TAYLOR JONES
SOFTWARE CHEF AT IZEA

Breaking up your Monolith into Microservices

About the Author.

Software Chef at IZEA. Slowly simmering in the Florida heat. Taylor is a contributor to the Codeship blog. He mainly writes about how to transition to a microservice structure with your software team.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

Breaking up your Monolith into Microservices

Advocates for microservices tend to pitch the pattern as a remedy for bloated, messy, monolithic ailments. However, is the cost of transforming a monolith into a series of microservices worth it? There are arguments for the merits and sins of each, but what about microservices attracts so many companies and developers?

The most common use case for switching to a microservice-based infrastructure is converting an existing monolith into microservices. I like to refer to this process as "decomposing" an application, because we're taking a tightly coupled piece of code and breaking it down into smaller services.

This could be anything from just adding one or two microservices to support your main application, to completely reorganizing every aspect of your software architecture into microservices.

ABOUT THIS EBOOK

In this book you will learn about the basics of "decomposing" a monolith into microservices and why I think this is a worthwhile effort for your project.

Take It Slow

When developing software, we often split up large tasks into smaller sections in order to better carve out a step-by-step plan. With microservices, this plan of attack should look quite similar. We want to break down the decomposition of the application into smaller pieces. In many ways, we're treating it like a compost pile: We want to gradually add the right ingredients over time so it decomposes into rich and useful resources.

When you first introduce the idea of microservices to your development team, odds are they'll be quite receptive to it. Microservices are trendy in the development community as of late (this book was published in November 2016), so people are pretty eager to give them a shot. The more romanticized versions of the pitch tend to go like this:

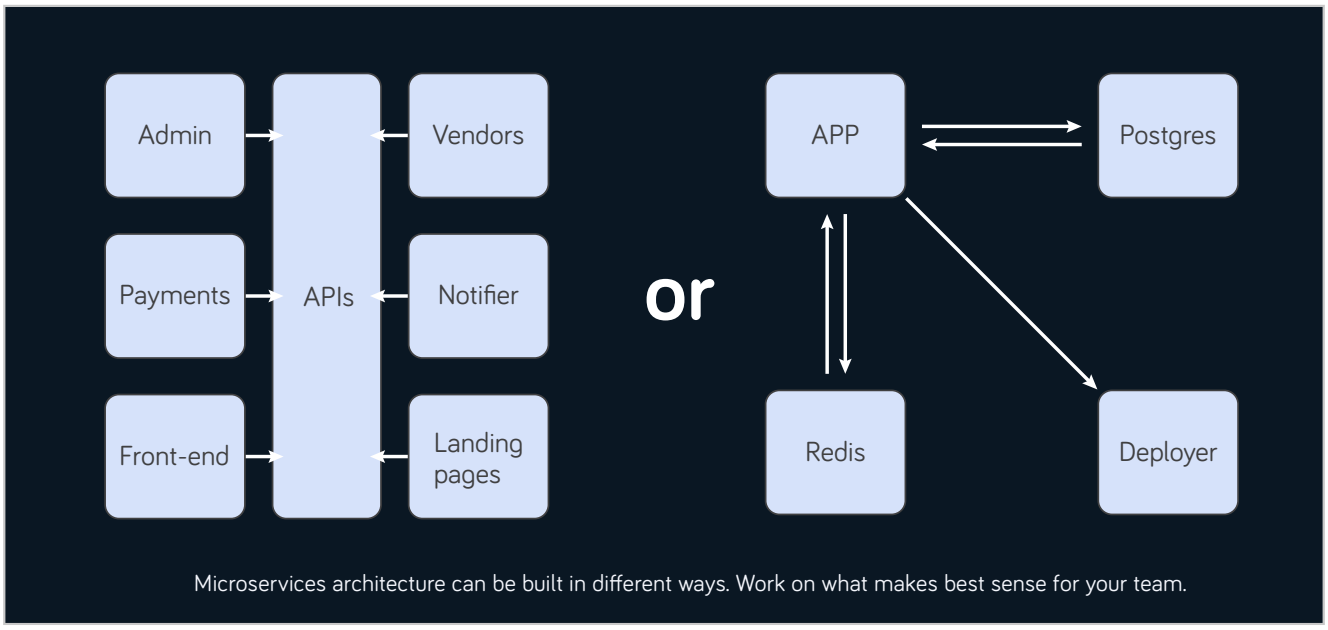
"Changing one thing won't break 200 other things in your app anymore. Isn't that awesome?"

However, the journey to a less-coupled and dependent app is far from a two-week process. Odds are, you're going to spend months (possibly even years) breaking down each aspect of your application into a separate, functional service.

To learn more about how you can structure conversations with your team about breaking up your monolith watch

Codeship's webinar video about "Building Applications with Microservices" here. Ethan Jones and Manuel Weiss give good pointers to conversations starters that help you think about the right thinks when trying to decompose your monolith.

It's also very important to look at decomposition as a long-term technical debt task. Clients and investors don't really care whether you're extracting an old feature into a microservice. They're concerned if you're delivering a product they expect. You and your team will need to push forward the developmental priorities of the company while having a taste of what microservice architecture looks like and getting a feeling for where to push forward with decomposition next. Again, this is very much like technical debt that you're always aware of.



