Skill Up. Level Up

Day 30: Spring Framework Intro

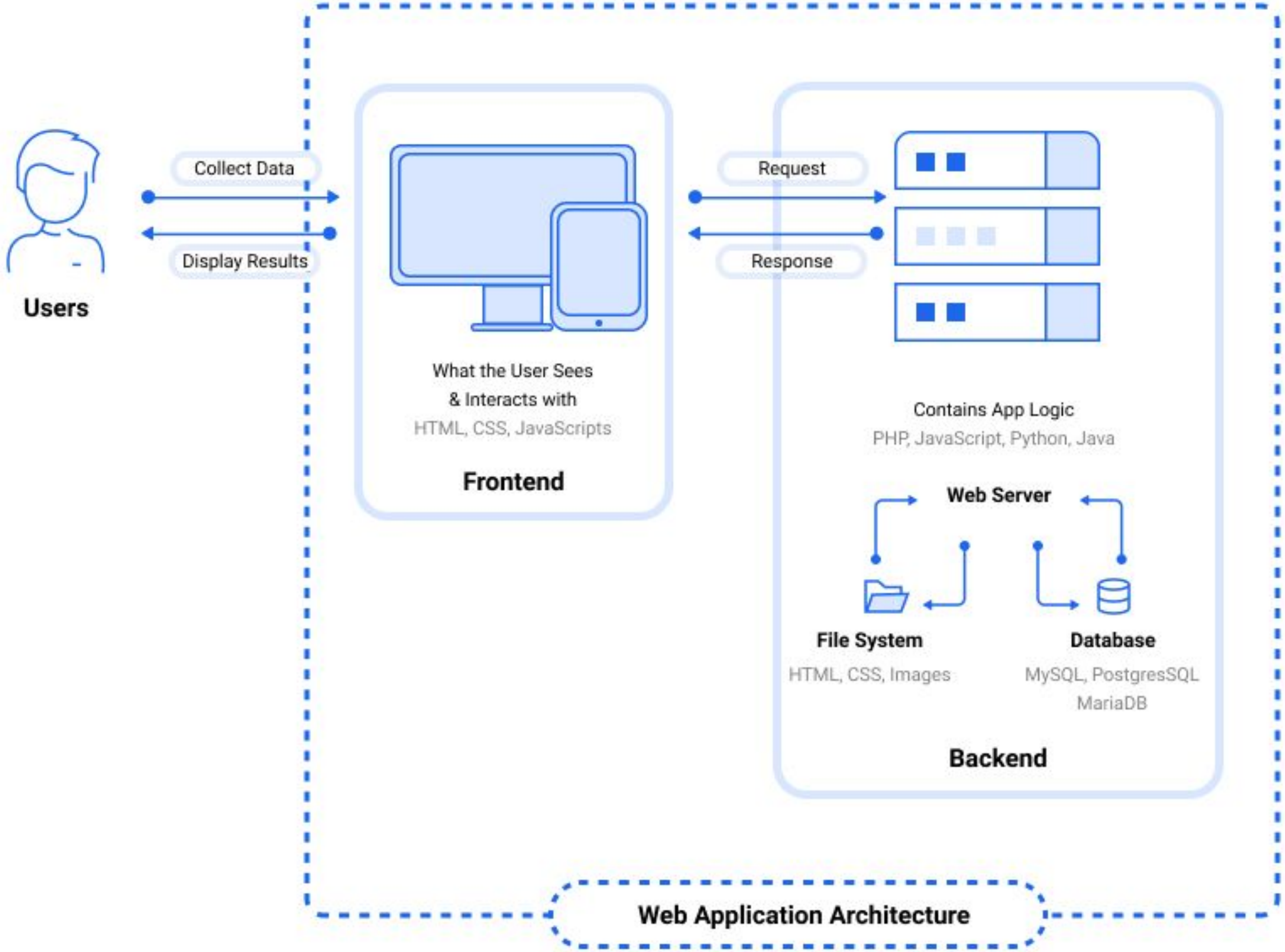
www.guvi.in

- Web application Software architecture-Single Tier,Two-Tier,
- Communcation Between Client -Server
- Monolithic & microservices
- What is Rest API?
- Intro to REST
- Intro to Java Containers & Servlets
- Introduction to Software Design Patterns

Web Application Architecture

Web Application Architecture

- Web application architecture is a mechanism that gives us a clarification that how the connection is established between the client and the server. It determines how the components in an application communicate with each other.

