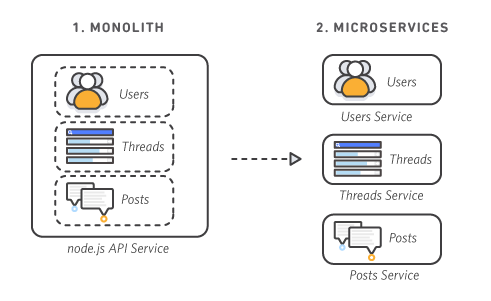
**Microservices Architecture**

**What is Microservice?**

* The goal of microservices architecture is to break down a large, monolith application into smaller.
* Isolation and independence. Each service runs in its own process, can have its own language and database.
* Communicate with other microservices through well-defined APIs, which will be using **API Gateway (Ocelot).**
  + Microservices are registered with Ocelot through its configuration.
  + When a client makes a request, it is directed to the Ocelot API Gateway.
  + Client only needs to interact with single API even though the request may have been routed to multiple microservices.
* Support continuous deployment practices, as each service can be developed, tested, and deployed independently.
* Ensure that failures in one service do not necessarily impact others.

**Monolith vs. Microservices Architecture**



|  |  |  |
| --- | --- | --- |
| Feature | Monolith Architecture | Microservices Architecture |
| Structure | Single, tightly integrated application | Multiple, independence and loosely coupled |
| Size | Large codebase | Smaller, focused codebase for individual services only |
| Development | Single team working on entire codebase | Multiple teams working on different services |
| Database | Shared database | Each microservice can have its own database |
| Testing | Testing to entire application | Testing individual service |
| Failure Impact | Single point failure will impact entire application | Failure only impact to its own service, reducing impact |
| Maintenance | Changes may require rebuilding and redeploying the entire application | Easier maintenance with the ability to update individual services |

**Pros of Microservices Architecture:**

* **Scalability**
* Microservices can be individually scaled based on the specific needs of the service which allows for more efficient resource utilization.
* **Flexibility and Agility**
* Microservices enable independent development and deployment of services. This agility is beneficial for teams working on different services without affecting the entire system.
* **Technology Diversity**
* Different services can use different technologies, languages, and frameworks. This flexibility allows teams to choose the best tools for the specific requirements of each service.
* **Fault Isolation**
* If one microservice fails, it doesn't necessarily bring down the entire system. Fault isolation makes it easier to identify and address issues in a specific service.
* **Continuous Deployment**
* Microservices support continuous integration and continuous deployment (CI/CD) practices, making it easier to release updates and new features frequently.
* **Autonomous Teams**
* Microservices enable teams to work independently on different services, promoting autonomy and faster development cycles.
* **Easier Maintenance**
* Maintenance and updates can be focused on specific services without disrupting the entire system, making it easier to manage and improve.
* **Better Scaling for Development Teams**
* Smaller teams can own and manage individual microservices, reducing coordination overhead and allowing for more focused development efforts.

**Cons of Microservices Architecture:**

* **Complexity**
* Managing a distributed system of microservices introduces complexity in terms of communication, coordination, and monitoring.
* **Increased Network Communication**
* Microservices rely on network communication, and excessive communication can introduce latency and potential points of failure.
* **Consistency Across Services**
* Ensuring consistency in terms of logging, monitoring, and other operational aspects can be challenging.
* **Data Management**
* Coordinating data consistency across microservices can be challenging. Distributed data management requires careful planning.
* **Operational Overhead**
* Operating and monitoring a system with many microservices can be more complex than managing a monolithic application.
* **Testing Challenges**
* End-to-end testing can be challenging due to the need to test interactions between multiple services. Mocking services may be necessary.
* **Initial Development Cost**
* Developing a microservices architecture from scratch can be more time-consuming and resource-intensive compared to building a monolithic application.
* **Security Concerns**
* Implementing security across microservices requires additional considerations, and the distributed nature of microservices introduces potential security challenges.

