

[[<https://www.javacodegeeks.com/2015/04/microservice-design-patterns.html>]]

[[<http://blog.arungupta.me/microservices-monoliths-noops/>]]

The main characteristics of a microservices-based application are defined in [Microservices, Monoliths, and NoOps](#). They are functional decomposition or domain-driven design, well-defined interfaces, explicitly published interface, single responsibility principle, and potentially polyglot. Each service is fully autonomous and full-stack. Thus changing a service implementation has no impact to other services as they communicate using well-defined interfaces. There are several advantages of such an application, but it's not a [free lunch](#) and requires a significant effort in NoOps.

But let's say you understand the required effort, or at least some pieces of it, that is required to build such an application and willing to take a jump. What do you do? What is your approach for architecting such applications? Are there any design patterns on how these microservices work with each other?

Functional decomposition of your application and the team is the key to building a successful microservices architecture. This allows you to achieve loose coupling (REST interfaces) and high cohesion (multiple services can compose with each other to define higher level services or application).

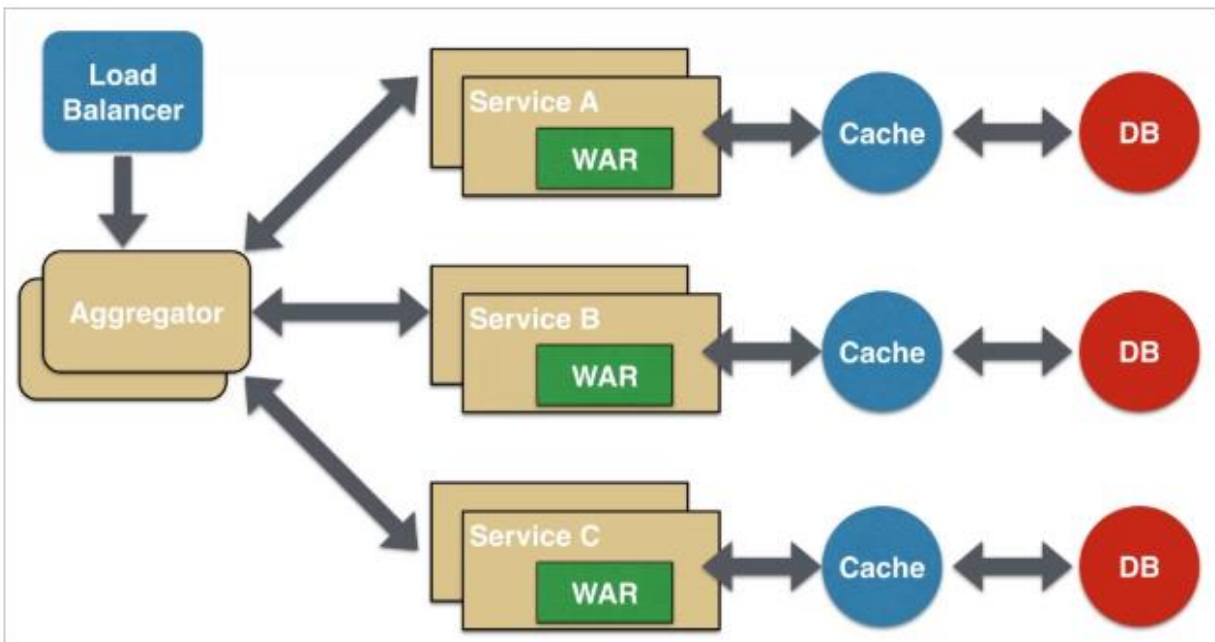
Verb (e.g. Checkout) or Nouns (Product) of your application are one of the effective ways to achieve decomposition of your existing application. For example, product, catalog, and checkout can be three separate microservices and then work with each other to provide a complete shopping cart experience.

Functional decomposition gives the agility, flexibility, scalability, and other *ilities but the business goal is still to create the application. So once different microservices are identified, how do you compose them to provide the application's functionality?

This blog will discuss some of the recommended patterns on how to compose microservices together.

