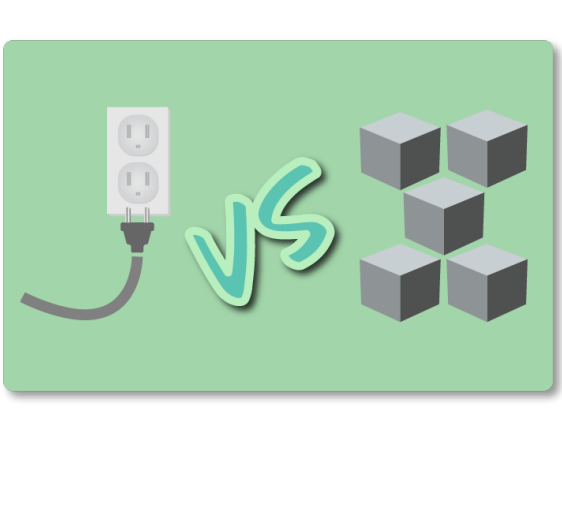
API vs. Microservices: A Microservice Is More Than Just an API

September 25, 2018
by SentinelOne

When writing software, consider both the implementation and the architecture of the code. The software you write is most effective when written in a way that logically makes sense. In addition to being architecturally sound, software should also consider the interaction the user will have with it and the interface the user will experience.

Both the concept of an API and the concept of a microservice involve the structure and interactions of software. A microservice can be misconstrued as simply an endpoint to provide an API. But microservices have much more flexibility and capabilities than that. This article will speak on the differences between APIs and microservices, plus detail some of the benefits a microservice can provide.

To get started, let's define our terms.



What Is an API?

First, let's define what an API is. According to Wikipedia, an API (application programming interface) is:

> a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of **communication** between various components.

An easy way to think about an API is to think of it as a contract of actions you can request for a particular service. APIs are in use today in a multitude of web applications, such as social media, banking software, and much more. The standardized contract allows for external applications to interface with another.

For instance, let's say you're building an application that's going to integrate with Facebook. You would be able to use the Facebook Graph API to access data inside Facebook, such as users, post, comments, and more. The API simplifies the complexity of trying to use the data inside Facebook and provides an easy-to-use way for the developer to access that data.

Common API Actions

In today's world, APIs are usually developed using a RESTful style. These APIs will have a series of verbs associating with HTTP actions, like the following:

- GET (get a single item or a collection)
- POST (add an item to a collection)
- PUT (edit an item that already exists in a collection)
- DELETE (delete an item in a collection)

The advantage of this consistency through different applications is having a standard when performing various actions. The four different HTTP verbs above correlate with the common CRUD capabilities that many applications use today. When working with different APIs in one application, this makes for a recognizable way to understand the implications of the actions taken across different interfaces.

If you're interested in working with an interactive example, take a look at Reqres. Reqres provides mock data for interfacing with a RESTful API and the actions you can take when interacting with an API.

Okay, now that we have that covered, let's take a look at microservices.

What Is a Microservice?

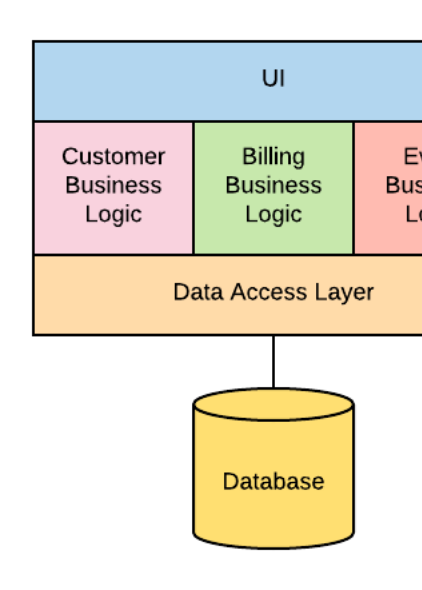
Wikipedia defines a microservice as:

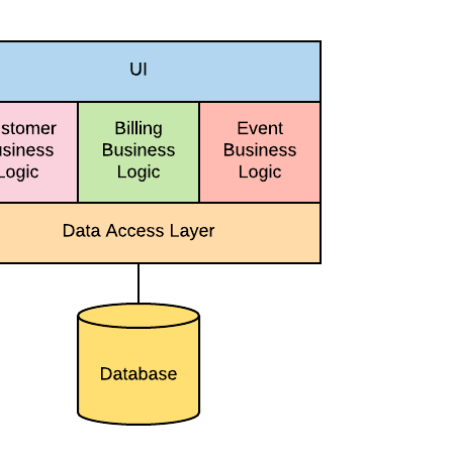
> a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight.

But before we dig deeper into what microservices are and how they can be useful, let's take a quick look into the monolith. Understanding how microservices differ from monoliths will give you a better sense of the benefits of moving to a microservices architecture.

The Precursor to Microservices: Monoliths

In the early days of software development (and continuing in many large enterprise environments today), there's the concept of a monolith. A monolith is a single application that holds a full collection of functionality, serving as one place to store everything. Architecturally, it looks like this:





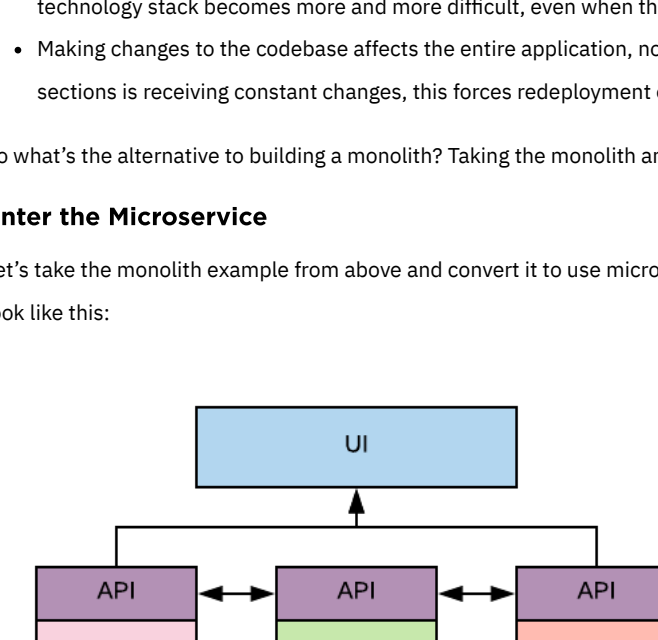
All of the components of the application reside in one area, including the UI layer, the business logic layer, and the data access layer. Building applications in a monolith is an easy and natural process, and most projects start this way. But adding functionality to the codebase causes an increase in both the size and complexity of the monolith, and allowing a monolith to grow large comes with disadvantages over time. Some of these include:

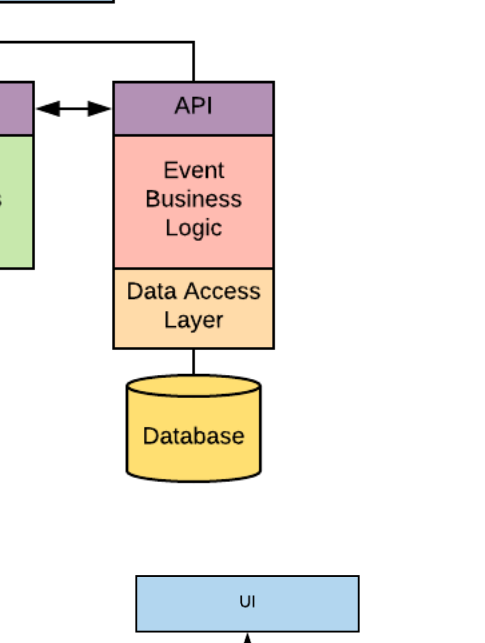
- Risk of falling into the big ball of mud anti-pattern, not having any rhyme or reason in their architecture and difficult to understand from a high level.
- Restriction of the technology stack inside the monolith. Especially as the application grows, the ability to move to a different technology stack becomes more and more difficult, even when the technology proves to no longer be the best choice.
- Making changes to the codebase affects the entire application, no matter how small. For example, if just one of the business logic sections is receiving constant changes, this forces redeployment of the entire application, wasting time and increasing risk.