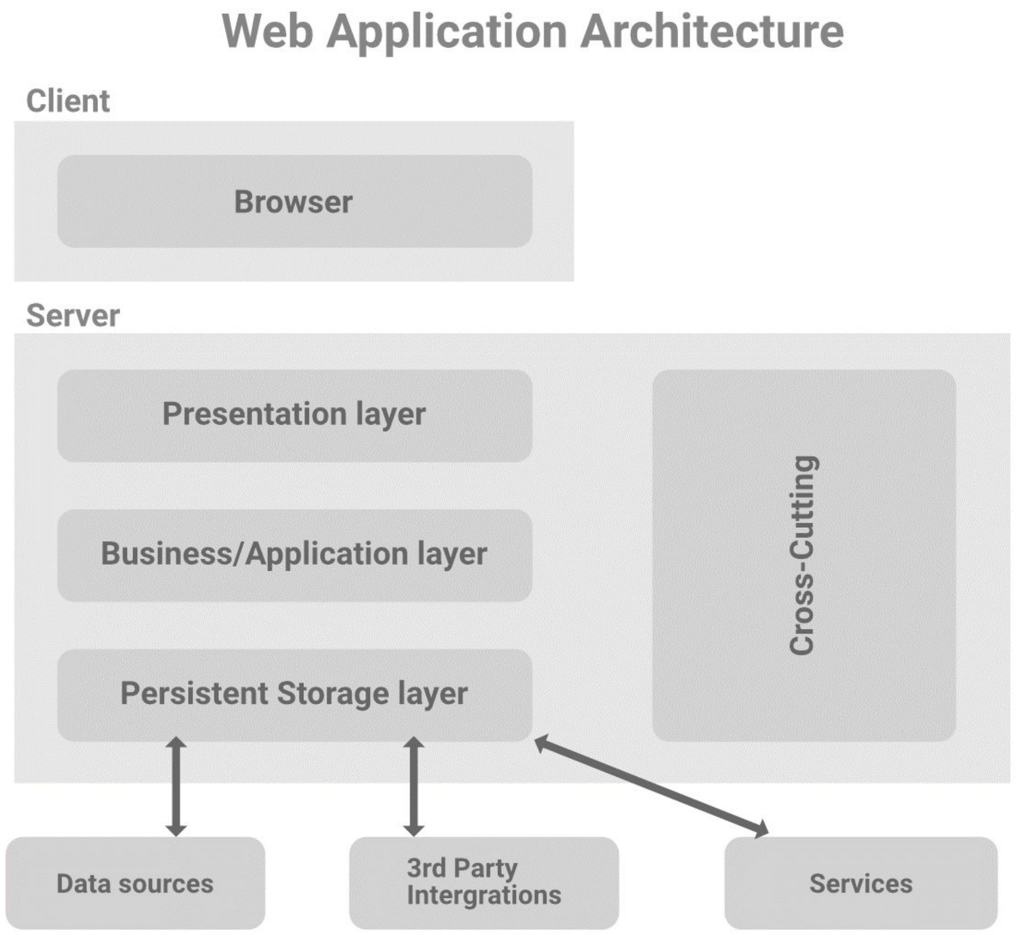

Users
Collect Data
Display Results
What the User Sees
& Interacts with
HTML, CSS, JavaScripts
Frontend
Request
Response
Contains App Logic
PHP, JavaScript, Python, Java
Web Server
File System
HTML, CSS, Images
Database
MySQL, PostgresSQL
MariaDB
Backend
Web Application Architecture

Web Application Three Tier Architecture Layers

- Web application architectural patterns are separated into many different layers or tiers which is called Multi- or Three-Tier Architecture. You can easily replace and upgrade each layer independently.

1. **Presentation Layer**
2. **Business Layer**
3. **Persistence Layer**

Architecture Layers



Types of Web Application Architecture

1. **Single Page Applications:** Today a lot of modern web applications are designed as single-page web applications that only include the most required elements and information.

2. **Microservices:** These are small and lightweight services that execute specific, single functionality. The components in the applications are not dependent on each other.

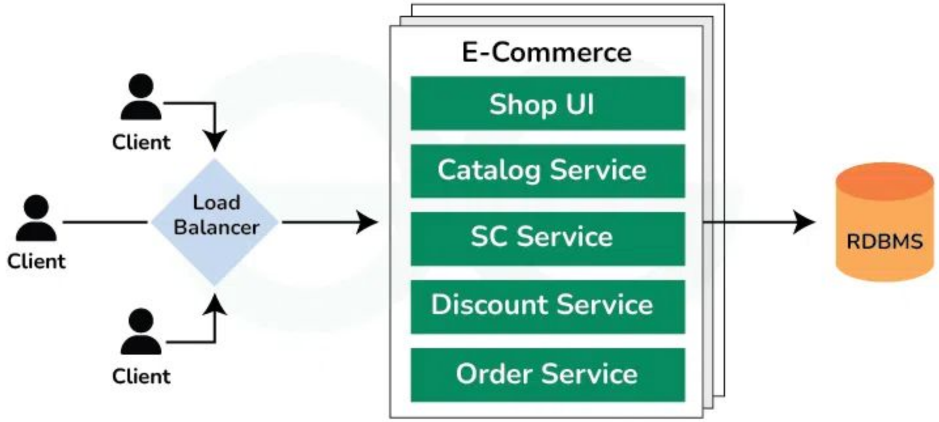
3. **Serverless Architectures:** In this approach, developers outsource the server and infrastructure management from a third-party cloud infrastructure services provider.

