



UNDERSTANDING MICROSERVICES

Marija Zaric, Borislav Gajic, Nemanja Romanic

Novi Sad, 2023

AGENDA

01 Common software architecture pattern

02 Microservices architecture

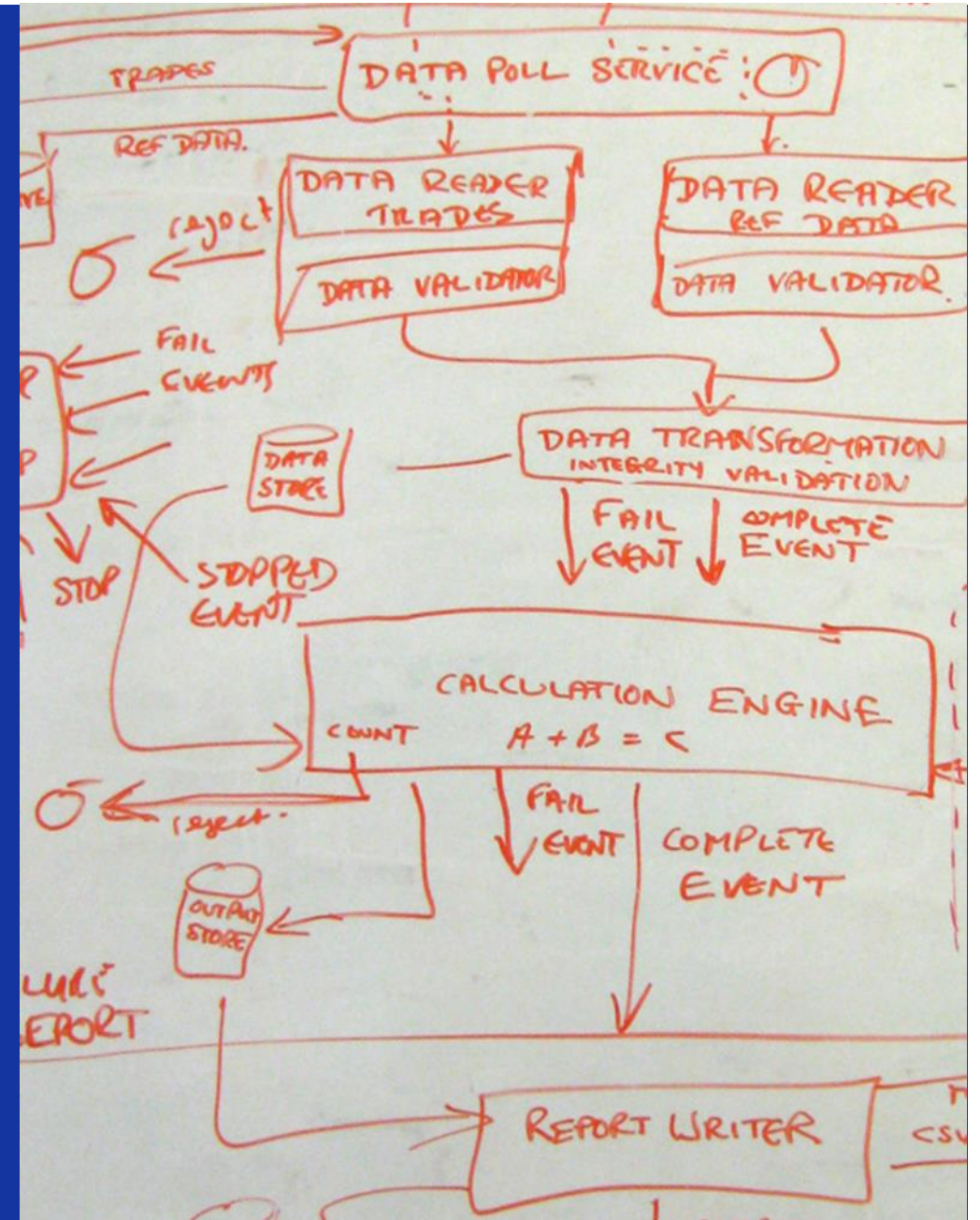
03 Liquibase

04 Docker

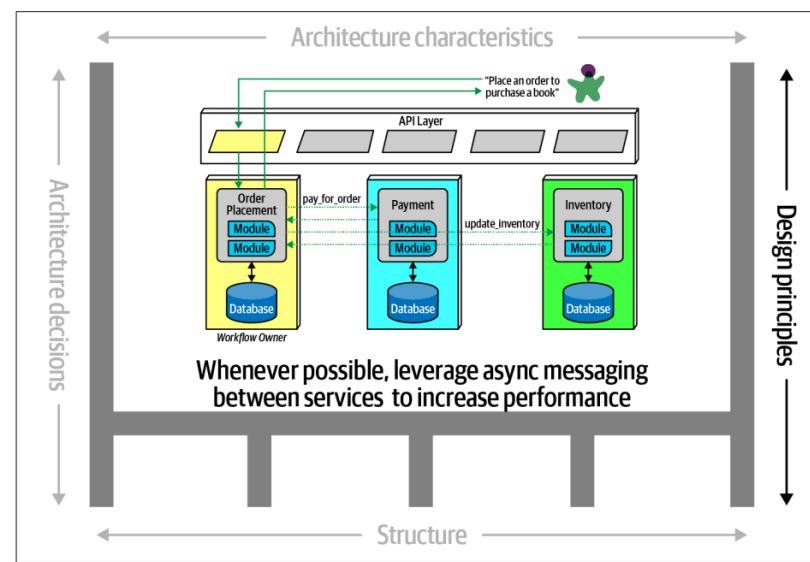
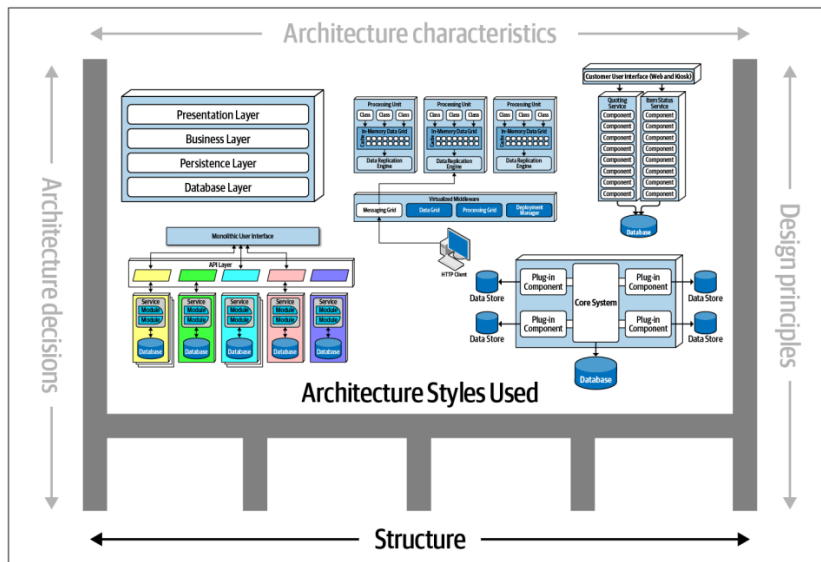
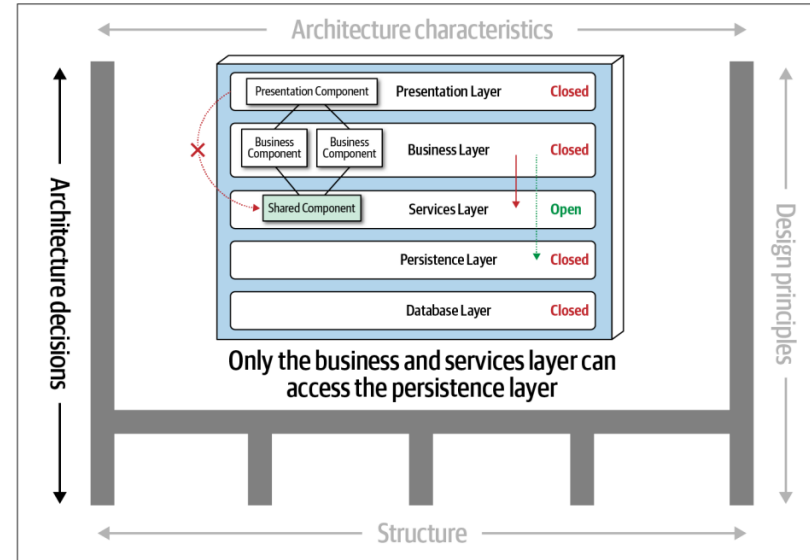
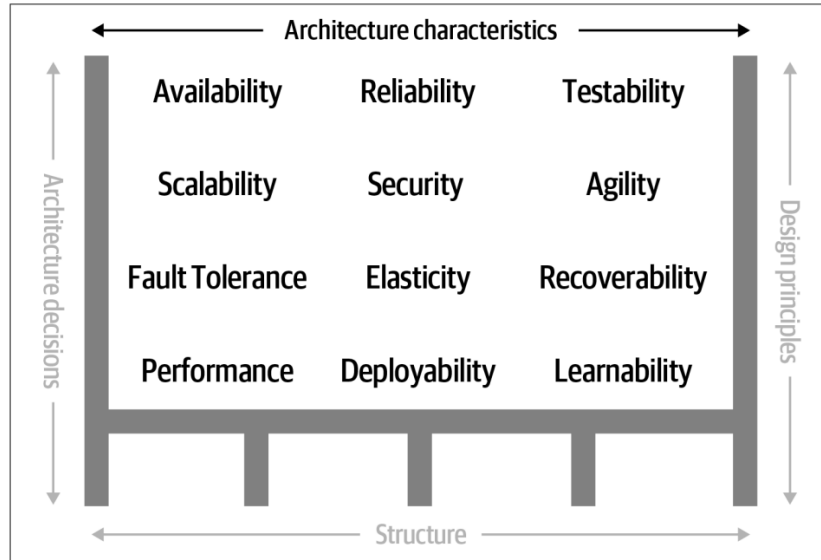
05 Splitting of monolithic application to microservices

