity. To solve those, you need to apply *Y-axis* scaling, or *functional decomposition*. Figure 1.6 shows how Y-axis scaling works: by splitting a monolithic application into a set of services.



A *service* is a mini application that implements narrowly focused functionality, such as order management, customer management, and so on.

***Microservices as a form of modularity***

*Modularity* is essential when developing large, complex applications. A modern application like FTGO is too large to be developed by an individual. It’s 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. Longlived, 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 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 building blocks, including the ability to deploy and scale them independently.

***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, Order Service has a database that includes the ORDERS table, and Customer Service has its database, which includes the CUSTOMERS table. At development time, developers can change a service’s schema without having to coordinate with developers working on other services. At runtime, the services are isolated from each other—for example, one service will never be blocked because another service holds a database lock.

***The FTGO microservice architecture***

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 frontend 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.

The frontend services include an API gateway and the Restaurant Web UI. The API gateway, which plays the role of a facade, provides the REST APIs that are used by the consumers’ and couriers’ mobile applications. The Restaurant Web UI implements the web interface that’s 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 the following:

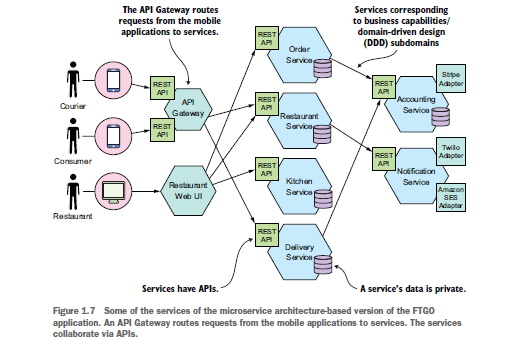
Order Service**—**Manages orders

Delivery Service—Manages delivery of orders from restaurants to consumers

Restaurant Service—Maintains information about restaurants

Kitchen Service—Manages the preparation of orders

Accounting Service—Handles billing and payments



Many services correspond to the modules described earlier. What’s different is that each service and its API are very clearly defined. Each one can be independently developed, tested, deployed, and scaled.

Also, this architecture does a good job of preserving modularity. A developer can’t bypass a service’s API and access its internal components.

***Benefits of the microservice architecture***

The microservice architecture has the following benefits:

 It 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.

 It allows easy experimenting and adoption of new technologies.

 It has better fault isolation.

***Drawbacks of the microservice architecture***

Certainly, no technology is a silver bullet, and the microservice architecture has a number of significant drawbacks and issues.

Here are the major drawbacks and issues of the microservice architecture:

 Finding the right set of services is challenging.

 Distributed systems are complex, which makes development, testing, and deployment

difficult.

 Deploying features that span multiple services requires careful coordination.

 Deciding when to adopt the microservice architecture is difficult.

***Summary***

 The Monolithic architecture pattern structures the application as a single deployable unit.

 The Microservice architecture pattern decomposes a system into a set of independently deployable services, each with its own database.

 The monolithic architecture is a good choice for simple applications, but 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 isn’t a silver bullet—there are significant drawbacks, including complexity.

 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.