WHAT ARE MICROSERVICES?

Introduction To Microservices

Microservice architecture, or simply microservices, is a distinctive method of developing software systems that has grown in popularity in recent years. In fact, even though there isn't a whole lot out there on what it is and how to do it, for many developers it has become a preferred way of creating enterprise applications. Thanks to its scalability, this architectural method is considered particularly ideal when you have to enable support for a range of platforms and devices—spanning web, mobile, Internet of Things, and wearables—or simply when you're not sure what kind of devices you'll need to support in an increasingly cloudy future.

While there is no standard, formal definition of microservices, there are certain characteristics that help us identify the style. Essentially, microservice architecture is a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

How the services communicate with each other depends on your application's requirements, but many developers use HTTP/REST with JSON or Protobuf. DevOps professionals are, of course, free to choose any communication protocol they deem suitable, but in most situations, REST (Representational State Transfer) is a useful integration method because of its comparatively lower complexity over other protocols.

To begin to understand microservices architecture, it helps to consider its opposite: the monolithic architectural style. Unlike microservices, a monolith application is always built as a single, autonomous unit. In a client-server model, the server-side application is a monolith that handles the HTTP requests, executes logic, and retrieves/updates the data in the underlying database. The problem with a monolithic architecture, though, is that all change cycles usually end up being tied to one another. A modification made to a small section of an application might require building and deploying an entirely new version. If you need to scale specific functions of an application, you may have to scale the entire application instead of just the desired components. This is where microservices can come to the rescue.

SOA vs. Microservices

"Wait a minute," some of you may be murmuring over your morning coffee, "isn't this just another name for SOA?" Service-Oriented Architecture (SOA) sprung up during the first few years of this century, and microservice architecture (abbreviated by some as MSA) bears a number of similarities. Traditional SOA, however, is a broader framework and can mean a wide variety of things. Some microservices advocates reject the SOA tag altogether, while others consider microservices to be simply an ideal, refined form of SOA. In any event, we think there are clear enough differences to justify a distinct "microservice" concept (at least as a special form of SOA, as we'll illustrate later).

The typical SOA model, for example, usually has more dependent ESBs, with microservices using faster messaging mechanisms. SOA also focuses on imperative programming, whereas microservices architecture focuses on a responsive-actor programming style. Moreover, SOA models tend to have an outsized relational database, while microservices frequently use NoSQL or micro-SQL databases (which can be connected to conventional databases). But the real difference has to do with the architecture methods used to arrive at an integrated set of services in the first place.

Since everything changes in the digital world, agile development techniques that can keep up with the demands of software evolution are invaluable. Most of the practices used in microservices architecture come from developers who have created software applications for large enterprise organizations, and who know that today's end users expect dynamic yet consistent experiences across a wide range of devices. Scalable, adaptable, modular, and quickly accessible cloud-based applications are in high demand. And this has led many developers to change their approach.

Examples of Microservices

As Martin Fowler points out, Netflix, eBay, Amazon, the UK Government Digital Service, realestate.com.au, Forward, Twitter, PayPal, Gilt, Bluemix, Soundcloud, The Guardian, and many other large-scale websites and applications have all evolved from monolithic to microservices architecture.

Netflix has a widespread architecture that has evolved from monolithic to SOA. It receives more than *one billion* calls every day, from more than 800 different types of devices, to its streaming-video API. Each API call then prompts around five additional calls to the backend service.

Amazon has also migrated to microservices. They get countless calls from a variety of applications—including applications that manage the web service API as well as the website itself—which would have been simply impossible for their old, two-tiered architecture to handle.

The auction site eBay is yet another example that has gone through the same transition. Their core application comprises several autonomous applications, with each one executing the business logic for different function areas.

Understanding Microservice Architecture

Just as there is no formal definition of the term microservices, there's no standard model that you'll see represented in every system based on this architectural style. But you can expect most microservice systems to share a few notable characteristics.

First, software built as microservices can, by definition, be broken down into multiple component services. Why? So that each of these services can be deployed, tweaked, and then redeployed independently without compromising the integrity of an application. As a result, you might only need to change one or more distinct services instead of having to redeploy entire applications. But this approach does have its downsides, including expensive remote calls (instead of in-process calls), coarser-grained remote APIs, and increased complexity when redistributing responsibilities between components.

