BOOTCAMP JAVA-MICROSERVICIOS | SESIÓN 02

BOOTCAMP JAVA-MICROSERVICIOS | Perú 2022

Por: Angelo Rafael Angulo Méndez

aangulom@emeal.nttdata.com

AGENDA

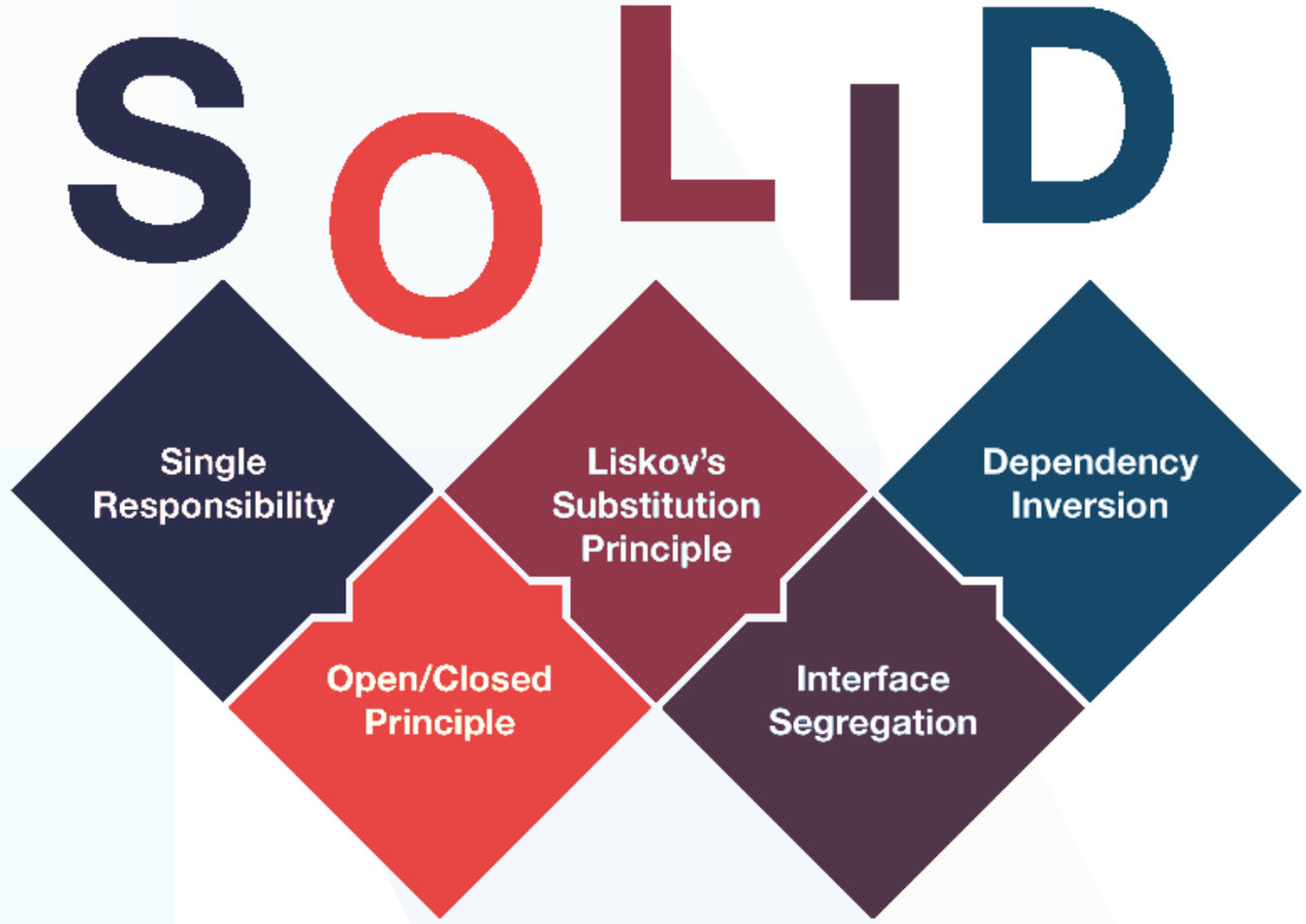
PRINCIPIOS SOLID

CONCEPTOS DE ARQUITECTURA
DE MICROSERVICIOS

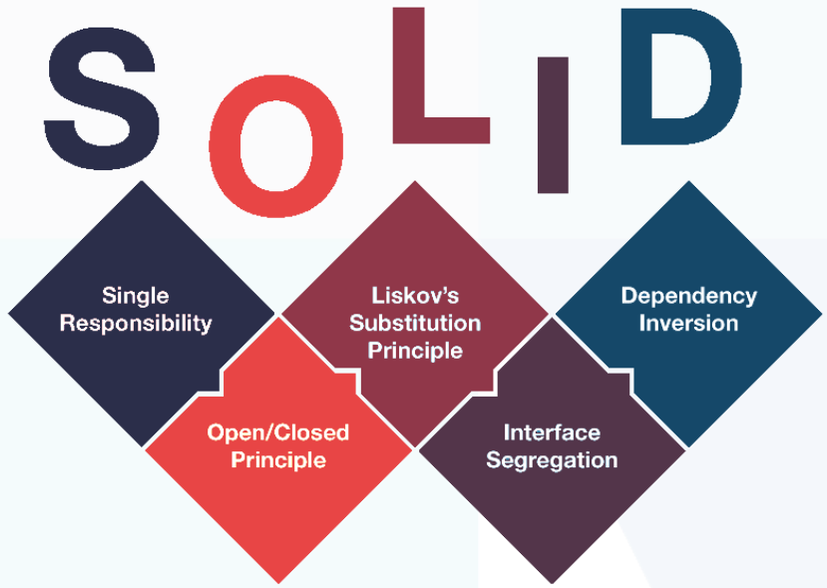
NTT DATA

PRINCIPIOS SOLID

Principios SOLID



Principios SOLID



- ✓ *Es un **acrónimo** introducido por Robert C. Martin a comienzos de la década del 2000 que representa **cinco principios básicos** de la programación orientada a objetos y el diseño.*
- ✓ *Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea **fácil de mantener** y ampliar con el tiempo.*
- ✓ *Los principios SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar **malos diseños** provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible.*