Aggregator Microservice Design Pattern

The first, and probably the most common, is the aggregator microservice design pattern. In its simplest form, Aggregator would be a simple web page that invokes multiple services to achieve the functionality required by the application. Since each service (Service A, Service B, and Service C) is exposed using a lightweight REST mechanism, the web page can retrieve the data and process/display it accordingly. If some sort of processing is required, say applying business logic to the data received from individual services, then you may likely have a CDI bean that would transform the data so that it can be displayed by the web page.



Another option for Aggregator is where no display is required, and instead it is just a higher level composite microservice which can be consumed by other services. In this case, the aggregator would just collect the data from each of the individual microservice, apply

business logic to it, and further publish it as a REST endpoint. This can then be consumed by other services that need it.

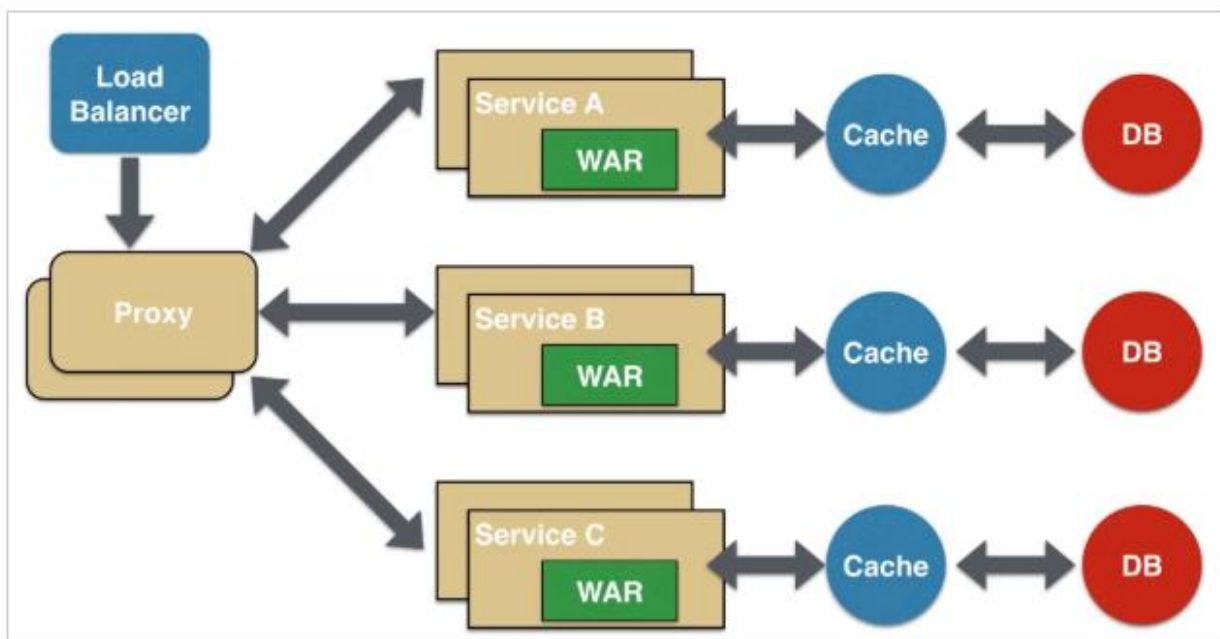
This design pattern follows the DRY principle. If there are multiple services that need to access Service A, B, and C, then its recommended to abstract that logic into a composite microservice and aggregate that logic into one service. An advantage of abstracting at this level is that the individual services, i.e. Service A, B, and C, and can evolve independently and the business need is still provided by the composite microservice.

Note that each individual microservice has its own (optional) caching and database. If Aggregator is a composite microservice, then it may have its own caching and database layer as well.

Aggregator can scale independently on X-axis and Z-axis as well. So if its a web page then you can spin up additional web servers, or if its a composite microservice using Java EE, then you can spin up additional WildFly instances to meet the growing needs.

