Case Study: a Decade of Microservices at a Financial Firm

Jonathan Allen

22-28 minutes

Key Takeaways

- Microservices are most appropriate for stateful processes, not easily scaled web servers.
- Each microservice needs its own network credentials and a unique number, or application key, that people can easily recognize in logs and records.
- A common application framework is necessary for ensuring timely updates of dependencies.
- Centralized configuration is vital for reducing errors when settings change.
- Microservices that make synchronous calls to other microservices should be avoided when possible.
- If your database provides tools for building an abstraction layer, use them.

Microservices are the hot new architectural pattern, but the problem with "hot" and "new" is that it can take years for the real costs of an architectural pattern to be revealed. Fortunately, the pattern isn't new, just the name is. So, we can learn from companies that have been doing this for a decade or more.

In this case study, we'll be looking at how microservices were implemented at a financial firm. When I had joined the firm, they had already been using the pattern for five to ten years and it continued for another half-decade under my supervision.

What is a Microservice?

Since the term "microservice" didn't exist when we started down this road, we called our 85+ microservices "process applications". The defining characteristic of our microservices was that each would perform one and only one activity. These activities included:

- Watching a network drive for files to process.
- Listening to a TCP port for messages from business partners. (Imagine someone streaming stock prices to you.)
- Aggregating messages from other services into a standardized format before processing them and pushing the results to databases and caches.
- Listening a message queue. (MSMQ at the time, but there are better options available today.)
- Hosting a specialized cache.
- Hosting a HTTPS port. (We used HTTP.sys for this, as it made it easier to deal with server and client certificates.)
- Watching a database table for records to process. (Usually polling on a timer, but these days I recommend using SqlDependency instead; the Chain ORM has this feature built in.)
- Passing WCF messages from desktop applications to a TCP port connected to business partners. (Again, newer technologies such as gRPC may be more appropriate.)

The one thing we didn’t do was divide each controller class in our websites into a separate deliverable. Partially because MVC was just barely getting started and we, like most MS shops at the time, were still mostly focused on WebForms for our website. But there were other reasons for not dividing up websites that I’ll cover later.

Microservices and Security

One of the policies I instituted was a hard rule that every microservice get its own Active Directory domain account. This meant each application’s configuration file didn’t include usernames and passwords for internal resources. (We did occasionally need them for external resources such as websites and FTP servers run by partner companies.)

Previously, we had the bad habit of using a single login shared across every application. And like far too many companies today, that single login was effectively an administrator for the databases. This made deployment easy, but also made the risk of catastrophe incredibly high.

Each microservice must have its own network credentials.

My reasoning for making this change was it allowed me to lock down the database. Each application could only perform the actions it actually needed to perform. This prevented bad outcomes such as a hacker using a vulnerability in our pricing server to download or destroy our user table.

Later we learned we accidentally built a fairly robust documentation system. Before we instituted the separate network credentials policy, we had to do exhaustive code searches to see what impact a change would have on our applications. Now we merely have to query a database to get a complete list of the tables, views, and stored procedures it has access to. This dramatically reduced the amount of regression testing needed when altering the database.

Naming and Numbering Each Service

While a unique name for each service is important, names are not enough to eliminate confusion; especially when you have many applications with very similar names.
The way we solved this problem was by assigning each service its own "application key". This is a short, numeric identifier you can use in things like configuration files and database audit columns.

Because each application key was registered in a single table, we also got a complete inventory of every application, microservice, tool, etc. that could possibly be in use. And we never found ourselves in a situation where we didn’t know what software we built or where it was running.

It may seem like I’m worrying about nothing, but as a consultant I often find myself working with clients who literally can’t tell me how many websites and microservices they have in operation. With no central repository, we find ourselves having to interview every developer, network admin, and department just to figure out what’s touching the database.

Application Frameworks

If you are going down the microservices route, it is critical you develop your own, company specific, application framework. And you need to dogmatically use this application framework in every microservice. Otherwise, there is no way you can keep up with the maintenance.

In the .NET world, your application framework could consist of a single NuGet package that defines the following:

- Configuration
- Logging
- Dependency injection
- Low-level database access
- Health monitoring
- Security components
- Common libraries

Now I’m not saying you have to write custom implementations of all of these. In fact, you probably want to use pre-defined packages that handle most, if not all, of these pieces. But you still need an application framework to stich everything together and define which version of each library will be used.