Second, the microservices style is usually organized around business capabilities and priorities. Unlike a traditional monolithic development approach—where different teams each have a specific focus on, say, UIs, databases, technology layers, or server-side logic—microservice architecture utilizes cross-functional teams. The responsibilities of each team are to make specific products based on one or more individual services communicating via message bus. That means that when changes are required, there won't necessarily be any reason for the project, as a whole, to take more time or for developers to have to wait for budgetary approval before individual services can be improved. Most development methods focus on projects: a piece of code that has to offer some predefined business value, must be handed over to the client, and is then periodically maintained by a team. But in microservices, a team owns the product for its lifetime, as in Amazon's oft-quoted maxim "You build it, you run it."

Third, microservices act somewhat like the classical UNIX system: they receive requests, process them, and generate a response accordingly. This is opposite to how many other products such as ESBs (Enterprise Service Buses) work, where high-tech systems for message routing, choreography, and applying business rules are utilized. You could say that microservices have smart endpoints that process info and apply logic, and dumb pipes through which the info flows.

Fourth, since microservices involve a variety of technologies and platforms, old-school methods of centralized governance aren't optimal. Decentralized governance is favored by the microservices community because its developers strive to produce useful tools that can then be used by others to solve the same problems. A practical example of this is Netflix—the service responsible for about 30% of traffic on the web. The company encourages its developers to save time by always using code libraries established by others, while also giving them the freedom to flirt with alternative solutions when needed. Just like decentralized governance, microservice architecture also favors decentralized data management. Monolithic systems use a single logical database across different applications. In a microservice application, each service usually manages its unique database.

Fifth, like a well-rounded child, microservices are designed to cope with failure. Since several unique and diverse services are communicating together, it's quite possible that a service could fail, for one reason or another (e.g., when the supplier isn't available). In these instances, the client should allow its neighboring services to function while it bows out in as graceful a manner as possible. For obvious reasons, this requirement adds more complexity to microservices as compared to monolithic systems architecture.

Finally, microservices architecture is an evolutionary design and, again, is ideal for evolutionary systems where you can't fully anticipate the types of devices that may one day be accessing your application. This is because the style's practitioners see decomposition as a powerful tool that gives them control over application development. A good instance of this scenario could be seen with *The Guardian's* website (prior to the late 2014 redesign). The core application was initially based on monolithic architecture, but as several unforeseen requirements surfaced, instead of revamping the entire app the developers used microservices that interact over an older monolithic architecture through APIs.

To sum up: Microservice architecture uses services to componentize and is usually organized around business capabilities; focuses on products instead of projects; has smart end points but not-so-smart info flow mechanisms; uses decentralized governance as well as decentralized data management; is designed to accommodate service interruptions; and, last but not least, is an evolutionary model.

Now let's take a closer look at how all of it actually plays out in practice…

How Microservice Architecture Works

"If you wish to converse with me," said Voltaire, "define your terms." Just as there is more than one programming language, there are many terms to describe similar concepts used by different developers. So to follow our brief overview of microservices here, it will help to have at least a basic grasp of the following concepts:

- Object Oriented Programming (OOP)—*a modern programming paradigm (see also SOLID)*
- Web service / API—*a way to expose the functionality of your application to others, without a user interface*
- Service Oriented Architecture (SOA)—*a way of structuring many related applications to work together, rather than trying to solve all problems in one application*
- Systems—*in the general sense, meaning any collection of parts that can work together for a wider purpose*
- Single Responsibility Principle (SRP)—*the idea of code with one focus*
- Interface Segregation Principle (ISP)—*the idea of code with defined boundaries.*

1) Monoliths and Conway's Law

To begin with, let's explore Conway's Law, which states: *"Organizations which design systems…are constrained to produce designs which are copies of the communication structures of these organizations."*

Imagine Company X with two teams: *Support* and *Accounting*. Instinctively, we separate out the high risk activities; it's only difficult deciding responsibilities like customer refunds. Consider how we might answer questions like "Does the Accounting team have enough people to process both customer refunds and credits?" or "Wouldn't it be a better outcome to have our Support people be able to apply credits and deal with frustrated customers?" The answers get resolved by Company X's new policy: Support can apply a *credit*, but Accounting has to *process a refund* to return money to a customer. The roles and responsibilities in this interconnected system have been successfully split, while gaining customer satisfaction and minimizing risks.

