

Microservices

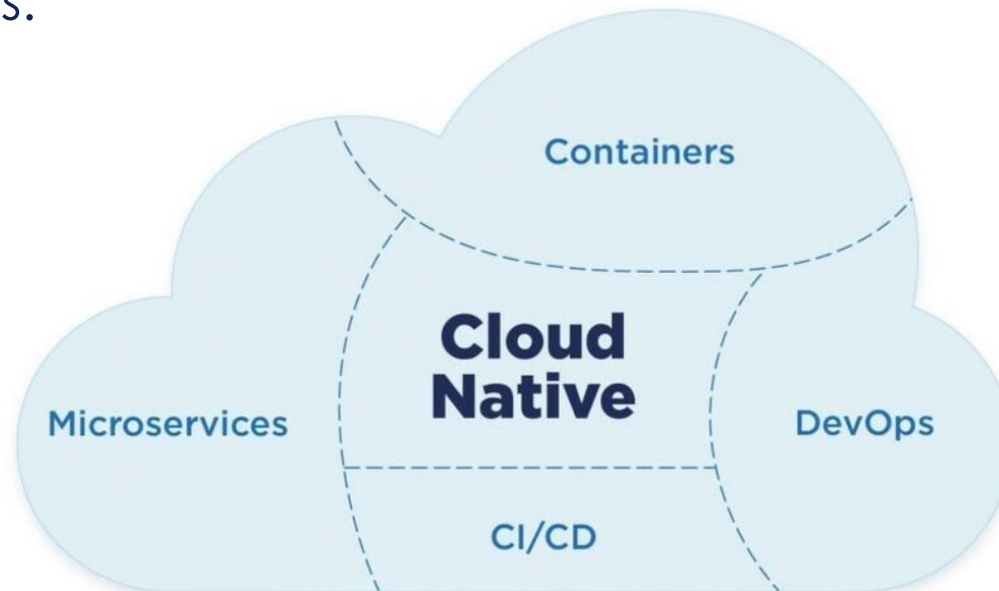
UA.DETI.IES

Overview

- ❖ Application architecture patterns are changing in the era of cloud computing.
- ❖ A convergence of factors has led to the concept of “cloud native” applications:
 - The general availability of **cloud computing platforms**
 - Advancements in **virtualization technologies**
 - The emergence of **agile and DevOps practices** as organizations looked to streamline and shorten their release cycles
- ❖ **Microservices** are cloud native applications composed of smaller, independent, self-contained pieces.

Cloud native applications

- ❖ Cloud computing environments are dynamic, with on-demand allocation and release of resources from a virtualized, shared pool.
 - This elastic environment enables more flexible scaling options, especially compared to the up-front resource allocation typically used by traditional on premises data centers.



Cloud native systems properties

- ❖ Applications or services (microservices) are **loosely coupled** with explicitly described dependencies
- ❖ Applications or processes are **run in software containers** as isolated units
- ❖ Processes are managed by using **central orchestration** processes to improve resource utilization and reduce maintenance costs

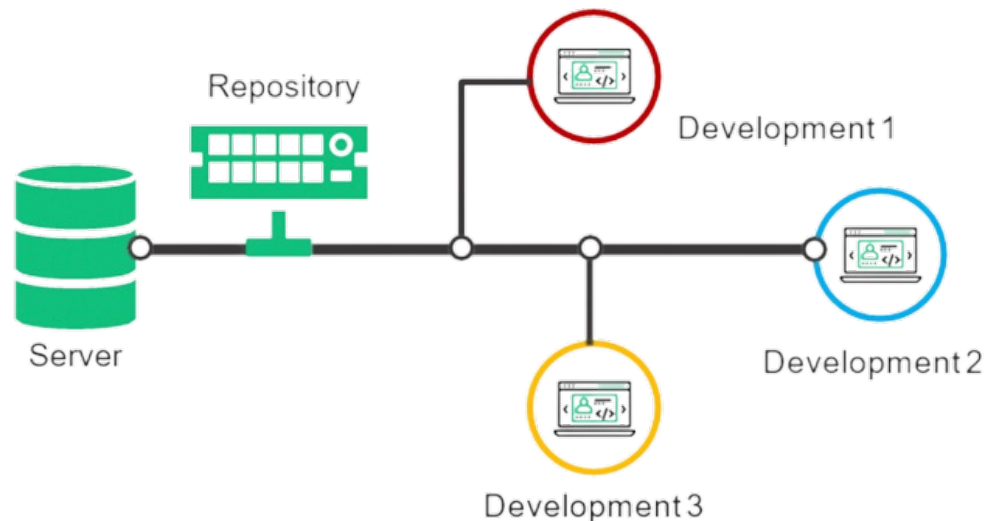
Microservices Twelve Factors

- ❖ The **12-factor application methodology** was drafted by developers at Heroku (PaaS provider)
- ❖ The characteristics mentioned in the 12 factors are not specific to cloud provider, platform, or language.
- ❖ The factors represent a **set of guidelines or best practices** for portable and resilient applications that will thrive in cloud environments (specifically software as a service applications).

<https://12factor.net>

#1 – Codebase

- ❖ There should be a one-to-one association between a versioned **Codebase** (a repository) and an application
 - Using code versioning line Git
- ❖ But the same codebase is used for many deployments

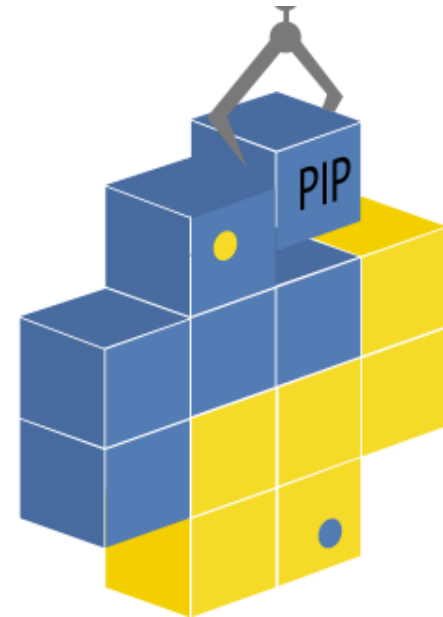


#2 – Dependencies

- ❖ Services should explicitly declare all **Dependencies**
- ❖ Services should not rely on the presence of system-level tools or libraries

MavenTM

Managing Maven
dependencies



#3 – Configurations

- ❖ **Configuration** that varies between deployment environments should be stored in the environment
 - Specifically in environment variables

```
.env.development

NODE_ENV=test
DB_NAME=test_db
DB_USER=username
DB_PASSWORD=password
DB_DOMAIN=localhost
DB_PORT=3000
HOST=localhost
```

```
.env.staging

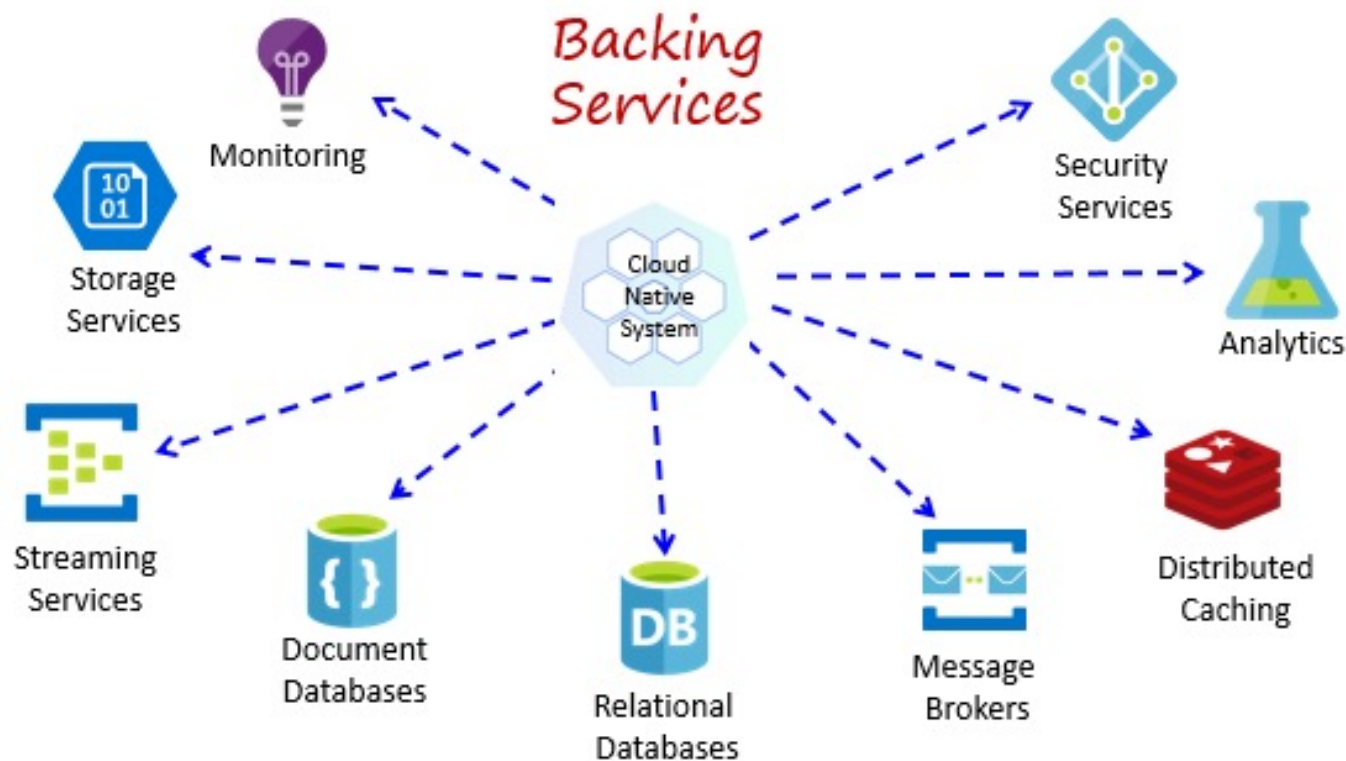
NODE_ENV=staging
DB_NAME=staging_db
DB_USER=staging_username
DB_PASSWORD=staging_password
DB_DOMAIN=staging_url
DB_PORT=5000
HOST=staging.example.com
```

```
.env.production

NODE_ENV=production
DB_NAME=prod_db
DB_USER=prod_username
DB_PASSWORD=prod_password
DB_DOMAIN=prod_url
DB_PORT=3000
HOST=example.com
```