Proxy Microservice Design Pattern

Proxy microservice design pattern is a variation of Aggregator. In this case, no aggregation needs to happen on the client but a different microservice may be invoked based upon the business need.

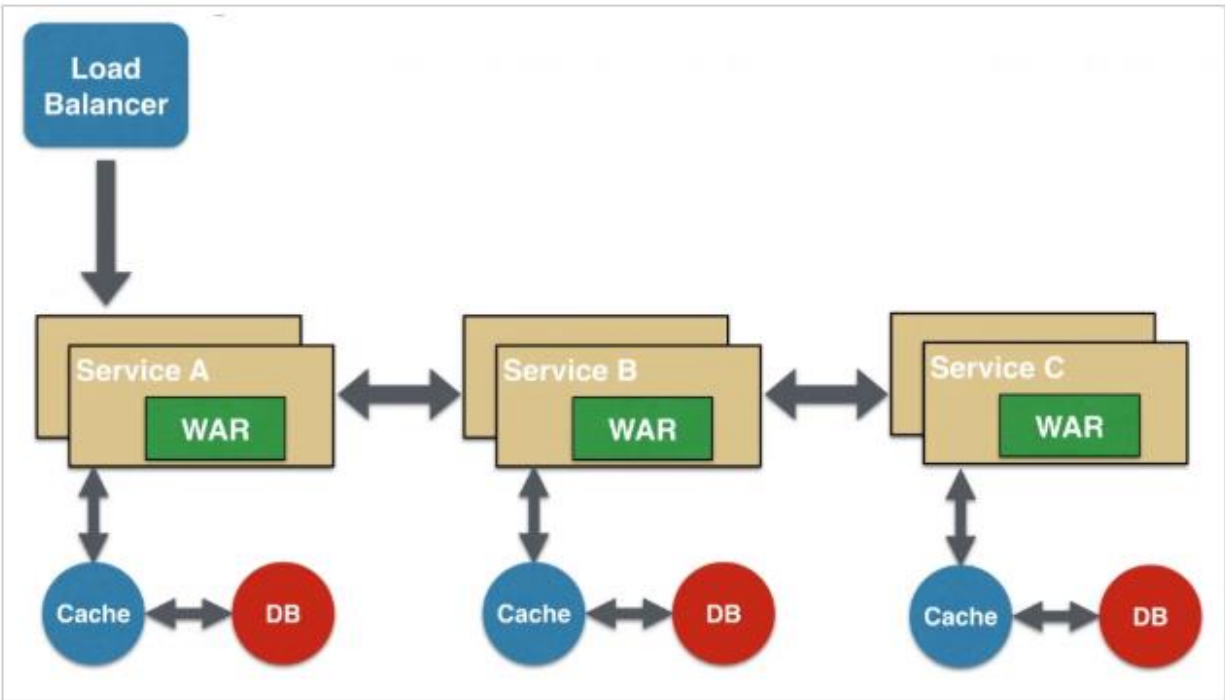


Just like Aggregator, Proxy can scale independently on X-axis and Z-axis as well. You may like to do this where each individual service need not be exposed to the consumer and should instead go through an interface.

The proxy may be a *dumb proxy* in which case it just delegates the request to one of the services. Alternatively, it may be a *smart proxy* where some data transformation is applied before the response is served to the client. A good example of this would be where the presentation layer to different devices can be encapsulated in the smart proxy.

Chained Microservice Design Pattern

Chained microservice design pattern produce a single consolidated response to the request. In this case, the request from the client is received by Service A, which is then communicating with Service B, which in turn may be communicating with Service C. All the services are likely using a synchronous HTTP request/response messaging.



The key part to remember is that the client is blocked until the complete chain of request/response, i.e. Service \leftrightarrow Service B and Service B \leftrightarrow Service C, is completed. The request from Service B to Service C may look completely different as the request from Service A to Service B. Similarly, response from Service B to Service A may look completely different from Service C to Service B. And that's the whole point anyway where different services are adding their business value.

