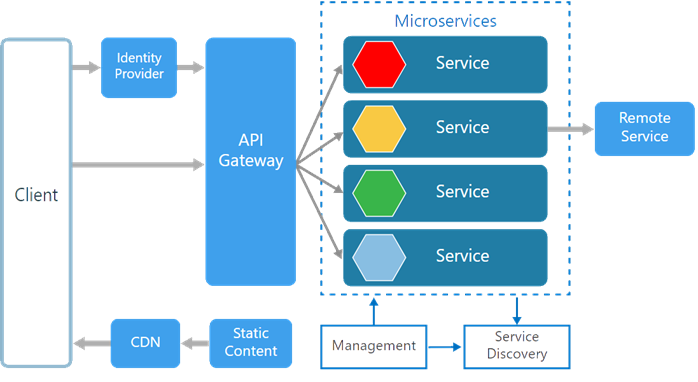
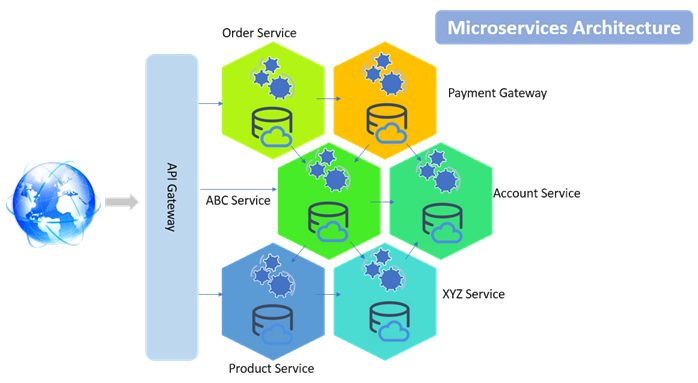
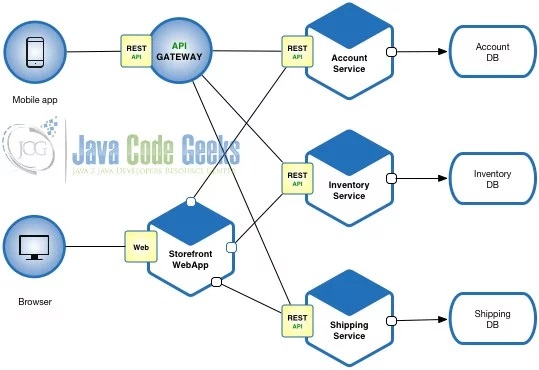
# Microservices

* Microservices architecture is an approach to create small services
* In Microservices we decompose a big application into multiple small services. Each microservice running in their own space that can communicate each other.
* Microservices are independent services directly calling their own database.
* Microservices are one of the hottest topics in the software industry, and many organizations want to adopt them.

The following is the diagrammatic representation of microservices architecture.



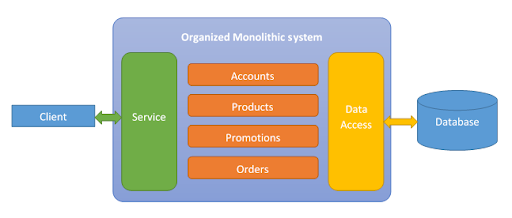




# Advantages of Microservices Architecture

* Microservices are smaller in size.
* Microservices are independent of each other, meaning that if one of the microservices goes down, there is little risk of the full application shutting down
* MicroServices can be written in different programming language, databases, and tools for each service
* Microservices are easier to develop, deploy, and debug
* Microservices can be scaled quickly and can be reused among different projects
* Microservices work well with containers like Docker
* Microservices are Support for continuous integration and delivery
* Better fault isolation keeps other services to work even though one got failed.
* Services can be deployed to different servers or even different datacentres.
* Interaction with other services in a well-defined protocol
* Microservices are Reliable and self-healing

# Monolithic System



# Monolithic System Limitations

* In Monolithic System If any single application function or component fails, then the entire application goes down. Imagine a web application with separate functions including payment, login, and history. If a particular function starts consuming more processing power, the entire application’s performance will be compromised.
* developers cannot work independently to develop or deploy their own modules and must remain totally dependent on others, increasing overall development time. So it Impacts both the development and application deployment stage.
* In monolithic architecture, the database remains the same for all the functionalities even if an approach of service-oriented architecture is followed, whereas in microservices each service will have their own database.

# problems with a monolithic system.

# Problem 1

As there is one codebase, it grows gradually. Every programmer, whether it's a UI Developer or a business layer developer, commits in same code base, which becomes very inefficient to manage. Suppose one developer only works in the JMS module, but he has to pull the whole codebase to his local and configure the whole module in order to run it on a local server. Why? He should only concentrate on the JMS module, but the current scenario doesn't allow for that.

# Problem 2

As there is one code base and modules are dependent on each other, minimal change in one module needs to generate all artifacts and needs to deploy in each server pool in a distributed environment.

Suppose in a multi-module project that the JMS module and business module are dependent on the data access module. A simple change in the data access module means we need to re-package the JMS module and business module and deploy them in their server pool.

# Problem 3

As monolithic software uses a three-tier architecture, three cross-functional teams are involved in developing a feature. Even though a three-tier architecture allows for separation of responsibility, in the long-run, the boundaries are crossed and the layers lose their fluidity and become rigid.

Suppose an inventory management feature has been developed. The UI, business layer, and data access layer have their own jobs. But everyone wants to take control of the main business part so that when defects come up, they can solve them and are not dependent on another layer's developer. Due to this competition, those boundaries end up being crossed, which results in inefficient architecture.