Microservices architecture can be built in different ways. Work on what makes best sense for your team.

Don't feel the need to start decomposing everything immediately. Take your time and work on what makes best sense for your team.

It works best when you look at decomposing your application as a long-term effort that might takes years to yield tangible dividends. Each piece of code extraction should be looked at as a surgical decision. It's possible to make a really dangerous mistake if you decompose too fast!

Freedom Isn't Always Free

Microservices architecture opens many more doors for using different development languages and frameworks. Microservices limit the restrictions of a limited set of languages shared by an app and allow developers to create using any means they please!

This obviously can lead to some very exciting decisions for developers to make. However, you might want to pump the brakes a bit before choosing the new language or framework with which to develop that microservice.

Learning new programming languages, frameworks, and tools can be a really fulfilling personal experience. They're a lot of fun to poke around with and learn from. However, even though I might be comfortable building personal

projects in some of these languages and frameworks, **something that scales in a production-ready manner for large corporations is an entirely different ballgame.**

You and two of your coworkers might think writing a service in Rust is a great idea. On paper, this idea can seem pretty sweet. You get to implement a pretty exciting language, all while making your business a better place.

But it's important to note that nothing stays the same when it comes to your team and coworkers. **Team churn is a real thing** whether you choose to think about it or not. Currently, your company might have the resources and skills among developers to create a really great microservice in Rust. However, will your team continue to invest in developers with the skillsets to maintain that Rust microservice? Depending on how big your investment is on that microservice, you might find yourself accidentally married to it.

When you're choosing a language, you're choosing to invest in dealing with it for the foreseeable future.

Think beyond the current state and size of your team. In the worst case, you'll be heading down a frustrating path that could waste a good deal of development resources to fully correct.

The DevOps Burden

IIf you've decided to move forward with decomposing an application, I strongly encourage you to increase your investments in DevOps. A trademark characteristic of microservices is that they work really well with continuous deployment. There's a lot to be said about the process. Microservices enable you to quickly deploy your smaller services whenever they're ready. This makes for a better deployment process in theory, since you aren't constantly redeploying a huge chunk of code.

That's in theory.

The more services you create and maintain, the more lengthy your deployment process will become. You'll need to be able to effectively deploy multiple services within quick succession. If you only have a limited set of developers trained in deployment, things can get out of hand quickly.

This is why it's important to teach developers how to own their code from development to deployment. Ideally, they should be comfortable simply sending off their code to

be deployed and monitored. They should also be ready for the worst case scenario of deploying and monitoring a service themselves. In addition, your developers should be well versed and aware of common production errors for a microservice. I've experienced numerous deploy errors that could have been solved more easily if more developers were familiar with common production errors. It could be the difference of a few seconds of downtime to a few minutes.

Furthermore, this complexity can be aggravated by the stability and diversity of the languages that you use. To reiterate my previous point, not all new languages and frameworks deploy as well as they function in development. It's wise to take smaller risks when using cutting-edge languages and frameworks; some of them might not be as production-ready as you would like to believe.

There are a lot of moving parts when it comes to decomposing a monolithic application. The key to making this work is to let teams efficiently plan out their time and efforts. They'll need time to work on new features and fixes, while also taking effective steps to decompose the application.

I encourage you to read Camille Fournier's article about architectural patterns of microservices in the real world.

With microservices, you have an immense freedom to choose any language, tool, or framework you want. This freedom can be exciting, but you can get caught up in building something that's just cutting edge instead of something that's more practical for your stability and production needs.

You'll also find that the DevOps process is given a different kind of complexity when you decompose a monolith. You gain the advantage of not having to totally redeploy every time you make a relevant change. However, this creates a need to have more support and ownership of the multitude of new services that you're adding to your deployment process.