A common trap developers fall into is the belief they can keep up with changing technology. And maybe they can when using a monolithic architecture. But when the next heartbleed incident occurs, you don’t have time to check the version number of each and every library on dozens or hundreds of microservices. Instead, you want to update the application framework to the safe version of the library, then push that application framework update onto all of the microservices.

This last point is where an application framework differs from a "starter kit". A starter kit defines where you begin, but applications will quickly diverge from it. And this is not acceptable when you are managing more than a handful of services. When I was working for this company, I was personally responsible for managing over 50 services. There is no way I could have done that if each one was setup slightly differently and I had to relearn it each time I touched it.

The application framework also gives you a much-needed layer of abstraction. For example, let us say you discover that SmtpClient is giving you trouble and you want to switch to MailKit. The application framework can provide an adapter that mimics the SmtpClient API, but actually sends emails via MailKit. This will markedly reduce the amount of effort needed to port all of your microservices to a different dependency.

Self-reporting Services

Services need to be able to self-report their status. External monitoring is still important, but self-reporting gives you a level of insight that external monitoring tools don’t have access to.

In our case each service fired a heartbeat on a timer. This was logged directly to the database because our database was our most robust resource. (It had a multi-site failover cluster and really, if it went down most of our services were not going to be useful anyways.) With mature products such as Azure Application Insights, you probably don't want to roll your own like we did.

Our heartbeat included the following information:

- Application Key
- Application Name
- Application Version
- Application Framework Version
- Hosting Server (or virtual machine)
- Current Timestamp
- Start time (when the service was last restarted)

Another process would monitor the heartbeat table, watching for services that weren't checking in. It could also detect when someone accidentally installed a service with production settings onto a non-production host.

From there you can build on additional information such as the service's opinion of its own health. For example, a service that listens for streaming messages from partner companies can indicate when it feels too much time has passed since the last message. Or a file processor can report how many records it last imported and what percentage of the records were found to be flawed.

This can be done cheaply at first with a generic "status" column. Later more sophisticated metrics added as needed in dedicated reporting tables.

Virtual Machines? Docker? Containers? Serverless?

Something I'm not going to talk about is how you host your service.

That’s an implementation detail which developers don’t really need to think about. Your only important question is "Can I run multiple copies of this service at the same time?", all other decisions can be deferred.

When I started, half our microservices were running as desktop applications. As in you literally needed to log into a server and start them. The reason I was originally hired is I had experience converting applications into Windows Services.

But they didn’t have to be Windows Services. I could have taken the same code, dropped it into a web server managed by IIS. Or packed it into a Docker container. For some of them, a serverless host would have worked equally well. As long as any interaction between the service and its host is abstracted by the application framework, you have more than enough flexibility here.

Centralized Configuration

Over the years we tried numerous ways to deal with configuration. We tried manual configuration, build scripts, even storing configuration directly in source control. Ultimately, all the methods failed for the same reason: they weren’t scalable. As the number of microservices increase, the time it takes to alter a shared setting grows linearly and the odds of making a mistake grows geometrically.

The solution that ultimately worked for us was centralized configuration. Applications were built with a single configuration value named "Environment". This pointed it to a file share with the list of environments and matching database server names. For example: "production:MainDB.company.net".

The configuration database held every setting needed for each application. But there’s more to it than that. Common settings, such as were to send critical alerts, were defined first. Then each application had the option to override a common setting or introduce an application-specific setting. Furthermore, specific instances of the microservice could further override the setting.

This meant that when we needed to change something common, we only had to modify one entry and every service would load the new setting. (When this happened was application specific, as some microservices would be working on long running data processing jobs.)

This wasn't a perfect solution. There was still the possibility that a non-production instance of a microservice could be pointed to the production environment. What we really needed separate AD accounts for production and non-production services, but that was beyond our organizational capabilities at the time.

And really, the file server holding environment/database server names was silly. We should have just put the database name directly into the configuration file and relied on DNS if the database server needed to be moved.