# Problem 4

In many projects, I have seen that there is a developer team and another support team. The developer team only develops the project, and after it's released, they hand it over to the support team. I personally don't support this culture. Although some knowledge transfer happens during the handover, it doesn't solve the problem. For critical incidents, the support team has to get help from the developer team, which hurts their credibility.

# Problem 5

As our system is monolithic, so is our team management. Often, we create teams base on the tier — UI developers, backend developers, database programmers, etc. They are experts in their domains, but they have little knowledge about other layers. So when there's a critical problem, it encompasses each layer, and the blame game starts. Not only that, but it takes additional time to decide which layer's problem it is and who needs to solve the issue

# Microservice Benefits

# Benefit 1

As in monolithic software, you only develop in one language, say .Net or Java, as the code base. But with microservices, as each service is independent and each service is a new project, each service can be developed in any language that is best fits for the requirement.

**Benefit 2**

The developer is only concentrated on a particular service, so the code base will be very small, and the developer will know the code very well.

**Benefit 3**

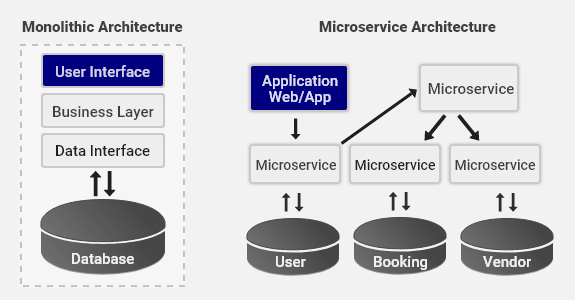
When one service needs to talk with another service, they can talk via API, specifically by a REST service. A REST service is the medium to communicate through, so there is little transformation. Unlike SOA, a microservice message bus is much thinner than an ESB, which does lots of transformation, categorization, and routing.

**Benefit 4**

There is no centralized database. Each module has its own, so there's data decentralization. You can use NoSQL or a relational database depending on the module, which introduces that polyglot persistence.

As microservices communicate through REST, the transformation scope is very small — only one service is dependent on another service via API call.

# Monolithic vs Microservices Architecture



# Microservices and the 12 Factor App

A collection of guidelines to help to build good microservice-based applications is the Twelve-Factor app. It is a collection of 12 tips for the development of horizontally scalable applications.

The 12 factors app is a methodology to build applications that are suited for modern architectures (e.g. cloud and containers and that operates well in DevOps operating environments)

**Why 12 factors App**

12 factors app leads to applications that are portable, high scalable and easily automatable.

**Factor 1: Codebase**

One single code base many deployments

Store your app code base in a version control system (e.g. git), dedicate one repository for one app, create many deployments out of that repository.

Build on top of one codebase, fully tracked by a Version Control System (VCS). Deployments should be automatic, so everything can run in different environments without work. You should always have one repository for an individual application to ease CI/CD pipelines.

**Factor 2: Dependencies**

Do not copy any dependencies to the project codebase, instead use a package manager. Always remember to use the correct versions of dependencies so that all environments are in sync and reproduce the same behavior.

**Factor 3: Config**

Store the config in Environment Variable. There should be a strict separation between config and code. The code should remain the same irrespective of where the application is being deployed, but configurations can vary.

**Factor 4: Backing Services**

Treat backing services as attached resources as your services should be easily interchangeable. You must be able to easily swap the backing service from one provider to another without code changes. This will ensure good portability and help maintain your system.

**Factor 5: Build, Run, Release**

A twelve-factor application requires a strict separation between Build, Release and Run stages. Every release should always have a unique release ID and releases should allow rollback. Automation and maintaining the system should be as easy as possible. Then you put everything together in something that can be released and installed in the environment and then be able to run it.

**Factor 6: Stateless Processes**

You should not be introducing state into your services, applications should execute as a single, stateless process. The Twelve-factor processes are stateless and share-nothing. This factor lies at the core of microservices architecture.

**Factor 7: Port Binding**

Your service should be visible to others via port binding. If you built a service, make sure that other services can treat this as a resource if they wish. The twelve-factor app is completely self-contained.

**Factor 8: Concurrency**

Break your app into much smaller pieces rather than trying to make your application larger (by running a single instance on the most powerful machine available). Small, defined apps allow scaling out as needed to handle the varying loads. Each process should be individually scaled, with Factor 6 (Stateless), it is easy to scale the services.

**Factor 9: Disposability**

Processes should be less time-consuming. Make sure you can run and stop fast. And that you can handle failure. Without this, automatic scaling and ease of deployment, development- are being diminished. You can achieve this with containers.

**Factor 10: Dev-Prod Parity**

Keep development, staging, and production as similar as possible so anyone can understand it and release it. Continuous deployment needs continuous integration based on matching environments to limit deviation and errors. This also implicitly encourages a DevOps culture where Software Development and Operations are unified. Containerization is a huge help here.

**Factor 11: Logs**

Treat logs as event streams. Logging is important for debugging and checking up on the general health of your application. At the same time, your application shouldn’t concern itself with the storage of this information. Instead, these logs should be treated as a continuous stream that is captured and stored by a separate service.

**Factor 12: Admin Processes**

Run admin/management tasks as one-off processes — tasks like database migration or executing one-off scripts in the environment. To avoid messing with the database, use the tooling you built alongside your app to go and check the database.