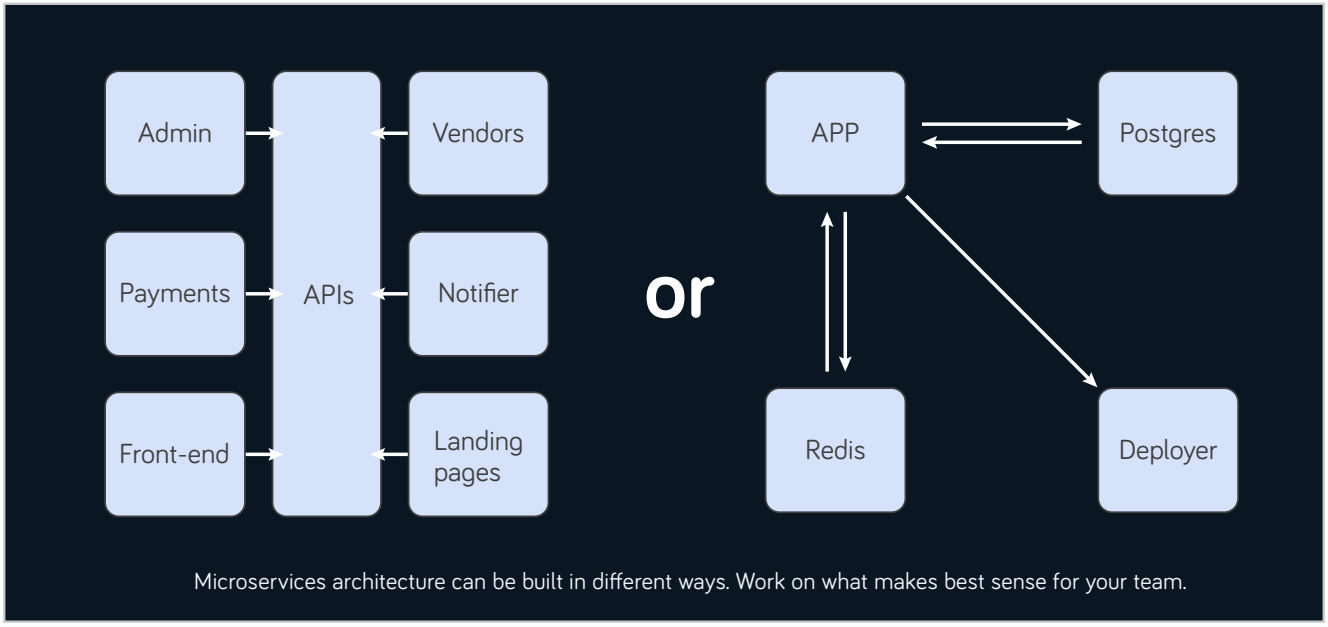
Microservices Best Practices

I ultimately believe that decomposing a monolith into microservices is a worthwhile effort. However, it's a long-term investment that may take a while to yield dividends, so it's important to make sure we're not just barreling ahead with little to no plan.

We need to think about the consumers of these services, and we need to craft good developer experience in order to make working with microservices easy and maintainable. To help us clarify some microservices best practices, let's recall exactly what a microservice is:

- A small service that does one thing well
- Independent
- Owner of its own data

Microservices exist in a larger ecosystem. It's important to not just think about the one microservice. We need to make sure we think about how it connects to the larger system of the network.



Microservices architecture can be built in different ways. Work on what makes best sense for your team.

Make Microservices Easy to Use

The needs of consumers of microservices all overlap. How can we ensure we're making a great microservice for everyone? There are a few things that are essential to creating a good developer experience with a microservice, and you should put enough effort into them to make sure you're doing a good job with them:

- **Documentation** Be clear and concise.
- **Version and revision history** Provide historical records of when things changed and why.
- **Live documentation** If possible, offer live responses (for example: APIs —> JSON response).

Other initiatives that can help make a big difference are:

- creating a dependency graph
- auto-generating documentation
- integration monitoring tools
- logging

We need to be able to surface errors before our service goes down.

Make Sure Your Microservices Can Coexist

In the end, it all comes down to two things:

- consistency
- conventions

Here are a few good questions you can ask yourself in order to check that you're developing your microservice in a sustainable way:

- **How does my microservice fit in with the others?**
- **Are my services capable of working together?**
  Think date formats and so on.
- **Am I taking single responsibility too far?**
  Don't try to force yourself into breaking up your monolithic apps. Sometimes it makes sense to not break everything up.

Your Best Practices for Microservices Checklist

Finally, if you want to build friendly, sustainable microservices, here's a good checklist:

- Helpful documentation
- Built with monitoring and troubleshooting in mind
- Easily deployable and scalable
- Easy to consume
- Coexists with established conventions

Always be sure to make your service easy to deploy and scale. Provide consistent error messaging and make sure your microservice consumers can hit the API directly in a nonproduction environment.

More Codeship Resources.

WEBINAR

An Introduction to Building your Apps with Microservices.

In this webinar you will learn about the advantages and challenges of using Microservices for your Apps.

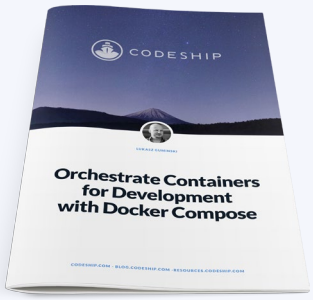
Download this eBook

EBOOKS

Orchestrate Containers with Docker Compose.

In this eBook you will learn how to easily recreate a microservice architecture with Docker Compose.

Download this eBook

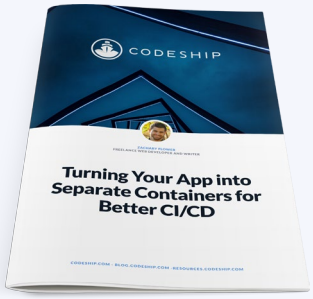


EBOOKS

Turning your App into separate Containers.

In this eBook you will learn how to speed up testing and deploying of your apps.

Download this eBook



About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs. Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.

- Works out of the box
- Preinstalled CI dependencies
- Optimized hosted infrastructure
- Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.

- Customizability & Full Autonomy
- Local CLI tool
- Dedicated single-tenant instances
- Deploy anywhere

LEARN MORE