Sam Newman

Building Microservices
by Sam Newman

Published by O•Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O•Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safaribooksonline.com). For more information, contact our corporate/ institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Mike Loukides and Brian MacDonald
Production Editor: Kristen Brown
Copyeditor: Rachel Monaghan
Proofreader: Jasmine Kwityn

Indexer: Judith McConville
Interior Designer: David Futato
Cove0()Designer:

978-1-491-95035-7

[LSI]

Table of Contents

CHAPTER 1

Splitting the Monolith

We•ve discussed what a good service looks like, and why smaller servers may be better for us. We also previously discussed the importance of being able to evolve the design of our systems. But how do we handle the fact that we may already have a large num, ber of codebases lying about that don•t follow these patterns? How do we go about decomposing these monolithic applications without having to embark on a big-bang rewrite?

The monolith grows over time. It acquires new functionality and lines of code at an alarming rate. Before long it becomes a big, scary giant presence in our organization that people are scared to touch or change. But all is not lost! With the right tools at our disposal, we can slay this beast.

It's All About Seams

We discussed in an earlier chapter that we want our services to be highly cohesive and loosely coupled. The problem with the monolith is that all too often it is the opposite of both. Rather than tend toward cohesion, and keep things together that tend to change together, we acquire and stick together all sorts of unrelated code. Likewise, loose coupling doesn•t really exist: if I want to make a change to a line of code, I may be able to do that easily enough, but I cannot deploy that change without potentially impacting much of the rest of the monolith, and I•ll certainly have to redeploy the entire system.

In his book Working E€ectively with Legacy Code (Prentice-Hall), Michael Feathers defines the concept of a seam/that is, a portion of the code that can be treated in isolation and worked on without impacting the rest of the codebase. We also want to identify seams. But rather than finding them for the purpose of cleaning up our code, base, we want to identify seams that can become service boundaries.

So what makes a good seam? Well, as we discussed previously, bounded contexts
make excellent seams since by definition they represent cohesive and yet loosely cou,
pled boundaries in an organization. So the first step is to start identifying these
boundaries in our code.

Most programming languages provide namespace concepts that allow us to group
similar code together. Java•s package concept is a fairly weak example, but gives us
much of what we need. All other mainstream programming languages have similar
concepts built in, with JavaScript very arguably being an exception.

Breaking Apart MusicCorp

Imagine we have a large backend monolithic service that represents a substantial
amount of the behavior of MusicCorp•s online systems. To start, we should identify
the high-level bounded contexts that we think exist in our organization, as we dis,
cussed in ???. Then we want to try to understand what bounded contexts the mono,
lith maps to. Let•s imagine that initially we identify four contexts we think our
monolithic backend covers:

Catalog
: Everything to do with metadata about the items we offer for sale

Finance
: Reporting for accounts, payments, refunds, and so on

Warehouse
: Dispatching and returning of customer orders, managing inventory levels, and the like

Recommendation
: Our patent-pending, revolutionary recommendation system, which is highly complex code written by a team with more PhDs than the average science lab

The first thing to do is to create packages representing these contexts, and then move
the existing code into them. With modern IDEs, code movement can be done auto,
matically via refactorings, and can be done incrementally while we are doing other
things. You•ll still need tests to catch any breakages made by moving code, however,
especially if you•re using a dynamically typed language where the IDEs have a harder
time of performing refactoring. Over time, we start to see what code fits well, and
what code is le• overand doesn•t really fit anywhere. This remaining code will often
identify bounded contexts we might have missed!

During this process we can use code to analyze the dependencies between these pack,
ages too. Our code should represent -0.01

allow us to see the dependencies between packages graphically. If we spot things that look wrongƒfor example, the warehouse package depends on code in the finance package when no such dependency exists in the real organizationƒthen we can investigate this problem and try to resolve it.

This process could take an afternoon on a small codebase, or several weeks or months when you•re dealing with millions of lines of code. You may not need to sort all code into domain-oriented packages before splitting out your first service, and indeed it can be more valuable to concentrate your effort in one place. There is no need for this to be a big-bang approach. It is something that can be done bit by bit, day by day, and we have a lot of tools at our disposal to track our progress.

So now that we have our codebase organized around these seams, what next?