Likewise, at the beginning of designing any software application, companies typically assemble a *team* and create a *project*. Over time, the team grows, and multiple projects on the same codebase are completed. More often than not, this leads to competing projects: two people will find it difficult to work at cross purposes in the same area of code without introducing tradeoffs. And adding more people to the equation only makes the problem worse. As Fred Brooks puts it, nine women can't make a baby in one month.

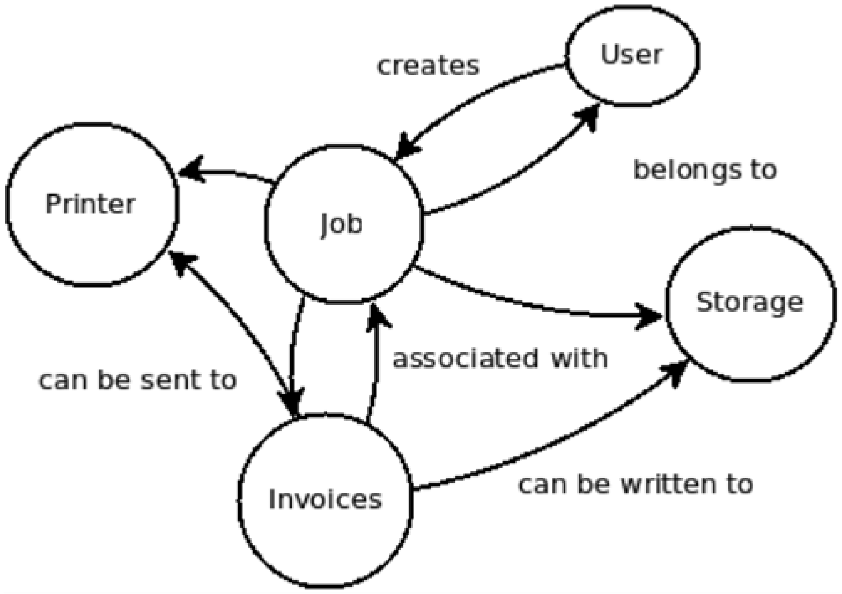
Moreover, in Company X or in any dev team, priorities frequently shift, resulting in management and communication issues. Last month's highest priority item may have caused our team to push hard to ship code, but now a user is reporting an issue, and we no longer have time to resolve it because of *this* month's priority. This is the most compelling reason to adopt SOA, including the microservices variety. Service-oriented approaches recognize the frictions involved between change management, domain knowledge, and business priorities, allowing dev teams to explicitly separate and address them. Of course, this in itself is a tradeoff—it requires coordination—but it allows you to centralize friction and introduce efficiency, as opposed to suffering from a large number of small inefficiencies.

Most importantly, smartly implementing an SOA or microservice architecture forces you to apply the Interface Separation Principle. Due to the connected nature of mature systems, when isolating issues of concern, the typical approach is to find a seam or communication point and then draw a dotted line between two halves of the system. Without careful thought, however, this can lead to accidentally creating two smaller but growing monoliths, now connected with some kind of bridge. The consequence of this can be marooning important code on the wrong side of a barrier: Team A doesn't bother to look after it, while Team B needs it, so they reinvent it.

2) Microservices: Avoiding the Monoliths

We've named some problems that commonly emerge; now let's begin to look at some solutions.

How do you deploy relatively independent yet integrated services without spawning accidental monoliths? Well, suppose you have a large application, as in the sample from our Company X below, and are splitting up the codebase and teams to scale. Instead of finding an entire section of an application to split off, you can look for something on the *edge* of the application graph. You can tell which sections these are because nothing depends on them. In our example, the arrows pointing to Printer and Storage suggest they're two things that can be easily removed from our main application and abstracted away. Printing either a Job or Invoice is irrelevant; a Printer just wants printable data. Turning these—Printer and Storage—into external services avoids the monoliths problem alluded to before. It also makes sense as they are used multiple times, and there's little that can be reinvented. Use cases are well known from past experience, so you can avoid accidentally removing key functionality.



