Apply the Strangler Application pattern to microservices applications

How to get microservices to work with your real-world apps

Kyle G. Brown

February 13, 2017

For many development teams, microservices may sound like a great idea, but they're not sure how they can start using it since they are currently using huge monolithic applications. That's where the Strangler Application pattern can help. This article shows you a few of the ramifications of applying the Strangler Application pattern -- when it applies, when it doesn't, and what it means for release management.

To view this video, **Microservices TV Episode 16: Strangling the Monolith** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

If you're like me, you're constantly reading articles and blogs about new development tools, new development and operations practices, or new architectural principles. The problem is that too often, these practices or architectural principles might look great on paper, or perhaps would work out fine if you were to implement them in a brand new (greenfield) application, but it's not quite as clear how you would implement them in an existing application.

However, we live in a world where **brownfield** applications—existing apps that were not built according to all the latest approaches, or using the latest tools—are more common. This is particularly true of the microservices architecture. Most of the development teams I talk to think microservices sound like a great idea, but they're not sure how they can start using them since they are currently working with huge monolithic applications.

Thus, it's great when you come across a solution that's really helpful in situations like this, and that's true of Martin Fowler's **Strangler Application pattern**. And even though he wrote it prior to the development of the microservices architecture, this pattern offers excellent guidance to teams that want to move to a microservices approach.

What is the Strangler Application pattern?

In a 2004 article on his website, Martin Fowler defined the Strangler Application pattern as a way of handling the release of refactored code in a large web application.

The Strangler Application is based on an analogy to a vine that strangles a tree that it's wrapped around. The idea is that you use the structure of a web application—the fact that web apps are built out of individual URIs that map functionally to different aspects of a business domain—to divide an application into different functional domains, and replace those domains with a new microservices-based implementation one domain at a time. This creates two separate applications that live side by side in the same URI space. Over time, the newly refactored application "strangles" or replaces the original application until finally you can shut off the monolithic application. Thus, the Strangler Application steps are **transform**, **coexist**, and **eliminate**:

- **Transform**—Create a parallel new site (for example, in Bluemix or even in your existing environment but based on more modern approaches).
- **Coexist**—Leave the existing site where it is for a time. Redirect from the existing site to the new one so the functionality is implemented incrementally.
- **Eliminate**—Remove the old functionality from the existing site (or simply stop maintaining it) as traffic is redirected away from that portion of the old site.

The great thing about applying this pattern is that it creates incremental value in a much faster timeframe than if you tried a "big bang" migration in which you update all the code of your application before you release any of the new functionality. It also gives you an incremental approach for adopting microservices—one where, if you find that the approach doesn't work in your environment, you have a simple way to change direction if needed.

When does the pattern work and when does it not work?

Of course, you can't assume that a pattern will apply to every situation. There are a few prerequisites that must be in place in order for this approach to be successful:

- **Web- or API-based monolith:** This is the first requirement. The whole purpose of the Strangler Application is that you have to have a way of easily moving back and forth between new and old functionality. If you're working with a web application, then the URL structure of that app gives you a framework for choosing what parts of the system are implemented in which way. On the other hand, Thick client applications or similar Native Mobile applications are not as well suited to this approach since they don't necessarily have a structure that allows you to pull the application apart in the same way.
- **Standardized URL structure (true use of URLs):** Even though web applications all work according to some standards imposed by the structure of the web (such as the use of HTTP and HTML), there are many different application architectures that can be used to implement those web applications. However, there is still a lot of leeway within this approach. For instance, when there is an intermediate layer underneath the server requests (as with a portal approach), you may have a problem if you are using URLs to divide the application up; the decision for switching and routing isn't made at the browser level, but deeper in the application, which is more complicated.
- **Meta UIs:** When the UI is a "meta" UI (such as when it is business-process based and/or constructed on the fly), the approach will still work, but the chunk is larger and must be implemented all at once. That's because you have to rework either the entire generation

framework or, at the very least, all of those places that interact with your current, non-microservices-based business model.

How not to apply the pattern

Just as there are cases where the pattern applies, there are likewise cases where it does *not* apply —or at least does not apply easily. These cases include:

- **Don’t apply it one page at a time:** The smallest sliver is a single microservice. You want to avoid having two different data access methods for your data at the same time to avoid consistency problems.
- **Don’t apply it all at once:** If you do your entire application at one time you’re not really applying the Strangler Pattern, are you?

How to apply the pattern

You have to interleave two different aspects of your application refactoring: refactoring your back end to the microservices design (the inside part), and refactoring your front end to accommodate the microservices and to make any new functional changes that are driving the refactoring (the outside part).

