**Design Patterns for Microservices**

Micorservices architecture has become the de facto choice for modern application development. Though it solves certain problems, it is not a silver bullet. It has several drawbacks and when using this architecture, there are numerous issues that must be addressed. This brings about the need to learn common patterns in these problems and solve them with reusable solutions. Thus, design patterns for microservices need to be discussed. Before we dive into the design patterns, we need to understand on what principles microservice architecture has been built:

1. Scalability
2. Availability
3. Resiliency
4. Independent, autonomous
5. Decentralized governance
6. Failure isolation
7. Auto-Provisioning
8. Continuous delivery through DevOps

Applying all these principles brings several challenges and issues. Let's discuss those problems and their solutions.

## 1. Decomposition Patterns

### a. Decompose by Business Capability

#### **Problem**

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

#### **Solution**

One strategy is to decompose by business capability. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except it’s business-oriented rather than technical.

### b. Decompose by Subdomain

#### **Problem**

Decomposing an application using business capabilities might be a good start, but you will come across so-called "God Classes" which will not be easy to decompose. These classes will be common among multiple services. For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc. How do we decompose them?

#### **Solution**

For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue. It uses subdomains and bounded context concepts to solve this problem. DDD breaks the whole domain model created for the enterprise into subdomains. Each subdomain will have a model, and the scope of that model will be called the bounded context. Each microservice will be developed around the bounded context.

**Note**: Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

### c. Strangler Pattern

#### **Problem**

So far, the design patterns we talked about were decomposing applications for greenfield, but 80% of the work we do is with brownfield applications, which are big, monolithic applications. Applying all the above design patterns to them will be difficult because breaking them into smaller pieces at the same time it's being used live is a big task.

#### **Solution**

The Strangler pattern comes to the rescue. The Strangler pattern is based on an analogy to a vine that strangles a tree that it’s wrapped around. This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services. The idea is to do it one domain at a time. This creates two separate applications that live side by side in the same URI space. Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.

## 2. Integration Patterns

### A. API Gateway Pattern

#### **Problem**

When an application is broken down to smaller microservices, there are a few concerns that need to be addressed:

1. How to call multiple microservices abstracting producer information.
2. On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
3. Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
4. How to handle different type of Protocols some of which might not be supported by producer microservice.

#### **Solution**

An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.
2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.
4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.
5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
6. It can also offload the authentication/authorization responsibility of the microservice.

### b. Aggregator Pattern

#### **Problem**

We have talked about resolving the aggregating data problem in the API Gateway Pattern. However, we will talk about it here holistically. When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

#### **Solution**

The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

### c. Client-Side UI Composition Pattern

#### **Problem**

When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices. In the monolithic world, there used to be only one call from the UI to a backend service to retrieve all data and refresh/submit the UI page. However, now it won't be the same. We need to understand how to do it.

#### **Solution**

With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section will make a call to an individual backend microservice to pull the data. That is called composing UI components specific to service. Frameworks like AngularJS and ReactJS help to do that easily. These screens are known as Single Page Applications (SPA). This enables the app to refresh a particular region of the screen instead of the whole page.

## 3. Database Patterns

### a. Database per Service

#### **Problem**

There is a problem of how to define database architecture for microservices. Following are the concerns to be addressed:

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.

2. Business transactions may enforce invariants that span multiple services.

3. Some business transactions need to query data that is owned by multiple services.

4. Databases must sometimes be replicated and sharded in order to scale.

5. Different services have different data storage requirements.

#### **Solution**

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly. For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service. Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

### b. Shared Database per Service

#### **Problem**

We have talked about one database per service being ideal for microservices, but that is possible when the application is greenfield and to be developed with DDD. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. What is the suitable architecture in that case?

#### **Solution**

A shared database per service is not ideal, but that is the working solution for the above scenario. Most people consider this an anti-pattern for microservices, but for brownfield applications, this is a good start to break the application into smaller logical pieces. This should not be applied for greenfield applications. In this pattern, one database can be aligned with more than one microservice, but it has to be restricted to 2-3 maximum, otherwise scaling, autonomy, and independence will be challenging to execute.

### c. Command Query Responsibility Segregation (CQRS)

#### **Problem**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. Then, how do we implement queries in microservice architecture?