The Reasons to Split the Monolith

Deciding that you•d like a monolithic service or application to be smaller is a good start. But I would strongly advise you to chip away at these systems. An incremental approach will help you learn about microservices as you go, and will also limit the impact of getting something wrong (and you will get things wrong!). Think of our monolith as a block of marble. We could blow the whole thing up, but that rarely ends well. It makes much more sense to just chip away at it incrementally.

So if we are going to break apart the monolith a piece at a time, where should we start? We have our seams now, but which one should we pull out first? It•s best to think about where you are going to get the most benefit from some part of your code, base being separated, rather than just splitting things for the sake of it. Let•s consider some drivers that might help guide our chisel.

Pace of Change

Perhaps we know that we have a load of changes coming up soon in how we manage inventory. If we split out the warehouse seam as a service now, we could change that service faster, as it is a separate autonomous unit.

Team Structure

MusicCorp•s delivery team is actually split across two geographical regions. One team is in London, the other in Hawaii (some people have it easy!). It would be great if we could split out the code that the Hawaii team works on the most, so it can take full ownership. We•ll explore this idea further in ???.

Figure 1-1. Splitting out our repository layers

Having the database mapping code colocated inside the code for a given context can help us understand what parts of the database are used by what bits of code. Hiber, nate, for example, can make this very clear if you are using something like a mapping file per bounded context.

This doesn•t give us the whole story, however. For example, we may be able to tell that the finance code uses the ledger table, and that the catalog code uses the line item table, but it might not be clear that the database enforces a foreign key relationship from the ledger table to the line item table. To see these database-level constraints, which may be a stumbling block, we need to use another tool to visualize the data. A great place to start is to use a tool like the freely available SchemaSpy, which can gen, erate graphical representations of the relationships between tables.

All this helps you understand the coupling between tables that may span what will eventually become service boundaries. But how do you cut those ties? And what about cases where the same tables are used from multiple different bounded contexts? Handling problems like these is not easy, and there are many answers, but it is doable.

Coming back to some concrete examples, let•s consider our music shop again. We have identified four bounded contexts, and want to move forward with making them nization so they can see how we•re doing. We want to make the reports nice and easy to read, so rather than saying, „We sold 400 copies of SKU 12345 and made $1,300,… we•d like to add more information about what was sold, instead saying, „We sold 400 copies of Bruce Springsteen•s Greatest Hits and made $1,300.… To do this, our reporting code in the finance package will reach into the line item table to pull out the title for the SKU. It may also have a foreign key constraint from the ledger to the line item table, as we see in Figure 1-2

Figure 1-2. Foreign key relationship

So how do we fix things here? Well, we need to make a change in two places. First, we need to stop the finance code fromreaching into the line item table, as this table really belongs to the catalog code, and we don•t want database integration happening once catalog and finance are services in their own rights. The quickest way to address this is rather than having the code in finance reach into the line item table, we•ll expose the data via an API call in the catalog package that the finance code can call. This API call will be the forerunner of a call we will make over the wire, as we see in Figure 1-3

Figure 1-3. Post removal of the foreign key relationship

At this point it becomes clear that we may well end up having to make two database calls to generate the report. This is correct. And the same thing will happen if these are two separate services. Typically concerns around performance are now raised. I have a fairly easy answer to those: how fast does your system need to be? And how fast is it now? If you can test its current performance and know what good perfor, mance looks like, then you should feel confident in making a change. Sometimes making one thing slower in exchange for other things is the right thing to do, espe, cially if *slower* is still perfectly acceptable.

But what about the foreign key relationship? Well, we lose this altogether. This becomes a constraint we need to now manage in our resulting services rather than in the database level. This may mean that we need to implement our own consistency check across services, or else trigger actions to clean up related data. Whether or not this is needed is often not a technologist•s choice to make. For example, if our order service contains a list of IDs for catalog items, what happens if a catalog item is removed and an order now refers to an invalid catalog ID? Should we allow it? If we do, then how is this represented in the order when it is displayed? If we don•t, then how can we check that this isn•t violated? These are questions you•ll need to get answered by the people who define how your system should behave for its users.

Example: Shared Static Data

Figure 1-4. Country codes in the database

I have seen perhaps as many country codes stored in databases (shown in Figure 1-4) as I have written `StringUtils` classes in-house in Java projects. This seems to imply that we plan to change the countries our system supports way more frequently than we•ll deploy new code, but whatever the real reason, these examples of shared static data being stored in databases come up a lot. So what do we do in our music shop if all our potential services read from the same table like this?