**How to manage Microservices:**

 **Monorepo**

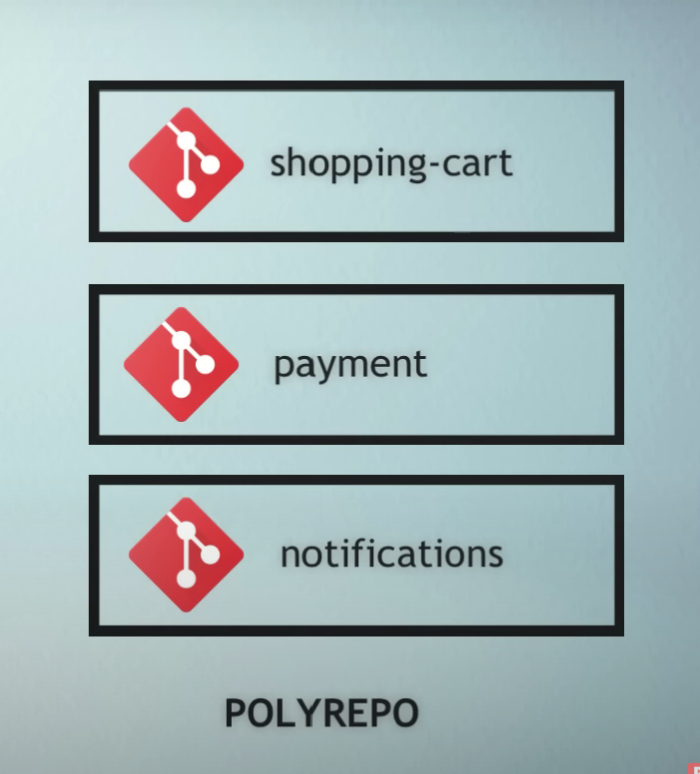
1 git repository for all the services

(contains many projects)

1 code base with folders

: a directory for each service / project

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * make the code management and development easier. * clone and work only with 1 repo. * changes can be tracked, tested and released together. * share code and configurations. | * easier to break this criterion (independent and isolated) and develop more tightly coupled code. * big source code, means git interactions becomes slow. * additional logic necessary to make sure only service is built and develop which had code changes. * all project and teams are affected if there is some issue (if main branch has been break, other services and their pipelines will be blocked) |

**Polyrepo**

1 git repository for each service   
 (own pipeline for each repository)  
 : code is completely isolated.

: clone and work on them separately.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * helps to keep an overview. * create shared secrets, CI/CD variables, runners. | * Cross-cutting changes is more difficult. * changes spread across projects must be submitted as separate merger requests instead of having a single, atomic MR. * switching between projects tedious * searching, testing and debugging, sharing resources is more difficult. |

**How to implement Microservices Architecture:**

1. **Code API in Visual Studio:**

Develop your microservices by coding APIs in Visual Studio. Each microservice should ideally focus on a specific business capability or domain.

**E.g.:**

* Create microservices for different modules or business capabilities such as CustomerWebAPI, ProductWebAPI and OrderWebAPI.
* Microservice is developed as a separate project in Visual Studio with its own set of APIs.

A screenshot of a computer

Description automatically generated

* Then, right-click on the each microservice, it can add container orchestrator support which used to add microservice into Docker Compose (Put all container into one)
* What will be contained in container orchestrator support?
  + Dockerfile (Contains all the commands a user could call on the command line to assemble an image.
  + Automatically by reading the instructions from a Dockerfile.



1. **Deploy Microservices with Docker:**

Deploy your Dockerized microservices to a container orchestration platform such as Docker Compose, Docker Swarm or Kubernetes. These platforms help manage the deployment, scaling, and lifecycle of your containers.

**E.g.:**

* Use Docker Compose because it is used for configuring multiple containers in the same host.
* By running the Docker Compose, it will start all the services defined in the docker-compose.yml file.