S – Single Responsability Principle (SRP)

```java
public class Customer {

    String customerId;
    String name;

    public String getCustomerId() {

    public void setCustomerId(String customerId) {

    public String getName() {

    public void setName(String name) {

    public void createCustomer(Customer customer) {

    public void updateCustomer(Customer customer) {

    public void deleteCustomer(Customer customer) {

}
```

https://www.enmilocalfunciona.io/principios-solid

- *Este principio establece que un componente o clase **debe tener una responsabilidad única, sencilla y concreta.***
- *Esto simplifica el código al evitar que existan clases que cumplan con múltiples funciones, las cuales son difíciles de memorizar y muchas veces significan una pérdida de tiempo buscando qué parte del código hace qué función.*

S – Single Responsability Principle (SRP)

```java
public class CustomerRefactor {

    String customerId;
    String name;

    public String getCustomerId() {

    public void setCustomerId(String customerId) {

    public String getName() {

    public void setName(String name) {

}
```

```java
public class CustomerService {

    public void createCustomer(Customer customer) {

    public void updateCustomer(Customer customer) {

    public void deleteCustomer(Customer customer) {

}
```

https://www.enmilocalfunciona.io/principios-solid

O – Open/Closed Principle (OCP)

```java
public class AreaCalculator {

    public static void main(String[] args) {
        System.out.println(calculateArea(new Rectangle(5.0,3.0)));
        System.out.println(calculateArea(new Triangle(5.0,3.0)));
    }

    private static double calculateArea(Shape shape) {
        double area = 0;
        if(shape instanceof Rectangle) {
            area = shape.width * shape.height;
        }
        if(shape instanceof Triangle) {
            area = (shape.width * shape.height) / 2;
        }
        return area;
    }
```

https://www.enmilocalfunciona.io/principios-solid

- *Establece que las entidades software (clases, módulos y funciones) deberían estar **abiertos para su extensión, pero cerrados para su modificación.***

- ***Abierto para extensión:***
  - *¿Cómo podemos hacerlo comportarse en nuevas y distintas formas a medida que la aplicación evoluciona, o para ajustarse a las necesidades de nuevas aplicaciones?*

- ***Cerrado para modificación:***
  - No se puede modificar el código de lo que hay.

O – Open/Closed Principle (OCP)

```java
public class Rectangle extends Shape  {

    public Rectangle(double width, double height) {
        super(width, height);
    }

    @Override
    public double calculateArea() {
        return  this.width * this.height;
    }
}
```

```java
public class Triangle extends Shape {

    public Triangle(double width, double height) {
        super(width, height);
    }

    @Override
    public double calculateArea() {
        return (this.width * this.height) / 2;
    }

}
```

```java
public class AreaCalculator {
    public static void main(String[] args) {
        System.out.println(calculateArea(new Rectangle(5.0,3.0)));
        System.out.println(calculateArea(new Triangle(5.0,3.0)));
    }

    private static double ~~calculateAreaDeprecated~~(Shape shape) {

    private static double calculateArea(Shape shape) {
        double area = 0;
        area = shape.calculateArea();
        return area;
    }
}
```

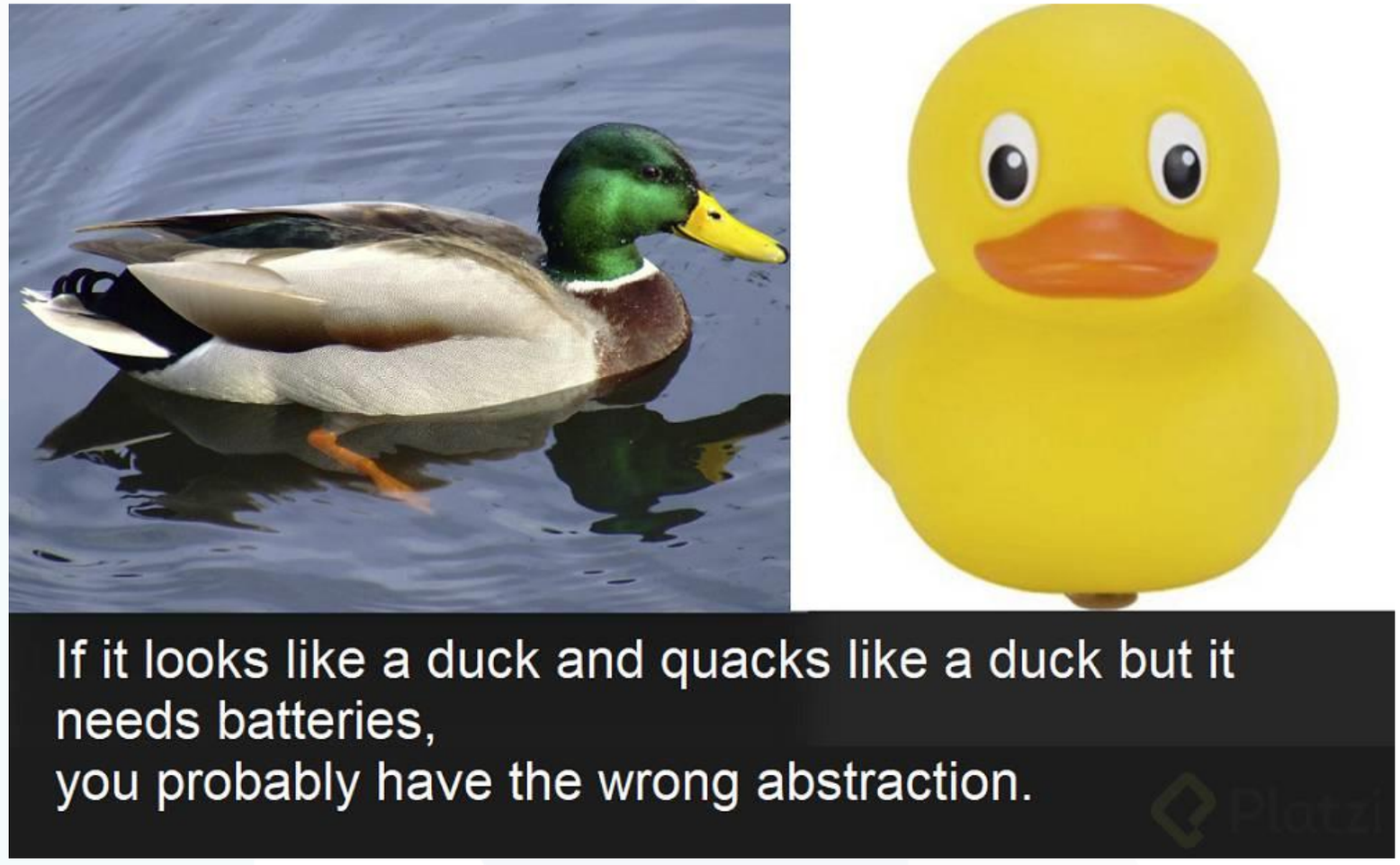
https://www.enmilocalfunciona.io/principios-solid