01

COMMON SOFTWARE ARCHITECTURE PATTERNS

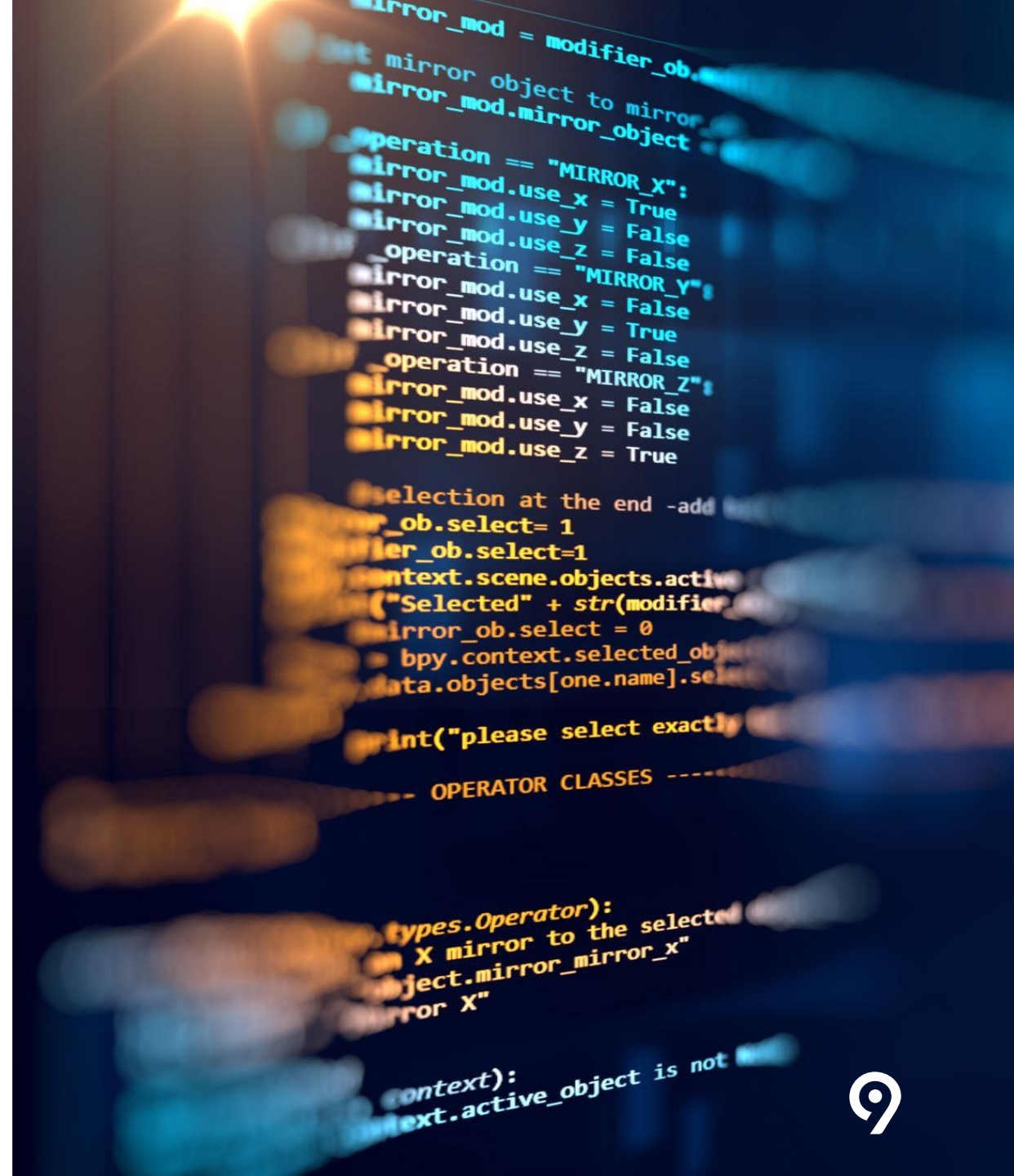


DEFINING SOFTWARE ARCHITECTURE

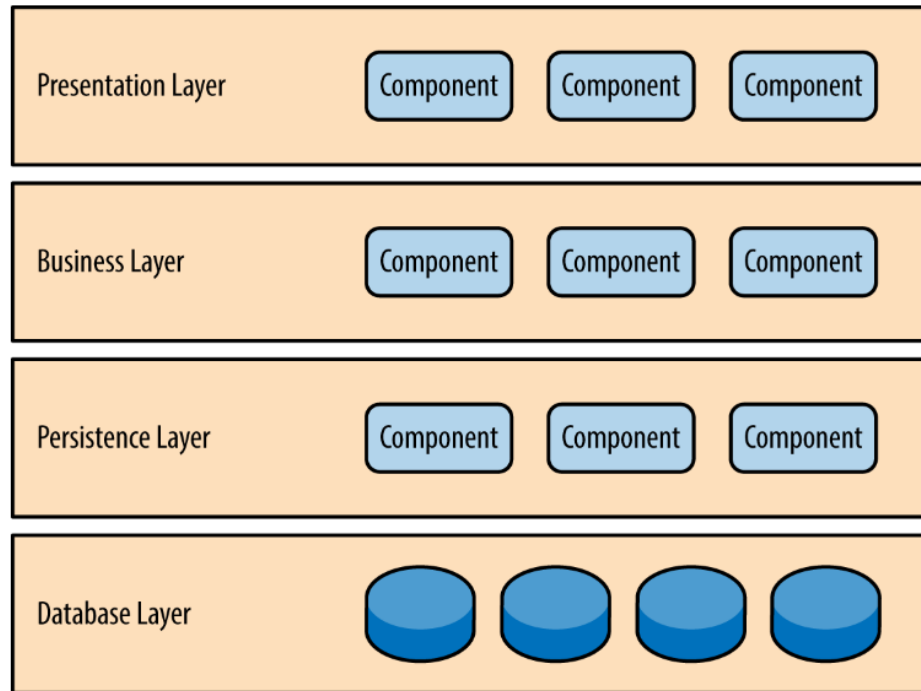


COMMON SOFTWARE ARCHITECTURE PATTERNS

- Layered (n-tier) architecture
- Event-driven architecture
- Microkernel architecture
- Space-based architecture
- Microservices architecture
- Pipeline architecture
- Service-based architecture



LAYERED (N-TIER) ARCHITECTURE



- Organized into horizontal layers
- Each layer performing a specific role within the application

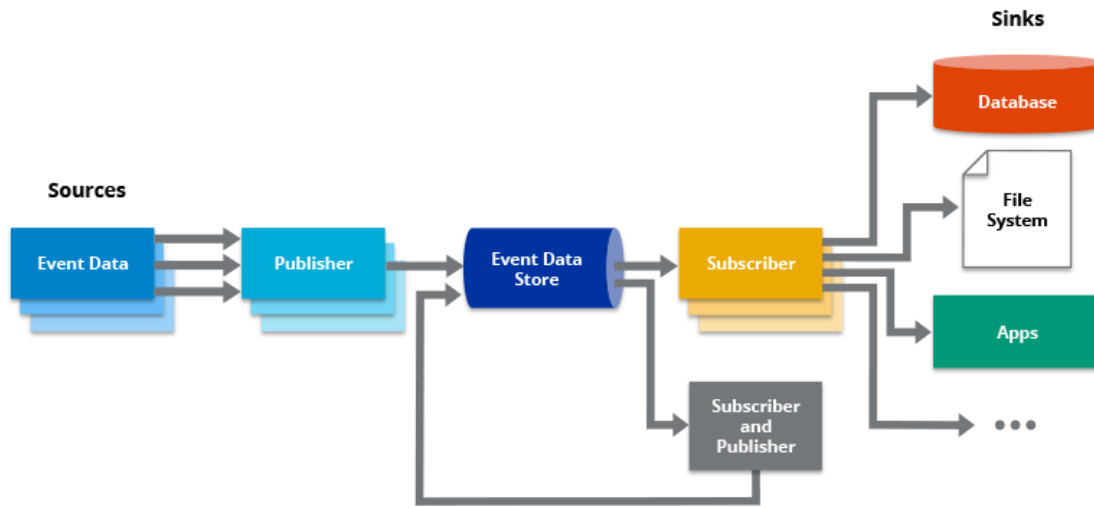
PROS:

- Maintainable (debugging, logging)
- Testable
- Easy to assign separate "roles"
- Easy to start with (intuitive) and "cheap"

CONS:

- Layer isolation, which is an important goal for the architecture, can also make it **hard to understand the architecture without understanding every module**.
- Code can end up slow thanks to what some developers call the “**sinkhole anti-pattern**.” Much of the code can be devoted to passing data through layers without using any logic.
- Coders can skip past layers to create **tight coupling** and produce a logical mess full of complex interdependencies.
- Source code can turn into a “**big ball of mud**” - the modules don’t have clear roles or relationships.
- **Monolithic deployment** is often unavoidable - small changes can require a complete redeployment of the application.

EVENT-DRIVEN ARCHITECTURE



- Centered around data that describes "**events**" (such as the click of a button, moving the scroll bar, etc)
- Act on events as they occur
- Two general categories:
 - Mediator - orchestrate with a central mediator
 - Broker topology - chain simple events

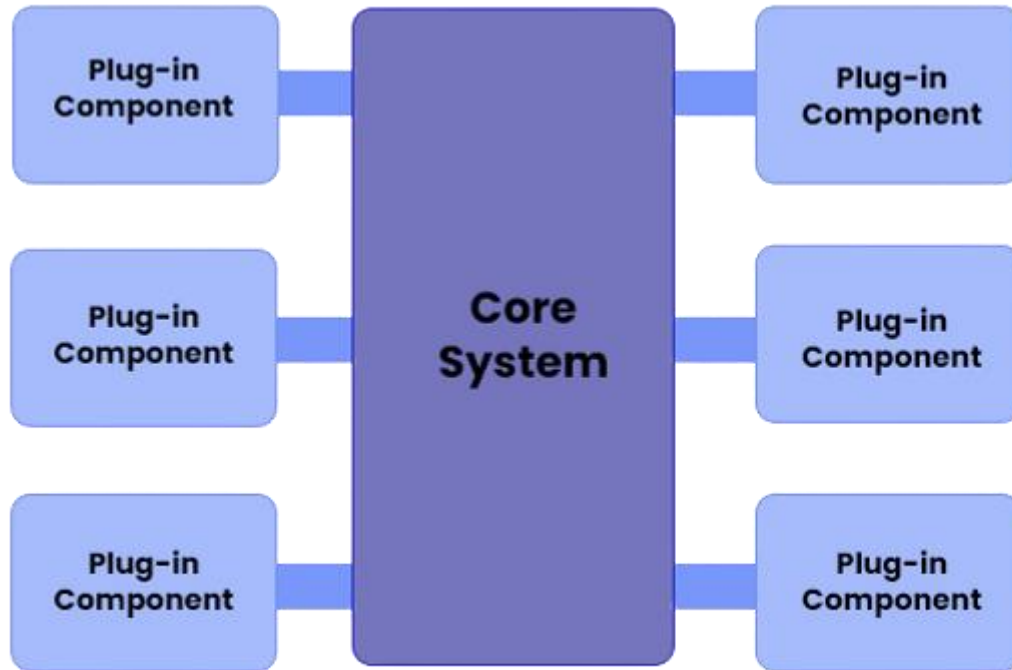
PROS:

- Agility
- Ease of deployment
- Performance
- Scalability

CONS:

- **Testability** - Some sort of specialized testing client or testing tool is required to generate events. Testing is also complicated by the asynchronous nature of this pattern.
- **Ease of development** - Development can be somewhat complicated due to the **asynchronous nature of the pattern** as well as contract creation and the need for more **advanced error handling** conditions within the code for unresponsive event processors and failed brokers.

MICROKERNEL ARCHITECTURE



- **Core system** - general business logic with no custom code for exceptional cases or complex conditional processes.
- **Plug-ins** - set of independent components that assist the core by providing specialized processing additional features via custom code.


PROS:

- Agility
- Ease of deployment
- Testability
- Performance

CONS:


- **Scalability** – Most microkernel architecture implementations are product-based and are generally smaller in size, **they are implemented as single units** and hence **not highly scalable**
- **Ease of development** – Requires thoughtful design and contract governance, making it rather complex to implement. **Contract versioning, internal plug-in registries, plug-in granularity, and the wide choices available for plug-in connectivity all contribute to the complexity.**

LAYERED (N-TIER) ARCHITECTURE

- New applications that need to be built quickly 
 - Enterprise or business applications that need to mirror traditional IT departments and processes
 - Applications requiring strict maintainability and testability standards
 - Teams with inexperienced developers who don't understand other architectures yet
-
- Not well suited if scalability is required
 - Not applicable if parallel processing is required




EVENT-DRIVEN ARCHITECTURE

- Applications that have asynchronous data flow systems 
 - Complex applications requiring seamless data flow or those applications that would eventually grow
 - User interfaces
-
- Not well suited for transactional support - event processor components are highly decoupled and distributed