Monolithic & Microservices

Monolithic Architecture

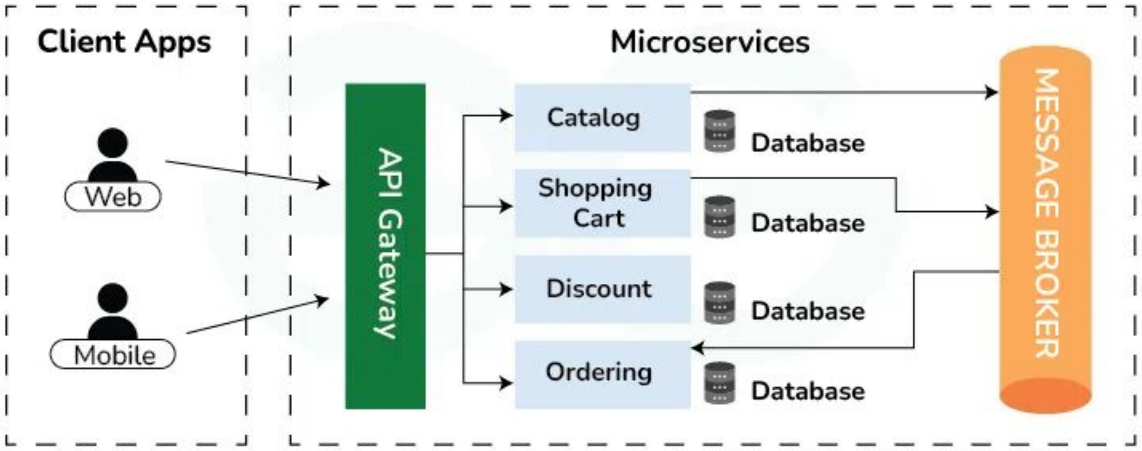
- A monolithic architecture is a traditional approach to designing software where an entire application is built as a single, indivisible unit.

- In this architecture, all the different components of the application, such as the user interface, business logic, and data access layer, are tightly integrated and deployed together.



Microservices Architecture

- In a microservices architecture, an application is built as a collection of small, independent services, each representing a specific business capability.
- These services are loosely coupled and communicate with each other over a network, often using lightweight protocols like HTTP or messaging queues.



Intro to REST, GraphQL

REST

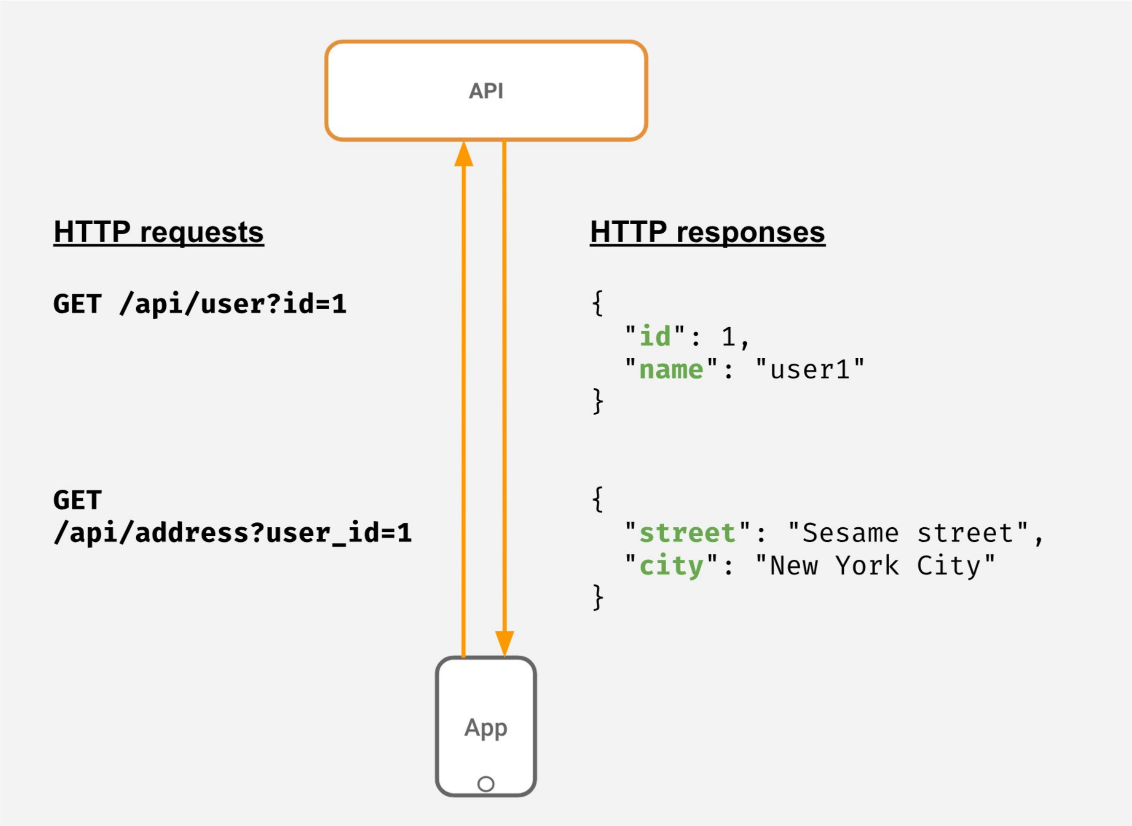
- REST is an acronym for ‘Representational State Transfer.’ It is a stateless and uniform format that decouples the client from the server.

