**Master Microservices with Spring Boot, Docker, Kubernetes**

<https://capgemini.udemy.com/course/master-microservices-with-spring-docker-kubernetes/>

## Instructors

[**Eazy Bytes**](https://capgemini.udemy.com/user/552e9eb07495d/)[**Madan Reddy**](https://capgemini.udemy.com/user/challa-madan/)Training on Java, Full Stack, Spring & ORM Java Full stack | Cloud enthusiast | Learner | Instructor

**GitHub page**:  
<https://github.com/eazybytes/microservices>  
---------------------------------------------------------------------------------------------------------------------------------------------------------  
**Microservices**: These days many organizations migrating monolithic application into microservices, the reason is microservice architecture makes application **easier to scale, faster to develop, enable innovation and also helps to organizations to accelerating the time to take new enhancement to the market**. So, microservice is going to be a demanding skillset for new, few years.

**Course agenda**:

* Welcome to the world of microservices.
* Building microservices building logic using spring boot.
* How do we right size our microservices and identify boundaries.
* How to containerize our microservices using Docker.
* Configurations management in microservices using spring cloud config.
* Service discovery and service registration in microservices using eureka.
* Building an edge server for microservices using spring cloud gateway.
* Making microservices resilient using Resiliency4J patterns.
* Observability and monitoring of microservices using Grafana, Prometheus etc.
* Securing microservices using OAuth2/OpenID, spring security.
* Event driven microservices using RabbitMQ, spring cloud Functions and stream.
* Event driven microservices using Kafka, spring cloud functions and stream.
* Container orchestration using Kubernetes.
* Deep dive on Helm (Kubernetes package manager).
* Deploying microservices into cloud Kubernetes cluster.
* Many best practices, techniques followed by real time microservice developers.

**Important points:**\* Explanation of DTO pattern  
<https://martinfowler.com/eaaCatalog/dataTransferObject.html>

**For documenting the APIs:**

<https://www.openapis.org/>

This is a very complex task and to document my rest service it needs lot of time that’s does not correct it’s going to be super easy with the help of the library called <https://springdoc.org/>

It has all the details and all the steps we need to follow to get started with documentation of REST API.

**<dependency>**

**<groupId>org.springdoc</groupId>**

**<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>**

**<version>2.3.0</version>**

**</dependency>**

This dependency is only supports for spring boot greater than or equal to 3.0  
<http://localhost:8080/swagger-ui/index.html>

Using this URL, we can access the swagger UI

**@Tag:** Provide some information specific to all the APIs present inside the controller class

**Introducing one plugin that we can install inside our spring boot which is going to improve our productivity whenever we try to write a java program. The tool is codeium.**

Goto this website codeium.com so, this is a generative AI tool which is built focusing on the coding only it is not a generic tool like a ChatGPT.

We install this option in our IntelliJ -> settings -> plugins -> search codeium -> install. Once install we must restart our IDE then we can be able to access the codeium features in our project.

Why we use application.yml file instead of an application. properties?

* **Readability and Structure:**  
  YAML (YAML Ain't Markup Language) provides a more readable and structured format compared to properties files. It supports hierarchical data, which is common in Spring Boot configurations.
* **Handling Arrays and Lists**:  
  YAML provides native support for lists and arrays, which can be tricky to represent in. properties files.
* Same YAML format is to use inside Docker, Kubernetes or any cloud providers go like AWS, GCP, AZURE etc. since many places this been used.

When to use application. properties?

* If we have a small configuration file or you prefer the simplicity of key-value pairs without a need for hierarchical or structured data, application. properties can still be a good choice.

**@MappedSuperClass** – This annotation indicates to the spring data jpa framework this class is going to act as a super class for all my entities wherever trying to extend the base entity class.

**ModelMapper** – We have a two such libraries to automatic mapping between entity and Dto class they are:   
modelmapper and mapstruct using these two libraries we can do conversion between Dto to entity and entity to Dto by adding few dependencies and writing a few lines of code. But this is not a good standard the reasons are in the spring boot framework officially not recommended these dependencies.

If we add these dependencies the client and architect or our leads, they may not agree bring these dependencies into our microservices. Because these are opensource and we never know there is some security issues with these libraries also these websites are not officially certificate as well.

Whenever we trying to update or delete custom methods that we have written use mainly two annotations are:  
**@Tansactional** and  
**@Modifying** – It tells to the spring data jpa framework this method is going to modify the data so that’s why please execute the query of this method inside a transaction. This reason we mention transactional annotation as well is there is some error happen rollback the transaction.