- *Herramientas bases para lograr mantenibilidad:*
  - *Abstracción.*
  - *Herencia.*
  - *Polimorfismo.*

L – Liskov Substitution Principle

Si para todo objeto o1 de tipo S existe un objeto o2 de tipo T tal que para todo programa P definido en función de T el comportamiento de P no cambia cuando o1 es substituido por o2, entonces S es un subtipo de T.

L – Liskov Substitution Principle



L – Liskov Substitution Principle

```java
public class RubberDuckDeprecated extends DuckDeprecated {


    public void quack() {
        System.out.println("Say quack...");
    }

    public void swim() {
        System.out.println("Swimming...");
    }

    @Override
    public void fly() {
        // TODO Auto-generated method stub

    }
}
```

- *Este principio establece que una **subclase puede ser sustituida por su superclase**. Es decir, podemos crear una subclase llamada RubberDuck, la cual deriva de la superclase Duck. Si al usar la superclase el programa falla o el comportamiento cambia, este principio no se cumple.*

L – Liskov Substitution Principle

```java
public class RubberDuck implements IQuack, ISwim {

    public void quack() {
        System.out.println("Say quack...");
    }

    public void swim() {
        System.out.println("Swimming...");
    }
}
```

```java
public class Duck implements IFly, IQuack, ISwim {

    public void fly() {
        System.out.println("Flying...");
    }

    public void quack() {
        System.out.println("Say quack...");
    }

    public void swim() {
        System.out.println("Swimming...");
    }
}
```

I – Interface Segretation Principle (ISP)

```java
public interface IAve {
    void volar();
    void comer();
    void nadar();
}

class Loro implements IAve{

    public void volar() {

    public void comer() {

    public void nadar() {
}

class Pinguino implements IAve{

    public void volar() {

    public void comer() {

    public void nadar() {
}
```

https://www.enmilocalfunciona.io/principios-solid

- *Este principio establece que los clientes **no deben ser forzados a depender de interfaces que no utilizan**. Es importante que cada clase implemente las interfaces que va a utilizar. De este modo, agregar nuevas funcionalidades o modificar las existentes será más fácil. .*

I – Interface Segretation Principle (ISP)

```
interface IAveRefactor {
    void comer();
}
interface IAveVoladora {
    void volar();
}

interface IAveNadadora {
    void nadar();
}

class Loro2 implements IAveRefactor, IAveVoladora {

    public void volar() {

    public void comer() {

}

class Pinguino2 implements IAveRefactor, IAveNadadora{

    public void comer() {

    public void nadar() {
}
```

https://www.enmilocalfunciona.io/principios-solid

D – Dependency Inversion Principle (DIP)

```java
public class CustomerController {

    @Value("${spring.application.name}")
    String name;

    @Value("${server.port}")
    String port;


    @Autowired
    private IGenericService<Customer> customerService;

    @Autowired
    private ICustomerMapper customerMapper;
```

```java
@Slf4j
@Service
@RequiredArgsConstructor
@Transactional
public class CustomerService implements IGenericService<Customer> {


    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private ICustomerMapper customerMapper;

    @Override
    public Flux<Customer> findAll(){
        log.debug("findAll executed");
        return customerRepository.findAll();
    }
```

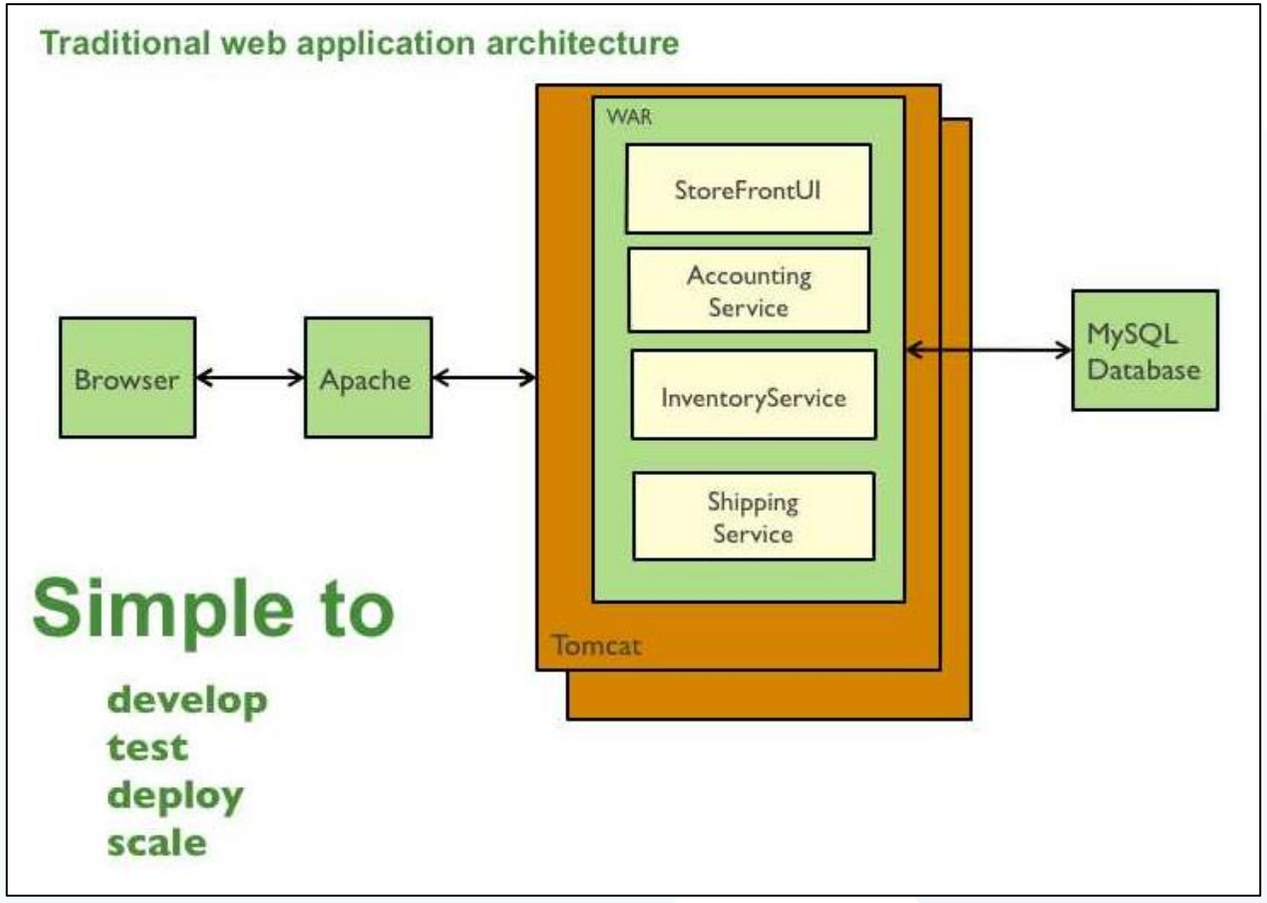
- ***Los módulos de alto nivel no deberían depender de módulos de bajo nivel.*** *Ambos deberían depender de abstracciones.*
- ***Las abstracciones no deberían depender de detalles****. Los detalles deberían depender de abstracciones.*