Well, we have a few options. One is to duplicate this table for each of our packages, with the long-term view that it will be duplicated within each service also. This leads to a potential consistency challenge, of course: what happens if I update one table to reflect the creation of Newmantopia off the east coast of Australia, but not another?

A second option is to instead treat this shared, static data as code. Perhaps it could be in a property file deployed as part of the service, or perhaps just as an enumeration. The problems around the consistency of data remain, although experience has shown that it is far easier to push out changes to configuration files than alter live database tables. This is often a very sensible approach.

A third option, which may well be extreme, is to push this static data into a service of its own right. In a couple of situations I have encountered, the volume, complexity, and rules associated with the static reference data were sufficient that this approach was warranted, but it•s probably overkill if we are just talking about country codes!

Personally, in most situations I•d try to push for keeping this data in configuration files or directly in code, as it is the simple option for most cases.

Example: Shared Data

Now let•s dive into a more complex example, but one that can be a common problem when you•re trying to tease apart systems; shared mutable data. Our finance code tracks payments made by customers for their orders, and also tracks refunds given to them when they return items. Meanwhile, the warehouse code updates records to show that orders for customers have been dispatched or received. All of this data is displayed in one convenient place on the website so that customers can see what is going on with their account. To keep things simple, we have stored all this informa, tion in a fairly generic customer record table, as shown in Figure 1-5.

So both the finance and the warehouse code are writing to, and probably occasionally reading from, the same table. How can we tease this apart? What we actually have here is something you•ll see often ƒ a domain concept that isn•t modeled in the code, and is in fact implicitly modeled in the database. Here, the domain concept that is missing is that of Customer.

Figure 1-6. Recognizing the bounded context of the customer

We need to make the current abstract concept of the customer concrete. As a transi, ent step, we create a new package called Customer. We can then use an API to expose Customer code to other packages, such as finance or warehouse. Rolling this all the way forward, we may now end up with a distinct customer service.

Example: Shared Tables

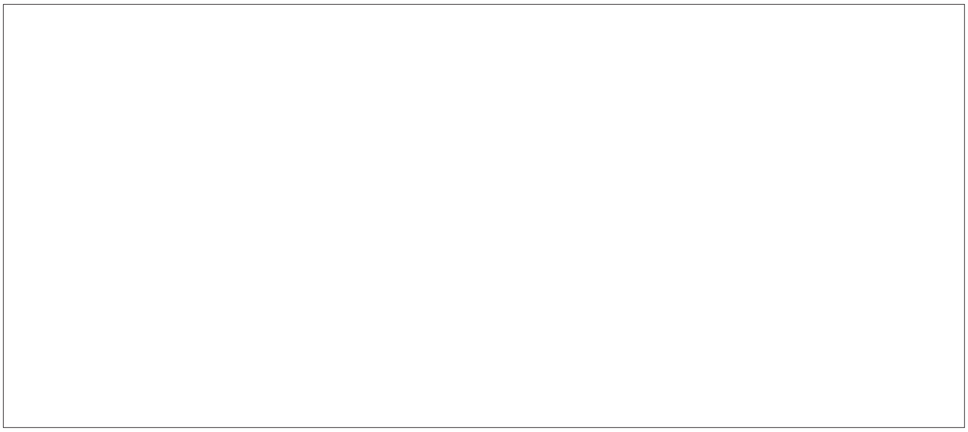
Figure 1-7. Tables being shared between different contexts

Figure 1-7 shows our last example. Our catalog needs to store the name and price of the records we sell, and the warehouse needs to keep an electronic record of inven, tory. We decide to keep these two things in the same place in a generic line item table. Before, with all the code merged in together, it wasn•t clear that we are actually con, flating concerns, but now we can see that in fact we have two separate concepts that could be stored differently.

Figure 1-8. Pulling apart the shared table

The answer here is to split the table in two as we have in Figure 1-8, perhaps creating a stock list table for the warehouse, and a catalog entry table for the catalog details.

Figure 1-9. Staging a service separation

With a separate schema, we•ll be potentially increasing the number of database calls to perform a single action. Where before we might have been able to have all the data we wanted in a single SELECT statement, now we may need to pull the data back from two locations and join in memory. Also, we end up breaking transactional integrity when we move to two schemas, which could have significant impact on our applications; we•ll be discussing this next. By splitting the schemas out but keeping the application code together, we give ourselves the ability to revert our changes or continue to tweak things without impacting any consumers of our service. Once we are satisfied that the DB separation makes sense, we can then think about splitting out the application code into two services.