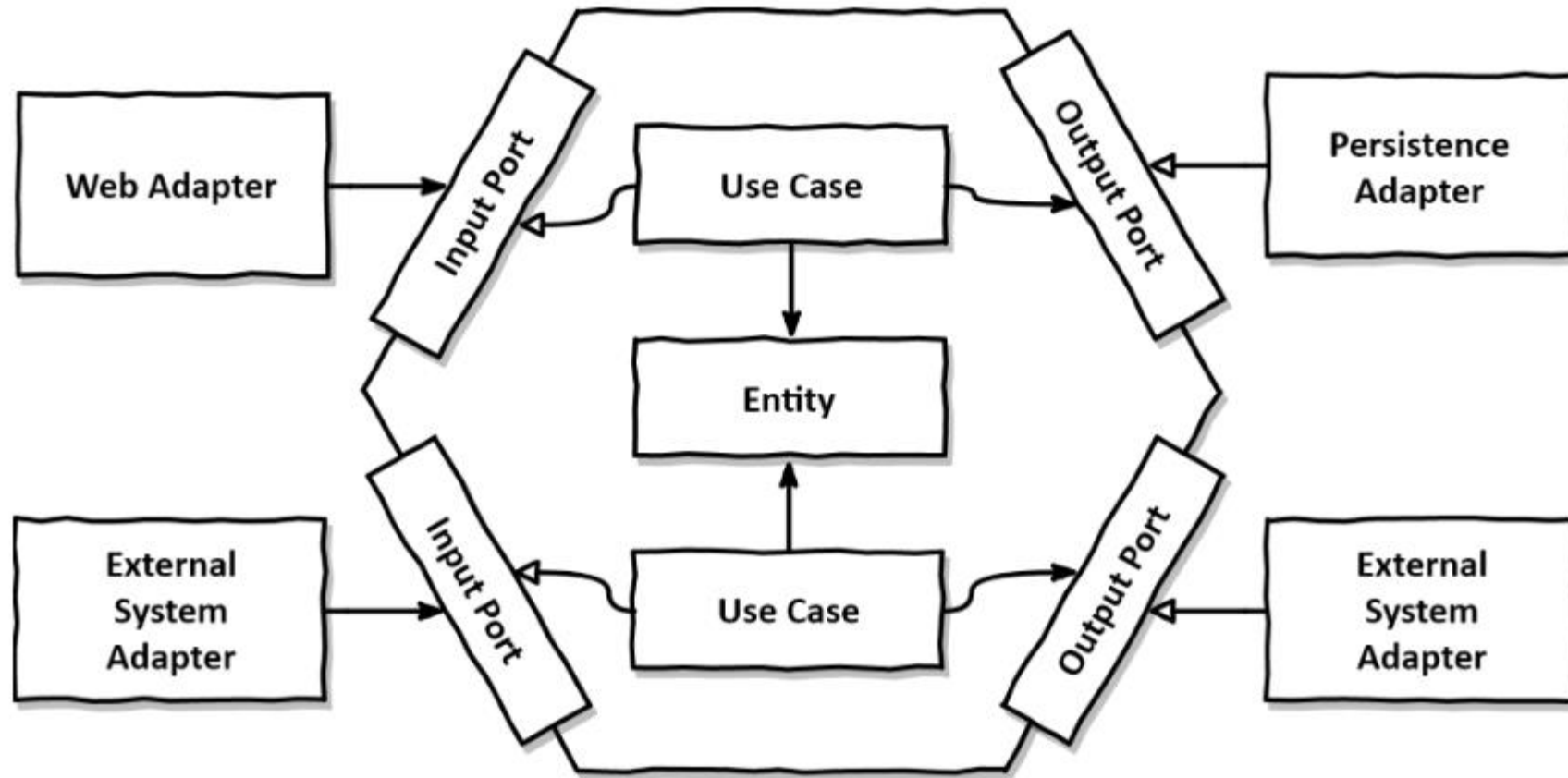


MICROKERNEL ARCHITECTURE

- Applications that are concerned with the separation between low-level functionalities and higher-level functionalities 
 - Applications with a clear division between basic routines and higher-order rules
 - Best used for enterprise applications
 - Best suited for development teams that are spread out
-
- Not well suited for larger applications if scalability is a requirement

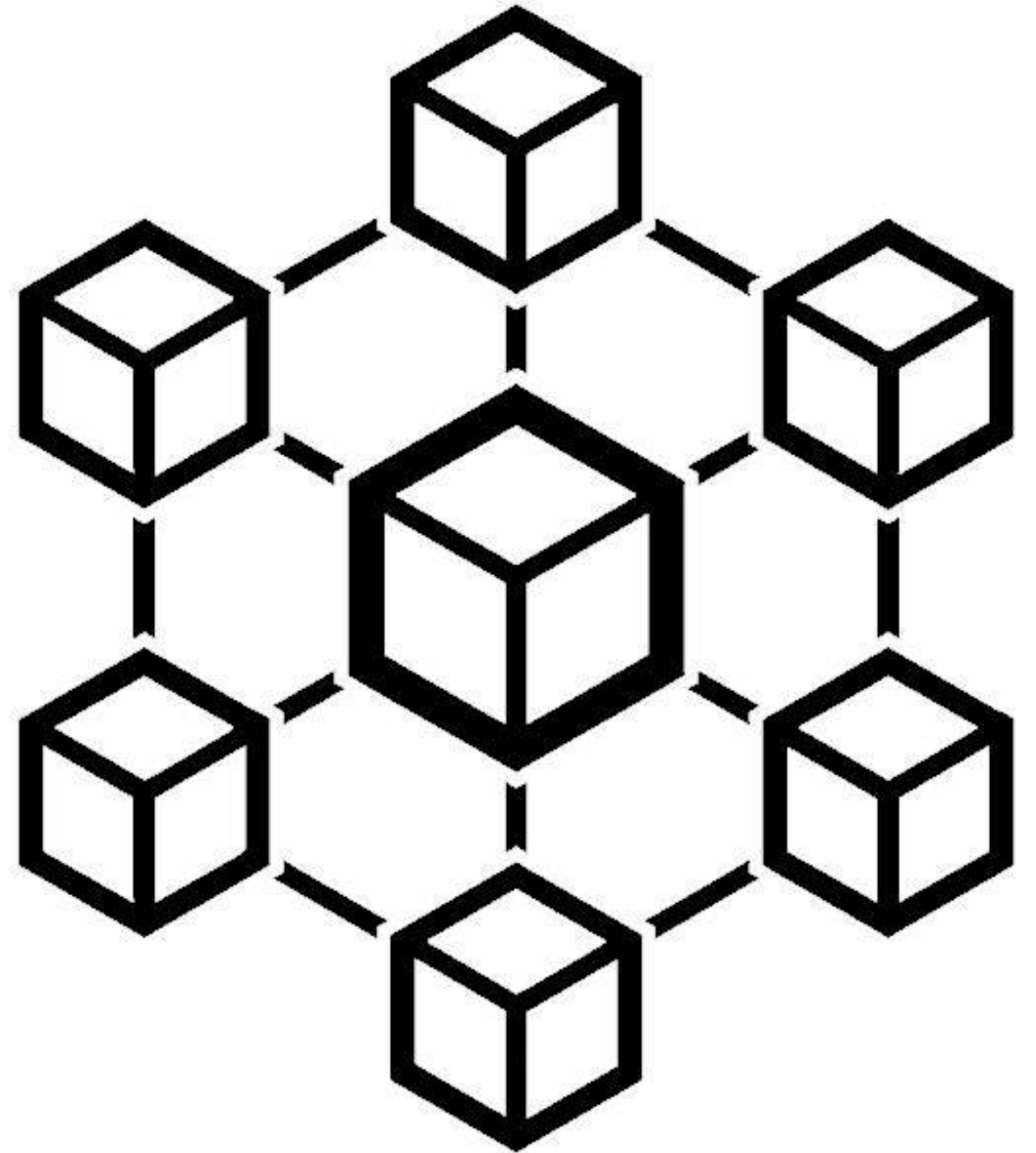


HEXAGONAL ARCHITECTURE - PORTS & ADAPTERS PATTERN

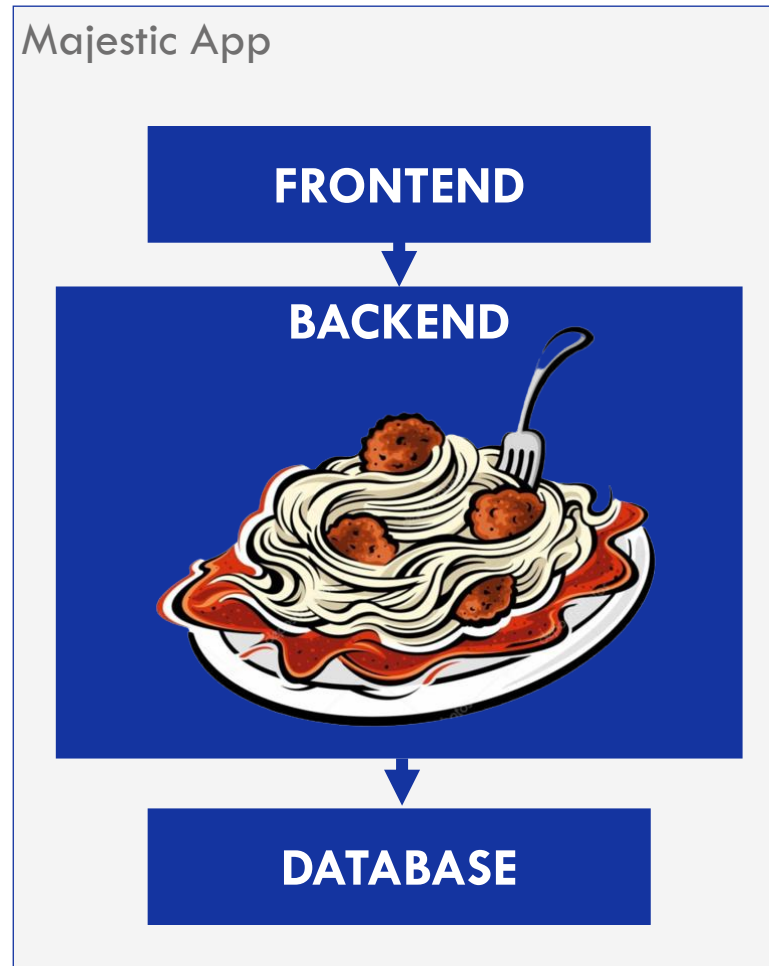
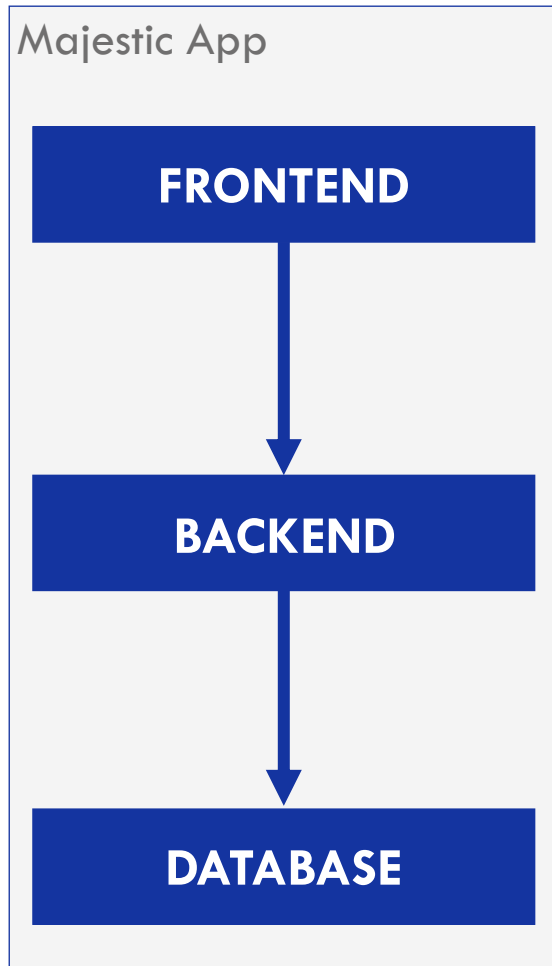


02

MICROSERVICES ARCHITECTURE



MOTIVATION



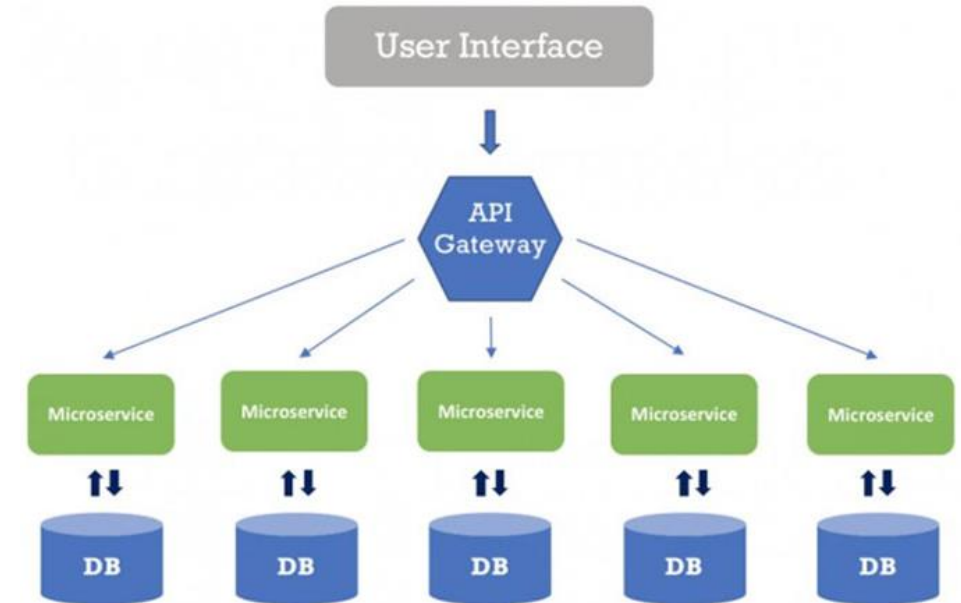
- **Codebases grow** as we write code to add new features.
- **Tightly coupled** code
- Business logic all over the place
- Fixing bugs and adding new features becomes more difficult
- Big Bang releases
- Harder for understanding

MICROSERVICES – DEFINITION

Small and Focused on Doing One Thing Well

There is no single definition for microservices. Over time, the consensus view has evolved. Some characteristics are:

- Microservices are small, **autonomous services** that work together
- Services in a microservice architecture are often processes that **communicate over a network** to fulfill a goal using technology-agnostic protocols (e.g., HTTP)
- Services are organized around **business capabilities (vertical split)**
- Services are implemented using **different programming languages**, databases, hardware, and software environments
- **Independently deployable** and released with automated processes



BENEFITS

Modularity

This makes the application **easier to understand, develop, test**, and become more resilient to architecture erosion.