- REST is initially fairly simple to implement and uses the HTTP transport layer to support creating, reading, updating, and deleting data (CRUD).

- Each operation in REST is characterized via an endpoint. For example, you may want to retrieve a user from your API, and you might have an endpoint like this:

  /api/users/1

REST API Calls



HTTP requests

```
GET /api/user?id=1
```

```
GET
/api/address?user_id=1
```

HTTP responses

```
{
  "id": 1,
  "name": "user1"
}
```

```
{
  "street": "Sesame street",
  "city": "New York City"
}
```

REST API calls

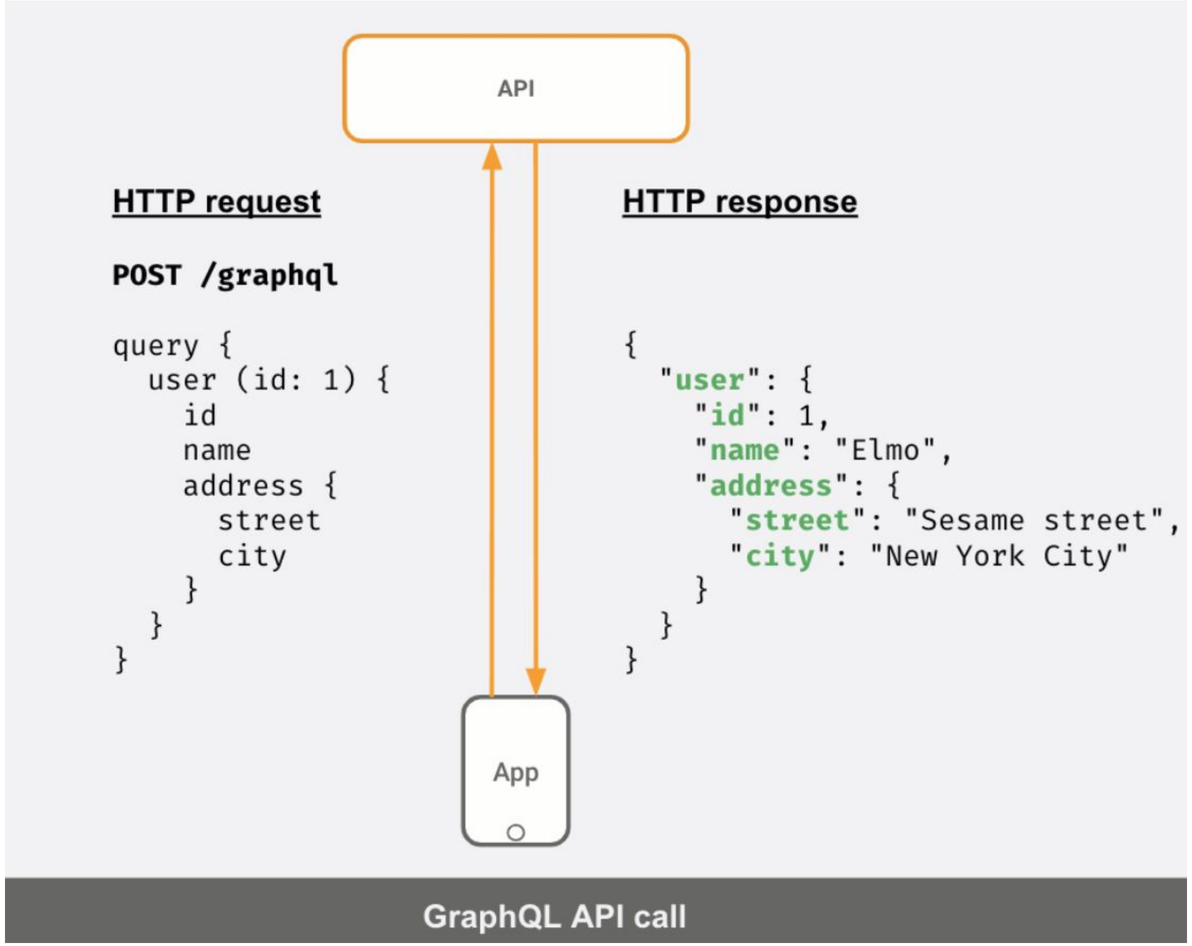
GraphQL

- GraphQL offers many advantages over REST, especially because it removes the multiple endpoints issue. Instead, graphQL has a single endpoint which allows you to declaratively tell the API what you want.

- Using our previous example of retrieving a user and their photos above, we would simply do the following in GraphQL:

```
query {
 user {
  name
  photos {
   id
   path
  }
 }
}
```

GraphQL Calls



HTTP request

```
POST /graphql

query {
  user (id: 1) {
    id
    name
    address {
      street
      city
    }
  }
}
```