Transactional Boundaries

Transactions are useful things. They allow us to say these events either all happen together, or none of them happen. They are very useful when we•re inserting data into a database; they let us update multiple tables at once, knowing that if anything fails everything gets rolled back, ensuring our data doesn•t get into an inconsistent state. Simply put, a transaction allows us to group together multiple different activities that take our system from one consistent state to anotherƒeverything works, or nothing changes.

Transactions don•t just apply to databases, although we most often use them in that context. Messages brokers, for example, have long allowed you to post and receive messages within transactions too.

customer order has been created, and also put an entry into a table for the warehouse team so it knows an order that needs to be picked for dispatch. We•ve gotten as far as grouping our application code into separate packages, and have also separated the customer and wharehouse parts of the schema well enough that we are ready to put them into their own schemas prior to separating the application code.

Within a single transaction in our existing monolithic schema, creating the order and inserting the record for the warehouse team takes place within a single transaction, as shown inFigure 1-10.

Figure 1-10. Updating two tables in a single transaction

But if we have pulled apart the schema into two separate schemas, one for customer-related data including our order table, and another for the warehouse, we have lost this transactional safety. The order placing process now spans two separate transac, tional boundaries, as we see inFigure 1-11. If our insert into the order table fails, we can clearly stop everything, leaving us in a consistent state. But what happens when the insert into the order table works, but the insert into the picking table fails?

Figure 1-11. Spanning transactional boundaries for a single operation

Try Again Later

The fact that the order was captured and placed might be enough for us, and we may decide to retry the insertion into the warehouse•s picking table at a later date. We could queue up this part of the operation in a queue or logfile, and try again later. For some sorts of operations this makes sense, but we have to assume that a retry would fix it.

In many ways, this is another form of what is called *eventual consistency*. Rather than using a transactional boundary to ensure that the system is in a consistent state when the transaction completes, instead we accept that the system will get itself into a con, sistent state at some point in the future. This approach is especially useful with busi, ness operations that might be long-lived. We•ll discuss this idea in more depth in ??? when we cover scaling patterns.

Abort the Entire Operation

Another option is to reject the entire operation. In this case we have to put the system back into a consistent state. The picking table is easy, as that insert failed, but we have a committed transaction in the order table. We need to unwind this. What we have to do is issue a *compensating transaction*, kicking off a new transaction to wind back what just happened. For us, that could be something as simple as issuing a `DELETE` statement to remove the order from the database. Then we•d also need to report back via the UI that the operation failed. Our application could handle both aspects within a monolithic system, but we•d have to consider what we could do when we split up the application code. Does the logic to handle the compensating transaction live in the customer service, the order service, or somewhere else?

rithms are hard to get right, so I•d suggest you avoid trying to create your own. Instead, do lots of research on this topic if this seems like the route you want to take, and see if you can use an existing implementation.

So What to Do?

All of these solutions add complexity. As you can see, distributed transactions are hard to get right and can actually inhibit scaling. Systems that eventually converge through compensating retry logic can be harder to reason about, and may need other compensating behavior to fix up inconsistencies in data.

When you encounter business operations that currently occur within a single transac, tion, ask yourself if they really need to. Can they happen in different, local transac, tions, and rely on the concept of eventual consistency? These systems are much easier to build and scale (we•ll discuss this more in ??)

If you do encounter state that really, really wants to be kept consistent, do everything you can to avoid splitting it up in the first place. Try really hard. If you really need to go ahead with the split, think about moving from a purely technical view of the pro, cess (e.g., a database transaction) and actually create a concrete concept to represent the transaction itself. This gives you a handle, or a hook, on which to run other oper, ations like compensating transactions, and a way to monitor and manage these more complex concepts in your system. For example, you might create the idea of an „in-process-order… that gives you a natural place to focus all logic around processing the order end to end (and dealing with exceptions).

Reporting

As we•ve already seen, in splitting a service into smaller parts, we need to also poten, tially split up how and where data is stored. This creates a problem, however, when it comes to one vital and common use case: reporting.

A change in architecture as fundamental as moving to a microservices architecture will cause a lot of disruption, but it doesn•t mean we have to abandon everything we do. The audience of our reporting systems are users like any other, and we need to consider their needs. It would be arrogant to fundamentally change our architecture and just ask them to adapt. While I•m not suggesting that the space of reporting isn•t ripe for disruptionƒit certainly isƒthere is value in determining how to work with existing processes first. Sometimes we have to pick our battles.