3) Service Objects and Identifying Data

So how do we go from monoliths to services? One way is through *service objects*. Without removing code from your application, you effectively just begin to structure it as though it were completely external. To do that, you'll first need to differentiate the *actions* that can be done and the *data* that is present as inputs and outputs of those actions. Consider the code below, with a notion of *doing something useful* and a *status of that task*.

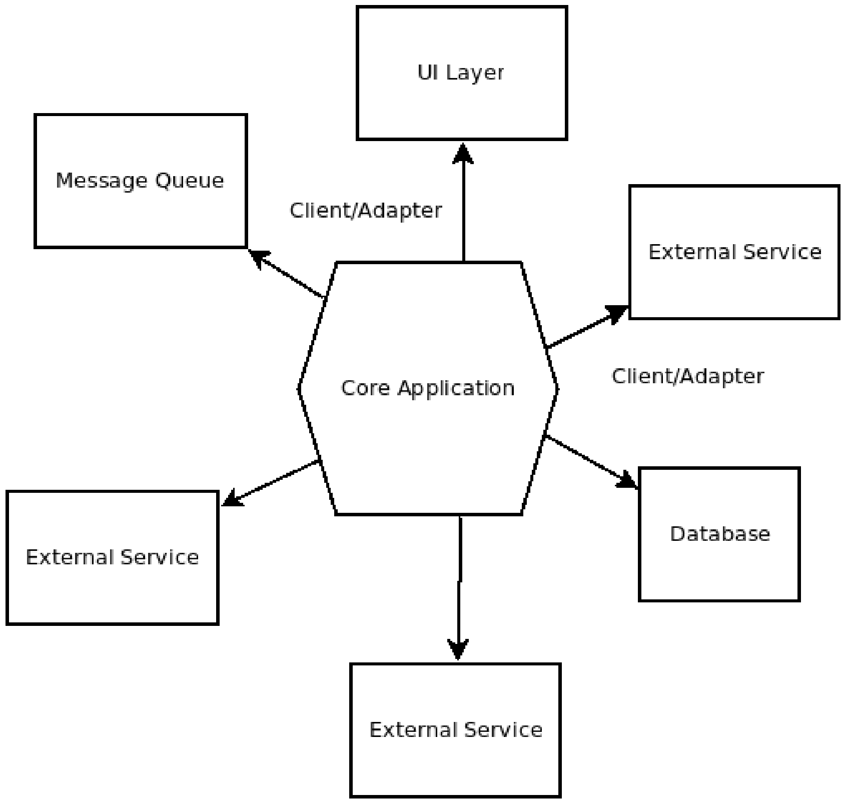
```
# A class to model a core transaction and execute it
class Job
  def initialize
    @status = 'Queued'
  end

  def do_useful_work
    ....
    @status = 'Finished'
  end

  def finished?
    return @status == 'Finished'
  end

  def ready?
    return @status == 'Queued'
  end
end
```

To prepare this to begin looking like a microservice, what's next?

```
# Service to do useful work and modify a status
class JobService
  def do_useful_work(job_status)
    ….

    job_status.finish!

   return job_status
  end
end

# A model of our Job’s status
class JobStatus
  def initialize
    @status = ‘Queued’
  end

  def finished?
    return @status == ‘Finished’
  end

  def ready?
    return @status == ‘Queued’
  end

  def finish!
    @status = ‘Finished’
  end
end
```

Now we’ve distinguished two distinct classes: one that models the data, and one that performs the operations. Importantly, our JobService class has little or no state—you can call the same actions over and over, changing only the data, and expect to get consistent results. If JobService somehow started taking place over a network, our otherwise monolithic application wouldn’t care. Shifting these types of classes into a library, and substituting a network client for the previous implementation, would allow you to transform the existing code into a scalable external service.

This is Hexagonal Architecture, where the core of your application and the coordination is in the center, and the external components are orchestrated around it to achieve your goals.



(You can read more about service objects and hexagonal architecture here and here.)

4) Coordination and Dumb Pipes

Now let's take a closer look at what makes something a microservice as opposed to a traditional SOA.