https://www.enmilocalfunciona.io/principios-solid

NTT DATA

ARQUITECTURA MICROSERVICIOS

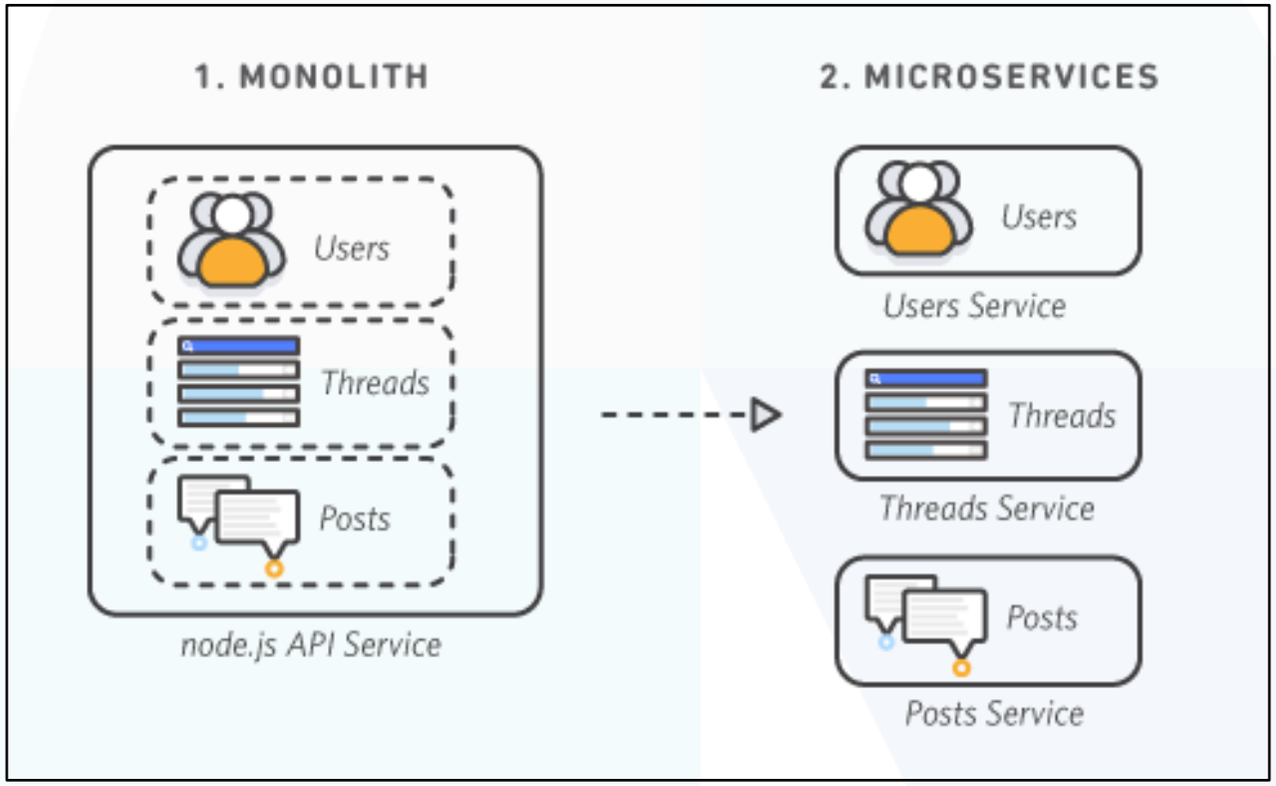
Arquitectura Monolítica



https://microservices.io/patterns/monolithic.html

- ✓ La aplicación puede ser difícil **de entender y modificar.**
- ✓ **IDE sobrecargado:** cuanto mayor es la base de código, más lento es el IDE
- ✓ **Contenedor web sobrecargado**: cuanto más grande es la aplicación, más tiempo tarda en iniciarse.
- ✓ **La implementación continua es difícil**: una gran aplicación monolítica también es un obstáculo para las implementaciones frecuentes.
- ✓ **Requiere un compromiso a largo plazo con una pila de tecnología**: una arquitectura monolítica lo obliga a estar casado con la pila de tecnología

Arquitectura de Microservicios



- ✓ La **arquitectura de microservicios** es una aproximación para el desarrollo de software que consiste en construir una aplicación como un conjunto de pequeños servicios, los cuales se ejecutan en su propio proceso y se comunican con mecanismos ligeros (normalmente una API de recursos HTTP).
- ✓ Cada microservicio se encarga de implementar una **funcionalidad completa del negocio**.
- ✓ Cada microservicio es desplegado de forma independiente y puede estar programado en **distintos lenguajes** y usar **diferentes tecnologías** de almacenamiento de datos.