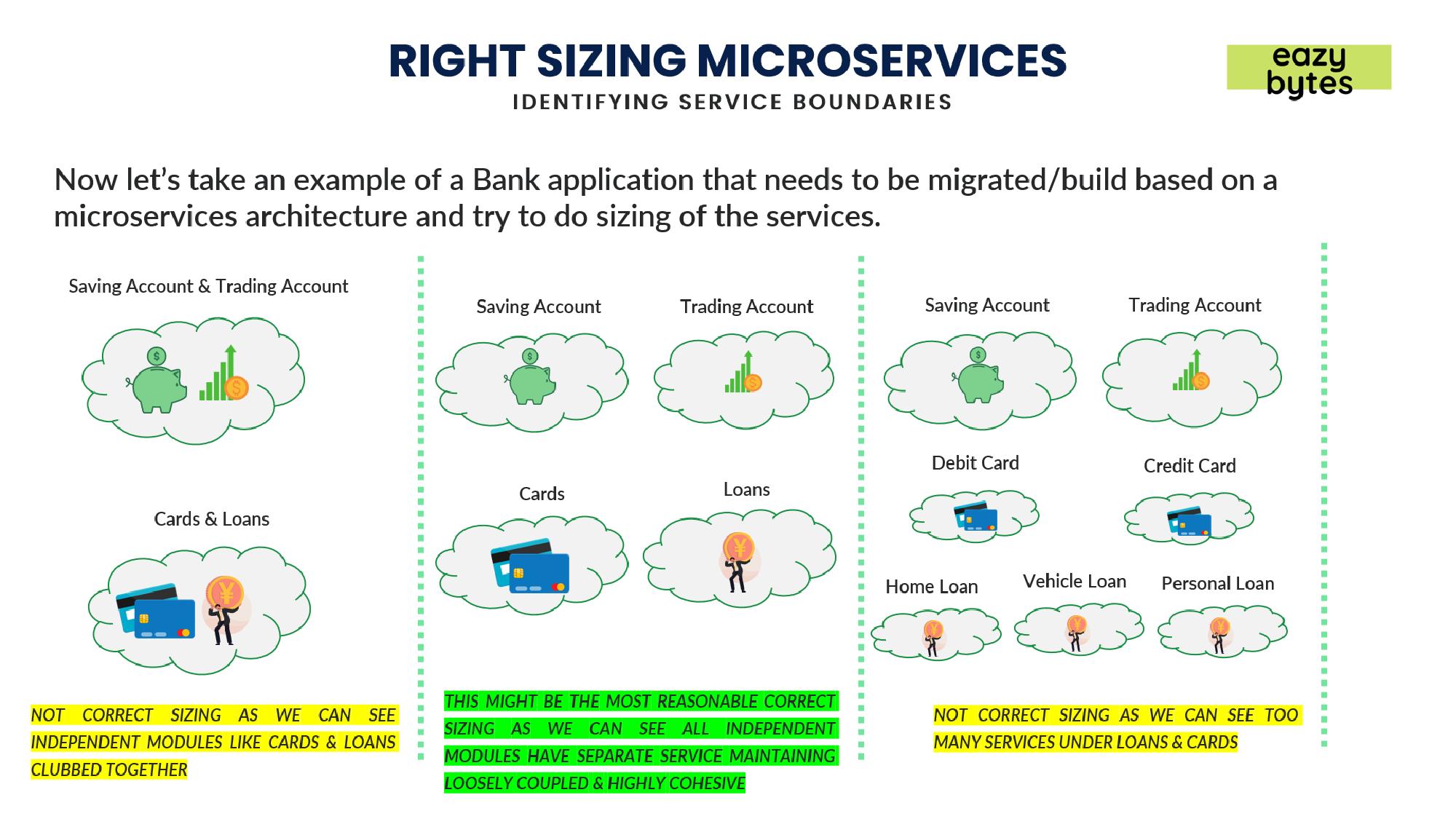
**Different annotations and classes that supports building REST services:**

1. @RestController – Can be used to put on top of a call, this will expose our methods as REST APIS, Developers can also use @Controller + @RequestBody for the same behaviour.
2. @ResponseBody – Can be used on top of the method to build a REST API when we are using @Controller on top of the java class.
3. ResponseEntity<T> - This is a class it allows developers to send response body, status, and headers on the HTTP response.
4. @ControllerAdvice – Is used to mark the class as a REST controller advice. Along with @ExceptionHandler, this can be used to handle exceptions globally inside app. We have another annotation @RestControllerAdvice which is same as @ControllerAdvice + @ResponseBody.
5. RequestEntity<T> - This is a class, allows developers to receive the request body, headers in HTTP request.
6. @RequestHeader & @RequestBody – Is used to receive the request and header individually.