#### **Solution**

CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views. The **event sourcing pattern** is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

### d. Saga Pattern

#### **Problem**

When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

#### **Solution**

A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

1. Choreography — When there is no central coordination, each service produces and listens to another service’s events and decides if an action should be taken or not.
2. Orchestration — an orchestrator (object) takes responsibility for a saga’s decision making and sequencing business logic.

## 4. Observability Patterns

### a. Log Aggregation

#### **Problem**

Consider a use case where an application consists of multiple service instances that are running on multiple machines. Requests often span multiple service instances. Each service instance generates a log file in a standardized format. How can we understand the application behavior through logs for a particular request?

#### **Solution**

We need a centralized logging service that aggregates logs from each service instance. Users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs. For example, PCF does have Loggeregator, which collects logs from each component (router, controller, diego, etc...) of the PCF platform along with applications. AWS Cloud Watch also does the same.

### b. Performance Metrics

#### **Problem**

When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens. How should we collect metrics to monitor application perfomance?

#### **Solution**

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

* Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
* Pull — the metrics services pulls metrics from the service e.g. Prometheus

### c. Distributed Tracing

#### **Problem**

In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?

#### **Solution**

We need a service which

* Assigns each external request a unique external request id.
* Passes the external request id to all services.
* Includes the external request id in all log messages.
* Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Spring Cloud Slueth, along with Zipkin server, is a common implementation.

### d. Health Check

#### **Problem**

When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances? With a load balancing pattern implementation.

#### **Solution**

Each service needs to have an endpoint which can be used to check the health of the application, such as /health. This API should o check the status of the host, the connection to other services/infrastructure, and any specific logic.

Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well.

## 5. Cross-Cutting Concern Patterns

### a. External Configuration

#### **Problem**

A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service. How do we avoid code modification for configuration changes?

#### **Solution**

Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at startup or on the fly.

Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties. These can be accessed by the application on startup or can be refreshed without a server restart.

### b. Service Discovery Pattern

#### **Problem**

When Micorservices come into the picture, we need to address a few issues in terms of calling services:

1. With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes.
2. Each service URL has to be remembered by the consumer and become tightly coupled.

So how does the consumer or router know all the available service instances and locations?

#### **Solution**

A service registry needs to be created which will keep the metadata of each producer service. A service instance should register to the registry when starting and should de-register when shutting down. The consumer or router should query the registry and find out the location of the service. The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it. There are two types of service discovery: client-side and server-side. An example of client-side discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

### c. Circuit Breaker Pattern

#### **Problem**

A service generally calls other services to retrieve data, and there is the chance that the downstream service may be down. There are two problems with this: first, the request will keep going to the down service, exhausting network resources and slowing performance. Second, the user experience will be bad and unpredictable. How do we avoid cascading service failures and handle failures gracefully?

#### **Solution**

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.

Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

### d. Blue-Green Deployment Pattern

#### **Problem**

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Also, the rollback will be a nightmare. How do we avoid or reduce downtime of the services during deployment?

#### **Solution**

The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let's assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic. All cloud platforms provide options for implementing a blue-green deployment. For more details on this topic, check out [this article](https://dzone.com/articles/blue-green-deployment-for-cloud-native-application).

There are many other patterns used with microservice architecture, like Sidecar, Chained Microservice, Branch Microservice, Event Sourcing Pattern, Continuous Delivery Patterns, and more. The list keeps growing as we get more experience with microservices. I am stopping now to hear back from you on what microservice patterns you are using.

In order for an application to be deployed in the cloud and enjoy features such as [auto scaling](https://dzone.com/articles/microservices-architecture-introduction-to-auto-sc), it first needs to be cloud native. In this article, we have a close look at the best practices for cloud native applications, popularly known as The 12 Factor App.

This is the first article in a series of six articles on best practices with cloud and Micorservices.

# The 12 Factor App: Best Practices in Cloud Native Applications and Micorservices

## The 12 Factor App

The 12 Factor App is a set of best practices that guide you to build a great cloud native application. These were framed by Heroku, based on their experiences with building cloud native applications.

### **Codebase - One Codebase Tracked in Version Control, Many Deploys**

You have the codebase in a version control system, and you extract and build it, then deploy it many times.