This benefit is often argued in comparison to the complexity of monolithic architectures.

Scalability

Microservices are implemented and deployed independently of each other (they run within independent processes) - **they can be monitored and scaled independently**.

Integration

Microservices are considered a viable means for modernizing existing monolithic software applications.

The process for Software modernization of legacy applications is done using an **incremental approach**.

Heterogeneity

We can decide to use **different technologies inside each one**.

This allows us to pick the right tool for each job, rather than having to select a standardized approach that often ends up being the lowest common denominator.

Resilience

If **one component** of a system **fails**, but that **failure doesn't cascade**, you can isolate the problem and the rest of the system can carry on working.

Deployment

We can make a change to a single service and **deploy it independently** of the rest of the system – faster deployment.

If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve.

Distributed development

Teams working on smaller codebases tend to be more productive.

Microservices allow us to better align our architecture to our organization by minimizing the number of people working on any codebase.

Replaceability

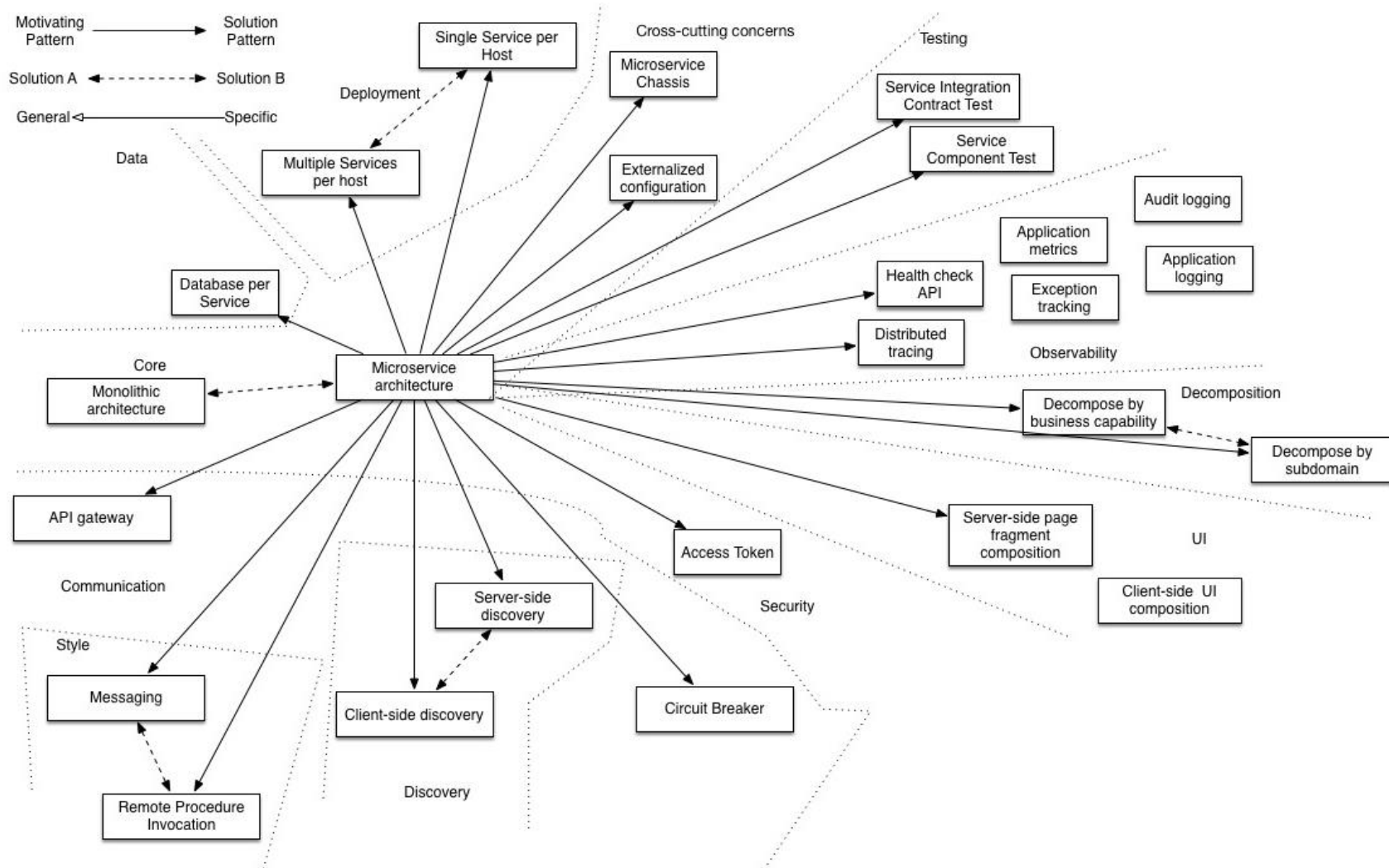
With our individual services being small in size, the cost to **replace them with a better implementation**, or even delete them altogether, is much easier to manage.

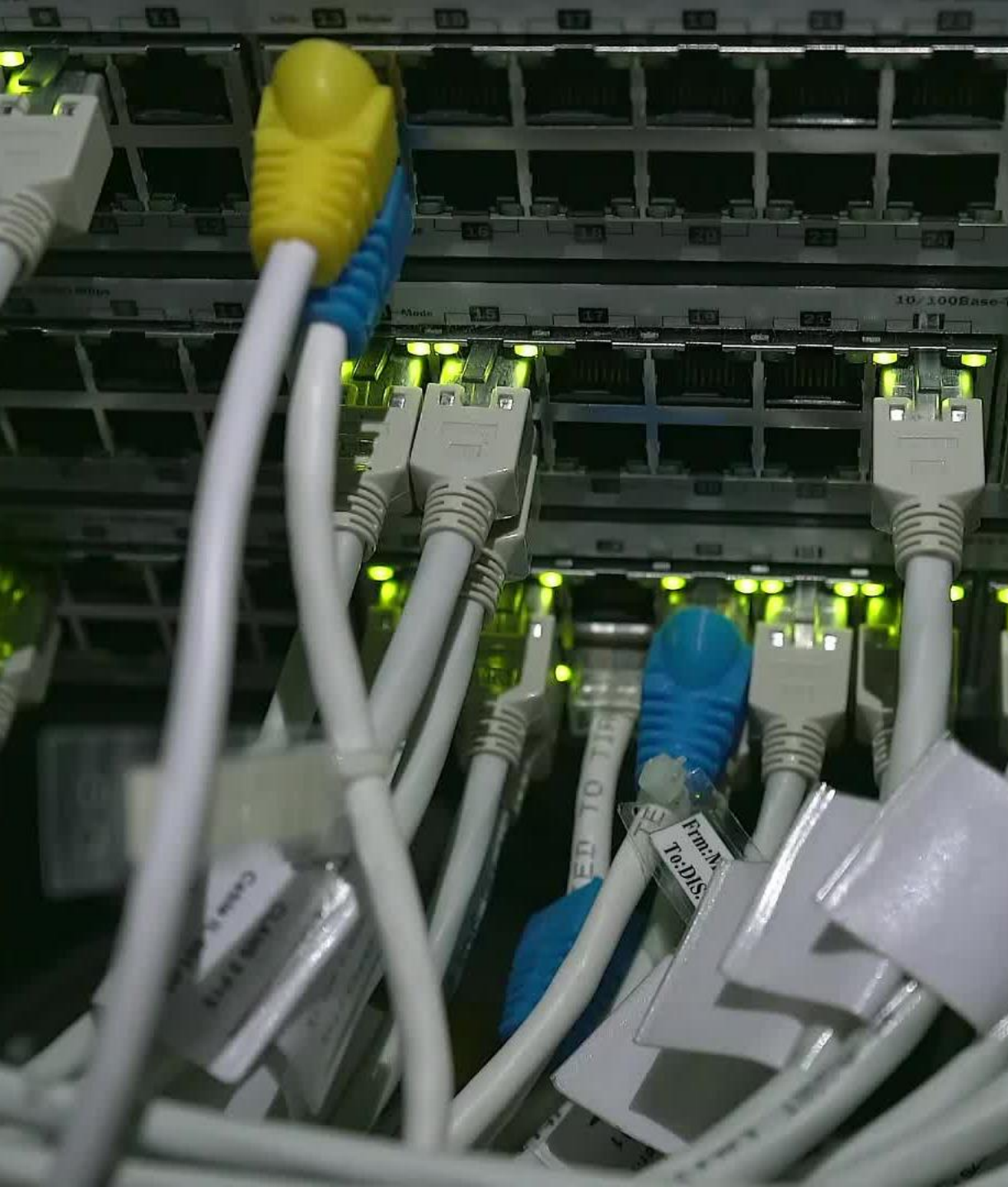


CRITICISM AND CONCERNS

- Communication between services can be complex
- More services equals more resources
- Testing
- Debugging
- Difficult to manage many services

MICROSERVICE PATTERNS





SERVICE COMMUNICATION

Synchronous communication

- http - rest

Asynchronous communication

- messaging

REST (FEIGN CLIENT, REST TEMPLATE)

```

@SpringBootApplication
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}

```

JAVA

```

@FeignClient("stores")
public interface StoreClient {

    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    Page<Store> getStores(Pageable pageable);

    @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")
    Store update(@PathVariable("storeId") Long storeId, Store store);

    @RequestMapping(method = RequestMethod.DELETE, value = "/stores/{storeId:\\d+}")
    void delete(@PathVariable Long storeId);

}

```

JAVA



MESSAGING (RABBITMQ)

Add dependency to project pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
    <version>3.1.0</version>
  </dependency>
</dependencies>
```

Create queue:

```
@Bean
public Queue myQueue() {
    return new Queue("myQueue", false);
}
```

Send message:

```
rabbitTemplate.convertAndSend("myQueue", "Hello world!");
```

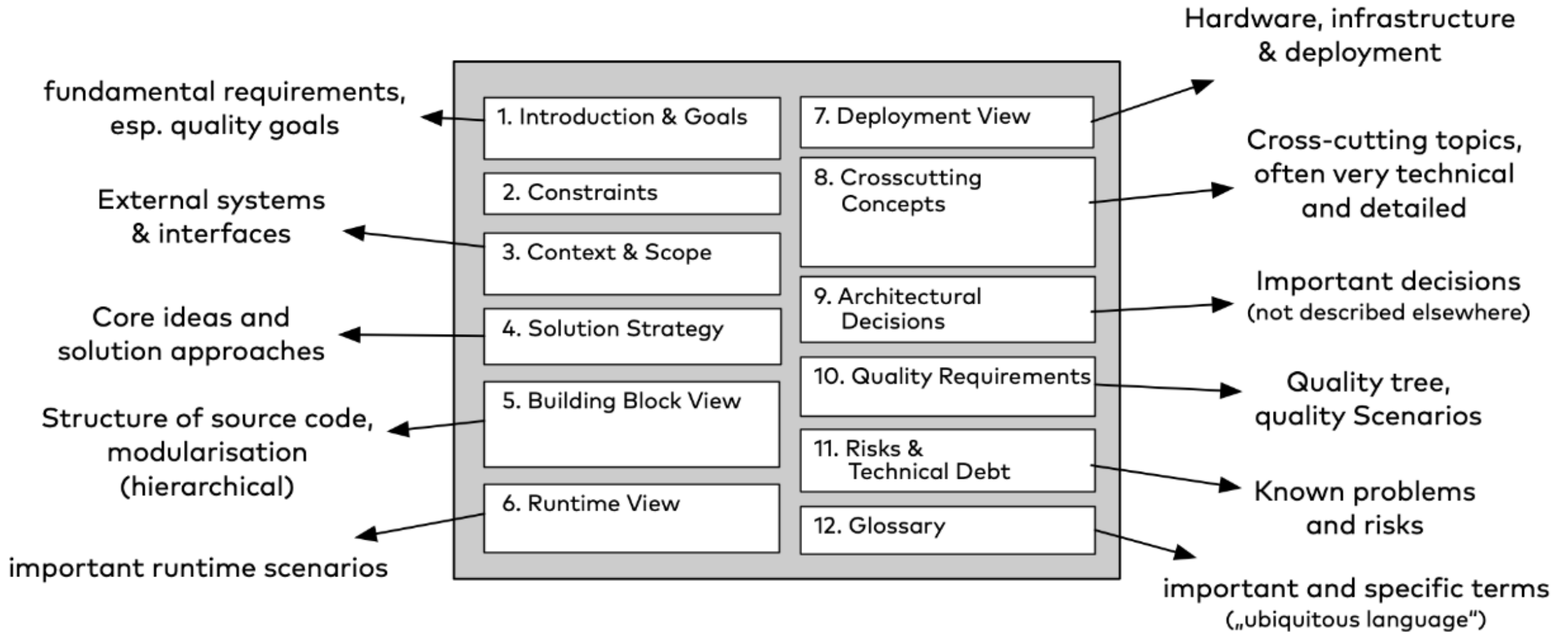
Add application properties

- spring.rabbitmq.port is used to specify the port to which the client should connect, and defaults to 5672.
- spring.rabbitmq.username is used to specify the (optional) username.
- spring.rabbitmq.password is used to specify the (optional) password.
- spring.rabbitmq.host is used to specify the host, and defaults to localhost.
- spring.rabbitmq.virtualHost is used to specify the (optional) virtual host to which the client should connect.

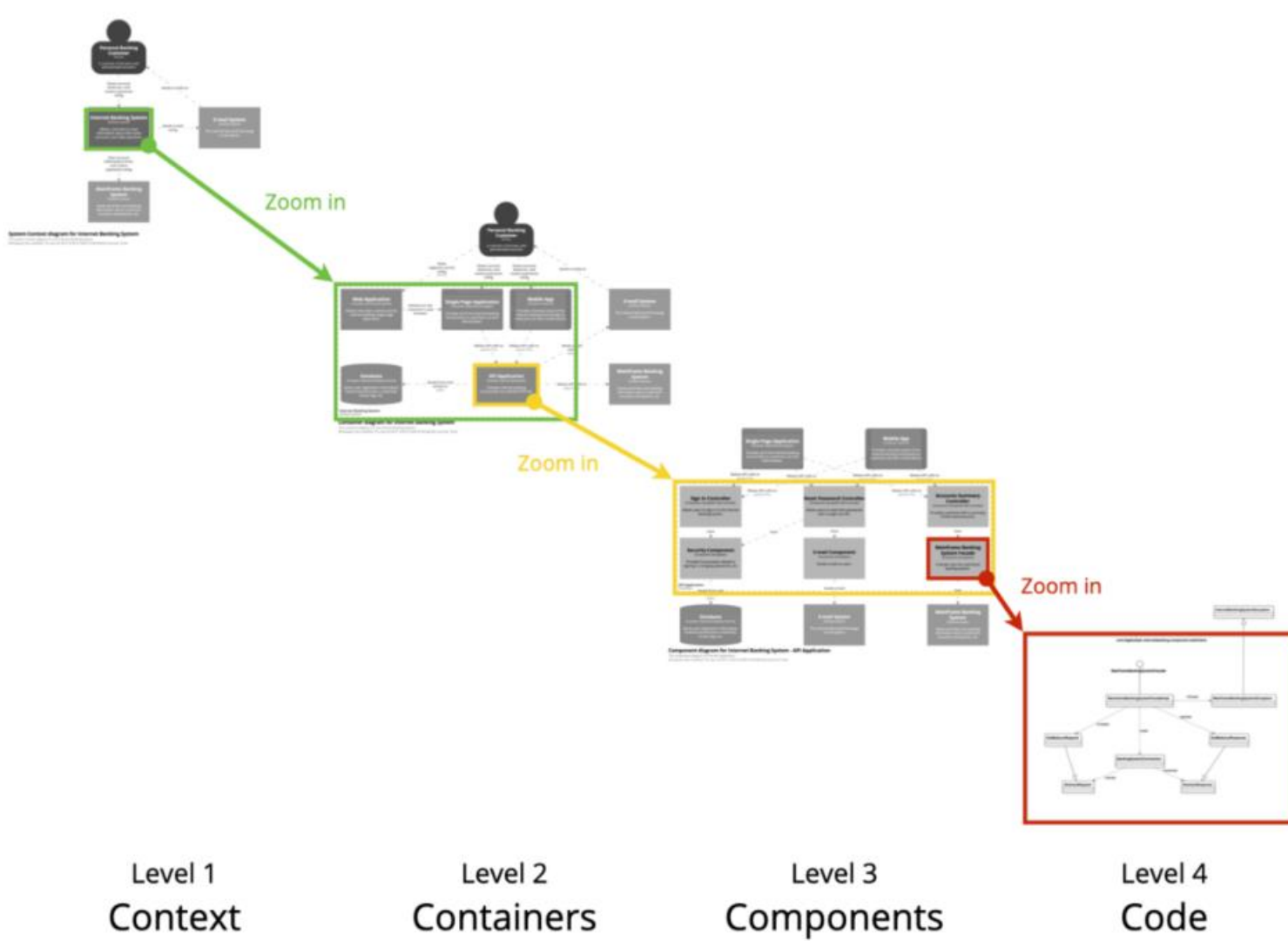
Consume message:

```
@RabbitListener(queues = "myQueue")
public void listen(String in) {
    System.out.println("Message read from myQueue : " + in);
}
```

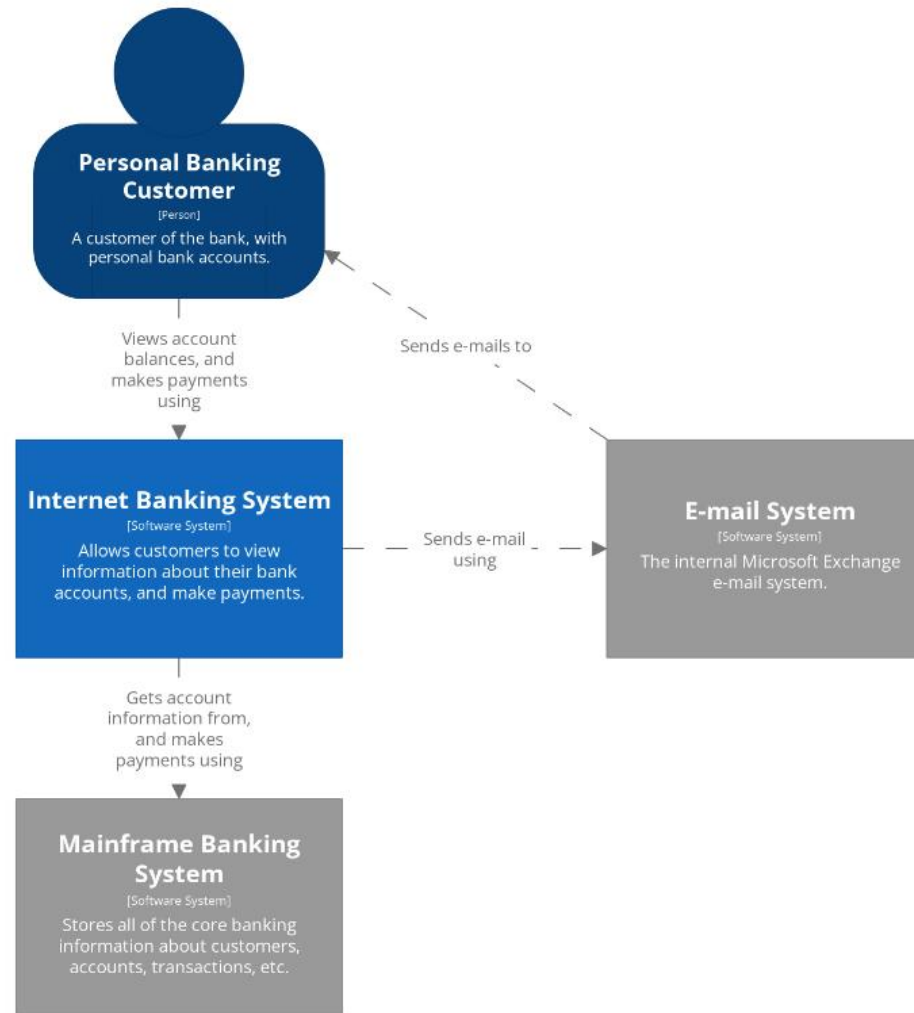
ARCHITECTURE DOCUMENTATION



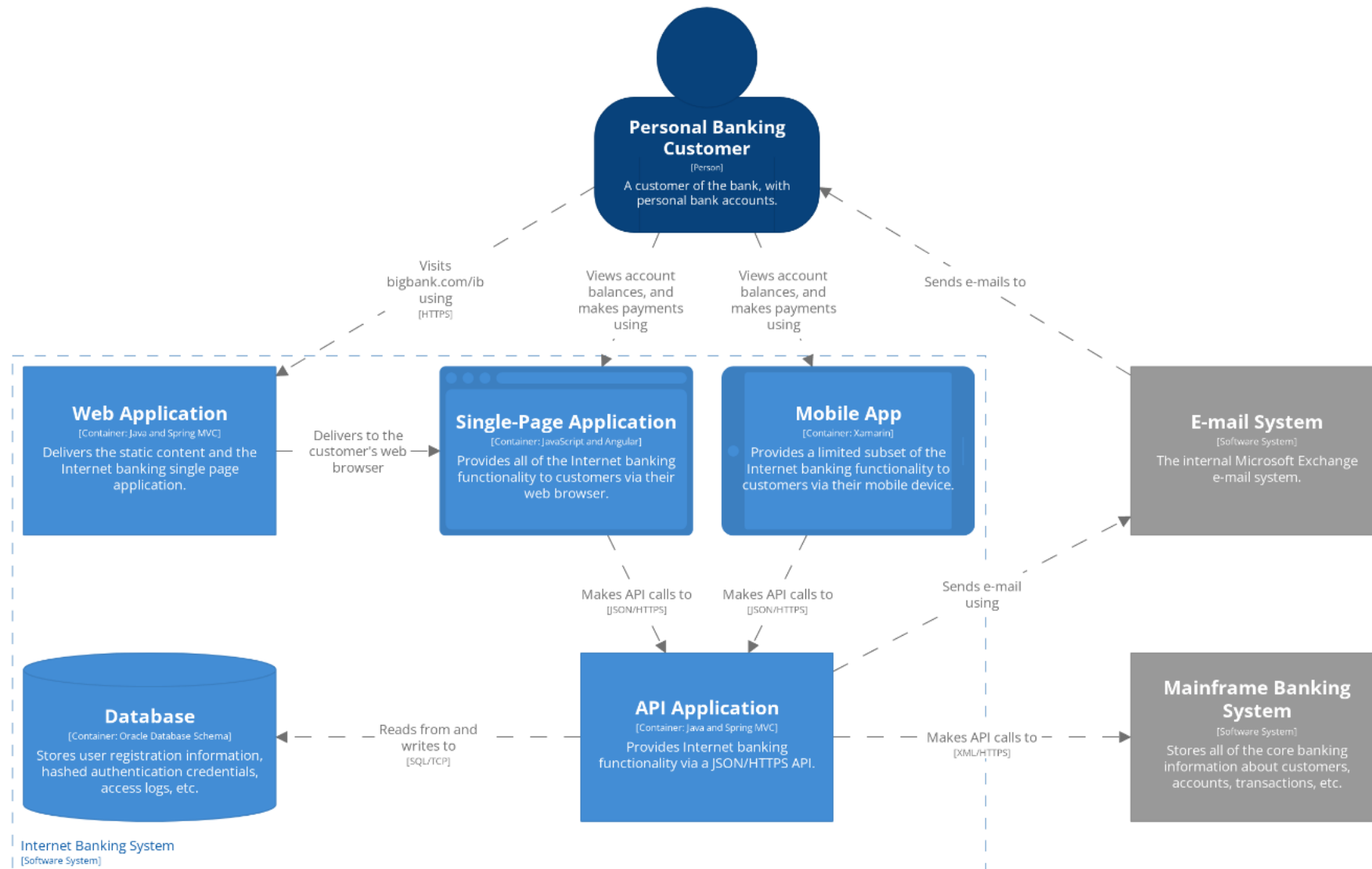
C4 MODEL



C4 MODEL – C1 CONTEXT



C4 MODEL – C2 CONTAINERS





03

LIQUIBASE



Liquibase

LIQUIBASE

- Open source solution for managing revisions of your database schema scripts
- Works across different types of SQL databases and support different file formats for defining DB structure
- The most attractive feature of Liquibase is its ability to roll changes back and forward from a specific point
 - You don't need to remember which scripts was executed at which time
 - Liquibase is doing for you this out of the box
- Tracking changes using its own tables in schema to ensure consistency and to avoid corruption due to incorrectly altered changelogs
- While it's running updates, it puts a "lock" on database so there is no possibility that by accident two changelogs are run concurrently.

CHANGELOGS

- Changelogs are written using domain-specific languages
- It can be written in JSON, YAML, XML or one of a few other supported formats.
- Consist of a series of changesets
- Changeset represent a single change to database
 - Creating table
 - Adding a column
 - Adding an index
 - Some SQL statements



CHANGELOGS