**Approaches to identify boundaries & right size microservices:**

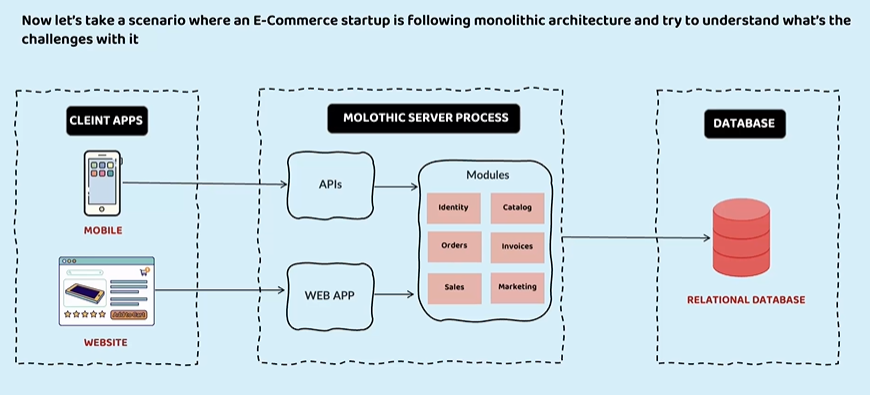
There are two types of sizing:  
1) Domain-Driven sizing.  
2) Event storming sizing.

<https://www.lucidchart.com/pages/>?