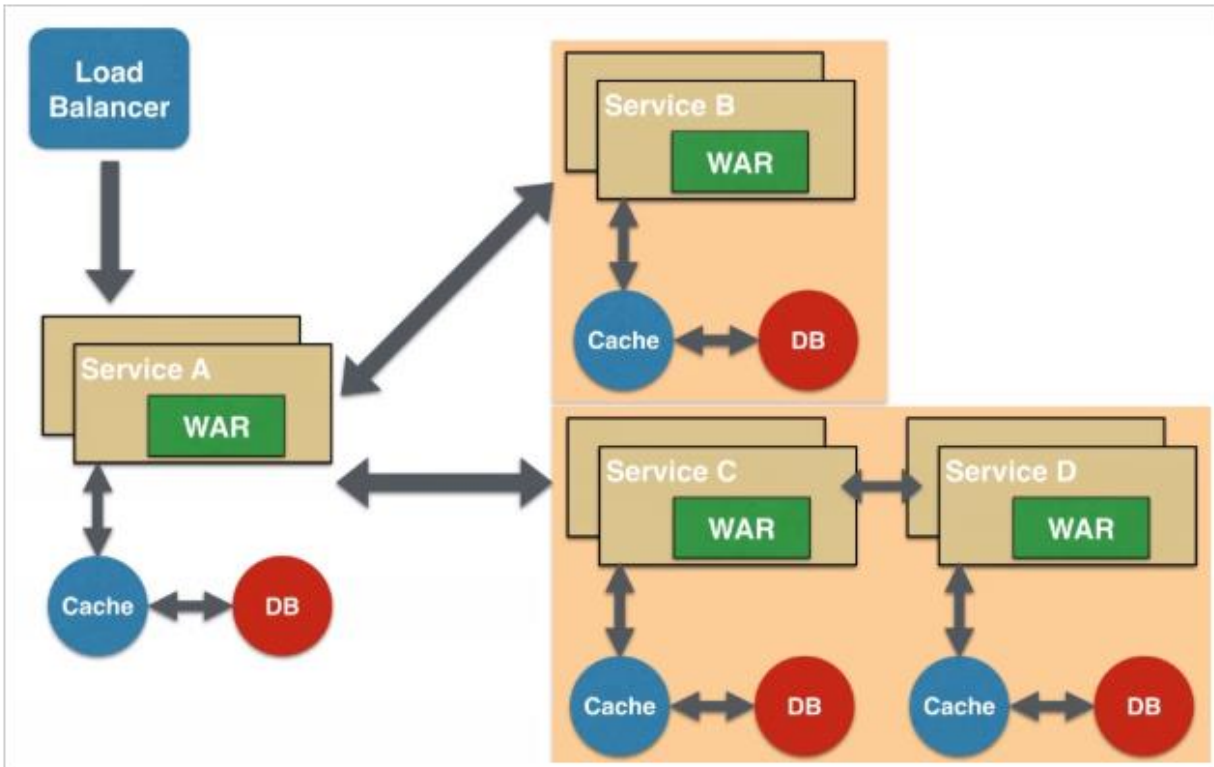
Another important aspect to understand here is to not make the chain too long. This is important because the synchronous nature of the chain will appear like a long wait at the

client side, especially if its a web page that is waiting for the response to be shown. There are workarounds to this blocking request/response and are discussed in a subsequent design pattern.

A chain with a single microservice is called *singleton chain*. This may allow the chain to be expanded at a later point.

Branch Microservice Design Pattern

Branch microservice design pattern extends Aggregator design pattern and allows simultaneous response processing from two, likely mutually exclusive, chains of microservices. This pattern can also be used to call different chains, or a single chain, based upon the business needs.