Perhaps the most important distinction is *side effects*. Microservices avoid them. To see why, let's look at an older approach: Unix pipes.

LS | WC -L

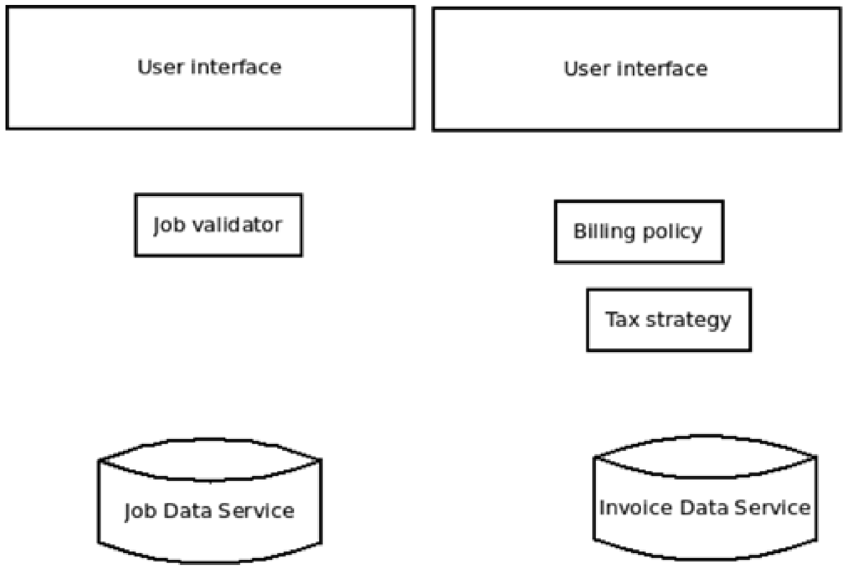
Above, two programs are chained together: the first lists all of the files in a directory, the second reads the number of lines in a stream of input. Imagine writing a comparable program, then having to modify it into the below:

LS | LESS

Composing small pieces of functionality relies on repeatable results, a standard mechanism for input and output, and an exit code for a program to indicate success or lack thereof. We know this works from observational evidence, and we also know that a Unix pipe is a “dumb” interface because it has no control statements. The pipe applies SRP by pushing data from A to B, and it’s up to members of the pipeline to decide if the input is unacceptable.

Let’s go back to Company X’s Job and Invoice systems. Each controls a transaction and can be used together or separately: Invoices can be created for jobs, jobs can be created without an invoice, and invoices can be created without a job. Unlike Unix shell commands, the systems that own jobs and invoices have their own users working independently. But without falling back to a *policy*, it’s impossible to enforce rules for either system globally.

Say we want to extract out the key operations that can be repeatedly executed—the services for sending an invoice, mutating a job status and mutating an invoice status. These are completely separate from the task of *persisting* data.



Here this allows us to wire together the discrete components into two pipelines:

User creates a manual invoice

- Adds data to invoice, status *created*
  — Invokes BillingPolicyService to determine when an invoice is payable for a given customer
- Invoice is issued to customer
- Persists to the invoice data service, status *sent*

User finishes a job, creating an invoice

- Validates job is completable
- Adds data to invoice, status *created*
  — Invokes BillingPolicyService to determine when an invoice is payable for a given customer
- Invoice is issued to customer
- Persists to the invoice data service, status *sent*

The invoice calculation related steps are idempotent, and it's then trivial to compose a *draft invoice* or preview the amounts payable by the customer by leveraging our new dedicated microservices.

Unlike traditional SOA, the difference here is that we have low-level details exposed via a simple interface, as compared to a high-level API call that might execute an entire business action. With a high-level API, in fact, it becomes difficult to rewire small components together, since the service designer has removed many of the seams or choices we can take by providing a one-shot interface.

By this point, the repetition of business logic, policy and rules leads many to traditionally push this complexity into a service bus or singular, centralized workflow orchestration tool. However, the crucial advantage of microservice architecture is not that we *never* share business rules/processes/policies, but that we push them into discrete packages, aligned to business needs. Not only does this mean that policy is distributed, but it also means that *you can change your business processes without risk*.