Inside part

1. **Identify the Bounded Contexts in your application design**—In his book *Domain-Driven Design*, Eric Evans states that a Bounded Context “defines the context within which a model applies.” A Bounded Context is a shared conceptual framework that constrains the meaning of a number of entities within a larger set of business models. In an airline application, flight booking would be one Bounded Context, while the airline loyalty program would be a different Bounded Context. Although they may share terms (such as “flight”), the way in which those terms are used and defined can be quite different.
2. **Choose the Bounded Context that is the smallest and least costly to refactor**—Rank your other Bounded Contexts in order of complexity from least complex to most complex. You’ll want to start with the least complex Bounded Contexts in order to prove out the value of your refactoring (and shake out any problems in adopting the process) before you take on the more complex (and potentially costly) refactoring tasks.
3. **Conceptually plan out the microservices within the context**—You plan out a rough URL structure and overall responsibility of your microservices by applying the Entity, Aggregate, and Service patterns from [Evans]. Note that at this point, we’re not actually trying to do a detailed design of all of these microservices! We’re just trying to get an understanding of which microservices likely exist so that we can use that “guess” in the next set of steps.

Outside part

The process of planning out the application of the Strangler Application pattern in the middle of a brownfield project is often complicated by the fact that you're not only changing your back-end implementation to incorporate microservices, but there are usually lots of front-end UI changes happening as part of the project as well. In fact, it's often the front-end changes (for instance, implementing a site as a single-page application or a mobile application) that drives the desire to

implement microservices in the first place. With a microservices architecture, you can't assume that you know exactly how the UI design is going to turn out, or exactly what changes the UI team are going to implement a priori. But you *can* make some simple assumptions that can help you get started with your design:

1. **Analyze the relationships between the screens in your existing UI**—What this practically means is that you need to group screens together into a set of logical **Flows**. A Flow is a simple concept—it's a group of screens that connect from one to another to perform a single user task like booking a flight in a travel website, or checking out a cart in a retail website. The notion here is that you have a limited number of Flows in your UI and that each Flow will correspond to a small set of microservices that represent the concepts that are being manipulated.
2. **Apply the principle of least astonishment to the related aspects of the model manipulation**—The principle of least astonishment is an old rule in software engineering and user interface design. It states that *the system should act the way that most people think it should act.* Try making some simple assumptions that apply this principle; for instance, if you have flows that query a particular domain aspect, you can assume that it is likely that this domain aspect should also be updated, even if you don't have a flow for that.
3. **Size your Chunk**—We discuss the concept of a chunk more in depth below, but for now, you can assume that it is a group of related epics. The issue to consider here is *Is it logical for a user to work within a single flow, or are the connections between flows so tight that a group of flows is the smallest reasonable element that a user can adopt?* For instance, it might be reasonable for profile management to be split out on its own from the rest of a retailing site because it doesn't connect tightly with the rest of the site. However, it would not be reasonable to split payment selection away from the rest of the checkout process and give payment a completely different UI than the rest of the checkout process. That would violate the principle of least astonishment, so applying that principle would lead you to keep those two together in a single logical unit.
4. **Choose whether to release an entire chunk at a time, or each chunk as a series of slivers**—We cover this aspect in the next section.

A release process for Strangler microservices

In the Bluemix Garage Method, we use certain concepts from Agile methods to help us organize our work. For instance, a **Hill** (or an MVP definition) is broken down through the inception process into **User Stories** that represent individual implementable elements of the MVP. Related User Stories are grouped into **Epics** that describe higher levels of system functionality.

That brings us back to the challenge of gradual system replacement through the Strangler Application pattern. Epics are grouped together into **Chunks**, which are tied together either by implementation restrictions or user-experience relationships that become initial large-scale rollout units. In most cases, a Chunk is bounded by a UI flow that defines a user task, and all the parts within that flow are implemented in a consistent way.

Squads implement Epics: A single Squad may implement one or more Epics within a Chunk, but the smallest element of implementation responsibility for a Squad is an Epic. The Squad adds the

User Stories that define that Epic to an ongoing **Ranked Backlog** of prioritized User Stories for implementation.

However, a Chunk is still a pretty big rollout unit when you're trying to receive intermediate feedback from a set of sponsor users. Therefore, we define a smaller release element that can be used internally to gain that feedback. A **Sliver** is the smallest releasable unit that incorporates a microservice and the changes that surround the microservice to make it usable to the end user. So even though a chunk may be an entire flow for booking a ticket on a bus or an airplane and all of the microservices needed to implement that flow, a Sliver may be a single page within that flow that represents a step such as showing the confirmation of the ticket booking, plus the microservice needed to retrieve that confirmation. So your Chunk would be made up of Slivers that are continuously released and made available in a limited way for feedback, but only released to the end user when the entire task represented by the Chunk is workable.

Conclusion

In this article, I've shown you a few of the ramifications of applying the Strangler Application pattern—when it applies, when it doesn't, and what it means for release management. I've also introduced a couple of new terms that we use in our consulting practice in the Bluemix Garage and that I hope you will find helpful. In particular, I've spotlighted a process for planning our your microservices development when applying the Strangler Application, and for managing the releases of functionality in your application.

Related topics

- Sign up for a free Bluemix trial
- Martin Fowler blog about the Strangler Application pattern
- "Domain-Driven Design: Tackling Complexity in the Heart of Software" by Eric Evans
- MicroservicesTV video series
- Introduction to microservices
- Implement a microservice-based architecture in Bluemix