Architecture Styles Research Document

Stijn Baltessen,

Wouter Vandenboorn,

Joeri Berman,

Floris Feddema,

Luc Urlings,

Davy de Haas

# Table of contents

[**Table of contents**](#_1501a187ob6c) **2**

[**Introduction**](#_21z0jenymag) **3**

[**N-Tier Architecture**](#_xluhqmu1z8w5) **4**

[Introduction to N-Tier](#_efxftddevkkc) 4

[Implementation and operation](#_5mi5lugvzb9m) 4

[Use cases](#_hns0topf688b) 5

[Advantages, Disadvantages and best practices.](#_tbn5xmdzd4ya) 6

[**Web-Queue-Worker Architecture**](#_ayf1fhlidxac) **7**

[Introduction to Web-Queue-Worker](#_14axuf8jwp2f) 7

[Implementation and operation](#_ydg4r3ua3kl4) 7

[Use cases](#_506jykro0snc) 8

[Advantages and Disadvantages](#_xfpdgz5ekpud) 8

[**Microservices architecture style**](#_g4endxdm5ggt) **9**

[Introduction to Microservices](#_8v8f6pwnmuli) 9

[Implementation and operation](#_9zz33vl60jwh) 9

[Use cases](#_r3fwe06u7m4m) 9

[Advantages and Disadvantages](#_c3v31k9ri7qh) 10

[Advantages](#_ov796srbgf1w) 10

[Disadvantages](#_xi8wiucf8hbj) 10

[**Event-driven architecture style**](#_vjcnjz4x8su) **12**

[Introduction to Event-driven architecture](#_93d3xemw8kpc) 12

[Implementation and operation](#_m7pis9svnhlg) 12

[Advantages and Disadvantages](#_i1wcbht7lztw) 13

[Advantages](#_hg7dmmy4j6dn) 13

[Disadvantages](#_q7d5px6lwmcp) 13

[**Conclusion**](#_lslxmvg23d3w) **14**

[**References**](#_mjkoo8xckcwa) **15**

[N-tier](#_j9qh9eu4bxmh) 15

[Web-Queue-Worker](#_drpzkzryk3u0) 15

[Microservices](#_t1i8fbsr19l0) 15

[Event-driven architecture](#_7edjfgqmw5h2) 15

# Introduction

In this document we will explore 4 different types of architecture styles for enterprise applications. We will research which style is best suited for our application. Based on the results of this research we will compare and decide on the architecture best suited for our application.

Based on the requirements of our application we will weigh the pros/cons and usages for each architecture style within a matrix.

# N-Tier Architecture

## Introduction to N-Tier

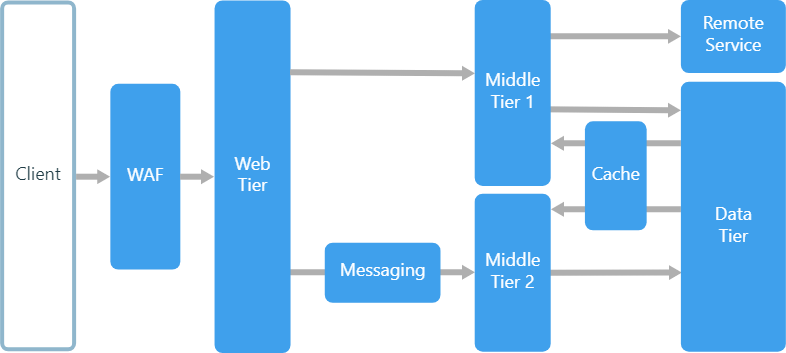
N-tier architecture divides an application into **logical layers** and **physical tiers**. These layers are meant as a way to separate responsibilities and manage dependencies. Each layer has a specific responsibility. A higher layer can use the services within a lower layer, but a lower layer cannot use the services within a higher layer.

## Implementation and operation

The tiers in a N-Tier architecture are physically separated, they run on separate (virtual) machines. A tier is able to directly communicate with another tier, or it can use a message queue(see messege queue “add reference”). A tier is able to contain multiple layers depending on the importance and workload of each layer. It might be better practice to host each layer within another tier. The way tiers are physically separated adds resilience, however you should take into account that it might add some latency.