### Dependencies - Explicitly Declare and Isolate Dependencies

Whenever you build a software application, say in Java, you need a number of dependencies such as frameworks. You may need to manage the versions of the libraries you need to use. Explicitly declare and isolate such dependencies.

### Config - Store Config in the Environment

There are a variety of environments where an application could execute, such as development, QA, staging, and production. Applications have different configurations in each of these environments. It is recommended to seperate the configuration and store it within the environment itself.

Another option is to store the configuration information in a centralized repository.

An good example is Spring Cloud Config Server. The configuration is stored in the config server, which can then be mapped to the environment.

### Backing services - Treat Backing Services as Attached Resources

The term backing services refers to the other systems that an application needs to talk to, and this also includes the databases. All such services need to be considered as attached resources.

All of these need to be configurable, and it should be easy to switch from one backing service to another. This switch should be possible with just a switch in configuration.

### Build, Release, Run - Strictly Separate Build and Run Stages

The build, release, and run stages need to be strictly separated.

You need to be able to build a deployable component, such as a JAR, WAR, or an EAR, that is independent of the environment. There should not be any change in this component as the deployment environment changes.

A release is a phase where we take this reusable component, and combine it with a specific configuration for a target environment.

The next phase is to take the released entity, create a container out of it, and run it in the environment.

### Processes - Execute the App as One or More Stateless Processes

Ideally an application should be stateless. But, in case you have state, where you store the state of an application determines how flexible it is. If you store state in a central data store such as Redis, it makes the application very flexible. You no longer need sticky sessions. You can have any instance of an app, answer any request.

### Port binding - Export Services via Port Binding

You should be able to deploy applications as services by tying them with ports.

### Concurrency - Scale Out via the Process Model

There are two kinds of scaling that can be applied to an application - horizontal and vertical. Vertical scaling refers to increasing the hardware infrastructure, such as increasing the CPU processing power, or increasing the amount of physical memory available to the application. Clearly, there are limits to such an approach.

Horizontal scaling refers to the possibility of dynamically increasing or decreasing the number of instances of an application, depending on the system needs.

Your applications should be built to be able to dynamically adapt to changing number of instances of various services.

### Disposability - Maximize Robustness with Fast Startup and Graceful Shutdown

If one of the instances of the application is causing errors, or is slow in responding to requests, or is not responding at all, it should be possible to gracefully shut the instance down.

In addition, the other applications in the system should not be affected by this change in the environment.

You should be able to bring in new instances as they are needed, and take down instances when required. This property is known as disposability, and is a measure of the system's robustness.

### Development/Production Parity

There is a strong need to keep the development, QA, staging, and production stages of a deployment pipeline as similar as possible. The similarities should apply to the processes you follow, the technologies you make use of, and the infrastructure you employ.

If you have this parity among these stages, then most of the problems that could arise with the application, would appear in the earlier stages. Not many surprises would be in store for you at production.

### Logs - Treat Logs as Event Streams

Visibility is one of the most important requirements of a microservices architecture.

By treating each log message entered into a centralized logging system as an event, you get a sequence of actions that are performed on a request when it enters the system, right up to when it is completed or abandoned.

All one needs to do in order to debug a problem, is to go to the central dashboard and search for it.

### Admin processes - Run Admin/Management Tasks as One-off Processes

There are a number of one-off process that you need to run — batch programs, database migrations, scripts.

Treat one-off processes the same way as long running processes.

Have the same standards, have the code base in version control, follow standard deployment processes, and use the same environments

