The What, Why, and How of a Microservices Architecture

8 Keys to Help You Get Started Today

Hashmap Follow
Jun 7, 2018 · 13 min read

by Jetinder Singh

For many years now we have been building systems and getting better at it. Several technologies, architectural patterns, and best practices have emerged over those years. Microservices is one of those architectural patterns which has emerged from the world of domain-driven design, continuous delivery, platform and infrastructure automation, scalable systems, polyglot programming and persistence.

What is a Microservices Architecture in a Nutshell?

Robert C. Martin coined the term single responsibility principle which states “gather together those things that change for the same reason, and separate those things that change for different reasons.”

A microservices architecture takes this same approach and extends it to the loosely coupled services which can be developed, deployed, and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.

Key Benefits of a Microservices Architecture

As the constituent services are small, they can be built by one or more small teams from the beginning separated by service boundaries which make it easier to scale up the development effort if need be.

Once developed, these services can also be deployed independently of each other and hence its easy to identify hot services and scale them independent of whole application. Microservices also offer improved fault isolation whereby in the case of an error in one service the whole application doesn't necessarily stop functioning. When the error is fixed, it can be deployed only for the respective service instead of redeploying an entire application.

Another advantage which a microservices architecture brings to the table is making it easier to choose the technology stack (programming languages, databases, etc.) which is best suited for the required functionality (service) instead of being required to take a more standardized, one-size-fits-all approach.

How Do I Get Started with a Microservices Architecture?

Hopefully, you're now convinced that a microservices architecture can offer some unique advantages over traditional architectures and you've started thinking about this type of approach for your next project.

The very next question that comes to mind is "How do I start?"—and —"Is there a standard set of principles which I can follow to help me build a microservices architecture in a better way?"

Well, I'm afraid the answer is "No".

While that might not sound that promising, there are, however, some common themes which many organizations that have adopted microservices architectures have followed and with which they have ultimately found success. I'll discuss some of those common themes below.

1. How to Decompose

One of the ways to make our job easier could be to define services corresponding to business capabilities. A business capability is something a business does in order to provide value to its end users.

Identifying business capabilities and corresponding services requires a high level understanding of the business. For example, the business capabilities for an online shopping application might include the following..

- Product Catalog Management
- Inventory Management
- Order Management
- Delivery Management
- User Management
- Product Recommendations
- Product Reviews Management

Once the business capabilities have been identified, the required services can be built corresponding to each of these identified business capabilities.

Each service can be owned by a different team who becomes an expert in that particular domain and an expert in the technologies that are best suited for those particular services. This often leads to more stable API boundaries and more stable teams.

2. Building and Deploying

After deciding on the service boundaries of these small services, they can be developed by one or more small teams using the technologies which are best suited for each purpose. For example, you may choose to build a User Service in Java with a MySQL database and a Product Recommendation Service with Scala/Spark.

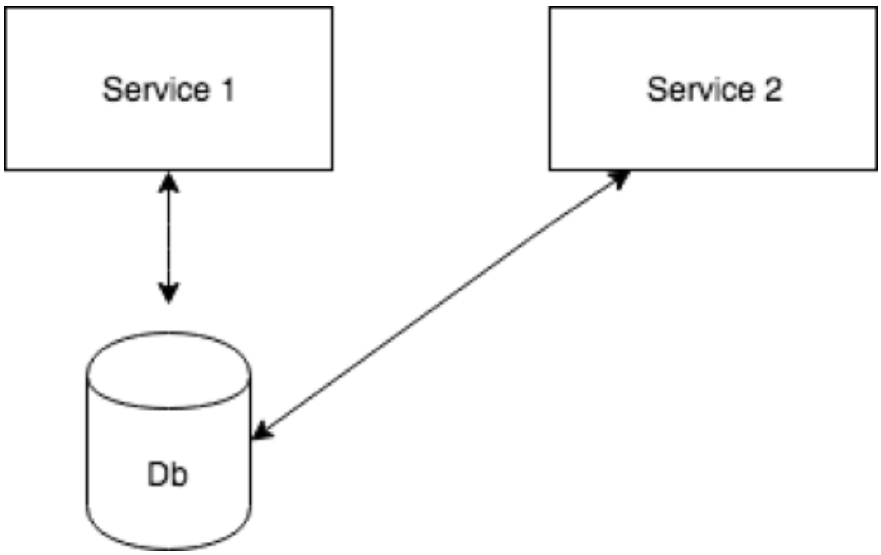
Once developed, CI/CD pipelines can be setup with any of the available CI servers (Jenkins, TeamCity, Go, etc.) to run the automated test cases and and deploy these service independently to different environments (Integration, QA, Staging, Production, etc).