Microservicios – Características

Los componentes son servicios

Organizada en torno a las funcionalidades del negocio.

Productos, no proyectos.

Extremos inteligentes, tuberías bobas.

Tener gobierno descentralizado permite usar tecnologías que se adapten mejor a cada funcionalidad

Gestión de datos descentralizada.

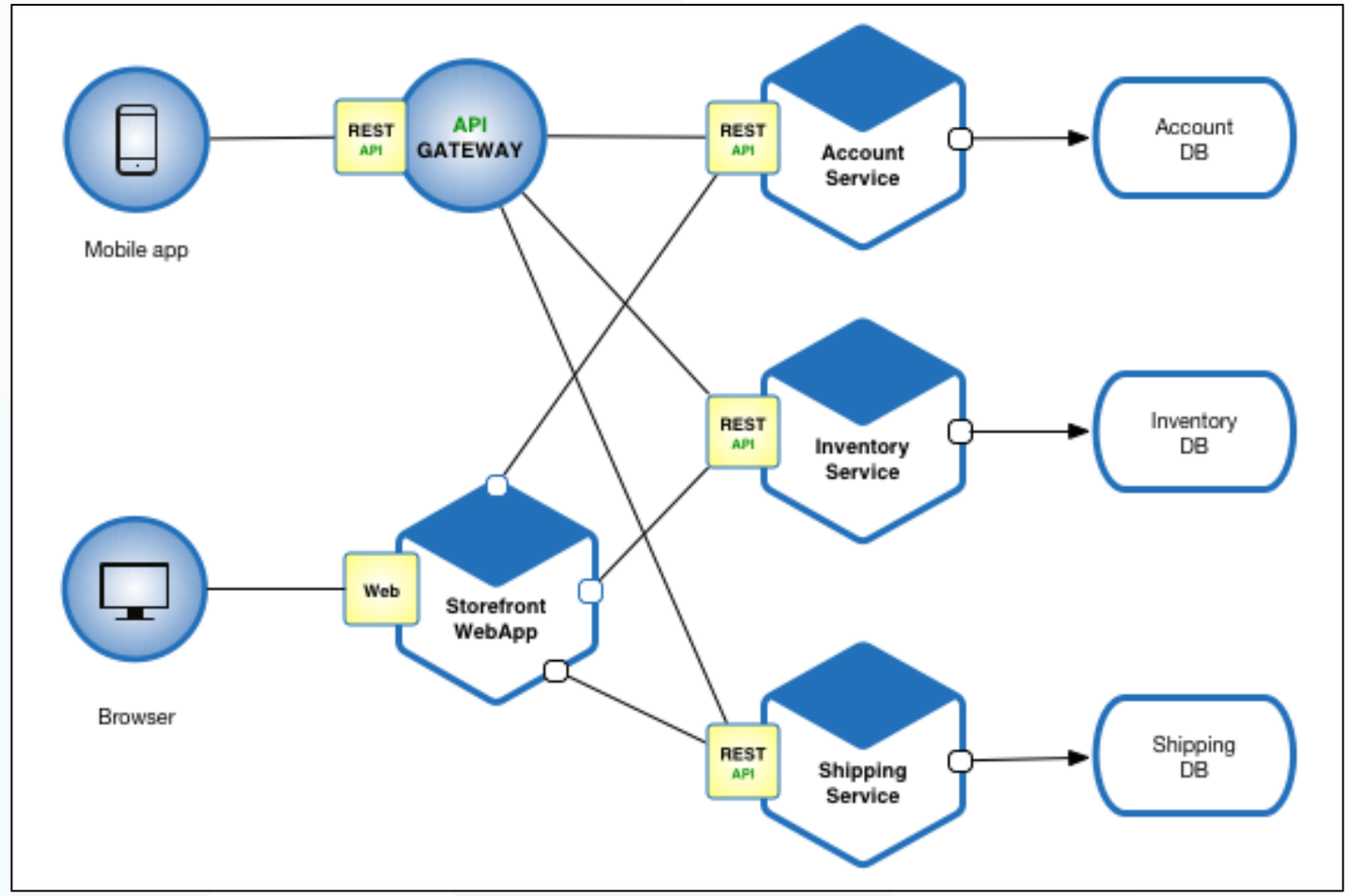
Diseño tolerante a fallos.

Automatización de la infraestructura.