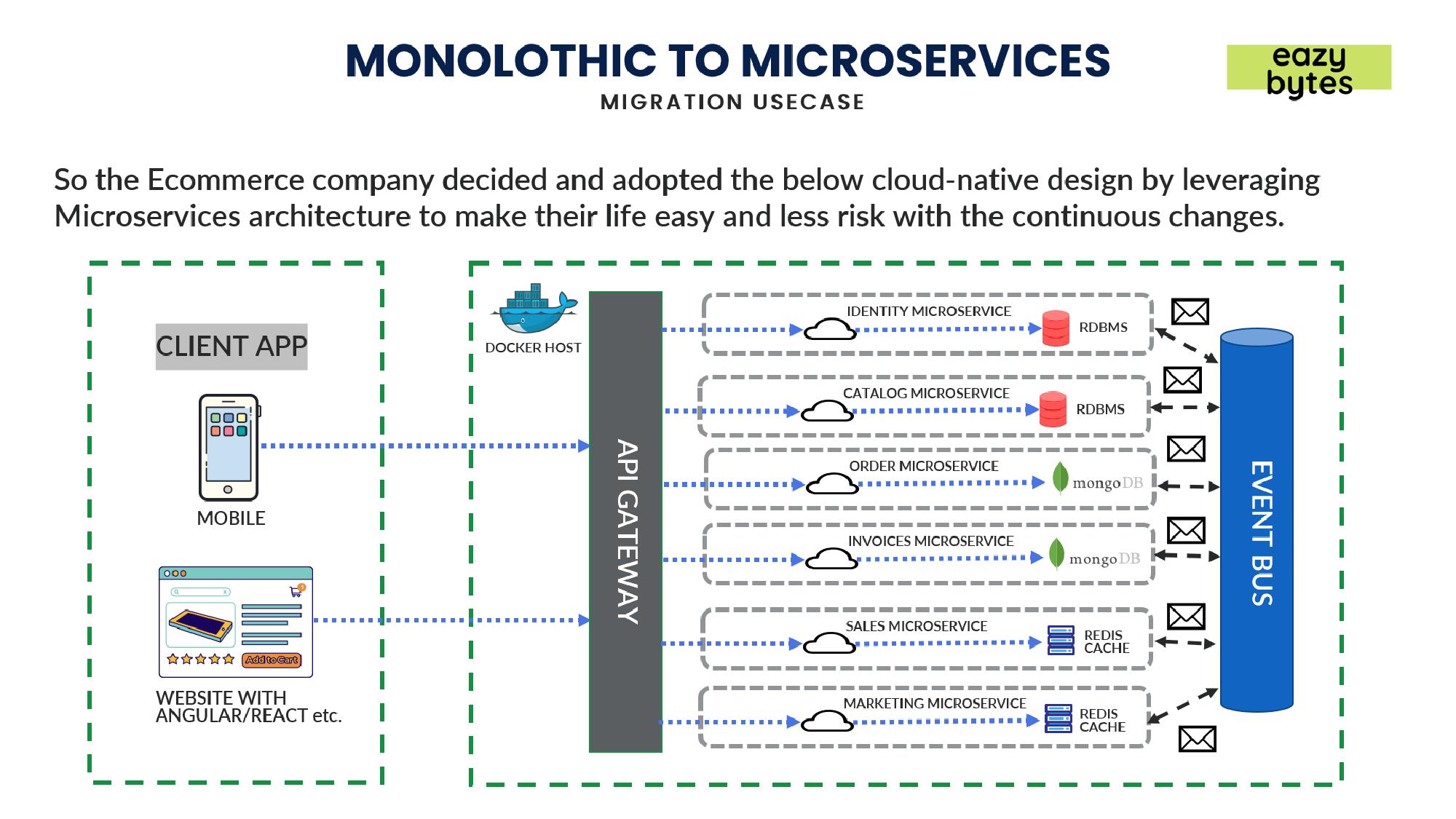


**Conversion of Monolithic application to Microservices:**

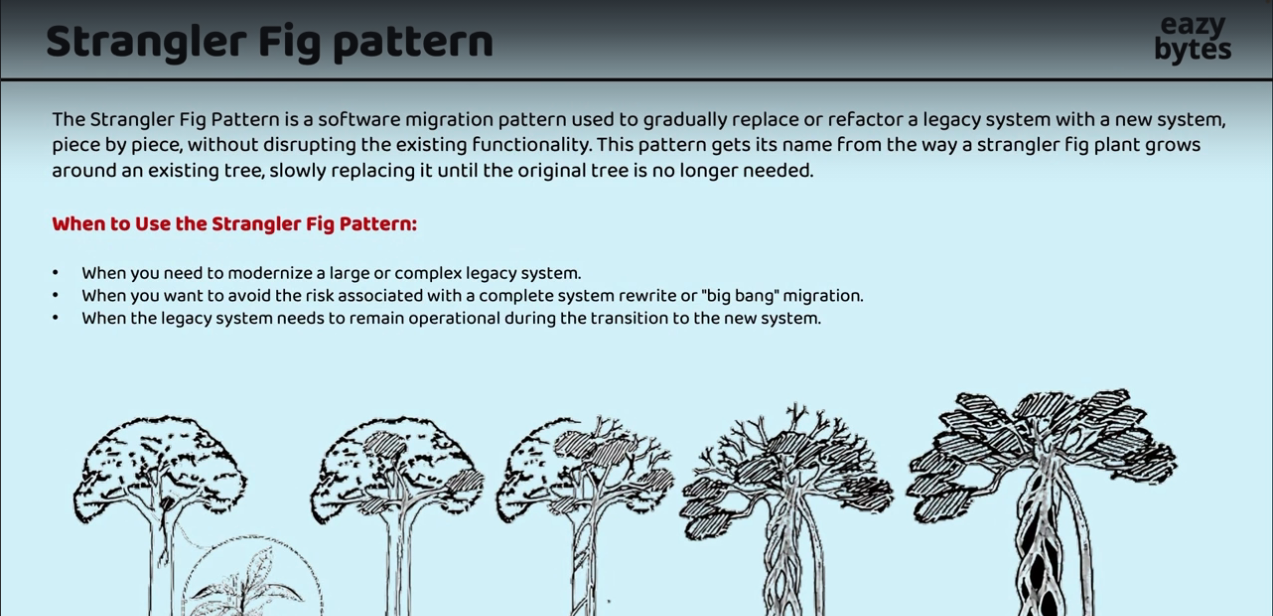