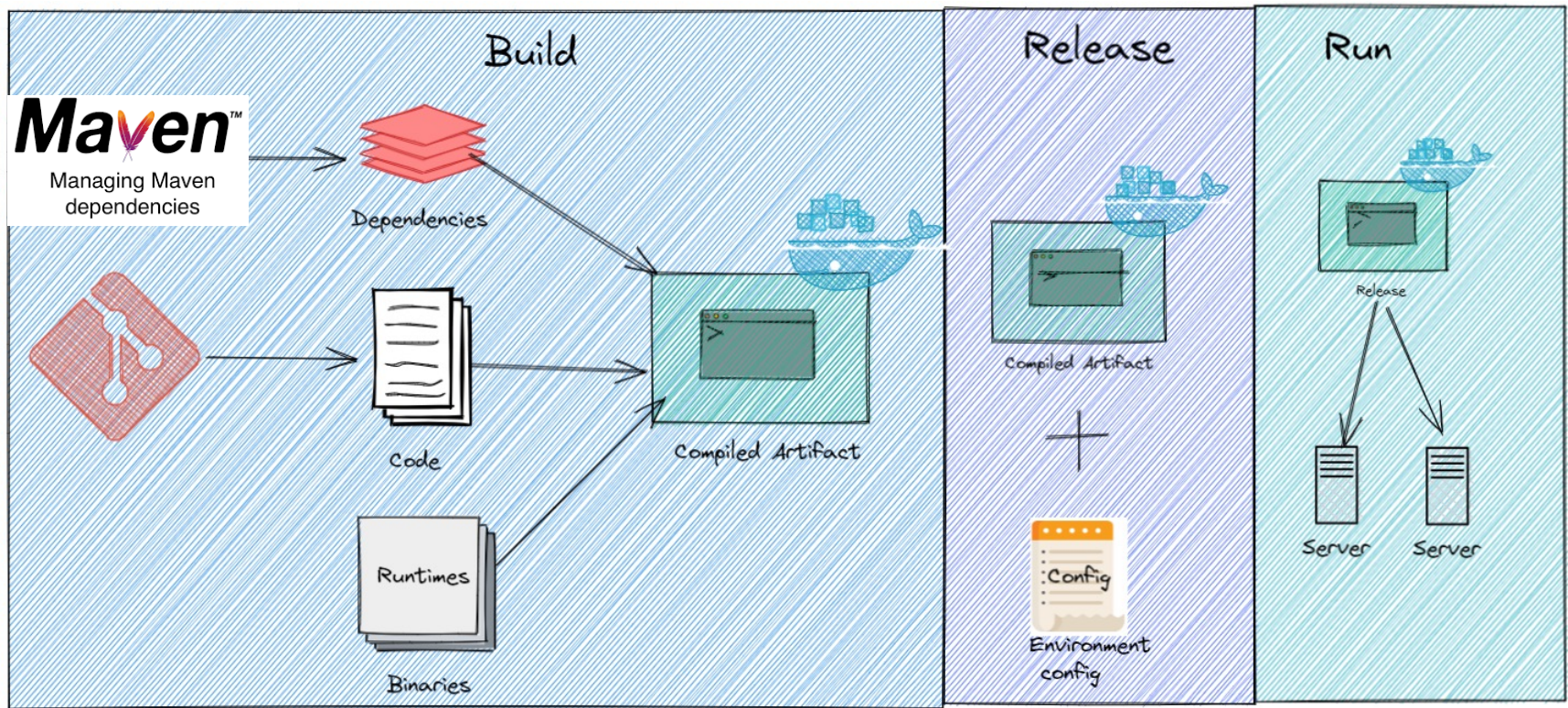

#4 – Backing services

- ❖ All **backing services** are treated as attached resources
 - which are managed (attached and detached) by the **execution environment**



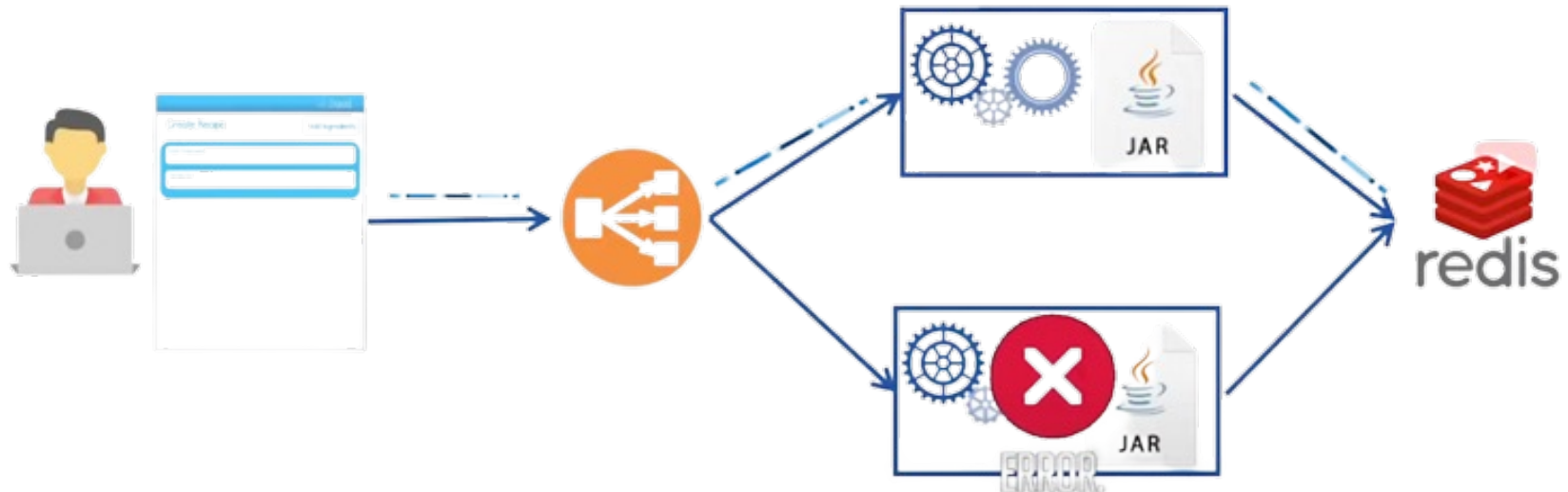
#5 – Build, release, run

- ❖ The delivery pipeline should have strictly separate stages: **build**, **release**, and **run**.



#6 – Processes

- ❖ Applications should be deployed as one or more stateless **processes**.
- ❖ Persisted data should be stored in an appropriate backing service.



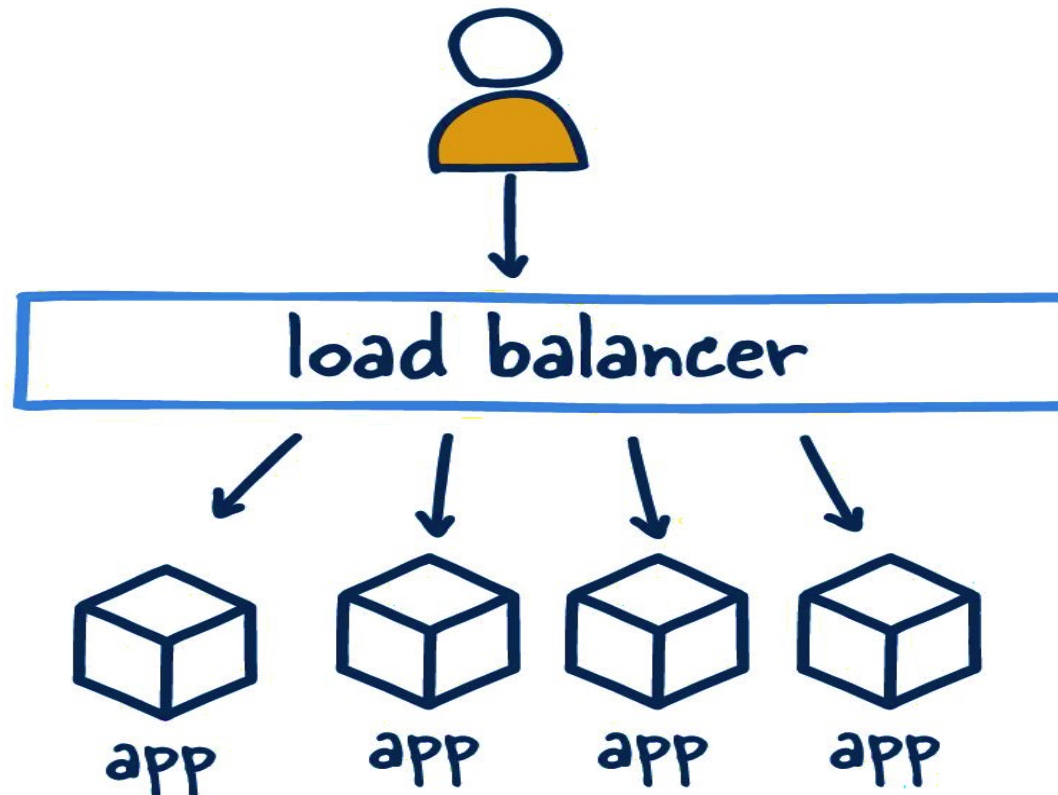
#7 – Port binding

- ❖ Self-contained services should be available through a **port binding**



#8 – Concurrency

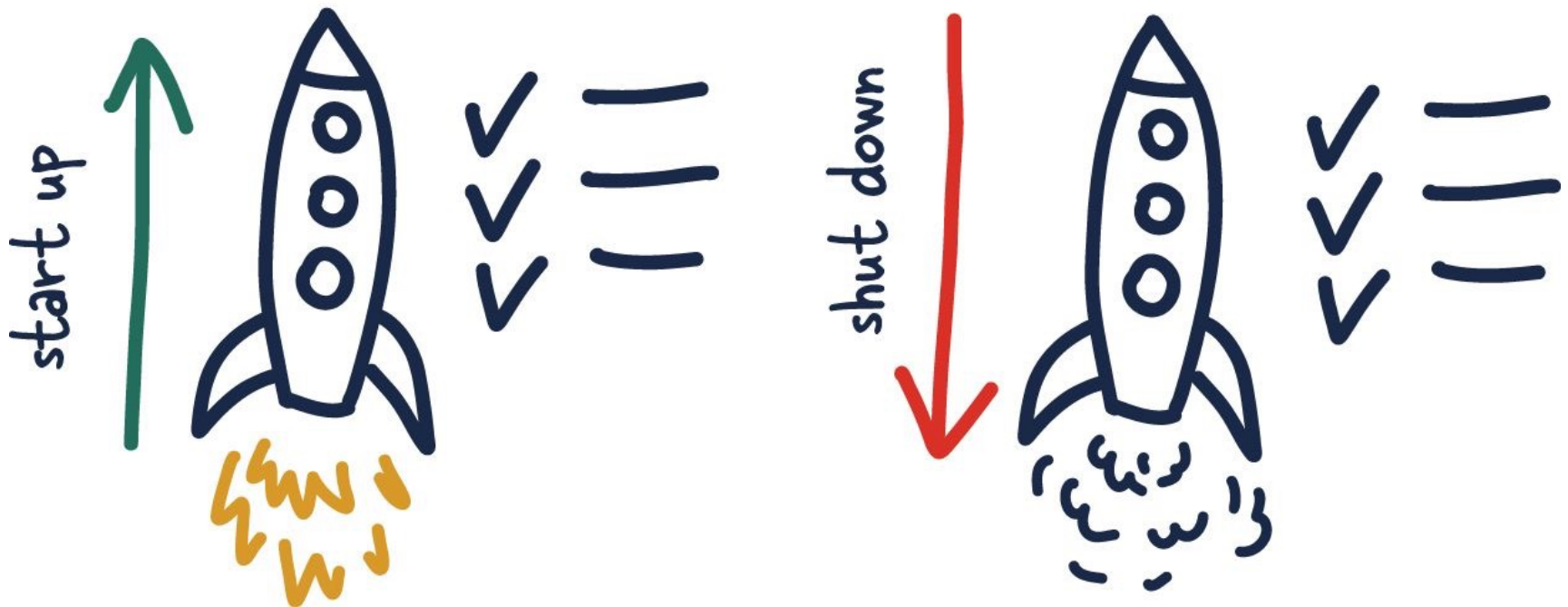
- ❖ **Concurrency** is achieved by scaling individual processes (horizontal scaling), to take advantage of OS.