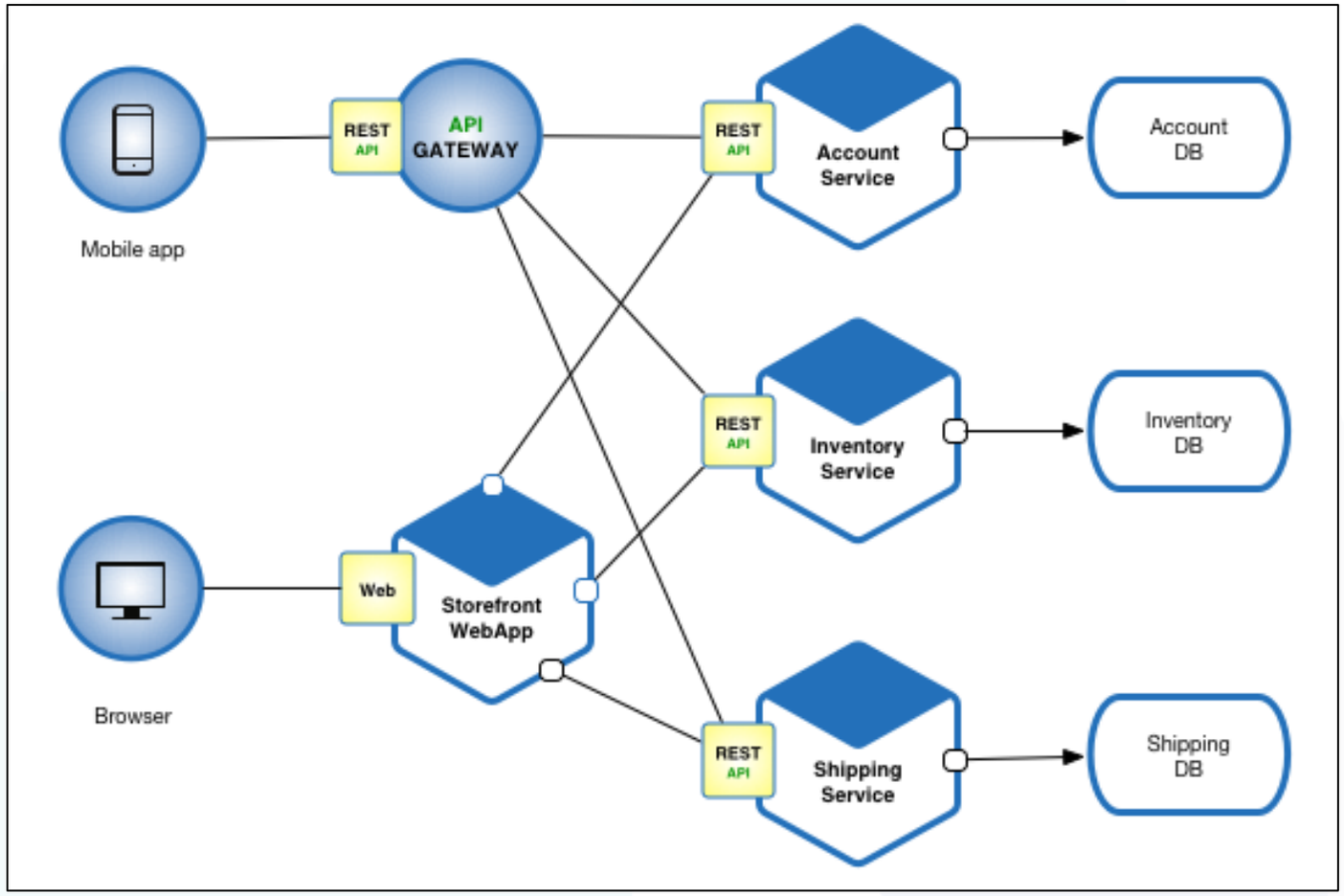
Diseño evolutivo.

https://es.wikipedia.org/wiki/Arquitectura_de_microservicios

Arquitectura de Microservicios

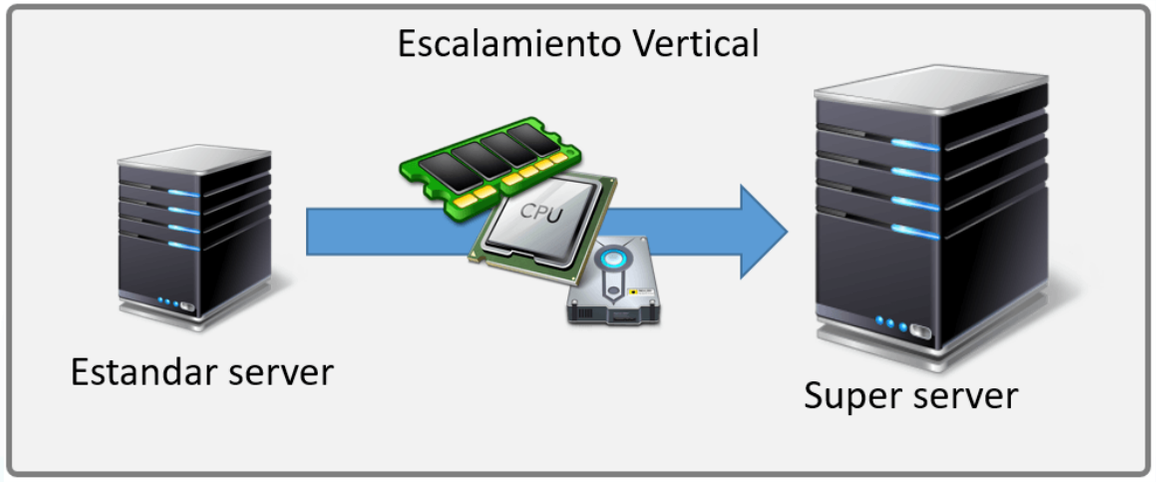


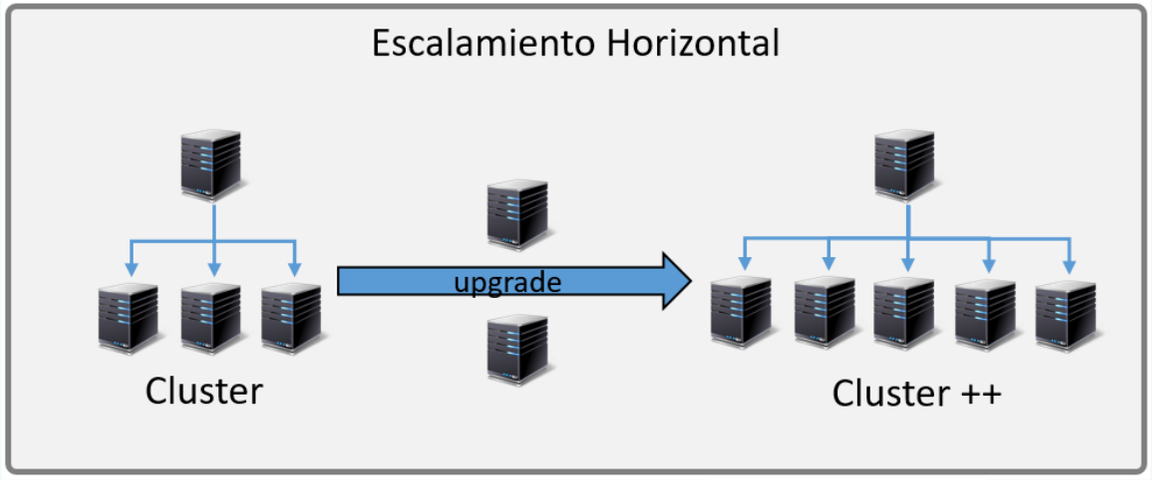
Escalamiento



- ✓ El **escalado** es una de las características más deseables en las aplicaciones y una de las principales preocupaciones para equipos de desarrollo y administración de servidores. Básicamente, se refiere a la **capacidad de crecimiento de la aplicación** para atender a un número cada vez mayor de solicitudes y usuarios con total normalidad y **sin degradaciones de servicio.**