**Monolithic Application:**

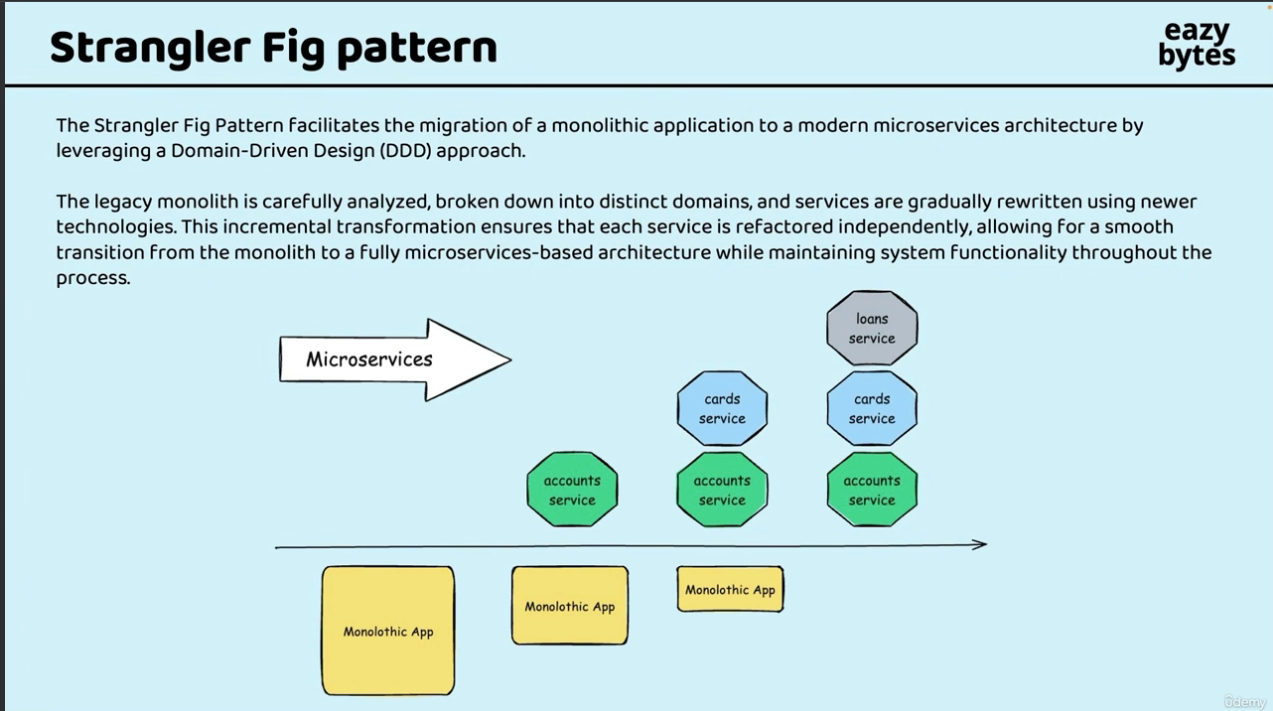


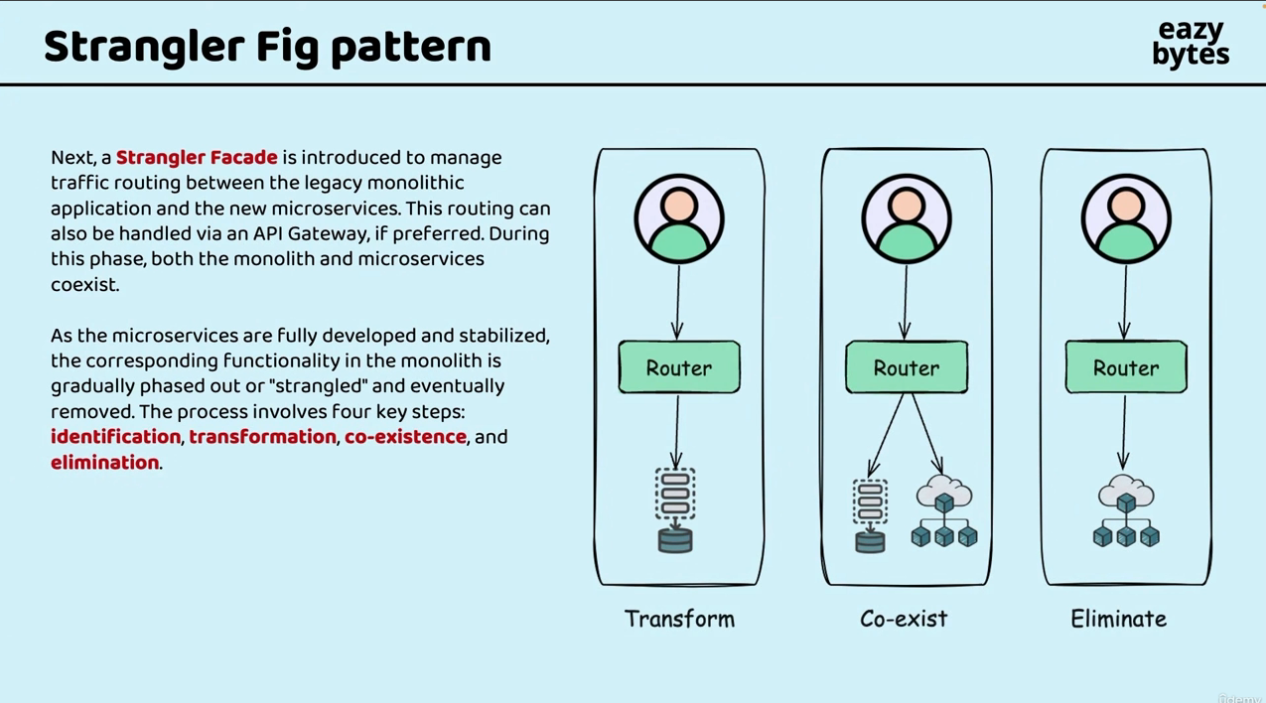
**Conversion of Microservices Application:**