Command and Control

One of the aspects I am quite proud of is the command and control. Each microservice exposed a WCF port that our in-house monitoring tools could attach to. We used WCF because it supported two-way messaging, allowing for the live streaming of log data.

But logs were not the important feature, many tools can do that. The command and control port also allowed us to see diagnostic metrics that were not reported in the logs. For example, the number items currently held in memory waiting to be processed. We could also see the configuration settings actually being used by the service, which may differ from those in the config file as these services were only restarted once every few months.

Previously I mentioned that our microservices were often triggered by a timer or file system watcher. The command and control tools allowed us to tell the service, "run now" or "reprocess file X". This was especially important when troubleshooting processes that ran infrequently, sometimes only once an hour or even only once a day. No one wants to wait 50 minutes to rerun a test after fixing some bad production data. And speaking of data ...

Database Access

As mentioned before, each microservice had its own network credentials. These were used by the database to determine which stored procedures were accessible to the service. I would like to stress the word "stored procedure". No service had direct table access.

This may seem weird in the era of ORMs, but ORMs such as NHibernate and Entity Framework have a problem with coupling. They expect to have full and unfettered access to the database and tightly bind their object model to the table schema.

This can work when you have one application touching the database, but not when you have a dozen, let alone a hundred. With that many applications we needed to protect our database from excessively tight coupling and our services from breaking changes.

Stored procedures do this by creating an encapsulation layer. The stored procedure itself forms an API and the tables are merely its hidden implementation detail. When changes to tables are needed, the stored procedure is either updated in a backwards compatible way or its version number is incremented, allowing the old and new version to temporarily run side-by-side.

And because each stored procedure was accompanied by the list of microservices that had permission to access it, we had a readymade list of regression tests to be performed whenever the stored procedure is modified.

You could simulate this by tossing a set of REST servers between your actual services and the database. But it would only be a simulation and would not get all the performance and isolation benefits of a fully encapsulated database. Which brings us to our next topic.

Web Servers and Microservices

Previously I mentioned that we didn't apply the microservices approach to our web servers. The reason why is simply because we saw zero benefit in it and whole lot of cost. Here is the chain of logic we used:

Ease of Deployment

Microservices are supposed to make deployments easier. This pre-supposes deployments are hard. So why are deployments hard? Deployments are hard because it takes time to safely shut-down stateful services, our "process applications". They have to be properly timed as to not disrupt employees and in many cases must be done during the 'idle' period between processing two jobs.

Web servers are not stateful. If you deploy a new version, IIS will gracefully handle any remaining requests and direct new requests to the newly loaded version. Which means deployments can happen at any time, though we prefer to do them after business hours to allow for additional testing.

Reliability and Scalability

Microservices are also meant to improve reliability and scalability. This makes a lot of sense, as you don't want process A to crash process B thru Z when it fails. And while process C may be limited

to a single instance, process D and E may allow for scaling out. So, breaking each of these processes into separate services makes a lot of sense.
But stateless web servers are perfectly scalable. As long as you have sufficient rack space, you can continue adding more and more web servers. Eventually other limitations such as database and network capacity will come into play, but the web servers themselves have no limits.

The Cost of Microservices

Even if there's no benefit, what's the harm?
Well, if a web server has to call other web servers, that's extra latency and more points of failure to contend with. Performance and reliability can't help but be reduce as any broken link in the chain can disrupt the whole process.

That's not the problem the hurt us most when we experimented with micro-web servers. The real issue was dependency management and deployments. Whenever we changed something in the UI, we invariably had to change the backend database calls to match. But since the UI and backend were in separate servers, both had to be modified and deployed at the same time. The UI web server was strongly coupled to every backend web server.

The other problem was composition, the combining of data from multiple sources. Ultimately everything came from the same database, but with a separate backend web server for customers, their clients, inventory, transactions, etc. it became a real nightmare to decide which web server should expose a given piece of data. Invariably, whatever we picked would be 'wrong' and have to be moved somewhere or be stitched together with data from other services.

So, in the end we aborted the experiment and continued to allow the web server to have direct database access.

Data Ownership and Change Logging

Returning to the topic of data, one of the most commonly asked questions when it comes to microservices is "Which service changed this record?". Invariably this is asked dozens of times a day by countless developers trying to understand the complex web of systems they work on.