## Summary

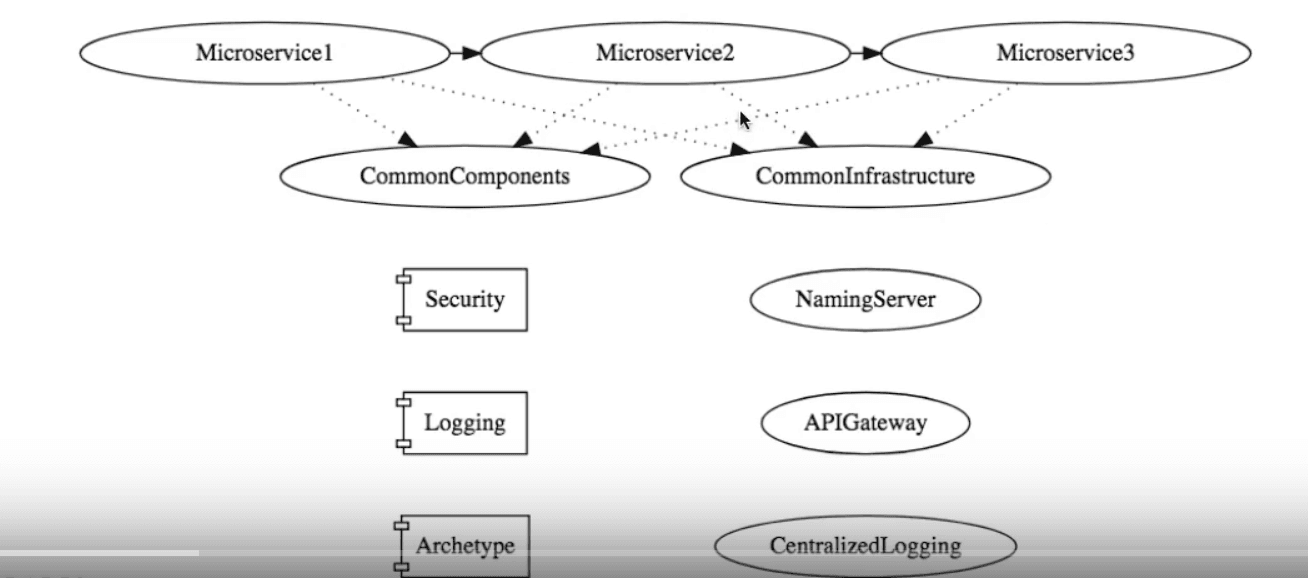
In this article, we looked at the best practices for cloud native applications, called the 12 Factor App.

* What is Event Driven Architecture?
* Why do we need Event Driven Architectures?
* What is the relationship between event driven architectures and microservices?
* What are the advantages of Event Driven Architectures?

## Best Practices With Cloud and Microservices

This is the second article in a series of six articles on best practices with cloud and microservices. This first part can be found here: [The 12 Factor App: Best Practices in Cloud Native Applications and Microservices](https://dzone.com/articles/the-12-factor-app-best-practices-in-cloud-native-a?preview=true).

## The Need for Event Driven Architecture

Microservices architectures have multiple small-sized microservices talking to each other. Here is one such architecture:

There are a set of common components — technical as well as infrastructure. Examples of technical components are Security and Logging components. Examples of infrastructure components include Naming Server, API Gateway, and Centralized Logging.

When implementing microservices, you would want to make them as event driven as possible. Why? Let's take an example

### Take a Use Case

Consider the use case of an online shopping application. Whenever there is an order, it is received by the order service. There are several actions that need to be carried out in order service:

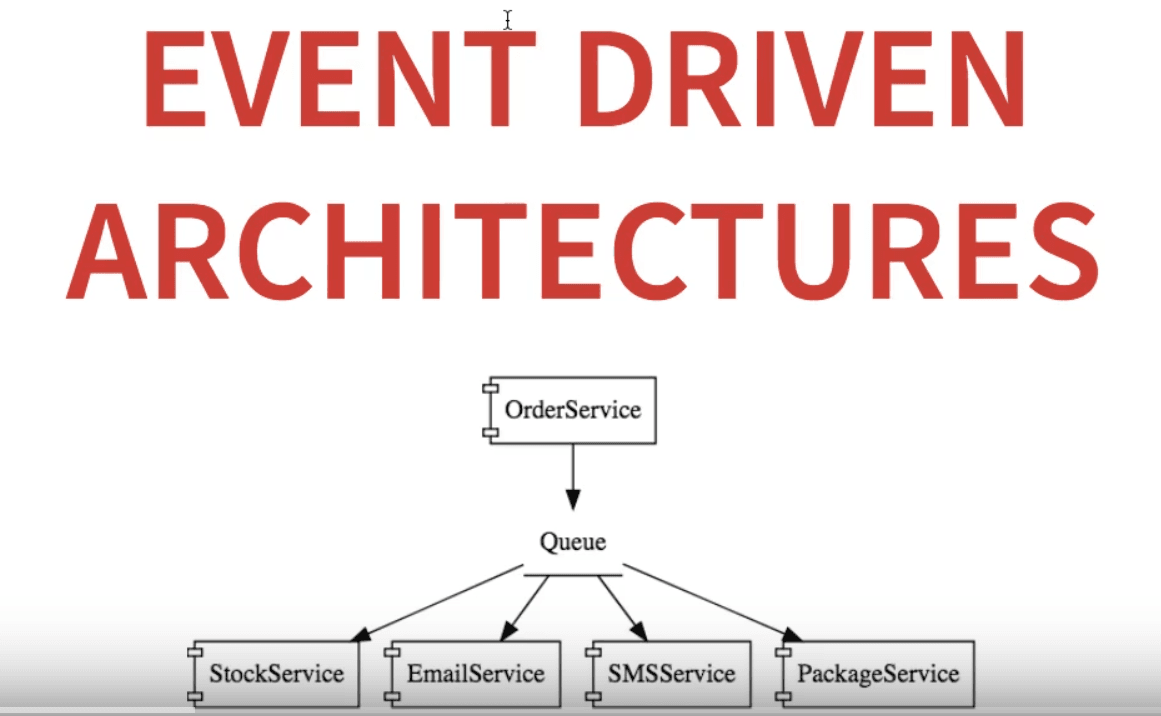
* Update the stock.
* Send out email and SMS.
* Notify the packaging team.