**How are migrating our legacy into the Microservices based application:** UsingStrangler Fig pattern

****

****

****

**Deployment, Portability and scalability of Microservices:**

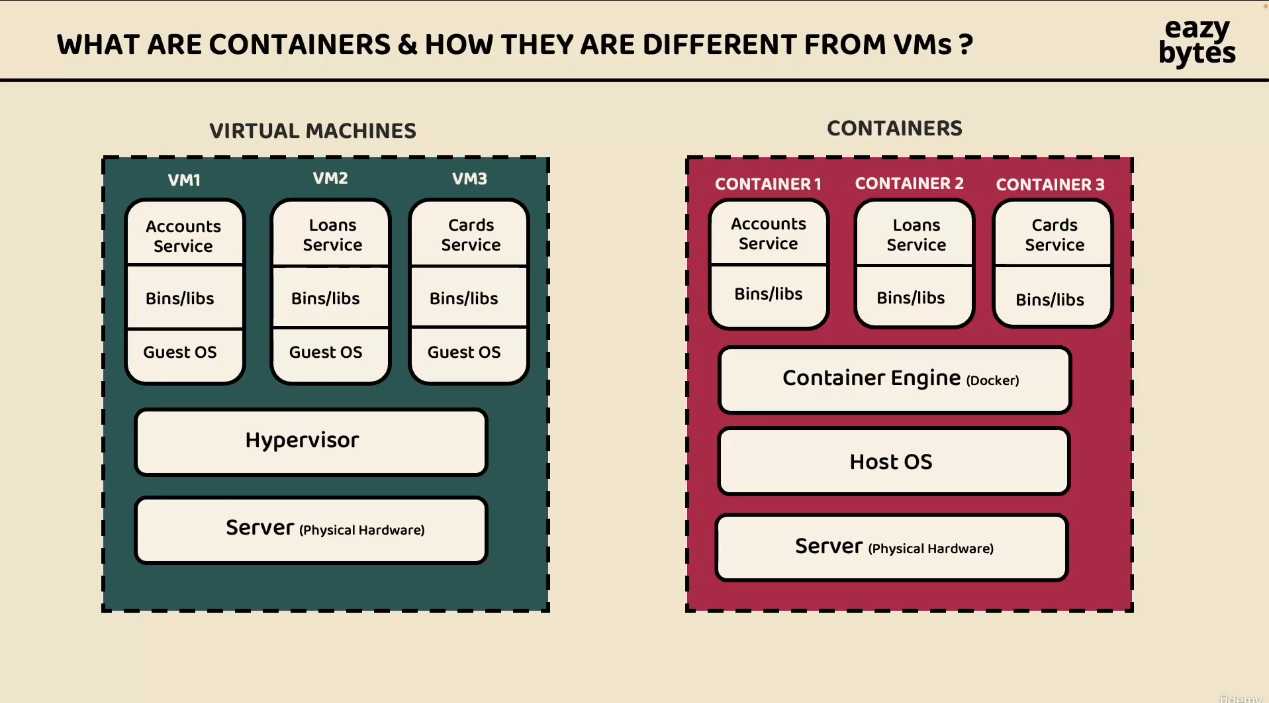
Developer -> GitHub Repository or an other version control system -> From there we need to deploy into the development environment -> Once dev build is stable we need to move to the UAT environment, SIT environment or QA environment -> Once the testing is completed we need to deploy Production replica environment -> Once the pre-production testing is completed eventually the deployment as to be done inside the production environment.

**Deployment** – How do we deploy all the tiny 100s of microservices with less effort and cost?  
**Portability** – How do we move our 100s of microservices across environments with less effort, configurations and cost?  
**Scalability** – How do we scale our applications based on the demand on the fly with minimum effort and cost?

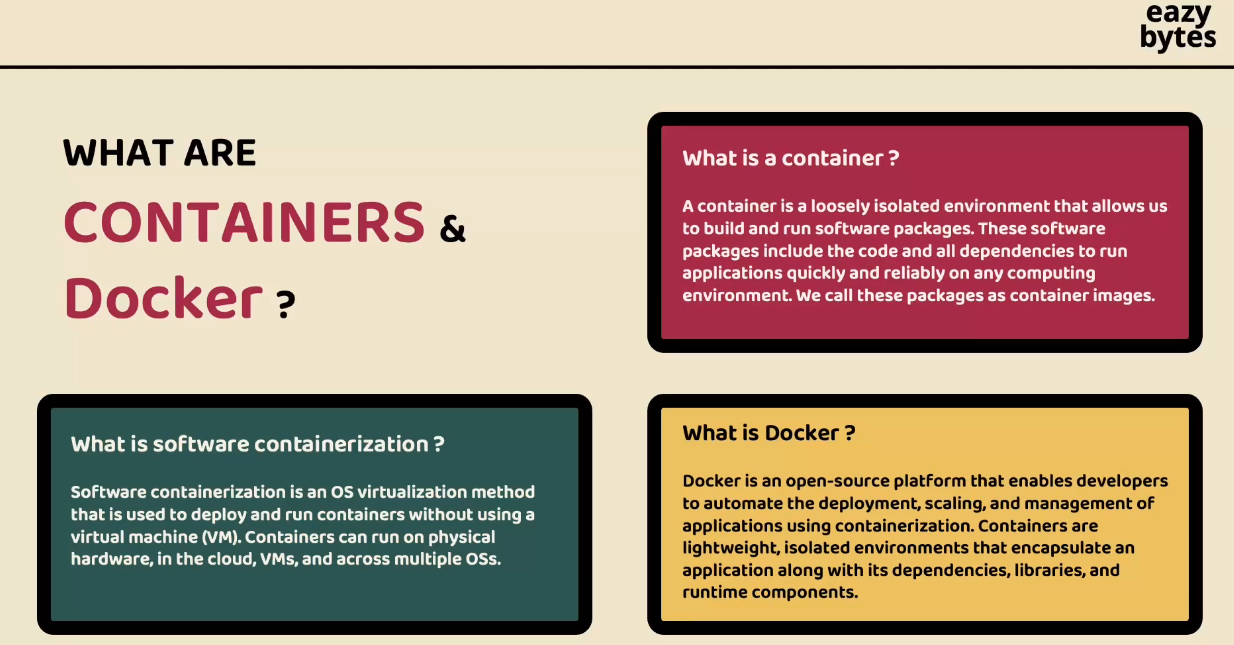
So, these are the three challenges that we face whenever we try to build microservices with a traditional monolithic mindset.