#9 – Disposability

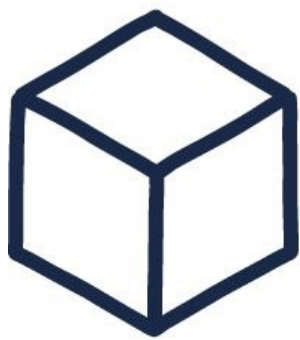
❖ Processes must be **disposable**

- Fast startup and graceful shutdown behaviors lead to a more robust and resilient system



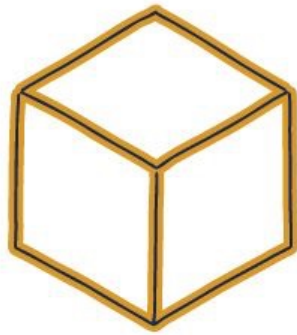
#10 – Dev/Prod parity

- ❖ All environments, from local development to production, should be as similar as possible



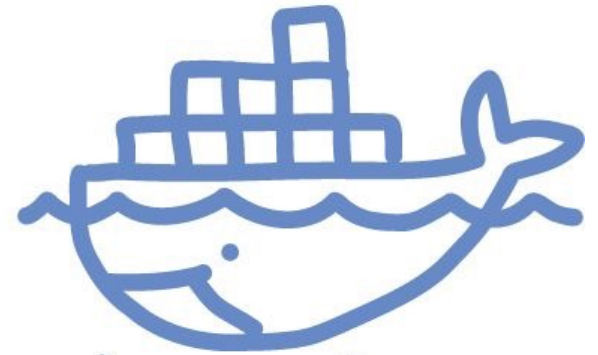
dev

=



prd

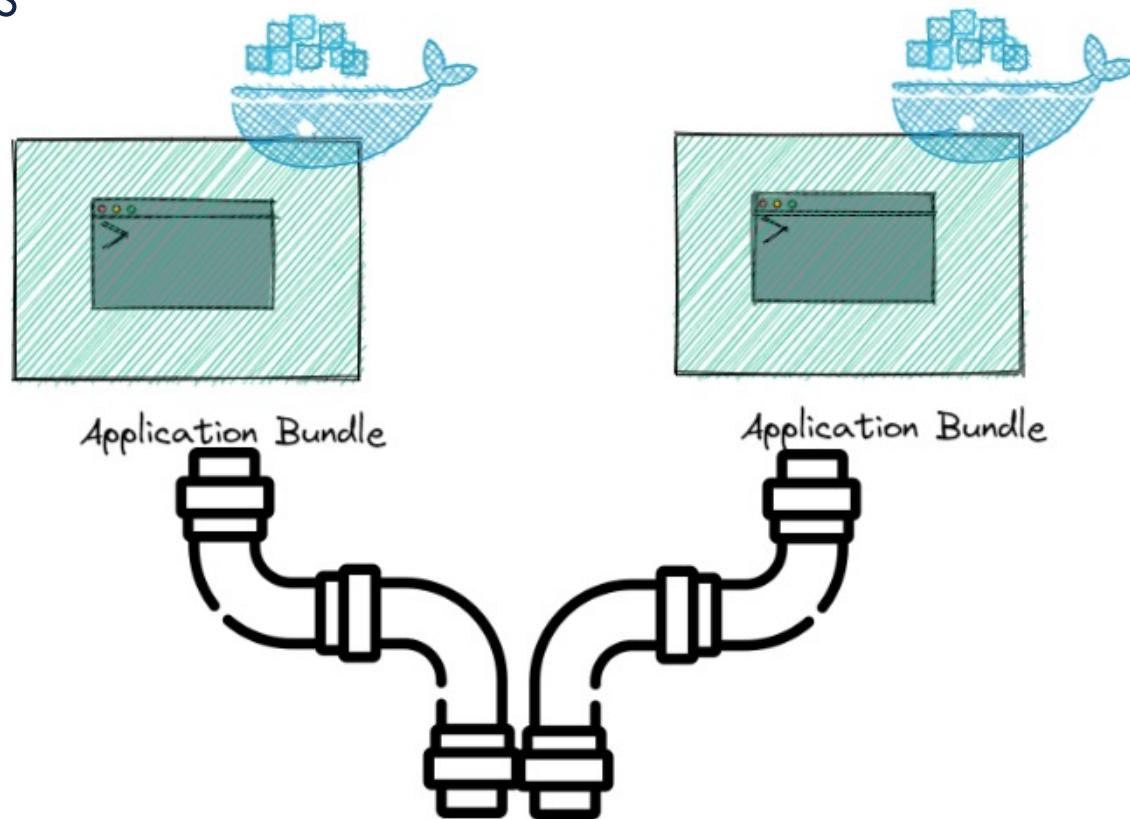
or



docker

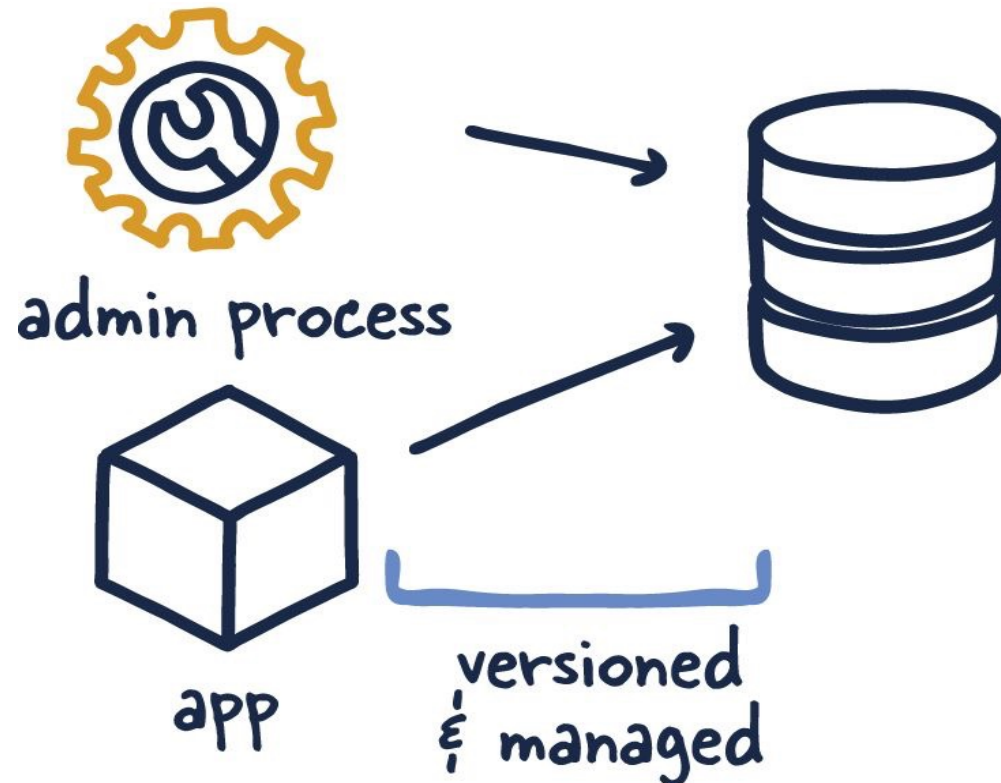
#11 – Logs

- ❖ Processes should produce **logs** as event streams and trust the execution environment to aggregate streams

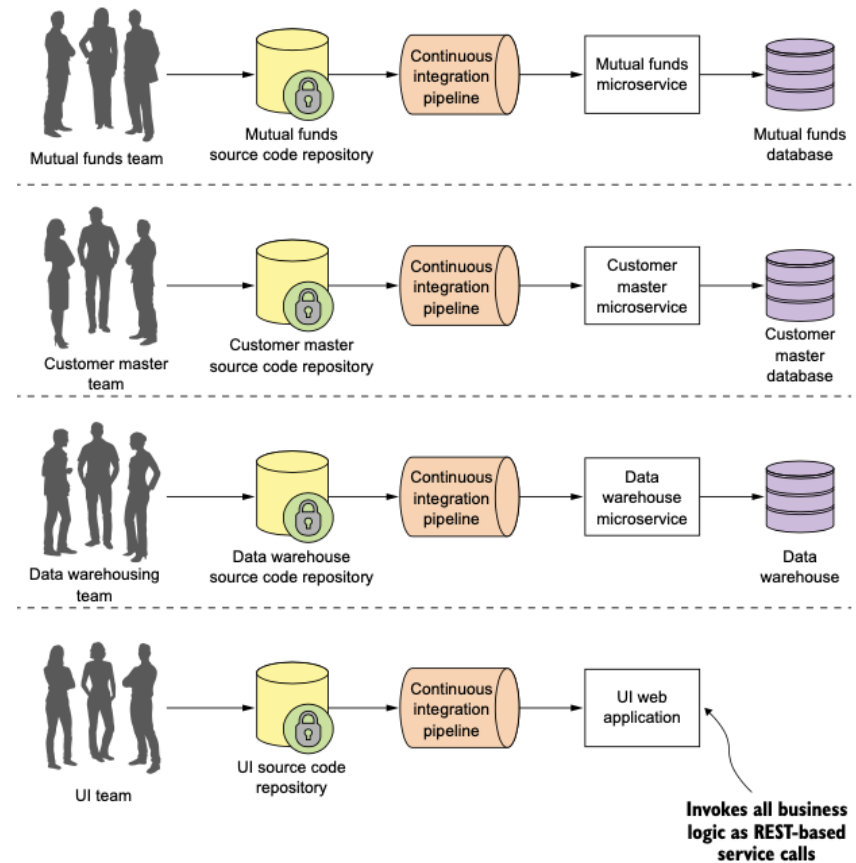
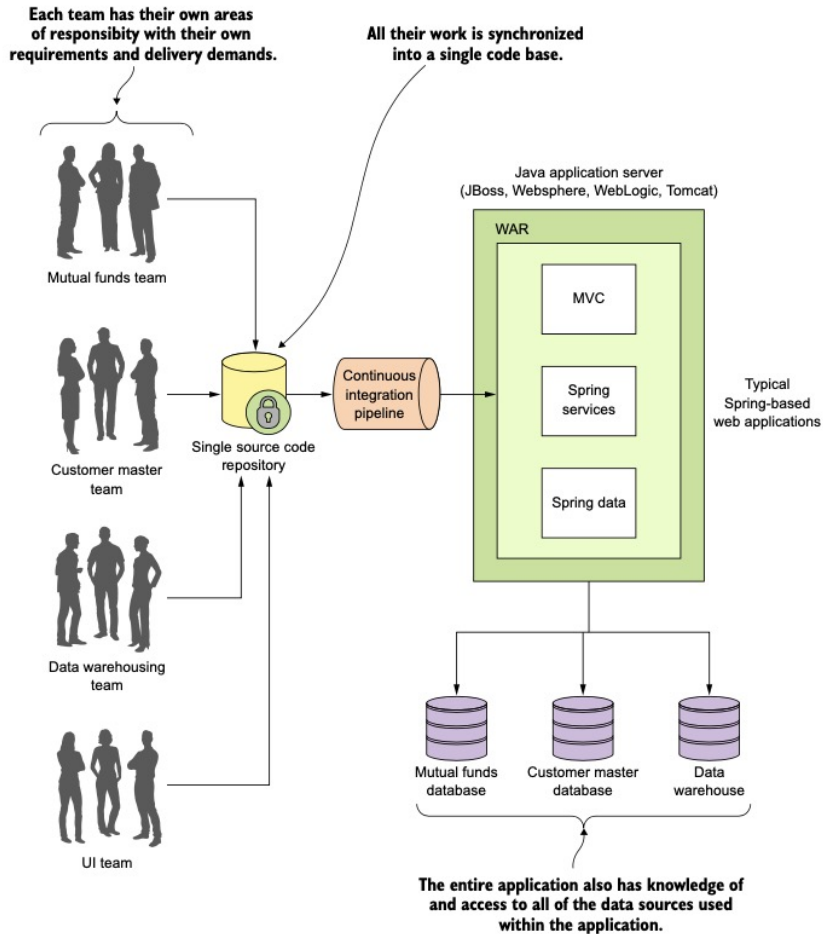