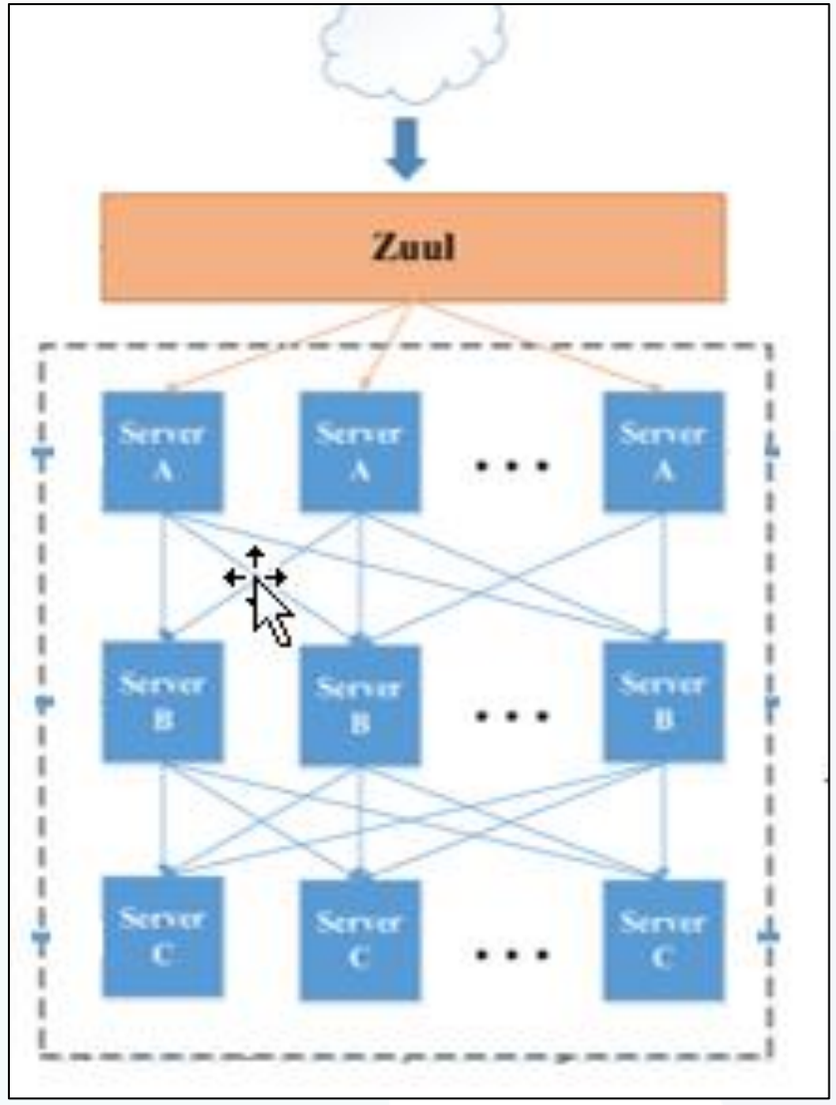
Escalamiento





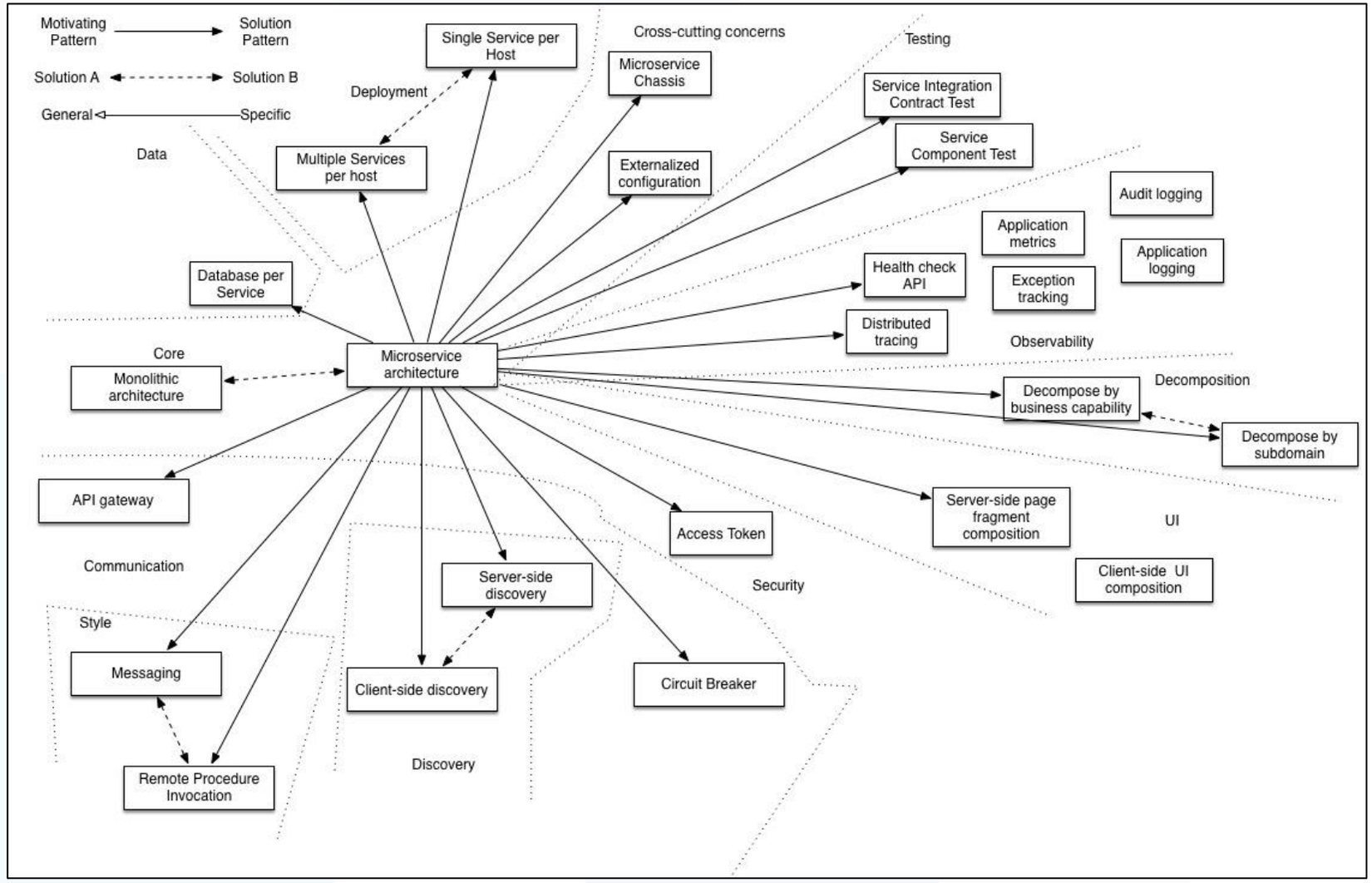
- ✓ El **escalado vertical** se consigue de una manera muy sencilla: aumentando los recursos del servidor. Principalmente, en lo que respecta a la capacidad de procesamiento, memoria y almacenamiento
- ✓ El **escalado horizontal** se consigue aumentando el número de servidores que atienden una aplicación. Para ello, un grupo de distintos servidores se configura para atender las peticiones de manera conjunta (es lo que se denomina Cluster) y la carga de trabajo se distribuye entre ellos a través de un balanceador. Cada uno de esos servidores se conoce como nodo y el escalado se realiza simplemente agregando un nuevo nodo al Cluster.