3. Design the Individual Services Carefully

When designing the services, carefully define them and think about what will be exposed, what protocols will be used to interact with the service, etc.

It is very important to hide any complexity and implementation details of the service and only expose what is needed by the service's clients. If unnecessary details are exposed, it becomes very difficult to change the service later as there will be alot of painstaking work to determine who is relying on the various parts of the service. Additionally, a great deal of flexibility is lost in being able to deploying the service independently.

The diagram below shows one of the common mistakes in designing microservices:

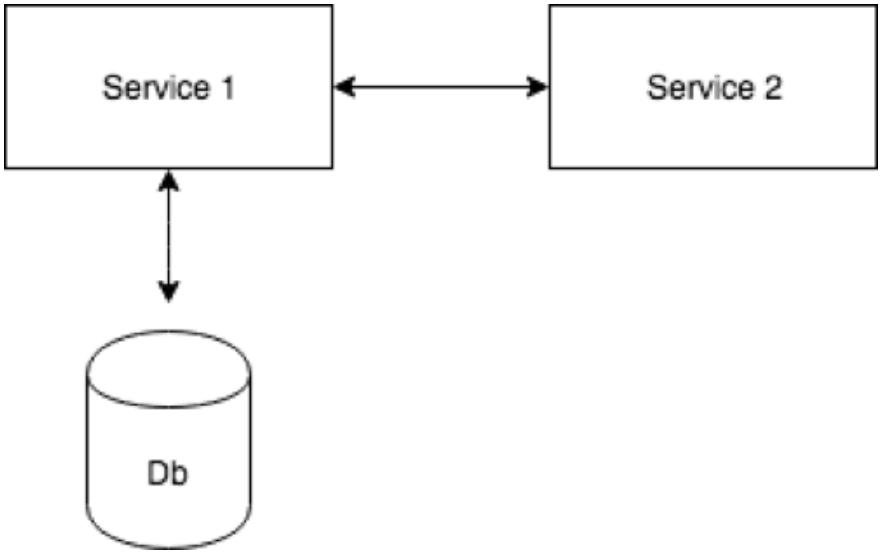


As you can see in the diagram, here we are taking a service (Service 1) and storing all of the information needed by the service to a database. When another service (Service 2) is created which needs that same data, we access that data directly from the database.

This approach might seem reasonable and logical in certain instances —maybe it's easy to access data in a SQL database or write data to a SQL database or maybe the APIs needed by Service 2 are not readily available.

As soon as this approach is adopted, control is immediately lost in determining what is hidden and what is not. Later on, if the schema needs to change, the flexibility to make that change is lost, since you won't know who is using the database and whether the change will break Service 2 or not.

An alternative approach, and I would submit the right way to tackle this, is below:



Service 2 should access Service 1 and avoid going directly to the database, therefore preserving utmost flexibility for various schema changes that may be required. Worrying about other parts of the system is eliminated provided you make sure that tests for exposed APIs pass.

As mentioned, choose the protocols for communication between services carefully. For example, if Java RMI is chosen, not only is the

user of the API restricted to using a JVM based language, but in addition, the protocol in and of itself is quite brittle because it's difficult to maintain backward compatibility with the API.

Lastly, when providing client libraries to clients to use the service, think about it carefully, because it's best to avoid repeating the integration code. If this mistake is made, it can also restrict changes being made in the API if the clients rely on unnecessary details.