HTTP response

```
{
  "user": {
    "id": 1,
    "name": "Elmo",
    "address": {
      "street": "Sesame street",
      "city": "New York City"
    }
  }
}
```

GraphQL API call

Intro to Java Containers & Servlets

What is Java Servlet?

- Java Servlets are the Java programs that run on the Java-enabled web server or application server.
- They are used to handle the request obtained from the web server, process the request, produce the response, and then send a response back to the web server.

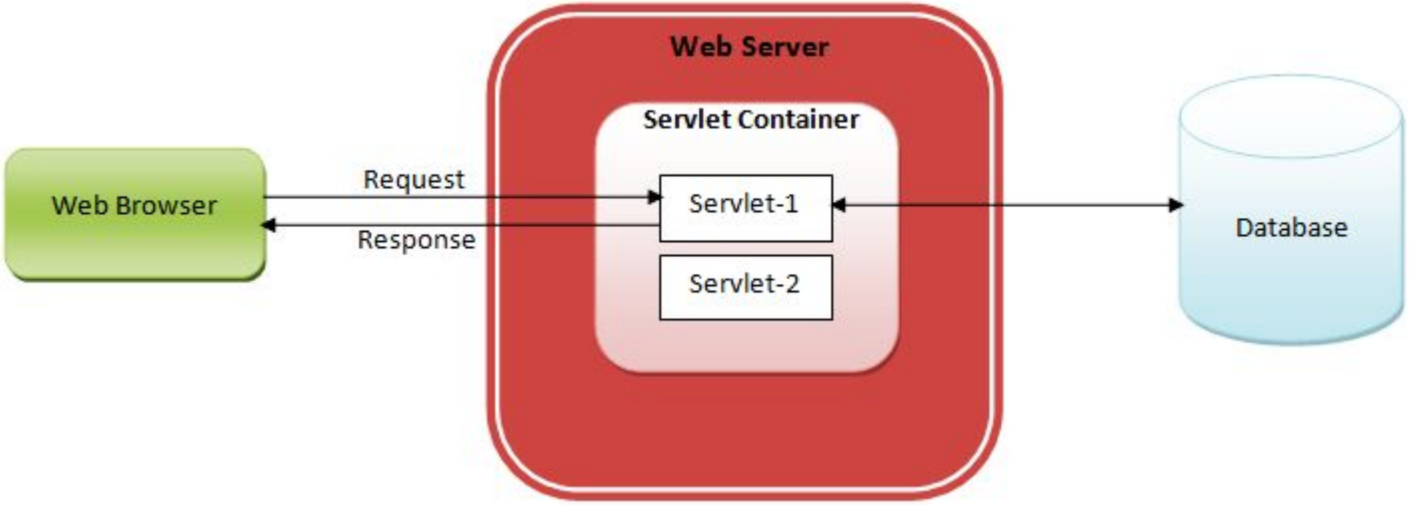
Properties of Java Servlet

The properties of Servlets are as follows:

- Servlets work on the server side.
- Servlets are capable of handling complex requests obtained from the web server.

Java Servlets Architecture

Servlet Architecture can be depicted from the image itself as provided below as follows:



Servlet Container

- Servlet container, also known as Servlet engine, is an integrated set of objects that provide a run time environment for Java Servlet components.
- In simple words, it is a system that manages Java Servlet components on top of the Web server to handle the Web client requests.