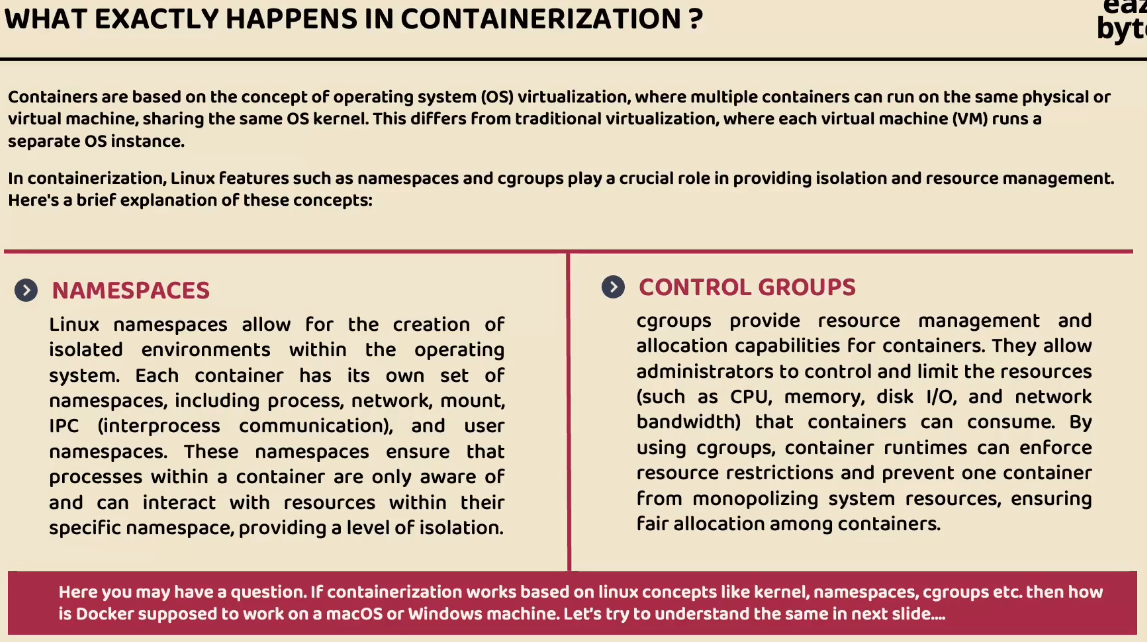
To overcome the above challenges, we should **containerize** our microservices. Why?  
Containers offers a self-contained and isolated environment for applications, including all necessary dependencies. By containerizing an application, it becomes portable and can run seamlessly in any cloud environment. Containers enabled unified management of applications regardless of the language or framework used.

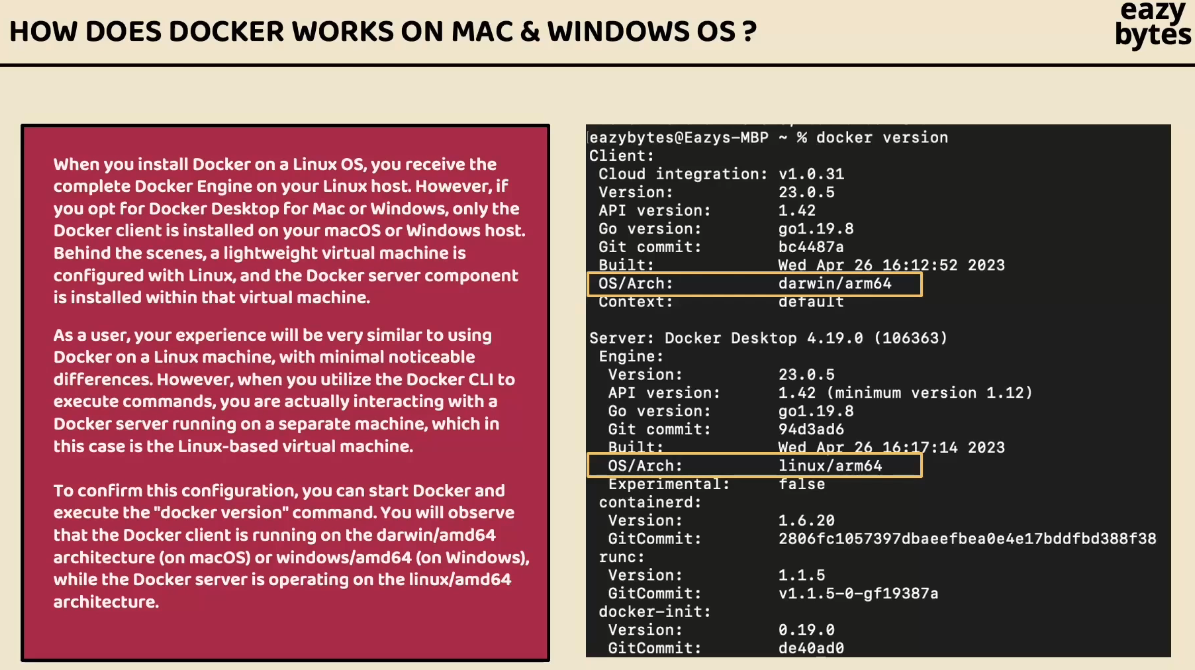
How do we containerize our maven application?  
-> **Docker** is an open-source platform that “provides the ability to package and run an application in a loosely isolated environment called a container”.