4. Decentralize Things

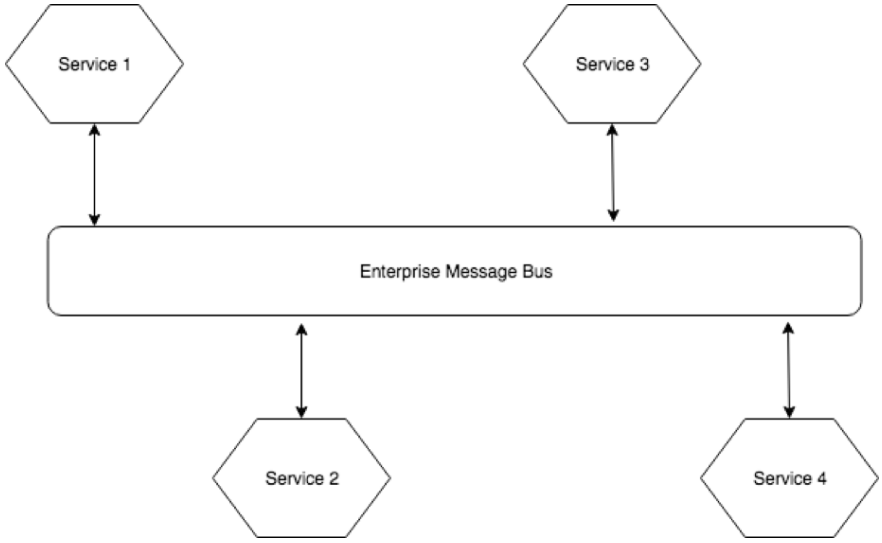
There are organizations who have found success with microservices and have followed a model where the teams who build the services take care of everything related to that service. They are the ones who develop, deploy, maintain and support it. There are no separate support or maintenance teams.

Another way to achieve the same is to have an internal open source model. By taking this approach, the developer who needs changes in a service can check out the code, work on a feature, and submit a PR himself instead of waiting for the service owner to pickup and work on needed changes.

For this model to work properly, the proper technical documentation is needed along with setup instructions and guidance for each service so that it's easy for anyone to pickup and work on the service.

Another hidden advantage of this approach is that it keeps developers laser focused on writing high quality code as they know that others will be looking at it.

There are also some architectural patterns which can help in decentralizing things. For example, you might have an architecture where the collection of services are communicating via a central message bus.



This bus handles the routing of messages from different services. Message brokers like RabbitMQ are a good example.

What tends to happen over time is people start putting more and more logic into this central bus and it starts knowing more and more about your domain. As it becomes more intelligent, that can actually become a problem as it becomes difficult to make changes which require coordination across separate dedicated teams.

My general advice for those types of architectures would be to keep them relatively “dumb” and let them just handle the routing. Event based architectures seem to work quite well in those scenarios.

5. Deploy

It’s important to write Consumer Driven Contracts for any API that is being depended upon. This is to ensure that new changes in that API don’t break your API.

In Consumer Driven Contracts, each consumer API captures their expectations of the provider in a separate contract. All of these contracts are shared with the provider so that they gain insight into the obligations they must fulfill for each individual client.

Consumer Driven Contracts must pass completely before being deployed and before any changes are made to the API. It also helps the provider to know what services are depending on it and how other services are depending on it.

When it comes to deploying independent microservices, there are two common models.

Multiple Microservices Per Operating System

First, **multiple microservices per operating system** can be deployed. With this model time is saved in automating certain things, for example, the host for each service does not have to be provisioned.

The downside of this approach is that it limits the ability to change and scale services independently. It also creates difficulty in managing dependencies. For instance, all the services on the same host will have to use same version of Java if they are written in Java. Further, these independent services can produce unwanted side effects for other running services which can be a very difficult problem to reproduce and solve.

One Microservice Per Operating System

Because of the above challenge, the second model, where **one microservice per operating system** is deployed, is the preferable choice.

With this model, the service is more isolated and hence it's easier to manage dependencies and scale services independently. But you may ask yourself "Isn't it expensive"? Well, not really.

The traditional solution for solving this problem is using Hypervisors whereby multiple virtual machines are provisioned on the same host. This solution approach can be cost inefficient as the hypervisor process itself is consuming some resources and, of course, the more VMs that are provisioned, the more resources will be consumed. And that's where the container model gets good traction and is preferred. Docker is one implementation of that model.

Making Changes to Existing Microservice APIs While In Production

Another common problem typically faced with a microservices model is determining how to make changes in existing microservice APIs when

others are using it in production. Making changes to the microservice API might break the microservice which is dependent on it.

There are different ways to solve this issue.

First, version your API and when changes are required for the API, deploy the new version of the API while still keeping the first version up. The dependent services can then be upgraded at their own pace to use the newer version. Once all of the dependent services are migrated to use the new version of the changed microservice, it can be brought down.

One problem with this approach is that it becomes difficult to maintain the various versions. Any new changes or bug fixes must be done in both the versions.

For this reason, an alternative approach can be considered in which another end point is implemented in the same service when changes are needed. Once the new end point is being fully utilized by all services, then the old end point can be deleted.

The distinct advantage to this approach is that it's easier to maintain the service as there will always be only one version of the API running.