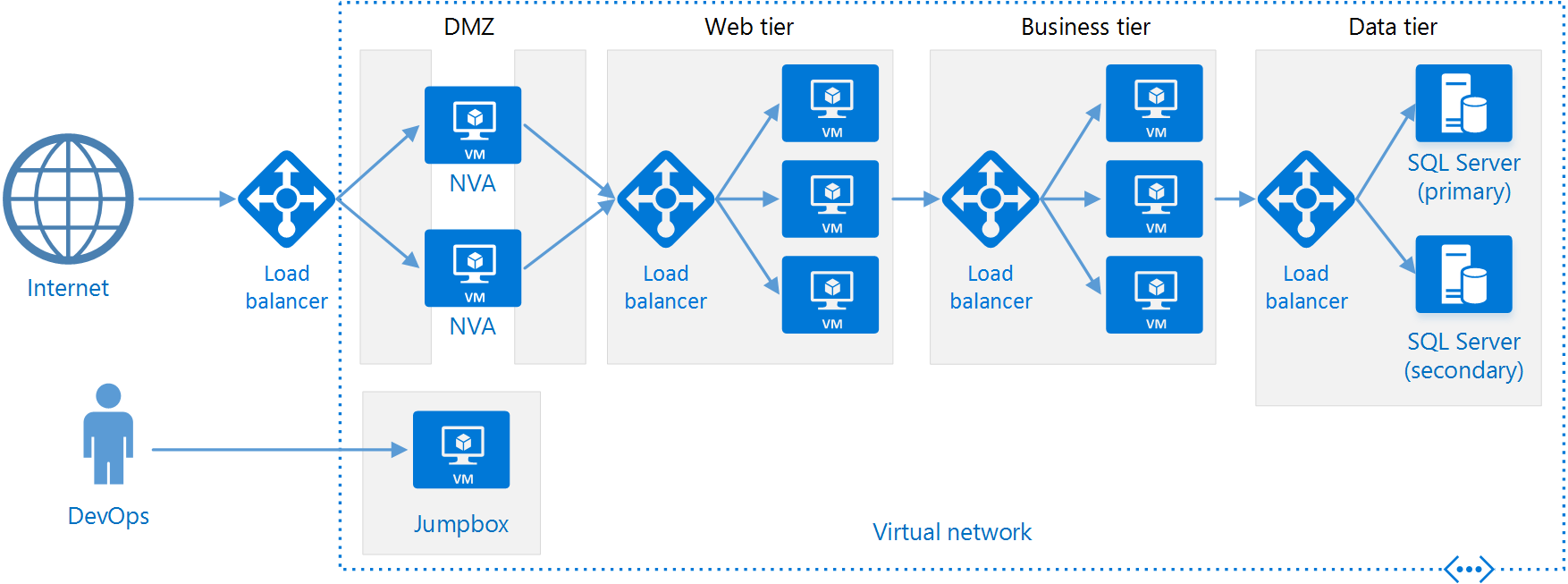
Traditionally an N-tier application consists of three tiers, a **Presentation Tier**, a **Middle Tier** and a **Database Tier**. In a three tier application the middle tier could be optional, however complex applications might use more tiers depending on the needs.

The image (taken from : <https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/images/n-tier-logical.svg>) shows a tiered application with two middle tiers. Each tier has their own functionality within the application (Single Responsibility)



The image below (taken from: <https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/images/n-tier-physical.png> ).

In this image we can find an example of a N-tier architecture where all tiers are consisting of virtual machines. Using load balance you can guarantee performance to a certain extent. It also provides the application with alot of resilience in case one of the vm’s goes down, in this case the load balancer can easily use the remaining vm’s to ensure operatability.



**Open and Closed layer architecture**

N-Tier architecture can be implemented in a closed or open way. A closed layer architecture can only call the next lower layer, an open layered architecture can call any tier lower then itself directly skipping the need to use layers in between.

## Use cases

Since all layers are separated physically it is advantageous to use managed services for some parts of the application. Parts like caching, messaging and data storage are examples of parts that might benefit from being on a managed service.

Some examples of applications to use N-tier architecture are:

* Simple web applications
* Migrating an existing locally hosted application to an cloud environment
* Managing application consisting of cloud hosted/external services and on-premise components

## 

## Advantages, Disadvantages and best practices.

Advantages of using this style of architecture are :

* Easily migratable to cloud platforms, or locally hosted platforms.
* Reasonably easily implemented due to a very structured approach of communication between layers and tiers.
* Easily scalable due to high migratability

Disadvantages of using this style of architecture are:

* You might end up with a middle layer that does nothing more than communicate requests from the presentation layer to the back-end layers. Impairing latency without providing any extra useful work.
* Due to the dependency relient architecture implementing independent features might require more work than expected, since all required layers need to be adjusted to meet the new needs.
* Since most communication is network based, security might become an issue in larger systems.
* When a closed layer architecture is used it might create unnecessary network traffic because a larey cannot “skip” unnecessary layers that just pass through the request..