IOC & Dependency Injection

Inversion of Control (IOC)

- Spring IoC (Inversion of Control) Container is the core of Spring Framework.
- It creates the objects, configures and assembles their dependencies, manages their entire life cycle.
- The Container uses Dependency Injection(DI) to manage the components that make up the application.

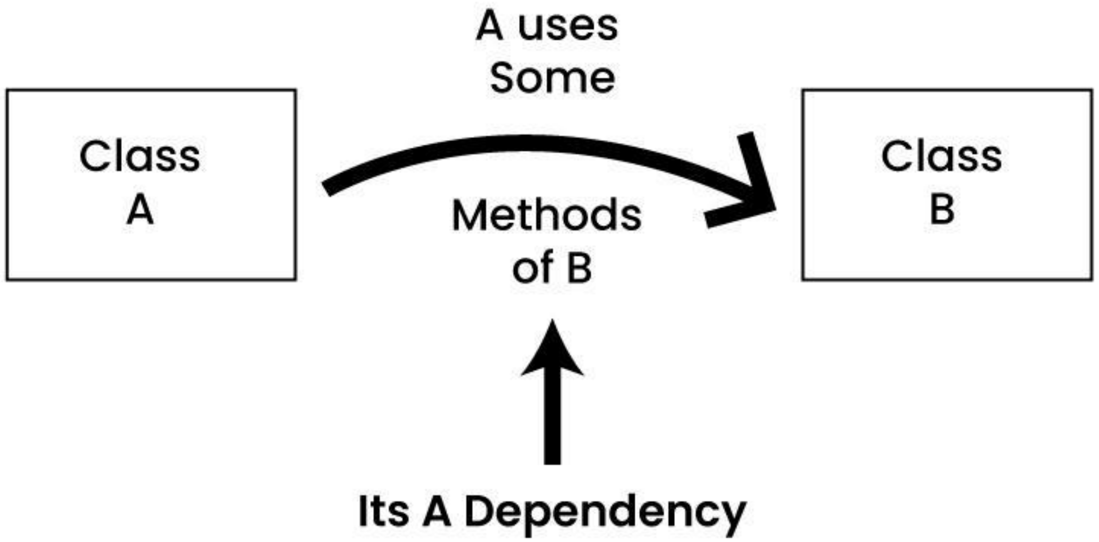
Features of Spring IoC

The followings are some of the main features of Spring IoC,

- Creating Object for us,
- Managing our objects,
- Helping our application to be configurable,
- Managing dependencies

Dependency Injection

In software design, dependency injection (DI) is a design pattern that aims to decouple objects from their dependencies. Instead of creating their own dependencies internally, objects receive them from an external source.



Class A

A uses Some Methods of B

Class B

Its A Dependency

Example – Dependency Injection

Imagine a coffee shop:

- The barista (object) class A needs coffee beans class B (dependency) to make coffee.
- Without dependency injection, the barista would have to grow, roast, and grind the beans themselves.
- This makes them tightly coupled to the bean-growing process, making it hard to change bean suppliers or use different roasts.

Spring Ecosystem

1. Field Injection

```
public class BusinessService {
    @Autowired
    private DataService dataService;
}
```

2. Constructor Injection

```
public class Employee {
 private String name; private String email;
 @Autowired  public Employee(String name, String email) {
 this.name = name; this.email = email;  }
}
```

3. Setter Injection

```
public class BusinessService {
    @Autowired
    public void setDataService(DataService dataService) {
     System.out.println("Setter injection");
     this.dataService = dataService;
    }
}
```

Tight Coupling

When two classes are tightly coupled, they are dependent on each other. If one class changes, the other class will also change.

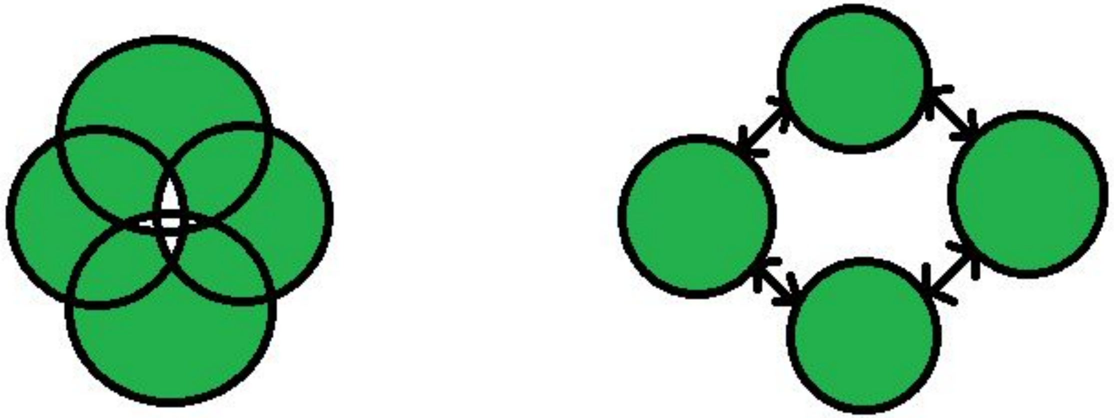
Example : If you want to change the skin, you would also have to change the design of your body as well because the two are joined together – they are tightly coupled. The best example of tight coupling is RMI(Remote Method Invocation).