#12 – Admin Processes

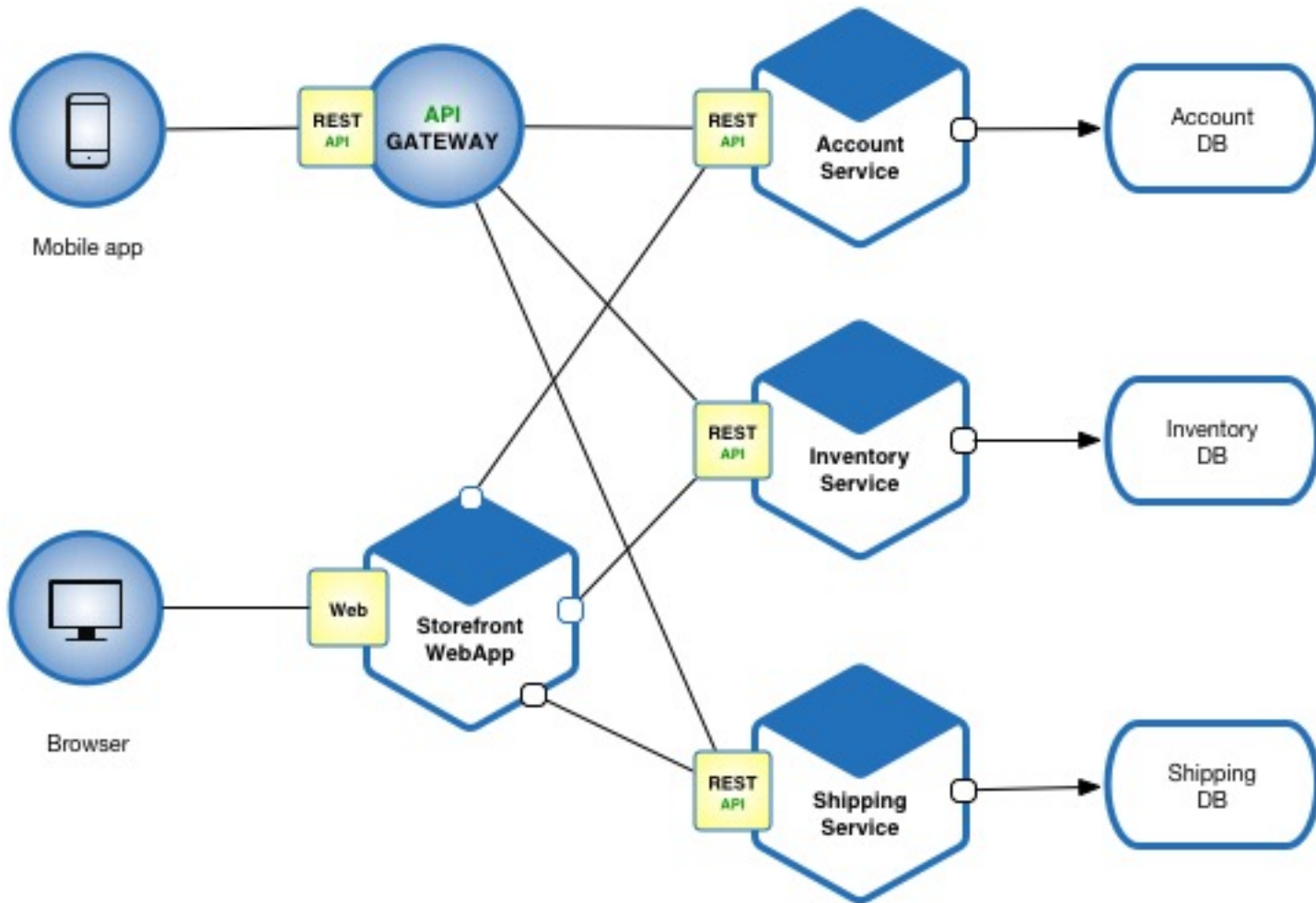
- ❖ If **admin** tasks are needed, they should be kept in source control and packaged alongside the application
 - to ensure that it is run with the same environment as the application



Monolithic vs. Microservices



Microservices



Microservices

- ❖ Although there is no standard definition for microservices, most reference Martin Fowler's seminal paper.
 - “Microservices: A new architectural term”
- ❖ The paper explains that microservices are used to compose complex applications by using:
 - **small**,
 - **independent** (autonomous),
 - **replaceable** processes that
 - **communicate** by using lightweight APIs that do not depend on language.
- ❖ Each microservice is a 12-factor application, with replaceable backing services providing a message broker, service registry, and independent data stores.

The meaning of “small”

- ❖ Many descriptions make parallels between the roles of individual microservices and chained commands on the UNIX command line:

```
$ ls | grep 'service' | sort -r
```

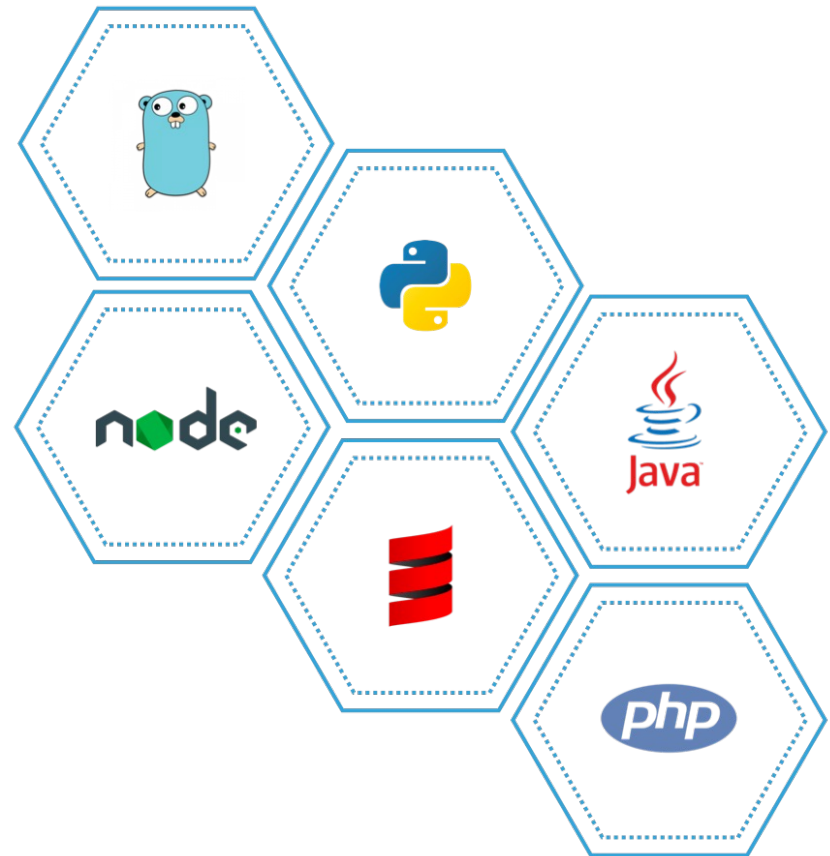
- These UNIX commands each do different things.
 - We can use the commands without awareness of what language the commands are written in, or how large their codebases are.
- ❖ The use of the word “small”, as applied to a microservice, essentially means that it is focused on purpose.
 - The microservice **should do one thing** and **do that one thing well**.

Independence and autonomy

- ❖ Each service must be capable of being started, stopped, or replaced at any time without tight coupling to other services.
 - Many other characteristics of microservice architectures follow from this single factor.
- ❖ If deploying changes requires simultaneous or time sensitive updates to multiple services, you are doing it wrong.
- ❖ There are a few possible outcomes:
 - You need to revise your versioning strategy to preserve the independent lifecycle of services.
 - The current service boundaries are not drawn properly, and services should be refactored into properly independent pieces.

Two key characteristics: Polyglot

- ❖ **Polyglot** is a frequently cited benefit of microservice-based architectures.
 - Being able to choose the appropriate language or data store to meet the needs of each service can be powerful and can bring significant efficiency.
 - Polyglot applications are only possible with language-agnostic protocols.

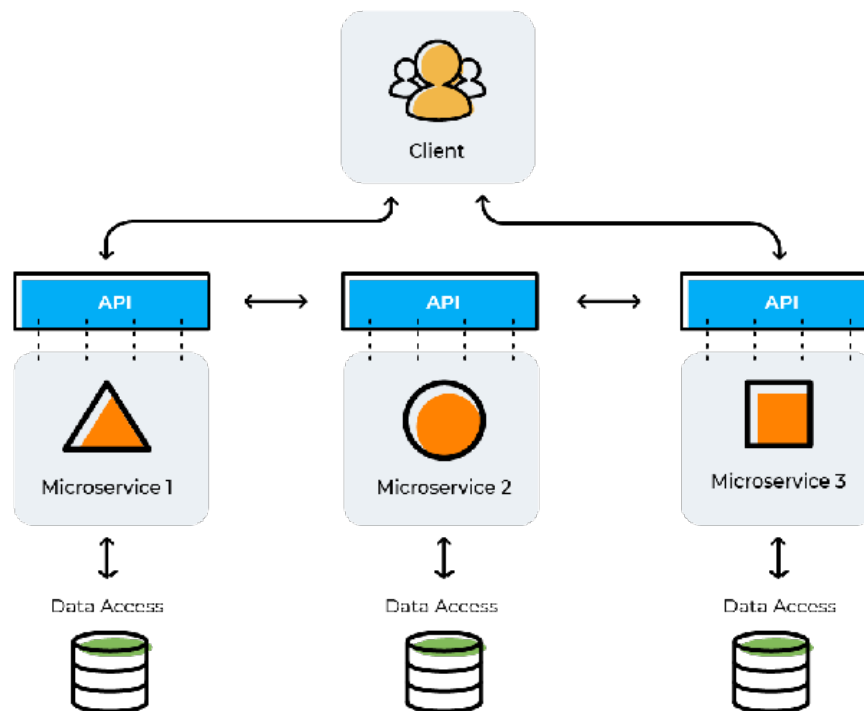


Two key characteristics: REST

❖ Representational State Transfer (REST)