```
<databaseChangeLog
  xmlns=http://www.liquibase.org/xml/ns/dbchangelog"....>
  <changeSet author="Code9User" id="createTable">
    <createTable tableName="app_user">
      <column autoIncrement="true" name="id" type="BIGINT">
        <constraints primaryKey="true"/>
      </column>
      <column name="email" type="VARCHAR(255)">
        <constraints nullable="false"/>
      </column>
      ....
    </createTable>
  </changeSet>

</databaseChangeLog>
```

CHANGELOGS

```
<databaseChangeLog
  xmlns=http://www.liquibase.org/xml/ns/dbchangelog"....>
  <changeSet author="Code9User" id="createTable">
    ...
  </changeSet>
  <changeSet author="Code9User" id="populateData">
    ...
  </changeSet>
  <changeSet author="Code9User" id="addIndexToUserTable">
    ...
  </changeSet>
  ....
</databaseChangeLog>
```

CHANGELOGS

```
<databaseChangeLog
  xmlns=http://www.liquibase.org/xml/ns/dbchangelog"....>
  <include file="com/levi9/code9/db/changelog/createTable.xml">
  <include file="com/levi9/code9/db/changelog/populateData.xml">
  <include file="com/levi9/code9/db/changelog/ addIndexToUserTable.xml">
  ...
</databaseChangeLog>
```

PRECONDITIONS

```
<databaseChangeLog
  xmlns=http://www.liquibase.org/xml/ns/dbchangelog"....>
<changeSet id="addColumn" author="liquibase">
  <preConditions onError="MARK_RAN" onFail="MARK_RAN">
    <tableExists tableName="person" />
    <not>
      <columnExists columnName="score" tableName="person" />
    </not>
  </preConditions>
  ...
</databaseChangeLog>
```

HOW LIQUIBASE WORKS

- First time when liquibase is run, two tables are created in schema
- DATABASECHANGELOG
- DATABASECHANGELOGLOCK

Id	author	filename	dateexecuted	orderexecuted	exectype	md5sum	description
createTable	Code9User	createTable.xml	2021-05-24 09:17:05.270418	1	EXECUTED	8:aa1....	Creation of tables

Id	locked	lockgranted	lockedby
1	false	null	null

STEPS FOR RUNNING LIQUIBASE

- Add liquibase as dependency (<https://www.baeldung.com/liquibase-refactor-schema-of-java-app>)
- Create changeset (inside of changelogs or not, it is up to developer)
- In configuration of application set path to the changelog file
- Perform corresponding changes on application code
 - E.g. create model which will do mapping in Java code
- Test application code together with the database changes
- When everything is working correctly, commit changes and start with deploying both, scripts and code
 - Advice:
 - Be sure that liquibase changes will be applied before codebase
 - Don't change already applied liquibase file

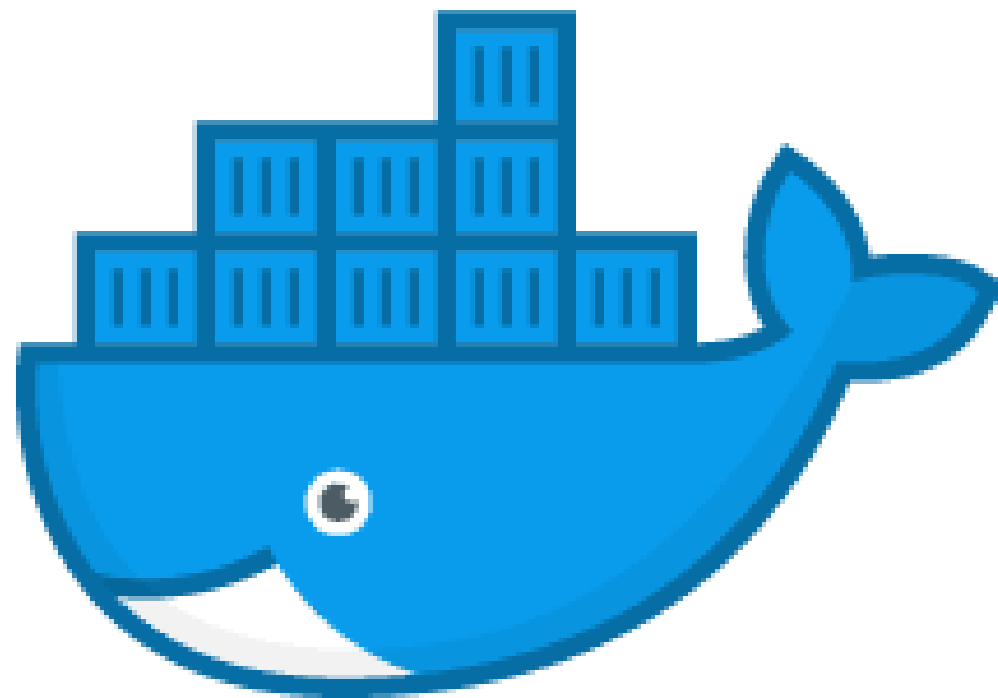
MOTIVATION FOR USING LIQUIBASE

- Easy for tracking changes on database
- Allow having different focus on development and different focus on database
- Same data state can be applied easily on different machine
- You can't delete script which other developer has written
- It can be easily applied rollback to some of previous version
- Schema generation using Hibernate is not recommended for most use cases



04

DOCKER



docker

INTRODUCTION TO DOCKER

Do you know the famous phrase *It works on my machine?*

- With Docker it is not an excuse anymore
- Docker allows you to locally run the same (or almost the same) environment which will be used in production



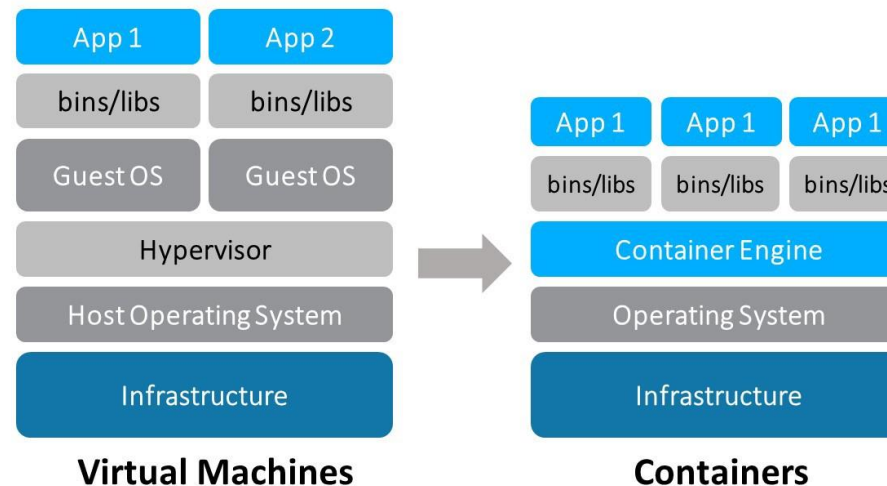
INTRODUCTION TO DOCKER

- Docker is a computer program that performs operating-system-level virtualization, also known as “containerization”
- Tool to make it easier to build, deploy and run applications using containers
- Containers allow us to package all the things that our application needs like such as libraries and other dependencies and ship it all as a single package.
- In this way, our application can be run on any machine and have the same behaviour.



DOCKER ≠ VIRTUAL MACHINE

- Docker is not a virtual machine (VM)
- A Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the kernel's functionality and uses resource isolation for CPU and memory, and separate namespaces to isolate the application's view of the operating system.
- Docker containers are simpler than virtual machines and using it we can avoid the overhead of starting and maintaining VMs



DOCKER TERMINOLOGY

- Images
 - The blueprints of our application which form the basis of containers.
- Containers
 - Created from Docker images and run the actual application.
- Docker Daemon
 - The background service running on the host that manages building, running and distributing Docker containers
- Docker Client
 - The command line tool that allows the user to interact with the daemon, issuing commands and managing containers
- Docker Hub
 - A registry of Docker images
 - "Directory" of all available Docker images

DOCKERFILE

- Text file that contains a list of commands that the Docker client calls while creating an image
- Simple way to automate the image creation process.
- Commands written in a Dockerfile are almost identical to their equivalent Linux commands
- Container image is built with ***docker build*** command
- ***docker build -t image-name .***



DOCKERFILE INSTRUCTIONS

- FROM – specifies the base image (an official image from Docker Hub or a customer image) upon which the new image will be built
- RUN – executes a command inside the Docker image during the image build process: installing packages, configuring software or setting up dependencies within the image
- COPY – copies files or directories from the host machine into the Docker image.
 1. argument: the source path (on the host)
 2. argument: the destination path (in the image)
- CMD/ENTRYPOINT – specifies the default command to be executed when a container is created from the image. Difference: CMD can be overridden at runtime by providing arguments to the 'docker run', which is not case with ENTRYPOINT.