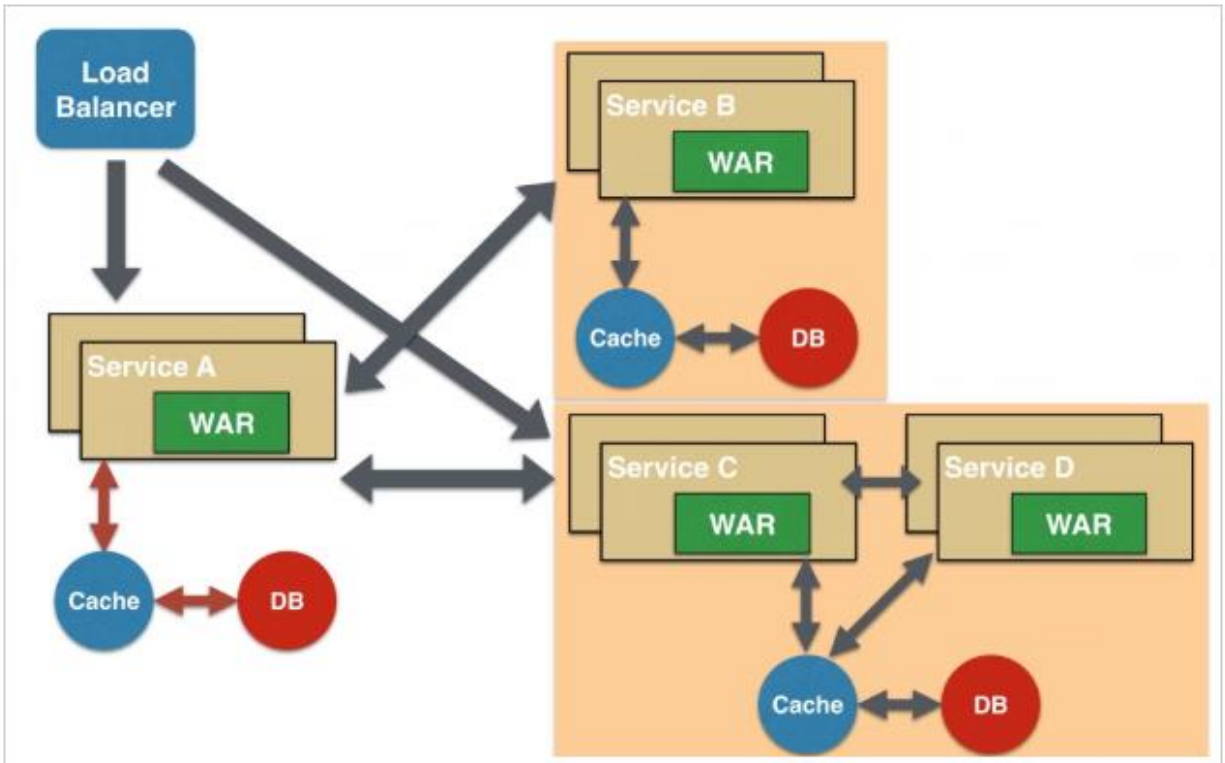
Service A, either a web page or a composite microservice, can invoke two different chains concurrently in which case this will resemble the Aggregator design pattern. Alternatively, Service A can invoke only one chain based upon the request received from the client. This may be configured using routing of JAX-RS or Camel endpoints, and would need to be dynamically configurable.

Shared Data Microservice Design Pattern

One of the design principles of microservice is autonomy. That means the service is full-stack and has control of all the components – UI, middleware, persistence, transaction. This allows the service to be polyglot, and use the right tool for the right job. For example, if a NoSQL data store can be used if that is more appropriate instead of jamming that data in a SQL database.

However a typical problem, especially when refactoring from an existing monolithic application, is database normalization such that each microservice has the right amount of data – nothing less and nothing more. Even if only a SQL database is used in the monolithic application, denormalizing the database would lead to duplication of data, and possibly inconsistency. In a transition phase, some applications may benefit from a shared data microservice design pattern.