architecture pattern

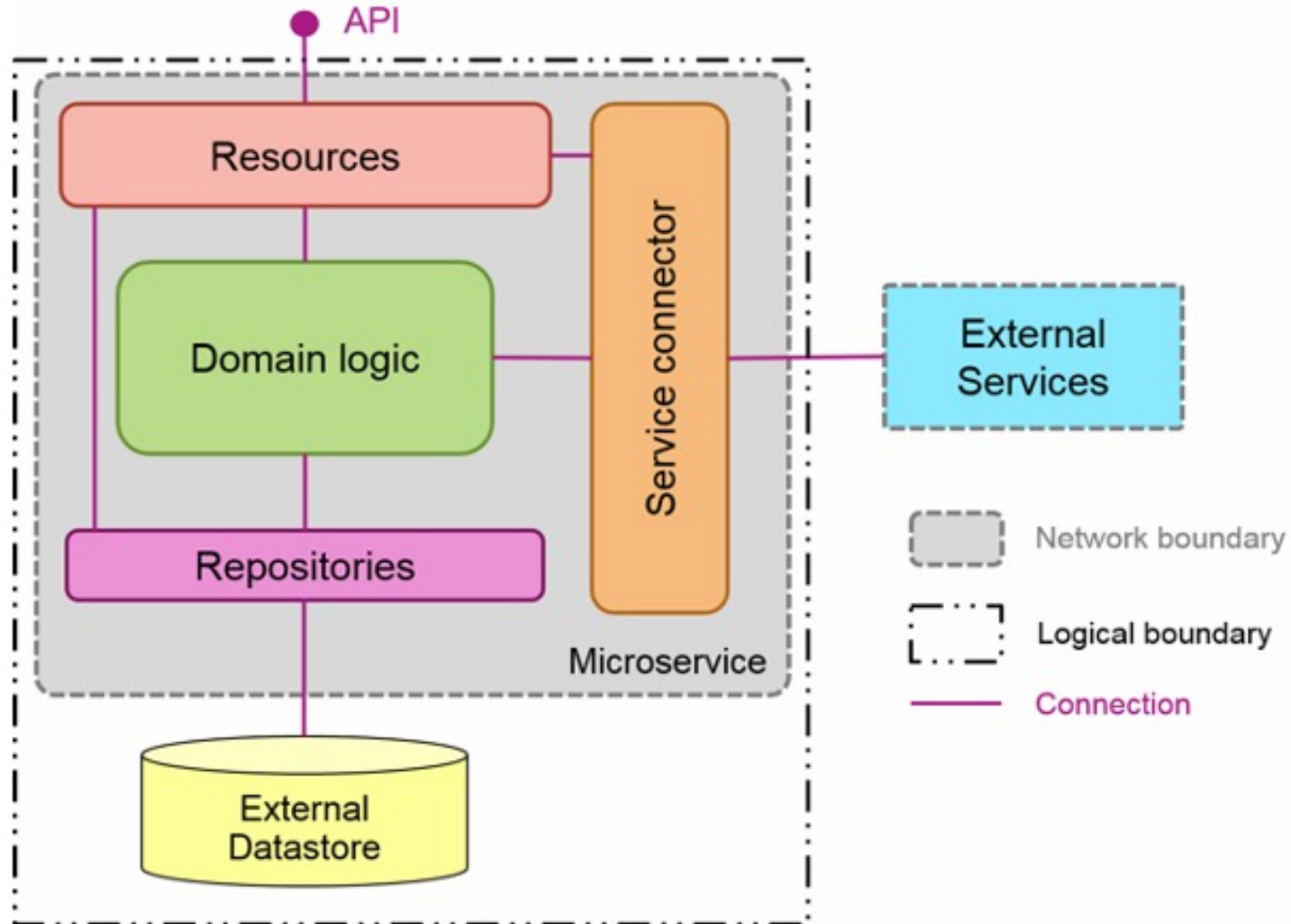
- It define guidelines for creating uniform interfaces that separate the on-the-wire data representation from the implementation of the service.
- RESTful architectures require the request state to be maintained by the client, allowing the server to be stateless.



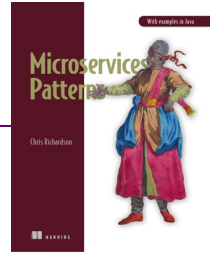
Microservice internal structure

- ❖ It should follow a few rules when creating classes, each performing one of these tasks:
 - Perform domain logic
 - Expose resources
 - Make external calls to other services
 - Make external calls to a data store
- ❖ These are general recommendations for code structure that do not have to be followed strictly.
 - The important characteristic is to reduce the risk of making changes.

Microservice internal structure



Microservices adoption antipatterns



- ❖ Microservices are a magic pixie dust
 - believing that a sprinkle of microservices will solve all of your development problems
- ❖ Microservices as the goal
 - making the adoption of microservices the goal and measuring success in terms of the number of services written
- ❖ Red Flag Law
 - retaining the same development process and organization structure that were used when developing monolithic applications.
- ❖ Scattershot adoption
 - multiple application development teams attempt to adopt the microservice architecture without any coordination
- ❖ Trying to fly before you can walk
 - Start practicing basic software development techniques, such as clean code, good design, and automated testing
- ❖ Focussing on Technology
 - focussing on technology aspects of microservices, most commonly the deployment infrastructure, and neglecting key issues, such as service decomposition

Creating Microservices in Java

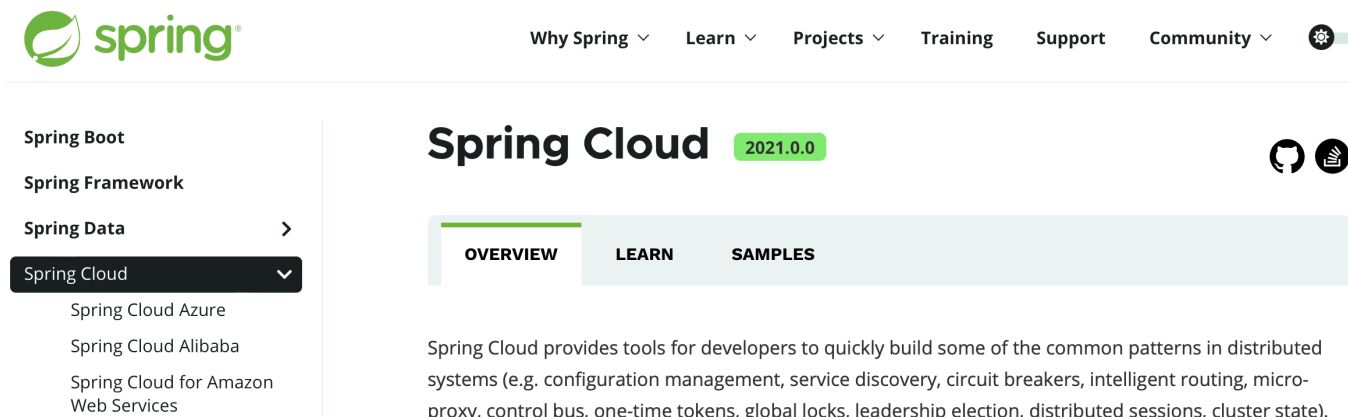
- ❖ How to identify and create the microservices?
 - with specific emphasis on how identified candidates are converted into RESTful APIs,
 - and then implemented in Java.
- 1. Java platforms
- 2. Versioned dependencies
- 3. Identifying services
- 4. Creating REST APIs

Java platforms

- ❖ Using **annotations** can simplify the codebase by eliminating boilerplate
 - but can reach a subjective point where there is “too much magic”.
- ❖ We can create perfectly functional microservices using nothing but low-level Java libraries.
- ❖ However, it is generally recommended to have something provide a reasonable base level of support for common, cross-cutting concerns.
 - It allows the codebase for a service to focus on satisfying functional requirements.

Spring Boot and Spring Cloud

- ❖ **Spring Boot** emphasizes convention over configuration
 - Uses a combination of annotations and classpath discovery to enable additional functions.
- ❖ **Spring Cloud** is a collection of integrations between third-party cloud technologies and the Spring programming model.



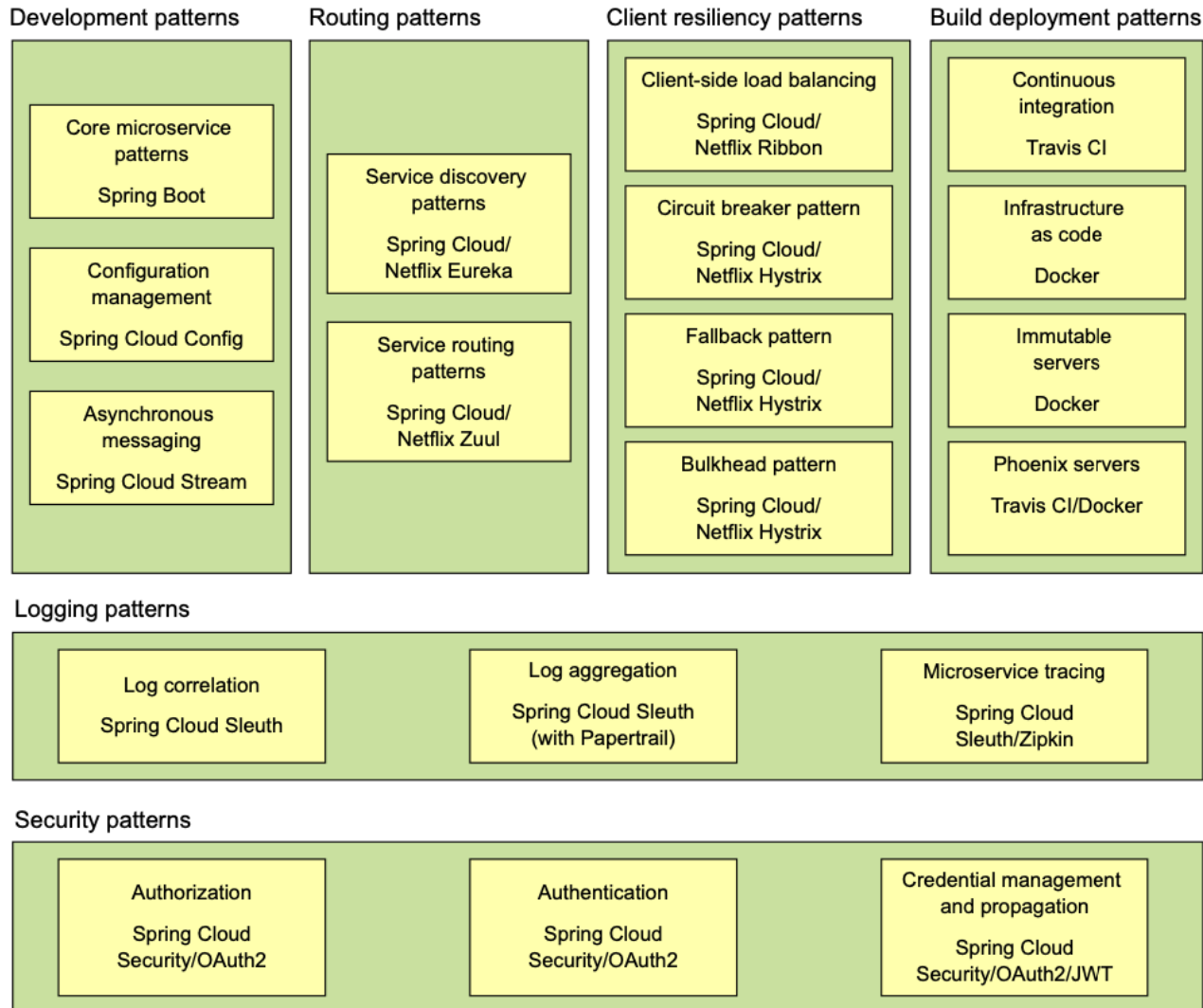
Spring Cloud

- ❖ We may develop microservices using **Spring Boot**.
 - These are all stand alone applications.
 - But suppose we now have to connect the various applications and build a distributed system.
- ❖ **Spring Cloud** deals with:
 - Complexity associated with distributed systems
 - Network issues, Latency overhead, Bandwidth issues, security issues.
 - Service Discovery
 - Service discovery tools manage how processes and services in a cluster can find and talk to one another.
 - Redundancy
 - Redundancy issues in distributed systems.
 - Load balancing
 - Performance issues
 - Deployment complexities
 - Requirement of DevOps skills