Microservice Pros and Cons

Microservices are not a silver bullet, and by implementing them you will expose communication, teamwork, and other problems that may have been previously implicit but are now forced out into the open.

One common issue involves sharing schema/validation logic across services. What A requires in order to consider some data valid doesn't always apply to B, if B has different needs. The best recommendation is to apply versioning and distribute schema in shared libraries. Changes to libraries then become discussions between teams. Also, with strong versioning comes dependencies, which can cause more overhead. The best practice to overcome this is planning around backwards compatibility, and accepting regression tests from external services/teams. These prompt you to have a conversation *before* you disrupt someone else's business process, not after.

As with anything else, whether or not microservice architecture is right for you depends on your requirements, because they all have their pros and cons. Here's a quick rundown of some of the good and bad:

Pros

- Microservice architecture gives developers the freedom to independently develop and deploy services
- A microservice can be developed by a fairly small team
- Code for different services can be written in different languages (though many practitioners discourage it)
- Easy integration and automatic deployment (using open-source continuous integration tools such as Jenkins, Hudson, etc.)
- Easy to understand and modify for developers, thus can help a new team member become productive quickly
- The developers can make use of the latest technologies
- The code is organized around business capabilities
- Starts the web container more quickly, so the deployment is also faster
- When change is required in a certain part of the application, only the related service can be modified and redeployed—no need to modify and redeploy the entire application
- Better fault isolation: if one microservice fails, the other will continue to work (although one problematic area of a monolith application can jeopardize the entire system)
- Easy to scale and integrate with third-party services
- No long-term commitment to technology stack

Cons

- Due to distributed deployment, testing can become complicated and tedious
- Increasing number of services can result in information barriers
- The architecture brings additional complexity as the developers have to mitigate fault tolerance, network latency, and deal with a variety of message formats as well as load balancing
- Being a distributed system, it can result in duplication of effort
- When number of services increases, integration and managing whole products can become complicated
- In addition to several complexities of monolithic architecture, the developers have to deal with the additional complexity of a distributed system
- Developers have to put additional effort into implementing the mechanism of communication between the services
- Handling use cases that span more than one service without using distributed transactions is not only tough but also requires communication and cooperation between different teams
- The architecture usually results in increased memory consumption
- Partitioning the application into microservices is very much an art

The Future of Microservice Architecture

Whether or not microservice architecture becomes the preferred style of developers in future, it’s clearly a potent idea that offers serious benefits for designing and implementing enterprise applications. Many developers and organizations, without ever using the name or even labeling their practice as SOA, have been using an approach toward leveraging APIs that could be classified as microservices.

We've also seen a number of existing technologies try to address parts of the segmentation and communication problems that microservices aim to resolve. SOAP does well at describing the operations available on a given endpoint and where to discover it via WSDLs. UDDI is theoretically a good step toward advertising what a service can do and where it can be found. But these technologies have been compromised by a relatively complex implementation, and tend not to be adopted in newer projects. REST-based services face the same issues, and although you can use WSDLs with REST, it is not widely done.

Assuming discovery is a solved problem, sharing schema and *meaning* across unrelated applications still remains a difficult proposition for anything other than microservices and other SOA systems. Technologies such as RDFS, OWL, and RIF exist and are standardized, but are not commonly used. JSON-LD and Schema.org offer a glimpse of what an entire open web that shares definitions looks like, but these aren't yet adopted in large private enterprises.

The power of shared, standardized definitions are making inroads within government, though. Tim Berners Lee has been widely advocating Linked Data. The results are visible through in data.gov and data.gov.uk, and you can explore the large number of data sets available as well-described linked data here. If a large number of standardized definitions can be agreed upon, the next steps are most likely toward *agents*: small programs that orchestrate microservices from a large number of vendors to achieve certain goals. When you add the increasing complexity and communication requirements of SaaS apps, wearables, and the Internet of Things into the overall picture, it's clear that microservice architecture probably has a very bright future ahead.

Written by: Tom Huston

Further Resources

- Ready! API
- Delving into the Microservices Architecture
- Martin Fowler on Microservices
- Microservices.io
- Microservices at InfoQ
- The Great Microservices vs Monolithic Apps Twitter Melee
- Microservices and PaaS