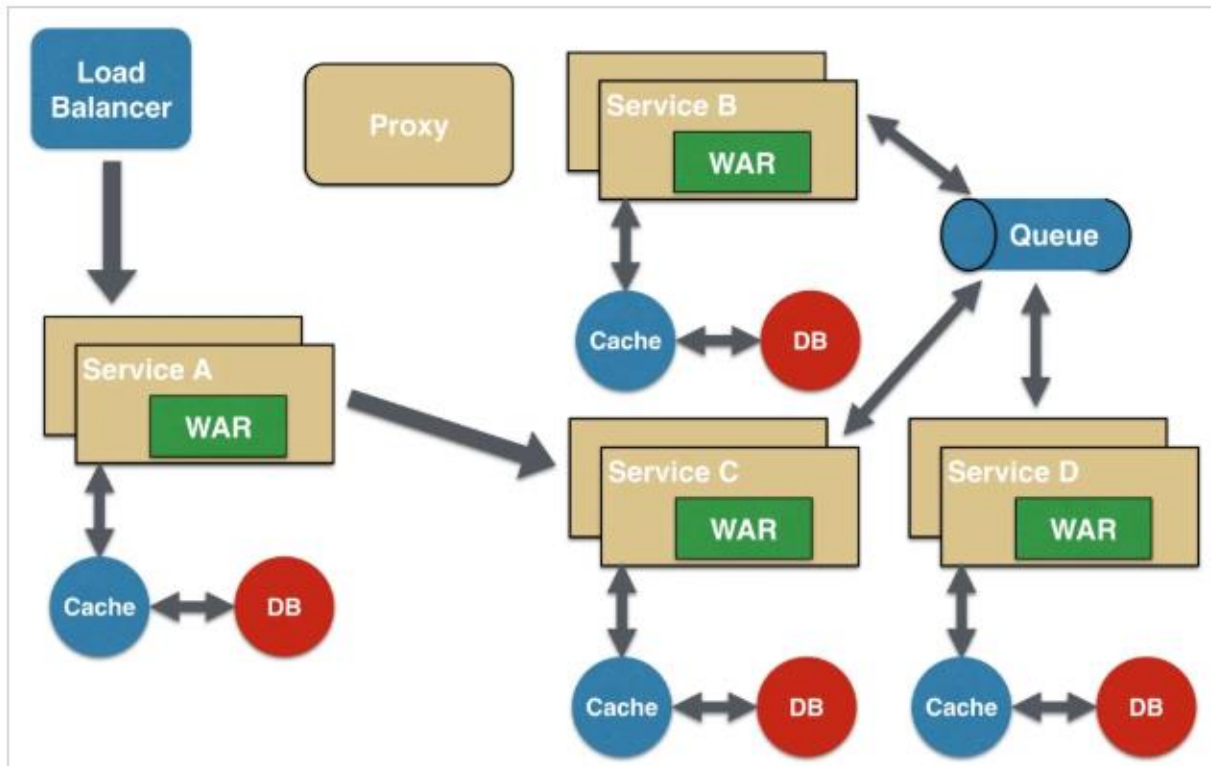
In this design pattern, some microservices, likely in a chain, may share caching and database stores. This would only make sense if there is a strong coupling between the two services. Some might consider this an anti-pattern but business needs might require in some cases to follow this. This would certainly be an anti-pattern for greenfield applications that are design based upon microservices.



This could also be seen as a transition phase until the microservices are transitioned to be fully autonomous.

Asynchronous Messaging Microservice Design Pattern

While REST design pattern is quite prevalent, and well understood, but it has the limitation of being synchronous, and thus blocking. Asynchrony can be achieved but that is done in an application specific way. Some microservice architectures may elect to use message queues instead of REST request/response because of that.



In this design pattern, Service A may call Service C synchronously which is then communicating with Service B and D asynchronously using a shared message queue. Service A -> Service C communication may be asynchronous, possibly using WebSockets, to achieve the desired scalability.

A combination of REST request/response and pub/sub messaging may be used to accomplish the business need.

[Coupling vs Autonomy in Microservices](#) is a good read on what kind of messaging patterns to choose for your microservices.

Hope you find these design patterns are useful.

What microservice design patterns are you using?

Reference: [Microservice Design Patterns](#) from our [JCG partner](#) Arun Gupta at the [Miles to go 2.0 ...](#) blog.