Spring Cloud – Main Projects

- ❖ Spring Cloud Azure
- ❖ Spring Cloud Alibaba
- ❖ Spring Cloud for AWS
- ❖ Spring Cloud Bus
- ❖ Spring Cloud CLI
- ❖ Spring Cloud for Cloud Foundry
- ❖ Spring Cloud Cluster
- ❖ Spring Cloud Commons
- ❖ Spring Cloud Config
- ❖ Spring Cloud Connectors
- ❖ Spring Cloud Consul
- ❖ Spring Cloud Contract
- ❖ Spring Cloud Function
- ❖ Spring Cloud Gateway
- ❖ Spring Cloud GCP
- ❖ Spring Cloud Netflix
- ❖ Spring Cloud Open Service Broker
- ❖ Spring Cloud Pipelines
- ❖ Spring Cloud Schema Registry
- ❖ Spring Cloud Security
- ❖ Spring Cloud Skipper
- ❖ Spring Cloud Sleuth
- ❖ Spring Cloud Stream
- ❖ Spring Cloud Stream Applications
- ❖ Spring Cloud Stream App Starters
- ❖ Spring Cloud Task
- ❖ Spring Cloud Vault
- ❖ Spring Cloud Zookeeper
- ❖ Spring Cloud App Broker
- ❖ Spring Cloud Circuit Breaker
- ❖ Spring Cloud Kubernetes
- ❖ Spring Cloud OpenFeign

Spring Cloud – Main Projects



Key projects – Cloud Config

<https://docs.spring.io/spring-cloud-config/docs/current/reference/html/>

❖ Configuration management

- client/server approach for storing and **serving distributed configurations** across multiple applications and environments.

```
@Configuration @RestController
```

```
@EnableAutoConfiguration
```

```
public class Application {
```

```
    @Value("${config.name}")
```

```
    private String name;
```

```
    @RequestMapping("/")
```

```
    public String home() { return "Hello " + name; }
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(Application.class, args);
```

```
    }
```

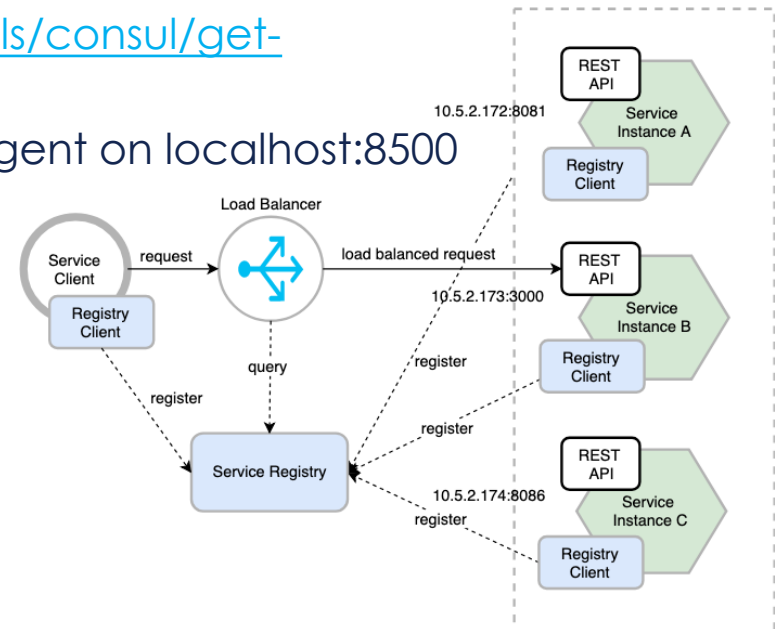
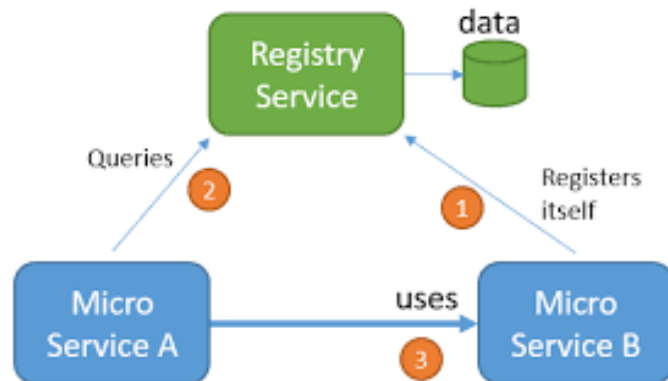
```
}
```

- Properties files can use Encrypt/Decrypt
 - e.g. password: {cypher}AUGUGUHI LUHUKJHGYGJHKG

Key projects – Naming/Discovery

<https://spring.io/guides/gs/service-registration-and-discovery/>

- ❖ Server/Middleware designed for **discovery and load balancing of web apps.**
- ❖ Eureka – Service registration and discovery
- ❖ **Consul** – Service Mesh
 - manages all service-to-service communication within a distributed (potentially microservice-based)
 - <https://learn.hashicorp.com/tutorials/consul/get-started?in=consul/getting-started>
 - @EnableDiscoveryClient - Consul agent on localhost:8500



Key projects – Dynamic scaling

❖ **LoadBalancer** (Ribbon)

❖ **Declarative REST** Client (OpenFeign)

- creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations

❖ **Circuit Breaker** (Resilience4, Sentinel, ...)

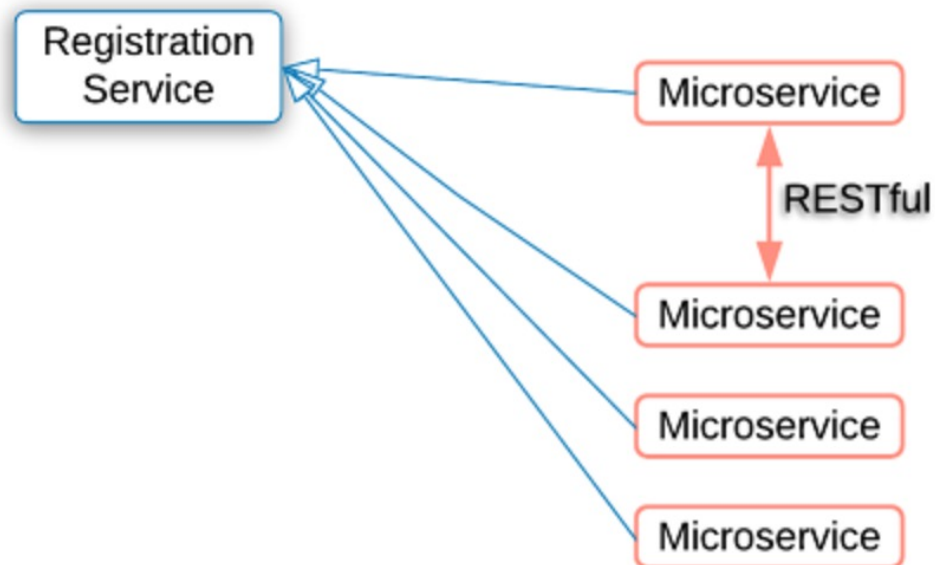
- to detect failures and encapsulates the logic of preventing a failure

❖ **Data Flow**

- provides tools to create complex topologies for streaming and batch data pipelines.

Spring Cloud: Example

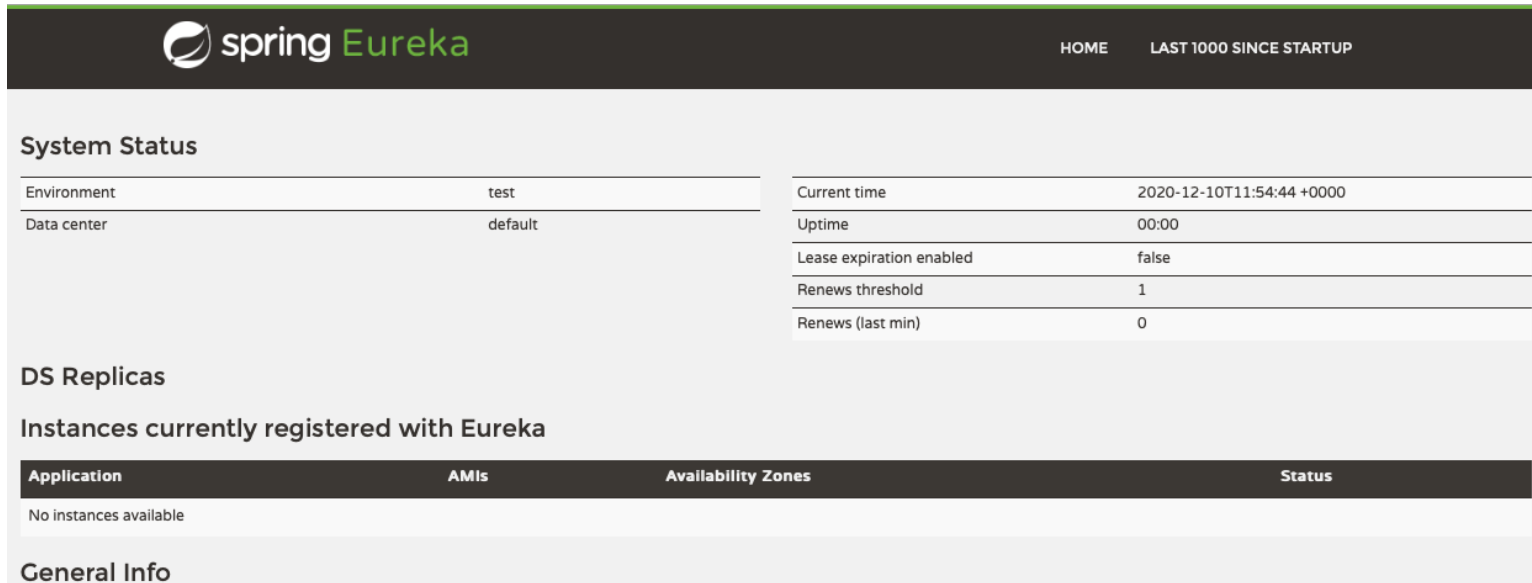
```
spring-cloud-getting-started git:(master) $ ls  
hello-eureka-server  
hello-service  
hello-web-client-service
```



<https://www.logicbig.com/tutorials/spring-framework/spring-cloud/hello-world.html>

Example – main class

```
// hello-eureka-server/../HelloEurekaServerMain.java
@SpringBootApplication
@EnableEurekaServer
public class HelloEurekaServerMain {
    public static void main(String[] args) {
        SpringApplication.run(HelloEurekaServerMain.class, args);
    }
}
```



The screenshot displays the Spring Eureka web interface. At the top, there's a navigation bar with the Spring Eureka logo and links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into several sections:

- System Status:** A table showing environment details.