Loose Coupling

When two classes are loosely coupled, they are not dependent on each other. If one class changes, the other class will not change.

Example : If you change your shirt, then you are not forced to change your body – when you can do that, then you have loose coupling. When you can't do that, then you have tight coupling. The examples of Loose coupling are Interface, Java Messaging Service.

Tight Coupling vs Loose Coupling



Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow

Loose coupling:

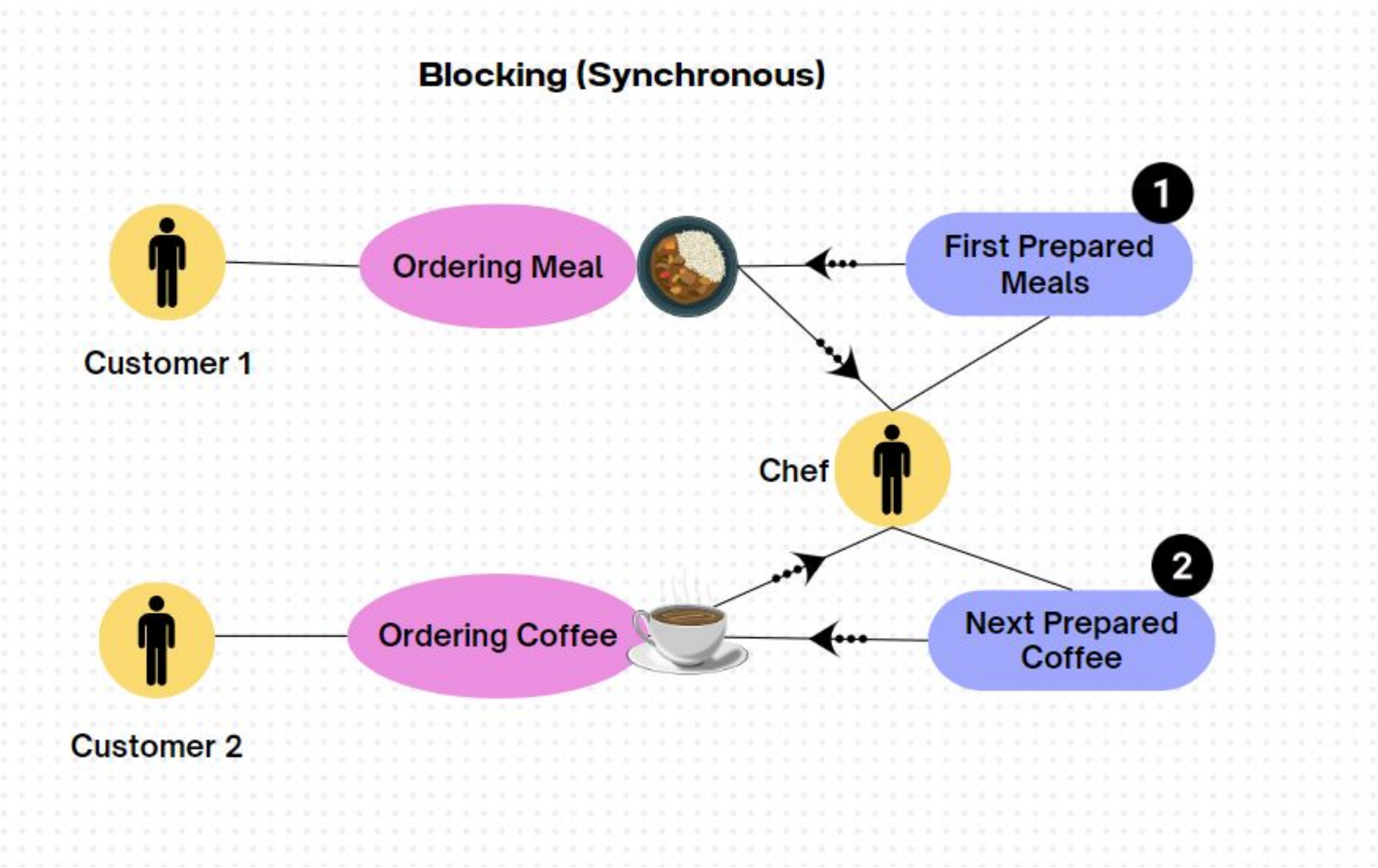
1. Less Interdependency
2. Less coordination
3. Less information flow

Blocking & Non-blocking web stacks

Blocking(synchronous) Processing

- Processing thread is waiting in case any I/O operation is performed
- CPU & RAM resources are wasted, while thread is waiting to the I/O results
- If all threads are waiting, new user requests are either put to the queue or dropped down. This leads to poor user experience
- If all threads are waiting, service becomes unresponsive for API clients. This leads to timeouts and API clients failure. Basically, failure leads to more failure.