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

1. **API Gateway (Ocelot):**

Implement an API Gateway using Ocelot. The API Gateway will be responsible for handling incoming requests, routing them to the appropriate microservices, and potentially providing additional functionalities like authentication, authorization, and load balancing.

**E.g.:**

* Implement Ocelot (Install Ocelot in NuGet Package)
* Define the deployment configurations, services, and networking.

1. **Configure Ocelot:**

Configure Ocelot to route requests to the respective microservices based on defined rules. Ocelot may also handle service discovery, load balancing, and other gateway-related tasks.

**E.g.:**

* Two terms in Ocelot configurations
  + Upstream: Used for incoming request (“Get”, “Post”, “Put”, “Delete”)
  + Downstream: Request to be routed to the API services (HTTP) A screen shot of a computer program

    Description automatically generated

1. **Frontend Communication:**

Frontend applications communicate with the API Gateway (Ocelot) instead of directly with individual microservices. This provides a single-entry point for clients and simplifies the communication model.

1. **Run the System:**

With Docker and Ocelot in place, your microservices architecture is ready to run. Docker containers for microservices are managed by the orchestration platform, and the API Gateway handles incoming requests and routes them to the appropriate services.

1. **Scale and Manage:**

Use the capabilities of Docker orchestration tools to scale your microservices horizontally based on demand. Docker Swarm or Kubernetes can manage the number of container instances running for each microservice.

1. **Monitoring and Management:**

Implement monitoring and management tools to track the health, performance, and logs of your microservices and the API Gateway. This helps ensure the reliability and availability of your system.

By following this approach, you create a modular, scalable, and maintainable microservices architecture. Docker facilitates containerization and deployment, while Ocelot acts as the API Gateway to manage the routing and gateway functionalities. This architecture allows for independent development, deployment, and scaling of microservices.

**What will be used during developing projects in microservices?**

* Visual Studio
* Docker
* Ocelot

**Summary**

Microservices architecture involves breaking down large applications into independent services that operate in isolation. These services run in separate processes, using individual languages and databases, and communicate through well-defined APIs, often managed by an API Gateway like Ocelot. This enables continuous deployment, allowing each service to be developed, tested, and deployed independently. Failures in one service do not necessarily impact others, promoting fault isolation. Microservices offer scalability, flexibility, and agility, supporting diverse technologies and enabling autonomous teams to work independently. While it simplifies maintenance and facilitates team scaling, challenges include increased complexity, network communication, consistency, and testing difficulties.

**Which project is suitable for Monorepo or Polyrepo?**

**Monorepo**

* **Code Sharing:**

When there is significant code sharing among microservices.

* **Consistent Versioning:**

When need consistent versioning across services.

* **Atomic Changes:**

When changes often involve multiple services that need to be deployed together.

* **Which Company used it?**

Google, Facebook (Meta), Twitter, Microsoft, and Uber, some of the most advanced technology companies globally, use monorepos. (Info published on 5 Sep 2023)

**Polyrepo**

* **Autonomous Teams:**

When different teams are responsible for different services and prefer autonomy in their development and deployment processes.

* **Scaling Development:**

When the organization is large, teams need to scale development independently.

* **Independence:**

When microservices are relatively independent, and changes in one service don't impact others significantly.

* **Which Company used it?**

Netflix, Amazon, and Lyft are famous companies using the multi-repo approach. (Info published on 4 Sep 2023)

**References:**

.NET Core Web API Microservice with SQL Server

<https://www.youtube.com/watch?v=2p01iafOxUw&list=PLzewa6pjbr3JQKhB_U_FiuYwQC70i-TyU&index=6>

ASP.Net Core API Gateway - Ocelot API Microservice:

<https://www.youtube.com/watch?v=k4l3Ptd4yjw>

Ocelot API Gateway JWT Authentication Tutorial:

<https://www.youtube.com/watch?v=P2osfctiHAc>