```
# Alpine Linux with OpenJDK JRE
FROM openjdk:8-jre-alpine
# copy WAR into image
COPY spring-boot-app-0.0.1-SNAPSHOT.war /app.war
# run application with this command line
CMD ["/usr/bin/java", "-jar", "-Dspring.profiles.active=default", "/app.war"]
```

STARTING CONTAINER

- When image is created, application can be run via command ***docker run name-of-image***
- One of the stuff which is commonly used is port mapping
- It is allowing access to application
 - -p flag in docker run command is doing port mapping
 - -p 3000:3000
 - Mapping between the host's port 3000 to the container's port 3000
- Command ***docker ps*** or ***docker container ls*** is showing all active docker containers
- Command for seeing all docker images on environment is ***docker image ls*** or ***docker images***

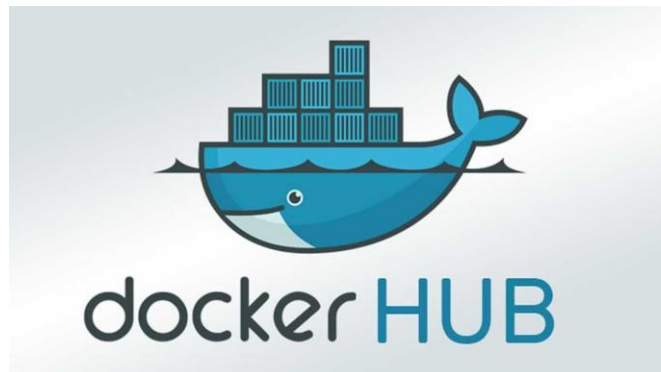
```
root@user: $ docker ps
```

CONTAINER ID	IMAGE	NAMES	CREATED	STATUS	PORTS
780464b51916	my_docker_image	my container	1 hours ago	Up 9 minutes	0.0.0.0:8080->8080, :::8080->8080/tcp



PUSHING DOCKER IMAGES

- Docker Hub repositories allow you share container images with your team, customers, or the Docker community at large.
- Images are pushed to Docker Hub through the `docker push` command. A single Docker Hub repository can hold many Docker images (stored as tags)
- To push an image to Docker Hub, you must first name your local image using your Docker Hub username and the repository name that you created through Docker Hub on the web.
- Steps:
 - `docker build -t <hub-user>/<repo-name>[:<tag>]`
 - `docker push <hub-user>/<repo-name>:<tag>`
- The image is then uploaded and available for use by your teammates and/or the community.



DOCKER COMPOSE

- Tool for defining and running multi-container Docker applications
- Usage of YAML file allowing to configure application's services
- With a single command, create and start all the services from YAML configuration
- ***docker-compose up*** from folder where file is stored or ***docker-compose -f file.yml up*** for some specific file