# Web-Queue-Worker Architecture

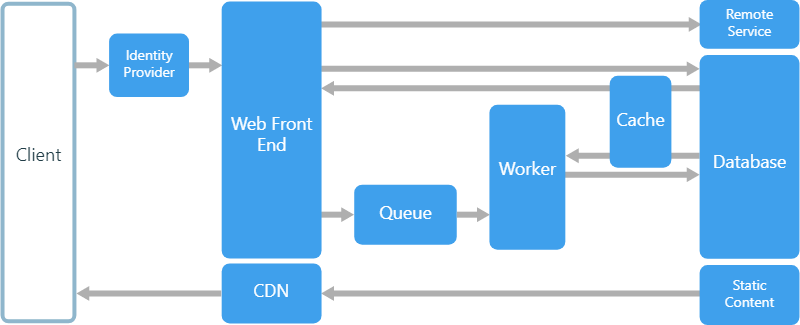
## Introduction to Web-Queue-Worker

Web queue worker architecture relies heavily on a web front end that serves client requests, and a worker that handles and performs resource-intensive, long-running workflows or batch jobs requested. It uses a message queue to communicate with the worker from the frontend, from this queue tasks are distributed and queued.

## Implementation and operation

The image below(taken from :<https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/images/web-queue-worker-logical.svg> ) shows a general overview of a Web-queue-worker architecture.

The front end communicates through a queue to the worker service. This queue can be distributed to multiple workers who process the jobs. between the worker, front end and database we can find a cache instance. The cache is used to store some variables from the database for quick access of commonly used data to improve performance and responsiveness..



*“The web and worker are both stateless. Session state can be stored in a distributed cache. Any long-running work is done asynchronously by the worker. The worker can be triggered by messages on the queue, or run on a schedule for batch processing. The worker is an optional component. If there are no long-running operations, the worker can be omitted.”*

[Source](https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/web-queue-worker):

## 

## Use cases

* Adding remote services
  + email
  + sms
  + identity authentication
* Integrating one or multiple databases
* Applications with a relatively simple domain
* Application heavy or long workloads or operations.
* If your application uses a lot of distributed management services.

## Advantages and Disadvantages

Advantages of using this type of architecture are:

* The front end and worker are decoupled and use asynchronous communication/messaging.
* Since the front end and worker are individual entities they can be scaled independently.
* clear separation of task management within the application
* On a small scale the architecture is relatively simple and easy to understand.

Disadvantages of using this type of architecture:

* When dealing with complex domains it can be hard to keep track and manage dependencies. Making it harder to maintain and update without proper planning and care for the structure of the architecture.
* Dependencies might be hidden if the front end and worker share modules or data.

# 

# Microservices architecture style

## Introduction to Microservices

*<< introductie omschrijving wat is Microservices>>*

A microservice architecture consists of a collection of **small, autonomous** services. Each service should be responsible for a single business capability.

## Implementation and operation

*<< hoe werkt het waarvoor kan je het implementeren. >>*

Each microservice can be **managed independently**, it can **restart**  **independently**, and it handles it **own data persistence**. Each microservice can be **accessed** by other services by a **well defined API Gateway**.

You can create an **management layer** which **balances** the **workload** of each microservices of one application. When a **microservice** which has a lot of load and **loses speed**, the **management** layer will take **action** and **create** another **node** that will also **host** the **same microservice** which **balances** the load **between** these **2 identical microservices**.

## Use cases

*<< waarvoor wordt het gebruikt, en in wat voor een soort applicaties >>*

In the following paragraphs you will read what an ideal application is of microservices, some of the most unique features and an example of a microservice from amazon and why it is a microservice.

An ideal application of microservices is when your application will be used by different implementations. Each service is independent so each implementation can use the data it wants to use.

So a website can use call the api and fill its site with the response, and the app can do the same. The ones asking for the data need to format it to their liking. This is why the clearly defined api is very different.

When you know your full application needs to use multiple coding languages it is also nice to use microservices because each one of them is independent.

If you make a generic enough service you can sell your different microservices to third external which use a combination of services to run their own implementation.

Example would be the services of amazon, they are independent, used by external party’s, they get easy load balancing, made of multiple languages. One of the services like Amazon Simple Storage Service (S3) is used by a lot of parties but also by amazon themselves.