You may be thinking, "I use a microservice in front of the database, so only that microservice could have changed it." If the microservice didn’t decide to change the data on its own, if it’s just responding to requests from other services, that’s not a real answer.

The way we solved this problem is through the "application key" concept. Every update to the database recorded both the user who initiated the change and the application key of the web server or microservice that actually made the change.

Of course, there is still the possibility of multiple microservices making contradictory changes to the same data. Thus, the concept of data ownership is still important. While in theory any microservice could read any field, whenever possible we designed the systems so only one microservice could write to a given field. When successful, this made reasoning about design changes much easier. When we failed and multiple services did write to the same field, at least we could use the list of stored procedures as a starting point for planning the next steps.

Decoupling and Message Queues

A much-touted benefit of microservices is they "decouple" parts of your system. In some cases, we found this to be true, in others it was a farce. To understand why, we must first define the term "decouple".

The working definition I use is "component A is not coupled to component B if a failure in B does not cause a failure in A". This is not the only valid definition, but in terms of estimating the robustness of a system I find it to be the most useful.

For example, we had various feeds from our partner companies. Each feed was its own microservice so if one feed failed it wouldn't take down the others. Each feed also wrote to two systems, a quote server and a pricing server. The quote server was essentially a cache that held raw data from the feeds. It was fast, but of limited use. The offering server ran the raw quotes through the complex bond price calculations needed prior to posting the offerings to our database. The database would be used by the web server, while the quote server was used by traders on near real-time terminals.

Originally, whenever the quote server or offering server failed it would bring down all of the feeds with it. Likewise, one errant feed that swamped a server could effectively break the other feeds. So, by the definition above, not only was each feed coupled to the servers, each feed was also coupled to all the other feeds.

It should be noted that swamping happened more often than you might suppose. Unlike the stock market, where each stock is independent, bonds are often pegged at each other's price or yield. For example, when the price of a US Treasury bond changes it can cause tens of thousands of corporate and municipal bonds to likewise change price.
In order to solve this problem for the quote server, we simply changed the messages to be one-way and in the event of a failure the feed would simply discard the message. This may seem reckless, but quotes change rapidly so missing one wasn't really a significant issue as it would be quickly replaced by the next. Furthermore, the quote server was such a simple piece of software that nothing short of a hardware failure would bring it down.

For the far more complicated offering server, we needed a more robust solution. It needed a way to read data at its own pace, with protections from feeds that could overwhelm it. To solve this, we turned to MS Message Queues. While today there is undoubtedly better message queueing products available, at the time MSMQ was more than sufficient to break the tight coupling between the feeds and the offering server.

In the case of a web server, this decoupling becomes much harder. If the user is waiting for information, one cannot simply send an asynchronous message to a queue to be processed at an indeterminate time. In some cases, the web server can proceed without the requested information, say YouTube or Netflix occasionally not showing your watch list. But in our case there was no ‘optional’ data and any failure in a microservice would mean the website fails as well.

Versioning Considerations

The vast majority of our microservices we truly independent of each other, only relying on message queues or the database itself. For them, the versioning issue was mitigated by ensuring messages and stored procedures were backwards compatible. If either changed in an unsafe way, then a new message queue or stored procedure was created instead of modifying the original.

For the few microservices that did directly communicate with each other, a more comprehensive plan was needed. Each service exposed an endpoint that provided its version number and environment. When a microservice started up, it would check the version numbers for all the other services that it depended on. If the check failed, it would refuse to start.

Conclusions

Microservices are most appropriate for stateful processes, not easily-scaled web servers. Use them in cases where you can isolate components, either fully or via message queues. Avoid using them when it will just create a chain of synchronous service calls. And most importantly, design a way to catalog your inventory of microservices so they don't get away from you.

About the Author

Jonathan Allen got his start working on MIS projects for a health clinic in the late 90's, bringing them up from Access and Excel to an enterprise solution by degrees. After spending five years writing automated trading systems for the financial sector, he became a consultant on a variety of projects including the UI for a robotic warehouse, the middle tier for cancer research software, and the big data needs of a major real estate insurance company. In his free time he enjoys studying and writing about martial arts from the 16th century.