```
version: "3.2"
services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
    restart: always
    container_name: 'rabbitmq'
    ports:
      - 5672:5672
      - 15672:15672
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
```

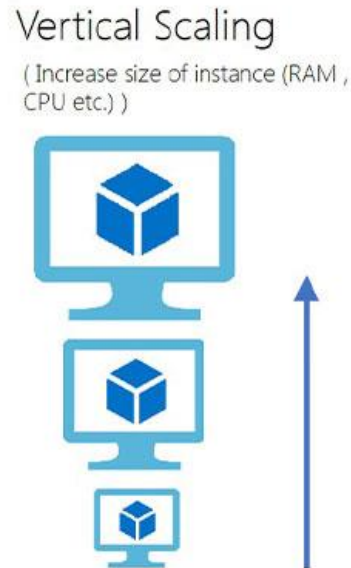
SCALING

- When demand for your application is soaring and you recognize a need to expand the app's accessibility, power and presence you need to do some type of scaling
- The heart of the difference is the approach to adding computing resources to your infrastructure.
- Two main types of scaling are:
 - Vertical scaling
 - With vertical scaling (a.k.a. “scaling up”), you’re adding more power to your existing machine
 - Horizontal scaling
 - In horizontal scaling (a.k.a. “scaling out”), you get the additional resources into your system by adding more machines to your network, sharing the processing and memory workload across multiple devices.



VERTICAL SCALING

- Vertical scaling refers to adding more resources (CPU/RAM/DISK) to your server (database or application server is still remains one) as on demand.
- Commonly used in applications and products of middle-range as well as small and middle-sized companies
- Means upgrade of server hardware

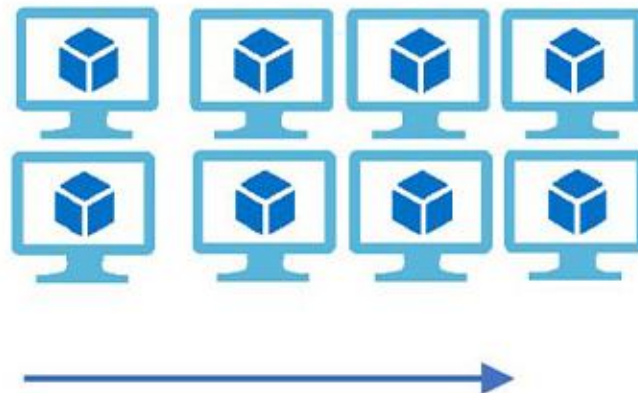


HORIZONTAL SCALING

- Horizontal Scaling is a must use technology – whenever a high availability of (server) services are required
- Involves adding more processing units or physical machines to your server or database.
- It involves growing the number of nodes in the cluster, reducing the responsibilities of each member node by spreading the key space wider and providing additional end-points for client connections
- Horizontal Scaling has been historically much more used for high level of computing and for application and services.

Horizontal Scaling

(Add more instances)



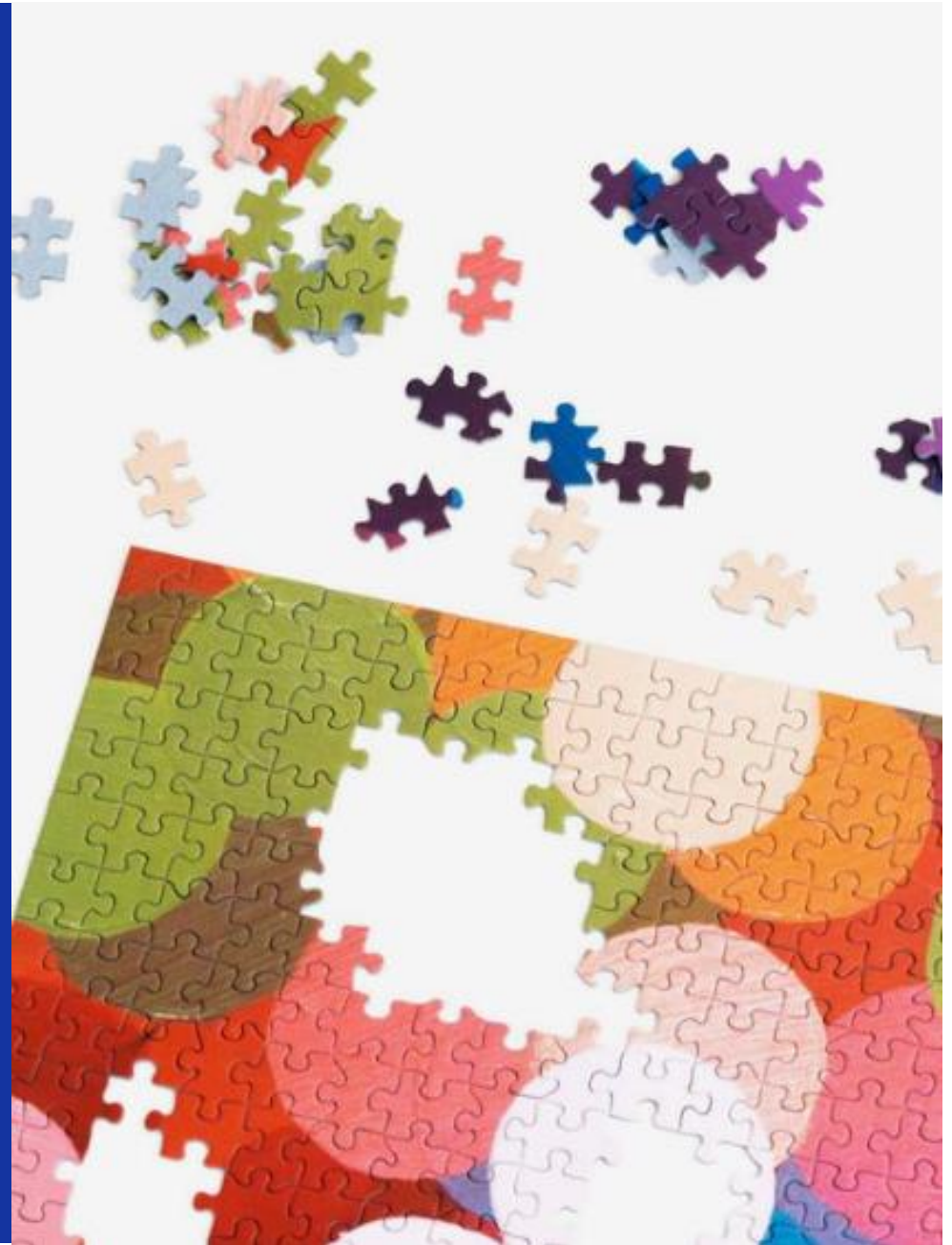
DOCKER AUTOSCALING

- Concept of autoscaling can be done via Docker Swarm or via Kubernetes, or via some Cloud provider
- Since Docker Swarm is not used so much anymore, in place came Kubernetes
- For understanding of Kubernetes you need to know Docker and his concepts (more about Kubernetes <https://kubernetes.io/docs/tutorials/>)
- Kubernetes allow us to make configuration when autoscaling need to be done
 - e.g when target CPU utilization came to 50% create new instance
 - e.g what will be min and what will be the max count of replicas

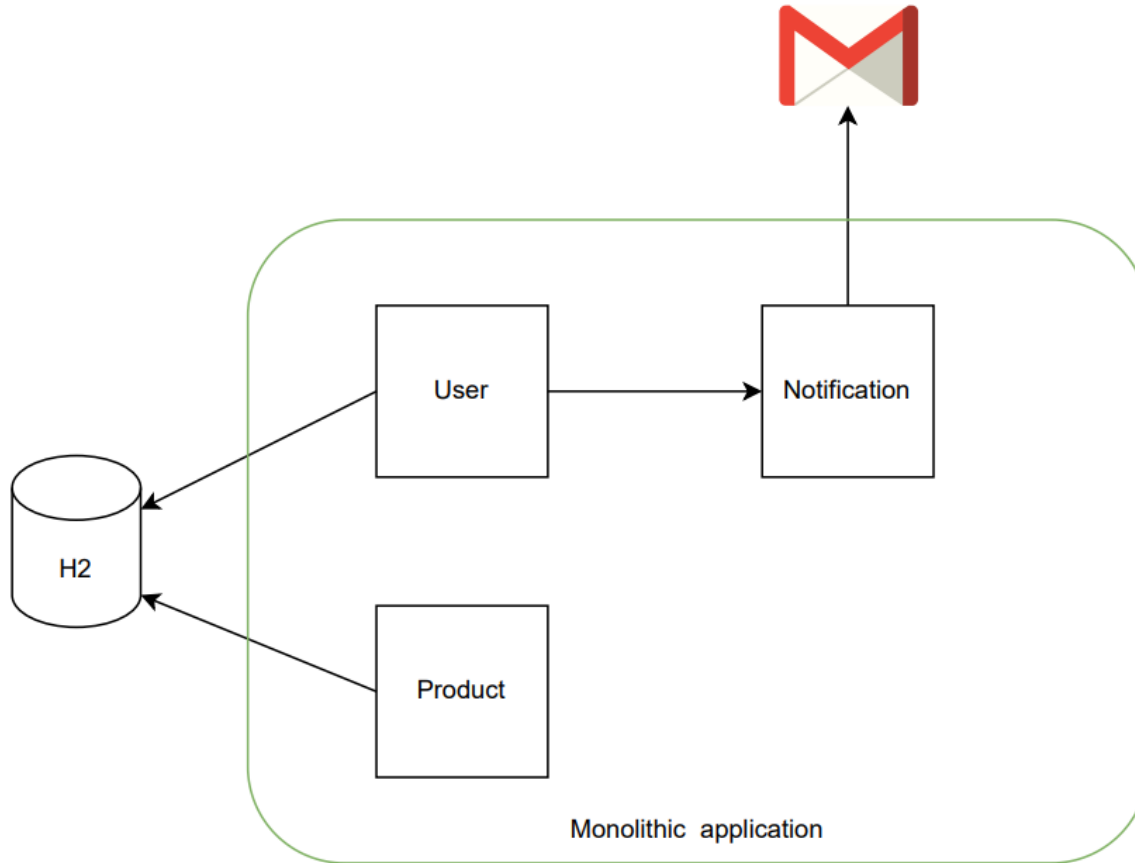


05

SPLIT OF MONOLITHIC APPLICATION TO MICROSERVICES



MONOLITHIC ARCHITECTURE (EXAMPLE)



Components:

- User module
- Notification module
- Product module

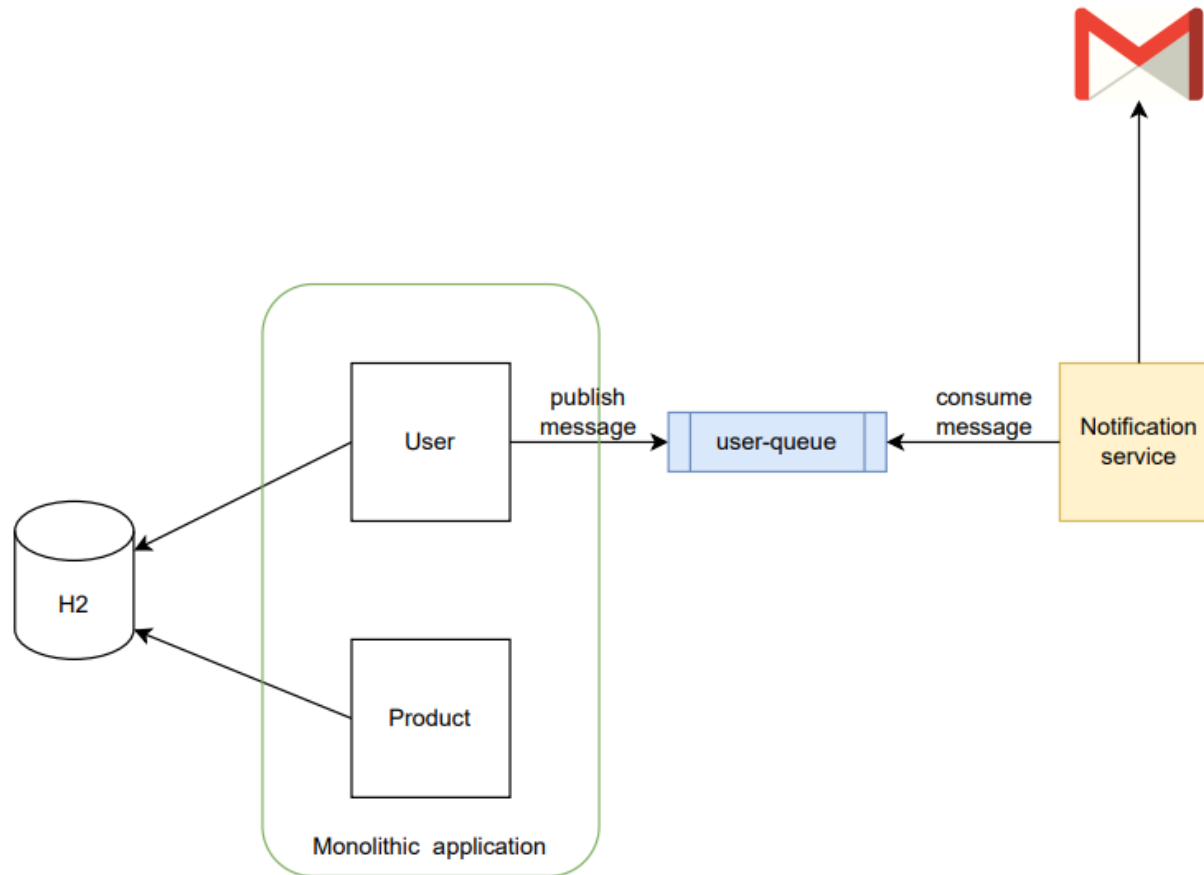
Resources:

- H2 database
- Mail server

GitHub repo:

- <https://github.com/nemanjaromanic/monolithic-app> (main branch)

MICROSERVICE ARCHITECTURE - STEP 1



Tasks:

1. Extract notification service from monolithic application
2. Publish user message to user-queue from monolith
3. Consume user message with notification service

For messaging use RabbitMQ

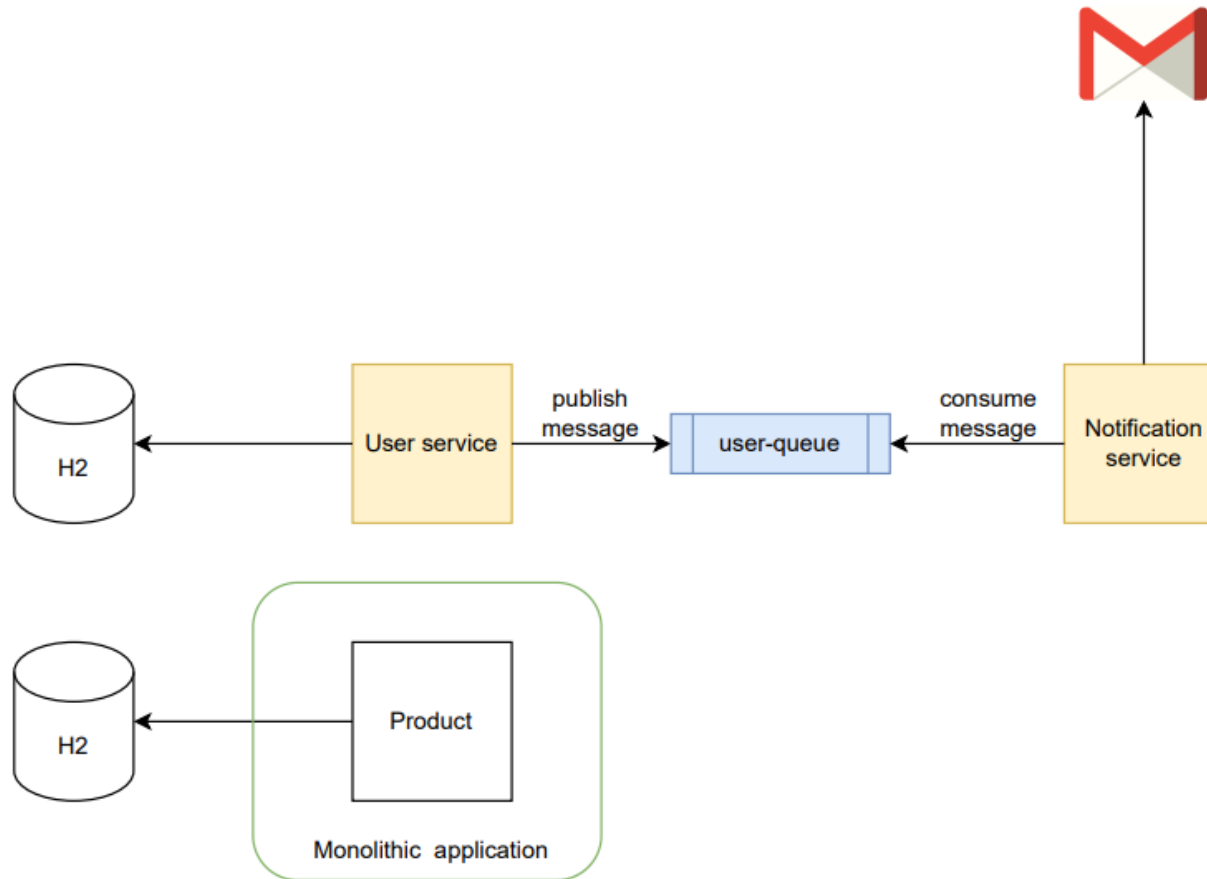
GitHub repo:

- <https://github.com/nemanjaromanic/monolithic-app> (refactor/split-monolith branch)
- <https://github.com/nemanjaromanic/notification-service> (main branch)

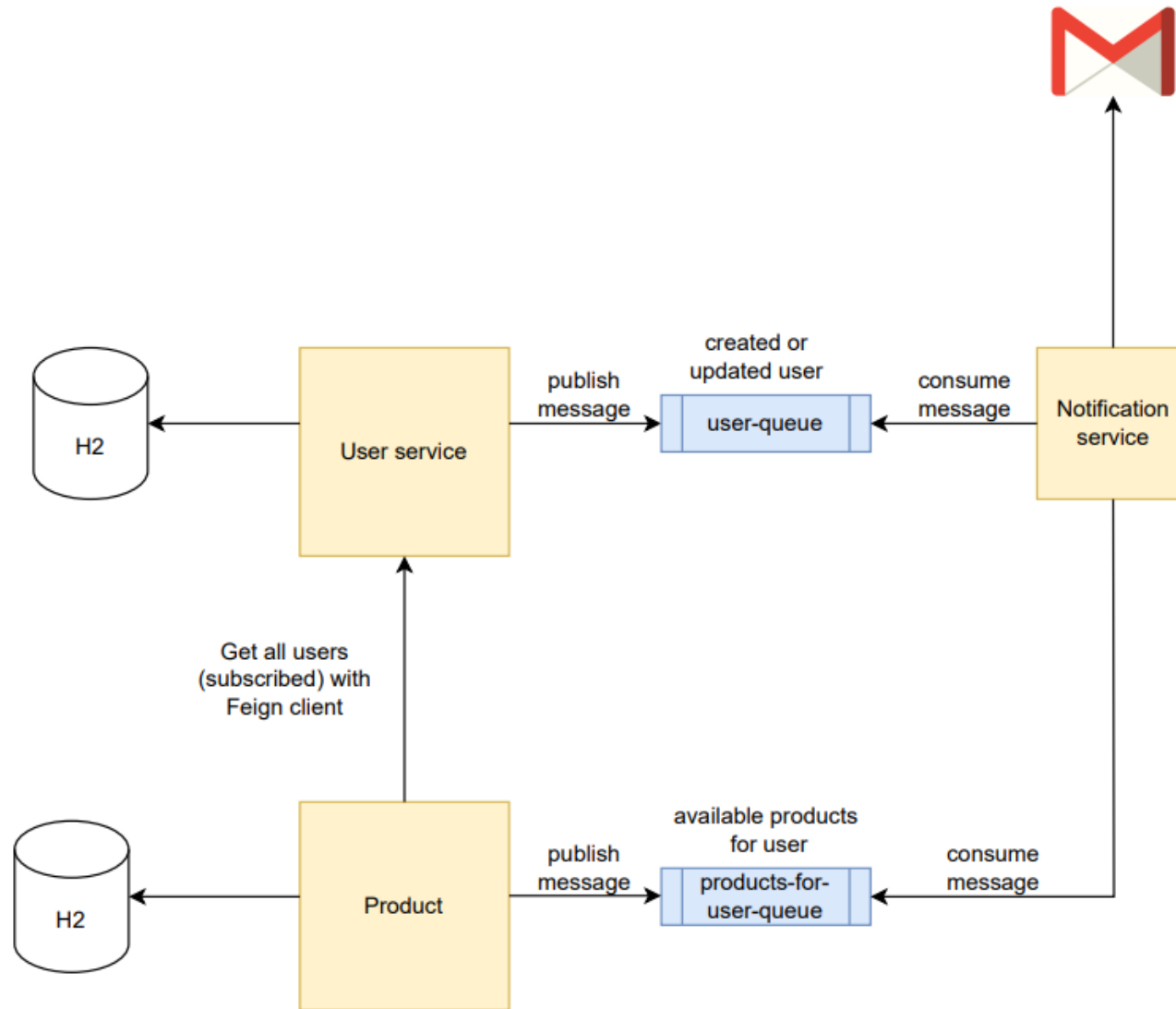
MICROSERVICE ARCHITECTURE - STEP 2

Tasks:

1. Extract user service
2. Extract user related tables to new DB used only by user service
3. Clean old code from monolithic application



MICROSERVICE ARCHITECTURE – FINAL STEP



Tasks:

1. Extract product service
2. Create endpoint on product service to send all available products in email to all subscribed users
 - Product service fetch all subscribed users from user service with Feign client
 - Product service publish messages to new queue
 - Notification service consume messages and send emails to subscribed users
3. Delete monolithic application

Q&A :)