So what's the alternative to building a monolith? Taking the monolith and breaking it up into microservices.

Enter the Microservice

Let's take the monolith example from above and convert it to use microservices. In that case, the application architecture would change to look like this:





There are a few key takeaways from this re-architecture:

- The broken out sections of the business logic, each encompassing a microservice. Instead of having a single boundary for the entire application, the application is broken into pieces. The complexity of the application is reduced, as the different services have well-defined interactions with each other. For example, this allows for the capability to assign align teams to each individual service, encompassing responsibility in an abstracted piece.
- The UI layer from before only needs to interface with the customer and event microservices, removing a dependency for the billing microservice on the UI.
- The billing microservice does not need to store data, so it doesn't have a data access layer or a database. Instead, it interacts and processes data directly from both the customer and event microservices.

With this kind of architecture comes a whole host of advantages:

- It's easier to separate concerns. These boundaries between areas help with development (you only need to concern yourself with your microservice, not the entire application) and with understanding the architecture of the application.
- Unlike with a monolith, a microservice can use a different tech stack as needed. Considering rewriting everything in a new language? Just change one microservice to use the new tech stack, assess the benefits gained, and determine whether to proceed.
- Deployments of the application as a whole become more focused. Microservices give you the flexibility to deploy different services as needed.

In the example above, notice the API sitting alongside the other portions of the microservice? We'll get into that. It's finally time to talk about the differences between APIs and microservices.

The Difference Between APIs and Microservices

Here are the main differences between APIs and microservices:

- An API is a contract that provides guidance for a consumer to use the underlying service.
- A microservice is an architectural design that separates portions of a (usually monolithic) application into small, self-containing services.

By definition, this means an API is usually a *portion* of a microservice, allowing for interaction with the microservice itself. Another way to think about this is that the API serves as a contract for interactions within the microservice, presenting the options available for interacting with the microservice.

However, if we look at the microservices diagram above, we can see that each microservice is built slightly differently based on its needs. Here are a few examples of different functions a microservice can have:

- Providing CRUD operations for a particular entity type, such as a customer, event, etc. This service would hold the ability to persist data in a database.
- Providing a means to accept parameters and return results based on (potentially intense) computations. The billing microservice above may take information on an event or customer and return the billing information required, without needing to store data.

With the above example, you can probably see that a microservice is capable of being more than just an API for a system. An entire application can encompass a series of microservices that use their own APIs for communication with each other. In addition, each of these microservices can abstract its own functionality, drawing logical boundaries for responsibility in the application and separating concerns to make for a more maintainable codebase.

Conclusion

Hopefully now you have a better understanding of what both APIs and microservices are. Code maintainability and quality are both key parts of a successful IT strategy. Microservices help you stay true to them. They keep your teams agile and help you meet customer demands by producing high-quality, maintainable code.

Are you working in a monolith codebase? Think about taking a portion of that monolith and moving it into a microservice of its own. Once you do that, you should be able to see the benefits of using microservices. In fact, you might even decide to convert the entire thing.

This post was written by Dave Farinelli. Dave is a senior software engineer with over eight years of experience. His specialty is in providing enterprise-level solutions for healthcare and insurance clients. Dave holds a B.S. in computer engineering from Kettering University in Flint, Michigan.

Like this article? Follow us on LinkedIn, Twitter, YouTube or Facebook to see the content we post.

Read more about Cyber Security

- How to Merge Log Files