#### Monolithic Approach

An initial approach could be to have a single monolith application to take care of all the functionality.

#### Microservices Approach

Alternatively, these are fairly independent activities, and the entire application can be structured to have microservices for them, in a straightforward manner.



In this approach, you create an order event for the request coming in, and place it in the Queue. The rest of the individual services listen in to the queue for order events, and do the processing subsequently.

#### How to Choose?

**How do we make the decision between these two application architectures?**

The nature of your application decides this.

If your system does not have a very high load at any given time, and also does not have any pressing scalability requirements, then you might want go with a monolith approach.

However, when your system is a large, handling millions of requests every day, and also has very stringent scalability requirements, you might be better off with the microservices approach.

**An event driven architecture would be the one best suited to your needs.**

## Advantages of Event Driven Architectures

### Improved Flexibility and Maintainability

One of the most important needs of an application is maintainability. Ease of maintainability comes with the proper **separation of concerns**.

In our example above, the OrderService is only responsible for collecting the order and placing it in the queue. It does not worry about how it is going to be processed, who is going to process it, and so on.

Similarly, the StockService is only responsible for updating the stock, and the same with the other microservices.

If there is a need for an additional processing step on an order, you write a new microservice to listen on the queue, and easily integrate it into the system.

Such an architecture is clearly extensible, and also easily maintainable, due to separation of concerns.

### High Scalability

Let's consider an example.

One fine day, a large volume of emails need to be sent out. In that case, you have the freedom to create a large number of instances of the EmailService to handle additional load. A similar thing can be done to the other microservices in the mix, depending on your needs.

This is a clear advantage of this architecture, over using a single component to handle all the functionality.

### Improved Availability

Let's say one of the services listening for order events from the queue, such as the PackageService, goes down.

In a monolithic approach, any one functionality going down would mean the application cannot process orders any more.

In the case of an event driven architecture, the PackageService going down would not prevent the OrderService from putting the order event into the Queue. The OrderService can notify the user that their request was received successfully.

A notification request would then be sent to the troubleshooting team about the PackageServer going down, and while it is being restored, the order event remains in the Queue. It can be processed by all the other services during this time, and when PackageService comes back up, it can process the event as if nothing untoward has happened.

### Good Reliability

Event driven architectures ensure a good standard of reliability for the system as a whole. However, the individual microservices can function with different levels of reliability. For example the StockService normally needs to ensure a high level of reliability, the EmailService and SMSService a medium level of reliability, and the PackageService — between low and medium levels of reliability.

### Good Performance and Responsiveness

Note that in our example event driven architecture, all that the OrderService does is receive the order and place it on the queue, before acknowledging the user. The user does not need to wait till all the steps are performed on an order. This has high high responsiveness, and is seen by the user as high performance.

## Summary

In this article, we had a look at event driven architectures. We saw that event driven architecture is ideal for implementing applications with high load and strict scalability needs.

We saw an example of an online shopping application that employs such an architecture. The advantages of using such an architecture include improved flexibility and maintainability, high scalability, imrpoved availability, good reliability, and good performance.