System Status	
Environment	test
Data center	default
- System Metrics:** A table showing operational metrics.

Current time	2020-12-10T11:54:44 +0000
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0
- DS Replicas:** A section for distributed storage replicas.
- Instances currently registered with Eureka:** A table header with columns: Application, AMIs, Availability Zones, and Status. Below the header, it states "No instances available".
- General Info:** A section for general information.

Example – hello-service

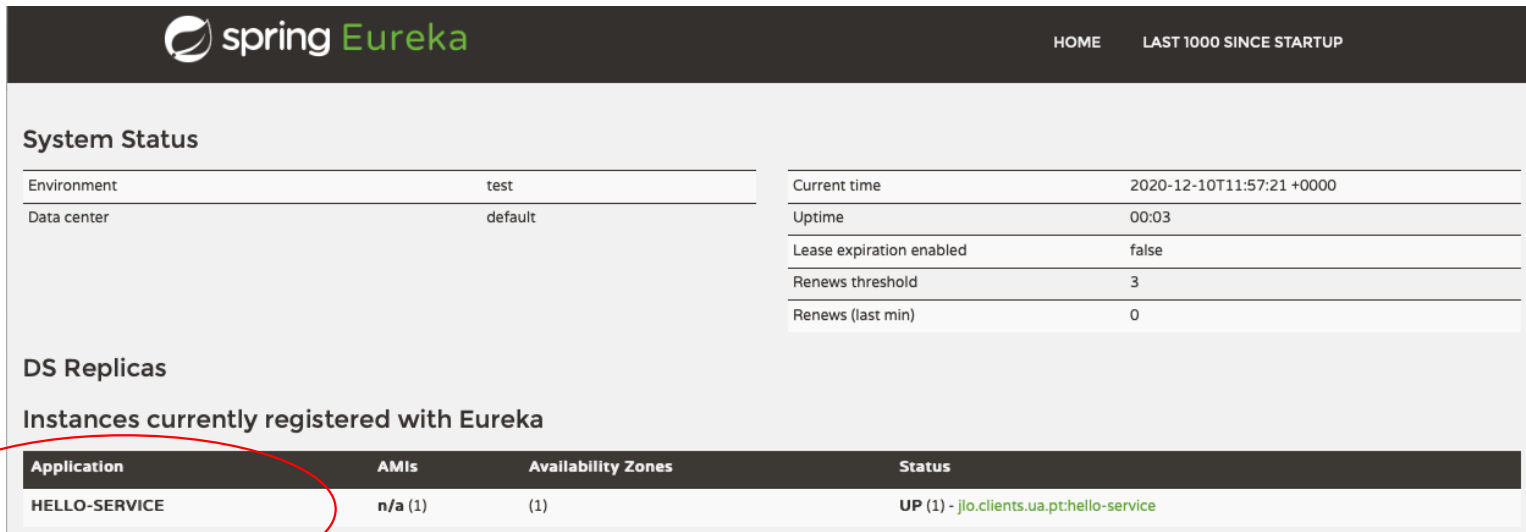
HelloServiceMain.java

HelloController.java

HelloObject.java

```
@SpringBootApplication
@EnableDiscoveryClient
public class HelloServiceMain{

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



The screenshot shows the Spring Eureka dashboard. The top navigation bar includes the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The 'System Status' section contains two tables. The left table lists 'Environment' as 'test' and 'Data center' as 'default'. The right table lists 'Current time' as '2020-12-10T11:57:21 +0000', 'Uptime' as '00:03', 'Lease expiration enabled' as 'false', 'Renews threshold' as '3', and 'Renews (last min)' as '0'. The 'DS Replicas' section is empty. The 'Instances currently registered with Eureka' section features a table with one entry, 'HELLO-SERVICE', which is circled in red. This entry shows 'n/a (1)' for AMIs, '(1)' for Availability Zones, and 'UP (1) - jlo.clients.ua.pt:hello-service' for Status.

System Status	
Environment	test
Data center	default

System Status	
Current time	2020-12-10T11:57:21 +0000
Uptime	00:03
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
HELLO-SERVICE	n/a (1)	(1)	UP (1) - jlo.clients.ua.pt:hello-service

Example – hello-web-client-service

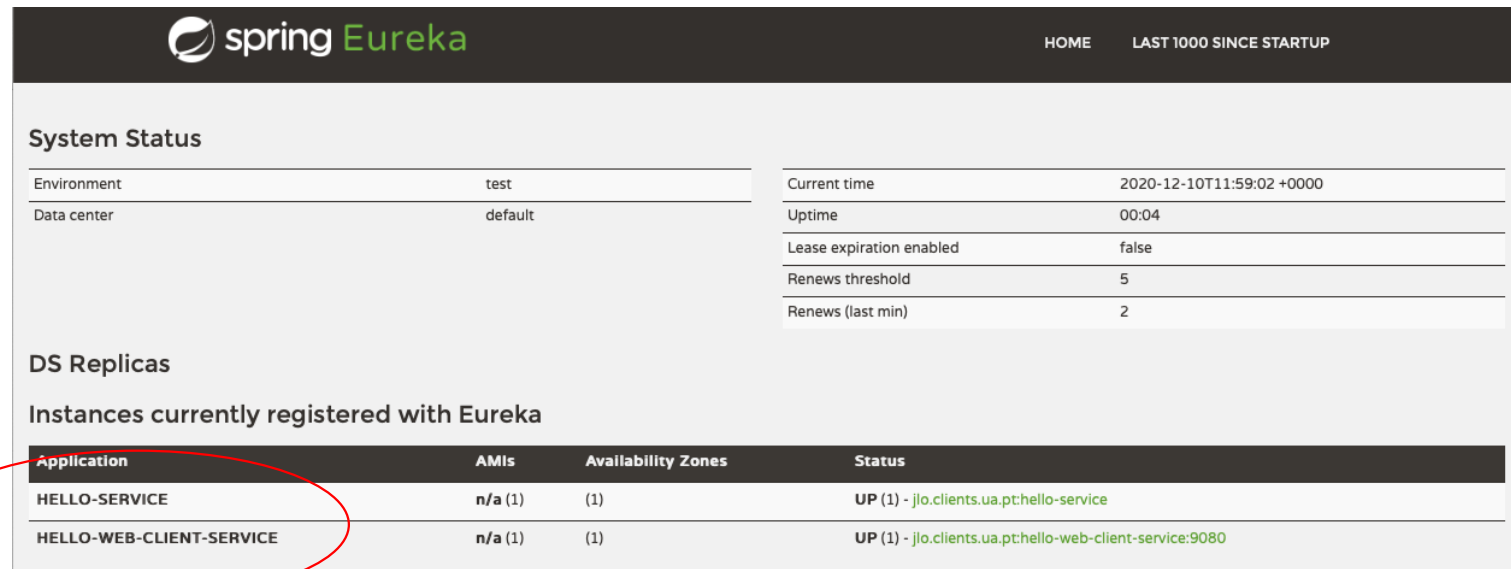
HelloObject.java

HelloWebClientController.java

HelloWebClientServiceMain.java

```
@SpringBootApplication
@EnableDiscoveryClient
public class HelloWebClientServiceMain {

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



The screenshot shows the Spring Eureka dashboard. At the top, there's a header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section contains two tables. The left table shows 'Environment: test' and 'Data center: default'. The right table shows 'Current time: 2020-12-10T11:59:02 +0000', 'Uptime: 00:04', 'Lease expiration enabled: false', 'Renews threshold: 5', and 'Renews (last min): 2'. The 'DS Replicas' section is empty. The 'Instances currently registered with Eureka' section contains a table with two rows: 'HELLO-SERVICE' and 'HELLO-WEB-CLIENT-SERVICE'. The 'HELLO-SERVICE' row is circled in red. The table has columns for 'Application', 'AMIs', 'Availability Zones', and 'Status'.

Application	AMIs	Availability Zones	Status
HELLO-SERVICE	n/a (1)	(1)	UP (1) - jlo.clients.ua.pt:hello-service
HELLO-WEB-CLIENT-SERVICE	n/a (1)	(1)	UP (1) - jlo.clients.ua.pt:hello-web-client-service:9080

Versioned dependencies

- ❖ Build tools like Apache Maven or Gradle provide easy mechanisms for defining and managing dependency versions.
 - Explicitly declaring the version ensures that the artifact runs the same in production as it did in the testing environment.
- ❖ Several tools are available to make creating Java applications easier, e.g.:
 - Spring Initializr
<http://start.spring.io/>
 - Open Liberty
<https://openliberty.io>

Documenting APIs

- ❖ The **Open API Initiative** (OAI) is a consortium focused on standardizing RESTful APIs descriptions.
 - <https://www.openapis.org>
- ❖ The OpenAPI specification is based on **Swagger**,
 - which defines the structure and format of metadata to create a representation of a RESTful API.
 - <https://github.com/OAI/OpenAPI-Specification>
- ❖ This definition is usually expressed in a single, portable file - name.json or name.yaml
- ❖ The swagger definition can further be used to generate client or server **stubs**
 - Simulate the behavior of existing code