6. Making Standards

When there are multiple teams taking care of different services independently, it's best to introduce some standards and best practices —error handling, for example. As might be expected, standards and best practices are not provided, each service would likely handle errors differently, and no doubt a significant amount of unnecessary code would be written.

Creating standards such as PayPal's API Style Guide is always helpful in long run. It's also important to let others know what an API does and documentation of the API should always be done when creating it. There are tools like Swagger which are very helpful in assisting in development across the entire API lifecycle, from design and documentation, to test and deployment. An ability to create metadata for your API and let users play with it, allows them to know more about it and use it more effectively.

Service Dependencies

In a microservices architecture, over time, each service starts depending on more and more services. This can introduce more problems as the services grow, for example, the number of service instances and their locations (host+port) might change dynamically. Also, the protocols and the format in which data is shared might vary from service to service.

Here's where API Gateways and Service Discovery become very helpful. Implementing an API Gateway becomes a single entry point for all clients, and API Gateways can expose a different API for each client.

The API gateway might also implement security such as verifying that the client is authorized to perform the request. There are some tools like Zookeeper which can be used for Service Discovery (although it was not built for that purpose). There are much more modern tools like etcd and Hashicorp's Consul which treat Service Discovery as a first class citizen and they are definitely worth looking at for this problem.

7. Failure

An important point to understand is that microservices aren't resilient by default. There will be failures in services. Failures can happen because of failures in dependent services. Additionally, failures can arise for a variety of reasons such as bugs in code, network time outs, etc.

What's critical with a microservices architecture is to ensure that the whole system is not impacted or goes down when there are errors in an individual part of the system.

There are patterns like Bulkhead and Circuits Breaker which can help you achieve better resilency.

Bulkhead

The Bulkhead pattern isolates elements of an application into pools so that if one fails, the others will continue to function. The pattern is coined Bulkhead because it resembles the sectioned partitions of a ship's hull. If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.

Circuit Breaker

The Circuit Breaker pattern wraps a protected function call in a circuit breaker object, which monitors for failures. Once a failure crosses the threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all for a certain configured timeout.

After the timeout expires some calls are allowed by circuit breaker to pass through, and if they succeed the circuit breaker resumes a normal state. For the period the circuit breaker has failed, users can be notified that a certain part of system is broken and the rest of the system can still be used.

Be aware that providing the required level of resiliency for an application can be a multi-dimensional challenge—take a look at Bilgin Ibryam's post for some great detail "It takes more than a Circuit Breaker to create a resilient application".

8. Monitoring and Logging

Microservices are distributed by nature and monitoring and logging of individual services can be a challenge. It's difficult to go through and correlate logs of each service instance and figure out individual errors. Just as with monolithic applications, there is no single place to monitor microservices.

Log Aggregation

To solve such problems, a preferred approach is to take advantage of a centralized logging service that aggregate logs from each service instance. Users can search through these logs from one centralized spot and configure alerts when certain messages appear.

Standard tools are available and widely used by various enterprises. ELK Stack is the most frequently used solution, where logging daemon, Logstash, collects and aggregate logs which can be searched via a Kibana dashboard indexed by Elasticsearch.

Stats Aggregation

Similar to log aggregation, stats aggregation such as CPU and memory usage can also be leveraged and stored centrally. Tools such as Graphite do a nice job in pushing to a central repository and storing in an efficient way.

When one of the downstream services is incapable of handling requests, there should be a way to trigger an alert, and that's where implementing health check APIs in each service become important—they return information on the health of the system.

A health check client, which could be a monitoring service or a load balancer, invokes the endpoint to check the health of the service instance periodically in a certain time interval. Even if all of the downstream services are healthy, there could still be a downstream communication problem between services. Tools such as Netflix's Hystrix project enable an ability to identify those types of problems.

One Last Thing

Now that we have covered what a microservices architecture is, why you'd want to deploy a microservices architecture, and thoughts on getting started, I'd like to offer up a final piece of advice:

> *— Start Small —*

When you are just starting to develop microservices, start modestly with just one or two services, learn from them, and with time and experience add more.

I wish you the best of success as you travel down this exciting microservices architecture path.

. . .

Feel free to share on other channels and be sure and keep up with all new content from Hashmap at https://medium.com/hashmapinc.

Jetinder Singh is Senior Tempus IIoT/IoT Developer at Hashmap working across industries with a group of innovative technologists and

domain experts accelerating high value business outcomes for our customers.