The Reporting Database

Reporting typically needs to group together data from across multiple parts of our organization in order to generate useful output. For example, we might want to enrich the data from our general ledger with descriptions of what was sold, which we get from a catalog. Or we might want to look at the shopping behavior of specific, high-value customers, which could require information from their purchase history and their customer profile.

In a standard, monolithic service architecture, all our data is stored in one big data, base. This means all the data is in one place, so reporting across all the information is actually pretty easy, as we can simply join across the data via SQL queries or the like. Typically we won•t run these reports on the main database for fear of the load gener, ated by our queries impacting the performance of the main system, so often these reporting systems hang on a read replica as shown in Figure 1-12.

Figure 1-12. Standard read replication

With this approach we have one sizeable upsideƒthat all the data is already in one place, so we can use fairly straightforward tools to query it. But there are also a couple of downsides with this approach. First, the schema of the database is now effectively a shared API between the running monolithic services and any reporting system. So a change in schema has to be carefully managed. In reality, this is another impediment that reduces the chances of anyone wanting to take on the task of making and co-coordinating such a change.

Second, we have limited options as to how the database can be optimized for either use caseƒbacking the live system or the reporting system. Some databases let us make optimizations on read replicas to enable faster, more efficient reporting; for example, MySQL would allow us to run a different backend that doesn•t have the overhead of managing transactions. However, we cannot structure the data differ, ently to make reporting faster if that change in data structure has a bad impact on the running system. What often happens is that the schema either ends up being great for one use case and lousy for the other, or else becomes the lowest common denomina, tor, great for neither purpose.

Finally, the database options available to us have exploded recently. While standard relational databases expose SQL query interfaces that work with many reporting tools, they aren•t always the best option for storing data for our running services. What if our application data is better modeled as a graph, as in Neo4j? Or what if we•d

of all the customers, making a separate call for each one. Not only could this be ineffi, cient for the reporting system, it could generate load for the service in question too.

While we could speed up some of the data retrieval by adding cache headers to the

Figure 1-14. Making use of materialized views to form a single monolithic reporting schema

Here, of course, the complexity of integration is pushed deeper into the schema, and will rely on capabilities in the database to make such a setup performant. While I think data pumps in general are a sensible and workable suggestion, I am less con,
vinced that the complexity of a segmented schema is worthwhile, especially given the challenges in managing change in the database.

Alternative Destinations

On one project I was involved with, we used a series of data pumps to populate JSON files in AWS S3, effectively using S3 to masquerade as a giant data mart! This approach worked very well until we needed to scale our solution, and at the time of writing we are looking to change these pumps to instead populate a cube that can be integrated with standard reporting tools like Excel and Tableau.

Event Data Pump

In ???, we touched on the idea of microservices emitting events based on the state change of entities that they manage. For example, our customer service may emit an event when a given customer is created, or updated, or deleted. For those microservi,
ces that expose such event feeds, we have the option of writing our own event sub,
scriber that pumps data into the reporting database, as shown in Figure 1-15.

make Cassandra easy to work with, much of which the company has shared with the rest of the world via numerous open source projects. Obviously it is very important that the data Netflix stores is properly backed up. To back up Cassandra data, the standard approach is to make a copy of the data files that back it and store them somewhere safe. Netflix stores these files, known as SSTables, in Amazon•s S3 object store, which provides significant data durability guarantees.

Netflix needs to report across all this data, but given the scale involved this is a non, trivial challenge. Its approach is to use Hadoop that uses SSTable backup as the source of its jobs. In the end Netflix ended up implementing a pipeline capable of processing large amounts of data using this approach, which it then open sourced as the Aegis, thus project. Like data pumps, though, with this pattern we still have a coupling to the

Summary

We decompose our system by finding seams along which service boundaries can emerge, and this can be an incremental approach. By getting good at finding these seams and working to reduce the cost of splitting out services in the first place, we can continue to grow and evolve our systems to meet whatever requirements come down the road. As you can see, some of this work can be painstaking. But the very fact that it can be done incrementally means there is no need to fear this work.

So now we can split our services out, but we•ve introduced some new problems too. We have many more moving parts to get into production now! So next up we•ll dive into the world of deployment.