Example

<https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore.yaml>

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
    post: ...
```

Example

Schemes

HTTP

Authorize

pet

Everything about your Pets

Find out more: <http://swagger.io>

POST

/pet

Add a new pet to the store

addPet

PUT

/pet

Update an existing pet

updatePet

GET

/pet/findByStatus

Finds Pets by status

findPetsByStatus

GET

/pet/findByTags

Finds Pets by tags

findPetsByTags

GET

/pet/{petId}

Find pet by ID

getPetById

POST

/pet/{petId}

Updates a pet in the store with form data

updatePetWithForm

DELETE

/pet/{petId}

Deletes a pet

deletePet

REST APIs: Use the correct HTTP verb

- ❖ **POST** operations may be used to Create(C) resources.
 - The distinguishing characteristic of a POST operation is that it is not idempotent (but it may).
 - For example, if a POST request is used to create resources, and it is started multiple times, a new, unique resource should be created as a result of each invocation.
- ❖ **GET** operations must be both idempotent and nullipotent.
 - They should cause no side effects and should only be used to Retrieve(R) information.
- ❖ **PUT** operations can be used to Update(U) resources.
 - PUT operations usually include a complete copy of the resource to be updated, making the operation idempotent.
- ❖ **PATCH** operations allow partial Update(U) of resources.
 - They might or might not be idempotent depending on how the delta is specified and then applied to the resource.
- ❖ **DELETE** operations are used to Delete(D) resources.
 - Delete operations are idempotent, as a resource can only be deleted once.
 - However, the return code varies, as the first operation succeeds (200), while subsequent invocations do not find the resource (204).

Machine-friendly, descriptive results

- ❖ The **HTTP status code** should be relevant and useful.
 - Use a 200 (OK) when everything is fine. When there is no response data, use a 204 (NO CONTENT) instead.
 - Beyond that technique, a 201 (CREATED) should be used for POST requests that result in the creation of a resource, whether there is a response body or not.
 - Use a 409 (CONFLICT) when concurrent changes conflict, or a 400 (BAD REQUEST) when parameters are malformed.
 - For more information, see the RFC2616 and RFC7231.
 - .. and this post: <https://dev.to/khaosdoctor/the-complete-guide-to-status-codes-for-meaningful-rest-apis-1-5c5>
- ❖ Consider also what data is being returned in the responses to make communication efficient.
 - The created/modified resource is usually included in the response, because it eliminates the need for the caller to make an extra GET request to fetch the created resource.

Resource URLs and versioning

- ❖ In general, resources should be nouns, not verbs, and endpoints should be plural.
- ❖ This technique results in a clear structure for create, retrieve, update, and delete operations:
 - POST /accounts Create a new item
 - GET /accounts Retrieve a list of items
 - GET /accounts/16 Retrieve a specific item
 - PUT /accounts/16 Update a specific item
 - PATCH /accounts/16 Update a specific item
 - DELETE /accounts/16 Delete a specific item
- ❖ Relationships are modeled by nesting URLs, for example, something like
 - /accounts/16/credentials
for managing credentials associated with an account.

Versioning

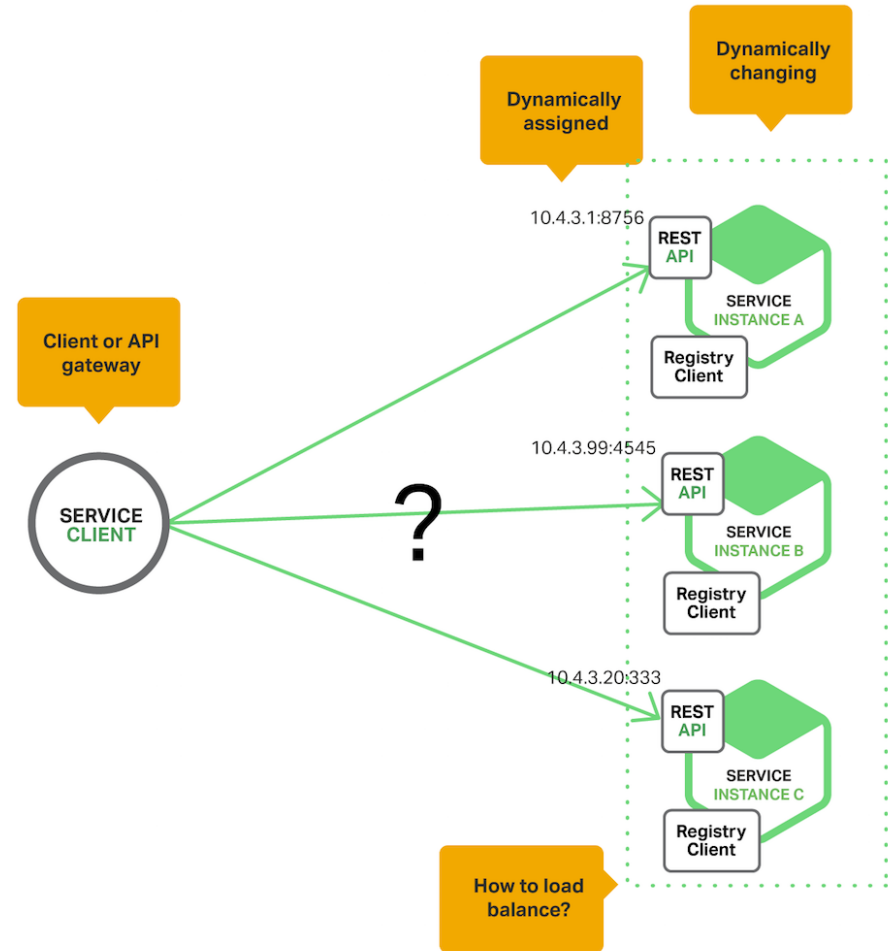
- ❖ One of the major benefits of microservices is the ability to allow services to **evolve independently**
 - Given that microservices call other services, that independence comes with a giant caveat. You cannot cause breaking changes in your API.
- ❖ If we do need to make breaking API changes for an existing service, there are generally a few ways to handle versioning a REST resource:
 - Header Versioning
 - One of the several header attributes in HTTP protocol is content version. Header-driven versioning uses this attribute to specify a service.
 - But .. information cannot be easily encoded in hypermedia links.
 - URI Versioning

Put the version in the URI

- ❖ The version should apply to the application, as a whole, for example:
 - /api/v1/accounts (not /api/accounts/v1).
- ❖ Advantages
 - It is easy to understand.
 - It is easy to achieve when building the services.
 - It is compatible with API browsing tools like Swagger and command-line tools like curl.
- ❖ Disadvantages
 - In a strict interpretation of the HTML standard, a URL should represent the entity and if the entity being represented didn't change, the URL should not change.
 - Another concern is that putting versions in the URI requires consumers to update their URI references.

Location services

- ❖ Microservices are designed to be easily scaled.
 - This scaling is done horizontally by scaling individual services.
- ❖ With multiple instances of microservices, we need a method for locating services and load balancing over the different instances of the service you are calling.



Service registry

- ❖ There are four reasons why a microservice might communicate with a service registry:
- ❖ **Registration**
 - After a service has been successfully deployed, the microservice must be registered with the service registry.
- ❖ **Heartbeats**
 - The microservice should send regular heartbeats to the registry to show that it is ready to receive requests.
- ❖ **Service discovery**
 - To communicate with another service, a microservice must call the service registry to get a list of available instances
- ❖ **De-registration**
 - When a service is down, it must be removed from the list of available services in the service registry.

Third-party registration # self-registration

- ❖ The registration of the microservice with the service registry can be done by the microservice or by a third party.
 - Using **self-registration** pulls the registration and heartbeat logic into the microservice itself.
 - Using a **third party** requires said party to inspect the microservice to determine the current state and relay that information to the service registry.
 - The advantage of using a third party is that the registration and heartbeat logic is kept separate from the business logic.
 - The disadvantage is that there is an extra piece of software to deploy and the microservice must be exposed to a health endpoint for the third party to poll.
- ❖ Many service registry solutions provide a **convenience library form registration**, reducing the complexity of the code required. Some service registry solutions are available:
 - Consul, <https://www.consul.io/>
 - Eureka, <https://github.com/Netflix/eureka>
 - ZooKeeper, <https://zookeeper.apache.org>

Fault tolerance

❖ Timeouts

- When making a request to another microservice, whether asynchronously or not, the request must have a timeout.

❖ Circuit breakers

- A circuit breaker is designed to avoid repeating timeouts. If the count of failures reaches a certain level, it prevents further calls from occurring.

❖ Bulkheads

- We need to make sure that a failure in one part of your application doesn't take down the whole thing.
- The simplest implementation of the bulkhead pattern is to provide fallbacks. They allow the application to continue functioning when non-vital services are down.
 - For example, in an online retail store there might be a service that provides recommendations for the user. If the recommendation service goes down, the user should still be able to search for items and place orders.

❖ Consuming APIs

❖ Producing APIs

Consuming APIs

- ❖ As the consumer of an API, we need to **validate the response** to check that it contains the information.
 - If we are receiving JSON, we also need to parse the JSON data before performing any Java transformations.
- ❖ When doing validation or JSON parsing, we must:
 - Only **validate the request against the variables or attributes** that we need
 - Do not validate against variables just because they are provided.
 - Accept unknown attributes
 - Do not issue an exception if we receive an unexpected variable.
- ❖ In this way, microservices are more resilient against changes.

Producing APIs

- ❖ When providing an API to external clients, do these two things when accepting requests and returning responses:
 - Accept unknown attributes as part of the request
 - If a service has called your API with unnecessary attributes, discard those values. Returning an error in this scenario just causes unnecessary failures.
 - Only return attributes that are relevant for the API being invoked
 - Leave as much room as possible for the implementation of a service to change over time.
- ❖ By following these two rules we may also changing the API in the future and adapting to changing requirements from consumers - Robustness Principle.

Deployment pipelines and tools

- ❖ Deployment artifacts often run on different systems
 - many systems must be configured to make the services work.
- ❖ Human intervention should be reduced to a minimum.
 - The code should be in a **Source Code Management** (SCM) tool like Subversion or GIT.
 - It should be built with a tool that **includes dependency management** like Maven or Gradle.
 - Choose one of the **repository management tools** (such as Nexus or Artifactory) to store versioned binary files of your shared libraries.
 - Build your services using one of the **CI-Tools** like Jenkins or Bamboo.
- ❖ After the automated build of the microservice **testing must be done**.
 - Scripting frameworks/infrastructures like Chef, Puppet, Ansible, and Salt can handle this situation.

Packaging options

- ❖ After building the microservice we need to decide on the **packaging format to deploy the application**.
 - A recurring goal is the creation of immutable artifacts that can be run without change across different deployment environments.
- ❖ The following packaging options are available, among others:
 - JAR/WAR file deployment on preinstalled middleware
 - Executable JAR file
 - Containers
 - Every microservice on its own virtualized server

JAR/WAR deployment

- ❖ This variant has been the **default way** for deploying Java EE applications.
 - the application gets hosted in different application servers in different staging environments (such as quality assurance, performance test, and user acceptance test).
- ❖ Problems:
 - Frequent deployments in a shorter time period (day, hour) can be hard to achieve, since some application deployments need a restart of the application server.
 - If different applications are hosted together on the same application server, the load from one application can interfere with the performance of the other application.
 - High variations in the load of an application cannot easily be adopted in the infrastructure.
- ❖ If this pattern is used with microservices, the relationship should be **one-to-one between the service and the preinstalled middleware server**.
 - This configuration is common, for example, in PaaS environments, where you push only the WAR to a freshly built, predefined server.

Executable JAR file

- ❖ One way to avoid depending on the deployment environment to provide dependencies is to **pack everything that is needed to run the service** inside an executable JAR file.
 - These fat JAR files eliminates problems introduced by dependency mismatches between development and test or production deployment environments.
 - These self-contained JAR files are generally easy to handle, and help create a consistent, reproducible environment for local development and test in addition to target deployment environments.

Containerization

- ❖ The previous strategies are all running on an operating system that must be installed and configured to work with the executable JAR file.
- ❖ A strategy to avoid the problems resulting from inconsistently configured operating systems is to **bring more of this operating system with you.**
 - This strategy leads to the situation that a microservice runs on an operating system with the same configuration on every stage.
- ❖ Containers are lightweight virtualization artifacts that are hosted on an existing operating system.
 - **Docker provides an infrastructure capability to build a complete file system to run the application.**

Every microservice on its own server

- ❖ Another alternative for packaging is to use the same **virtualized server as the basis for every stage**.
 - Create a server template with the operating system and other tools that are needed.
 - Use this template as the basis for all servers that host microservices, which is another form of immutable server.
- ❖ To manage these servers and server templates, we can use tools such as **Vagrant, VMware, and VirtualBox**.
- ❖ This configuration leads mainly to two variants:
 - One microservice packaged on a single virtualized server.
 - Multiple microservice packages on a single server

Resources & Credits



- ❖ Microservices Best Practices for Java, M. Hofmann, E. Schnabel, K. Stanley, IBM
– chapters 2, 3, 4 & 8



- ❖ Spring Microservices in Action, John Carnell, Manning Publications