## Advantages and Disadvantages

<< voor en nadelen van de architectuur >>

Microservices is notable because of its independent services this brings its own advantages and disadvantages. Each (dis)advantage will have a short summary which contains the reasoning of it being an (dis)advantage.

### Advantages

Agility through small teams/ Each microservice should be small enough it can be made by a small team. Smaller teams have faster communication and thus faster implementation compared to how much people work on it.

Agility through independence. Because each microservice is independent you can easily change the version without closing the full application.

Easy learning curve for newcomers or those who are not familiar with the microservice. A microservice is independent and should have a clearly defined API which executes its purpose.

Scalability with the help of management layer each service can be scaled to be able to operate correctly.

Isolation, When a fault happens in the microservice it is isolated to just the microservice. This also applies to faults in the data of a microservice.

### Disadvantages

Although microservices strength is the independence of each service it is also the creation of the pitfalls.

Complexity, Although the learning curve of one microservice is low. The full application which uses every microservice is much more complex to grasp for newcomers.

Testing, In the most ideal situations every microservice should be independable, this is the case but in full applications the data that one service provides will be used in the next. This makes testing difficult.

Complexity on a support level, when you have a new product which has a lot of services that use a lot of different languages. These are made by different teams. It might be difficult in the future to sustain the product with less teams which are not familiar with the same languages

Management, managing a big microservice environment needs a very mature devops culture. Consistency is good.

Versions, the pitfall the agility of version changing is that it breaks another service.

Difficulty, microservices is one of the harder architecture style and a team which does not hold the skill to work in this environment will likely fall to ruin or will lose valuable time.

# 

# Event-driven architecture style

## Introduction to Event-driven architecture

*<< introductie omschrijving wat is Event driven architecture>>*

Event driven architecture design is a style in which you have 2 parties; producers and consumers. and a middle party called ingestion, which decides to which consumer the produced events from the producer go.

producers don’t know which consumers are listening.

consumers see every produced event. But have no knowledge about one another.

There are 2 variants of event driven architecture. **Pub/Sub** publisher subscribers. subscribers indicate that they want to listen to a certain publisher. newcomers don’t see the old messages.

**Event streaming**  is a large log that is live being extended by new information, the consumers are responsible for where they are looking.

Use cases

*<< waarvoor wordt het gebruikt, en in wat voor een soort applicaties >>*

When There multiple systems that need to keep track of your data it is good to update them through this technique because it keeps everyone who is listening and wants to listen up to date.

When there is a certainty that a lot of data will be sent within your application which needs to be processed.

When a large group of subsystems depend on your data it is a good setup for this kind of architecture.

## Implementation and operation

*<< hoe werkt het waarvoor kan je het implementeren. >>*

The most important part of an event driven architecture is setting up how producers work. When there is a producer who broadcasts something to everyone subscribed or listening. You will need to make a way for people to subscribe or listen.

The easiest example is a blog that everyone wants to keep updated by, or youtubes notifcation bell. You have a way to indicate that you want to be notified.

## Advantages and Disadvantages

The event-driven architecture style is a very common used technique implemented in very various ways. It is mostly used for updating information for subsystems and notifying people. This brings advantages and disadvantages.

### Advantages

Producers are independent of consumers, this means that the server will always keep functioning and processing data even without clients using the application.

Consumers, they can come and go as they wish.

Distribution and scaling, is very easy because you can easily divide consumers over multiple systems or versions of systems.

Each subsystem can view the data they want.

### Disadvantages

Delivery successful? The producer doesn’t know if everyone who wants to listen has successfully received a broadcast.

Niche application. The choice that you have to make when deciding if you want this style is if you want to do things this is specifically designed for.

# Conclusion

*<< Matrix with the demands of our application measured against the architecture styles researched, also which one we are most likely to use. >>*

We will now create a matrix with one of its axis being the requirements of our application. On the other axis we will put the different architecture styles. Based on this we can decide which architecture style is the best fit for our application.

# References

## N-tier

* <https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/n-tier>
* <https://stackify.com/n-tier-architecture/>
* <https://medium.com/@pradnyapatil29/cloud-application-architectural-styles-part-1-bbda76f8ad3f>

## Web-Queue-Worker

* <https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/web-queue-worker>
* <https://medium.com/@pradnyapatil29/cloud-application-architectural-styles-part-1-bbda76f8ad3f>

## Microservices

* <https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/microservices>
* <https://aws.amazon.com/>

## Event-driven architecture

<https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/event-driven>

<https://docs.microsoft.com/en-gb/azure/architecture/guide/architecture-styles/event-driven>