Patrones de Arquitectura de Microservicios



- ✓ ¿Cómo mantengo la configuración actualizada y centralizada para cada nodo?.
- ✓ ¿Cómo administro las trazas / logs por nodo?.
- ✓ ¿Cómo los servicios se conocerán entre ellos?
- ✓ ¿Cómo el cliente sabrá a que servicio conectarse?

Patrones de Arquitectura de Microservicios



https://microservices.io/patterns/microservices.html

Patrones de Arquitectura de Microservicios

Decomposition patterns

The Database per Service pattern

The API Gateway pattern

The Client-side Discovery and Server-side Discovery

The Messaging and Remote Procedure Invocation patterns

The Single Service per Host and Multiple Services per Host patterns

Cross-cutting concerns patterns: Microservice chassis pattern and Externalized configuration

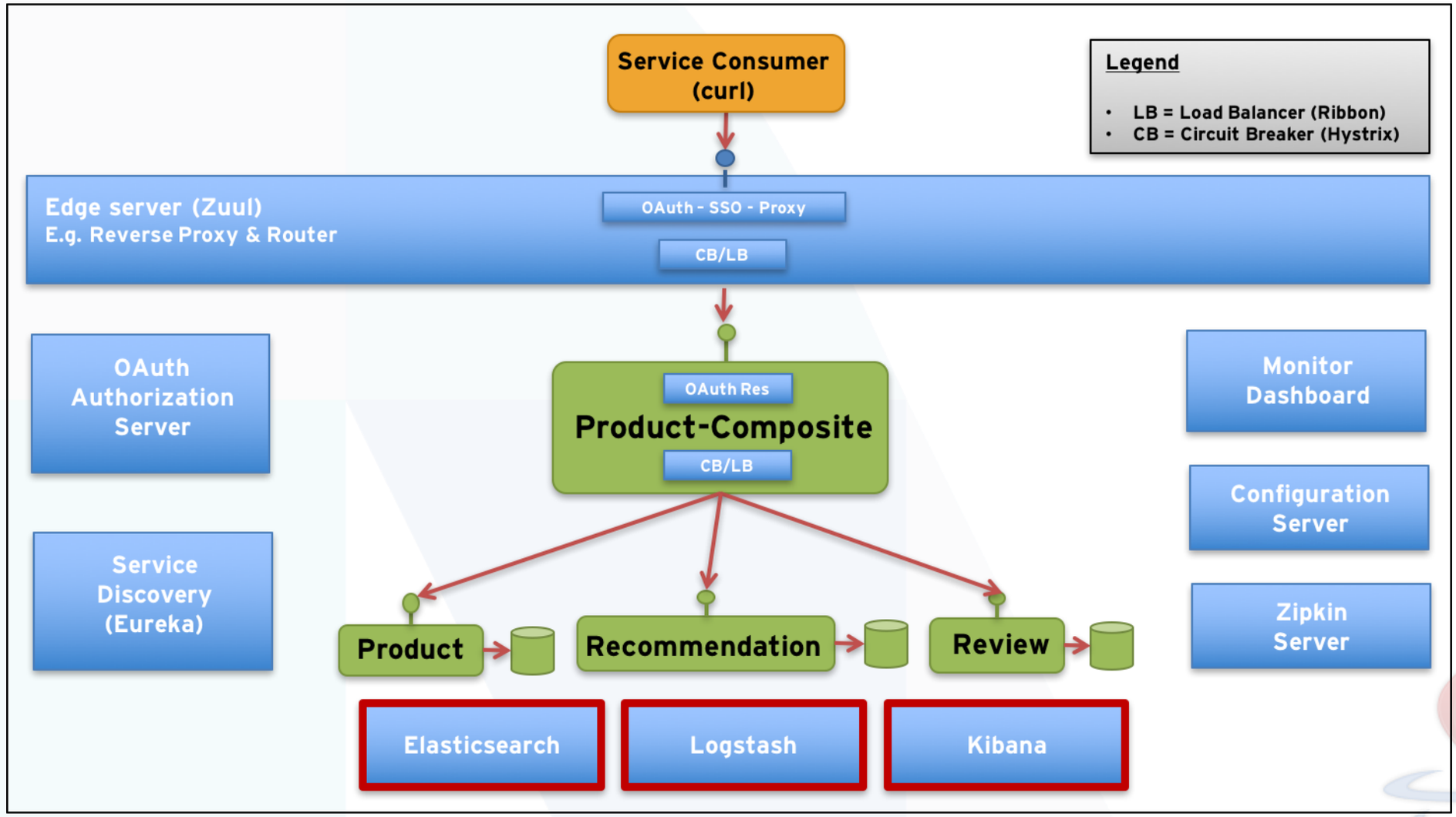
Testing patterns: Service Component Test and Service Integration Contract Test

Circuit Breaker

Access Token

https://microservices.io/patterns/microservices.html

Arquitectura de Microservicios



https://callistaenterprise.se/blogg/teknik/2017/09/13/building-microservices-part-8-logging-with-ELK/

NTT DATA

Gracias

aangulom@emeal.nttdata.com