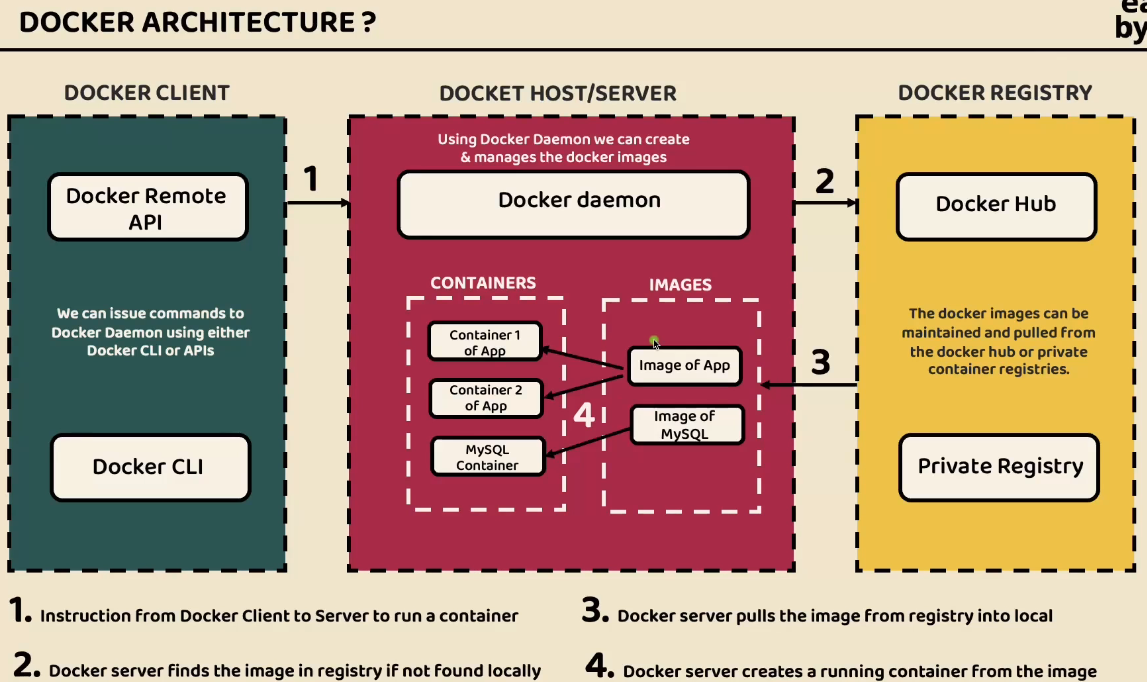


Containers they are going to become a light weight, creating a container destroying a container or restarting a container is going to take only few seconds. So, this is the very first advantage we have container compared to virtual machines.  
The other advantage is that we have with container is inside the same virtual machine they can have a separate isolated virtual environment.  
Deployment is very easy because it is a lightweight component. The main difference of container and virtual machines is containers they don’t need the Guest operating system not hypervisor to assign the resources, instead they will use the container engine which is Docker.









**Install Docker in our local system:**

Step – 1 <https://www.docker.com/> visit the official docker website. Install the application based on the operating system.

Need instructions to install docker visit this page - [https://docs.docker.com/desktop/?\_gl=1\*h620h1\*\_gcl\_au\*MTg3NzgyNjA4LjE3MjgyODk1MzY.\*\_ga\*MjAxMTMwODkyNC4xNzI3MTU4ODE4\*\_ga\_XJWPQMJYHQ\*MTcyODI4OTUzNS4yLjEuMTcyODI4OTcxMC4yMy4wLjA](https://docs.docker.com/desktop/?_gl=1*h620h1*_gcl_au*MTg3NzgyNjA4LjE3MjgyODk1MzY.*_ga*MjAxMTMwODkyNC4xNzI3MTU4ODE4*_ga_XJWPQMJYHQ*MTcyODI4OTUzNS4yLjEuMTcyODI4OTcxMC4yMy4wLjA).