Monolithic vs Microservices

Key sentences from the intro:

* The book talks about a fictional? app called Food-to-Go (FTGO) and tells a quick story about the problems they were facing
* They had spent two hours discussing why the development team was going to miss another critical release date
* Despite adopting agile, the pace of development was slowing down
* They were learning about the latest software development techniques, including continuous deployment and the microservice architecture
* The realized that they were suffering from a case of *monolithic hell* and that the cure was to adopt the microservice architecture
* But the microservice architecture and the associated state-of-the-art software development practices felt like an elusive dream.

# The Slow March towards Monolithic hell

* 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.
* it’s become an example of the Big Ball of Mud pattern: a “haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.”
* 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.

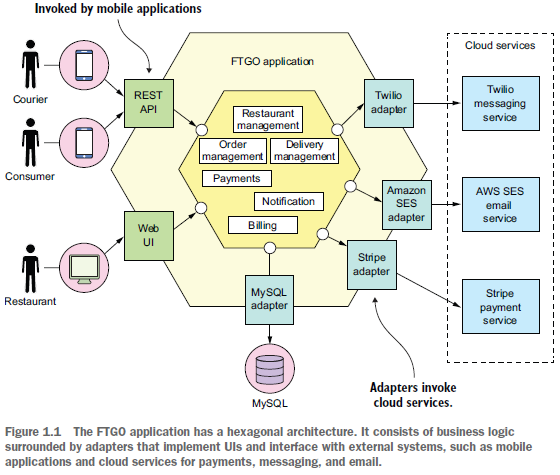
The next section describes the architecture of the FTGO application. Then it talks about why the monolithic architecture worked well initially. We’ll get into how the FTGO application has **outgrown its architecture** and how **that has resulted in monolithic hell**.

## 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, which is an architectural style described 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.

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, **including 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**.



**The monolithic architecture isn’t inherently bad. The FTGO developers made a good decision when they picked monolithic architecture for their application:**

## 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.**

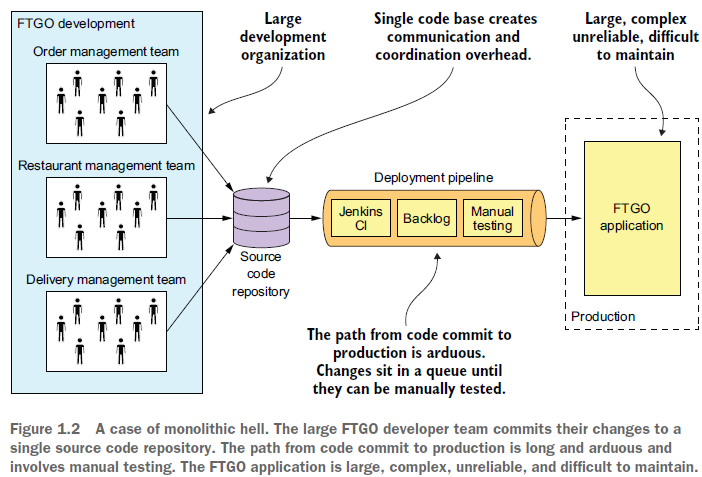
**Over time**, though**, development**, **testing**, **deployment**, **and scaling** became much **more**

**difficult**. Let’s look at why:

## Living in Monolithic Hell

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 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**.



Let’s look at why this has happened:

### Complexity Intimidates Developers

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

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. Moreover, because it’s so large, the application takes a long time to start up. As a result, the edit-build-run-test loop takes a long time, which badly impacts productivity.

### 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 to production once a month, usually late on a Friday or Saturday night. It’s heard that the state-of-the-art for Software-as-a-Service (SaaS) applications is ***continuous deployment*: deploying changes to production many times a day during business hours.**

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 distant dream. And adopting continuous deployment seems next to impossible.

FTGO has partially adopted agile. The engineering team is divided 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 un-releasable state. When the FTGO developers tried to solve this problem by using feature branches, their attempt resulted in lengthy, painful merges. Consequently, once a team completes its sprint, a long period of testing and code stabilization follows.

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 isn’t 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 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. 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.

### 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. It would be extremely expensive and risky to rewrite the entire monolithic application so that it would 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 backward 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, that’s not possible with a monolithic application.

# Microservices to Rescue