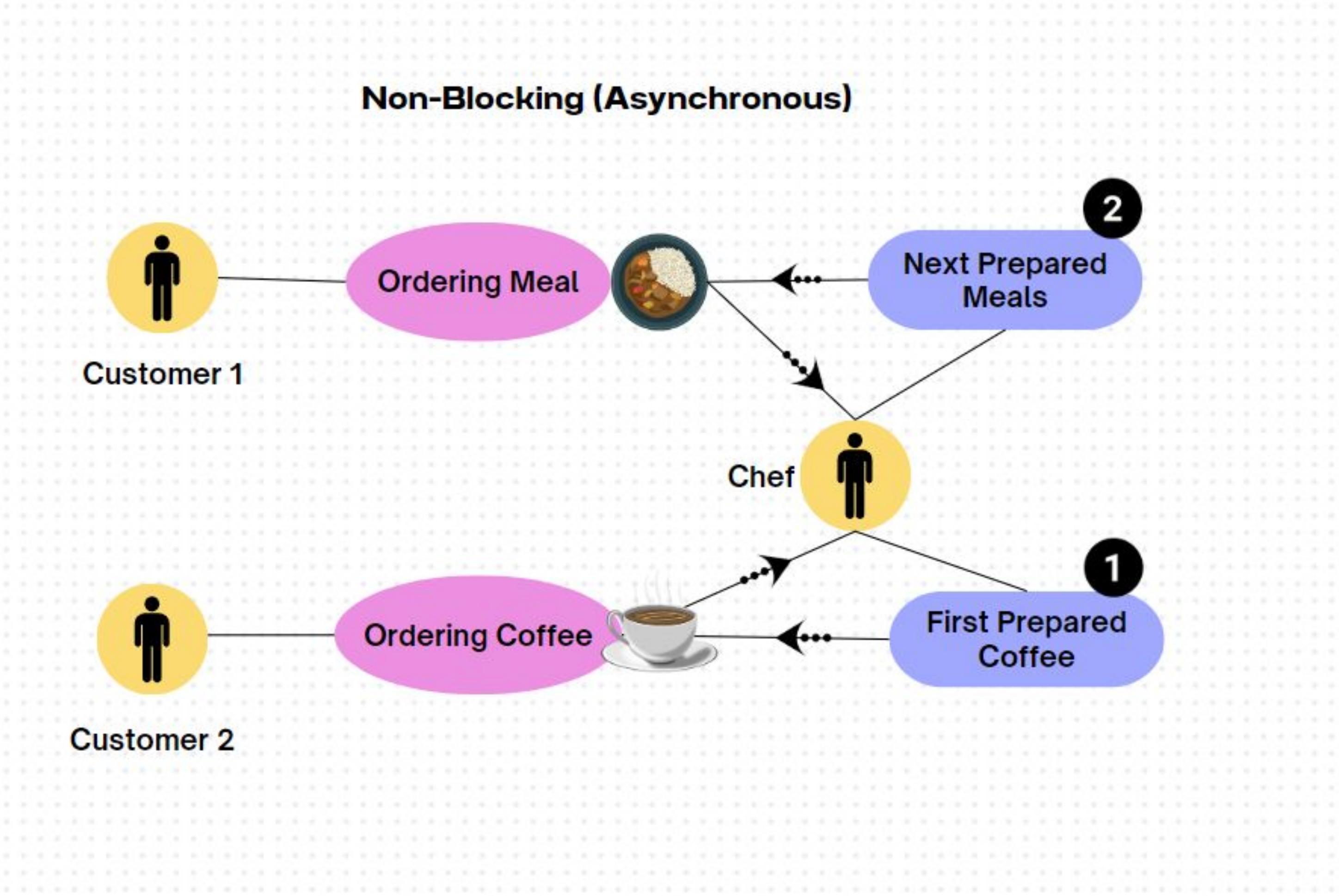
Blocking(synchronous) Processing



Non-Blocking(Reactive) Processing

- Not bound to specific processing thread
- Threads are not waiting in case I/O operation is performed
- Threads are reused between calls
- High CPU & RAM utilization
- Less threads are needed to serve same number of requests as in blocking case

Non-Blocking(Reactive) Processing



Thank You