1) **Java** = Google Guice, CDI, Dagger

   PHP = Symfony, Laravel, Drupal. Php

   C# = Castle Windsor

2) **@SpringBootApplication** annotation is used to mark the main class of a Spring Boot application. It also provides aliases to customize the attributes of **@EnableAutoConfiguration** and **@ComponentScan** includes annotations such as:

   @EnableAutoConfiguration = It tries to guess and configure beans that we likely to need based on our jar dependencies.

   @ComponentScan = It tells Spring to check configurations and components. It goes provided base package and gets dependencies required by **@Bean** or **@Autowired.**

   @Configuration = It indicates that a class declares one or more **@Bean** methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

3) **@Primary =** Gives priority to an indicated bean when there exist more than one beans that both can be auto wired at a time.

   @Qualifier = It helps us to choose the correct bean for dependency injection and avoid ambiguity when it finds multiple beans of the same type.

   Qualifier is more specific and has high priority. So, when both qualifier and primary are found, Primary will be ignored. During my research, I found a piece of code that explains relationship between these two annotations in terms of priority:

```java
@Configuration
public class Config {

    @Bean("bean1")
    @Primary
    public BeanInterface bean1() {
        return new Bean1();
    }

    @Bean("bean2")
    public BeanInterface bean2() {
        return new Bean2();
    }
}

@Service
public class BeanService {

    @Autowired
    @Qualifier("bean2")
    private BeanInterface bean;

    @PostConstruct
    public void test() {
        String name = bean.getName();
        System.out.println(name);
    }
}
```

The output is "bean2".

4) It says convention comes before configuration. It is like writing application settings in code rather than configuration files. Most of frameworks use convention over configuration approach. As long as we follow the convention, things happen quickly. It reduces developer flexibility but saves from workload.

For example, unless otherwise is stated, classes with annotations **@Entity** and **@Id** are based on the assumption that attribute names and class names match one-to-one tables and columns, which is called convention. But we can say **@Column**(name=”example”) if we use another name.

5) The main purpose of Aspect Oriented Programming is to divide the code into smaller parts and make it reusable and easy to understand. It complements to OOP. It is easy to

configure. Logging and validation is useful to make our code understandable. Some of the disadvantages of AOP is: hard to debug, poor toolchain support and maintenance.

6) Solid principles are for ensure that our software is flexible, reusable, maintainable, understandable, and also prevents code duplication. We should stick to them as much as possible in order not to face with problems as the project grows.

Single Responsibility: One class should only have one job to do.

Open-Closed Principle: We should be able to extend a class without changing its behavior. Following this principle is essential for writing code that is easy to maintain and revise. If our class fits with that principle it becomes extendable, meaning that the behavior of the class can be extended. And it is closed for modification, meaning the source code is set and cannot be changed.

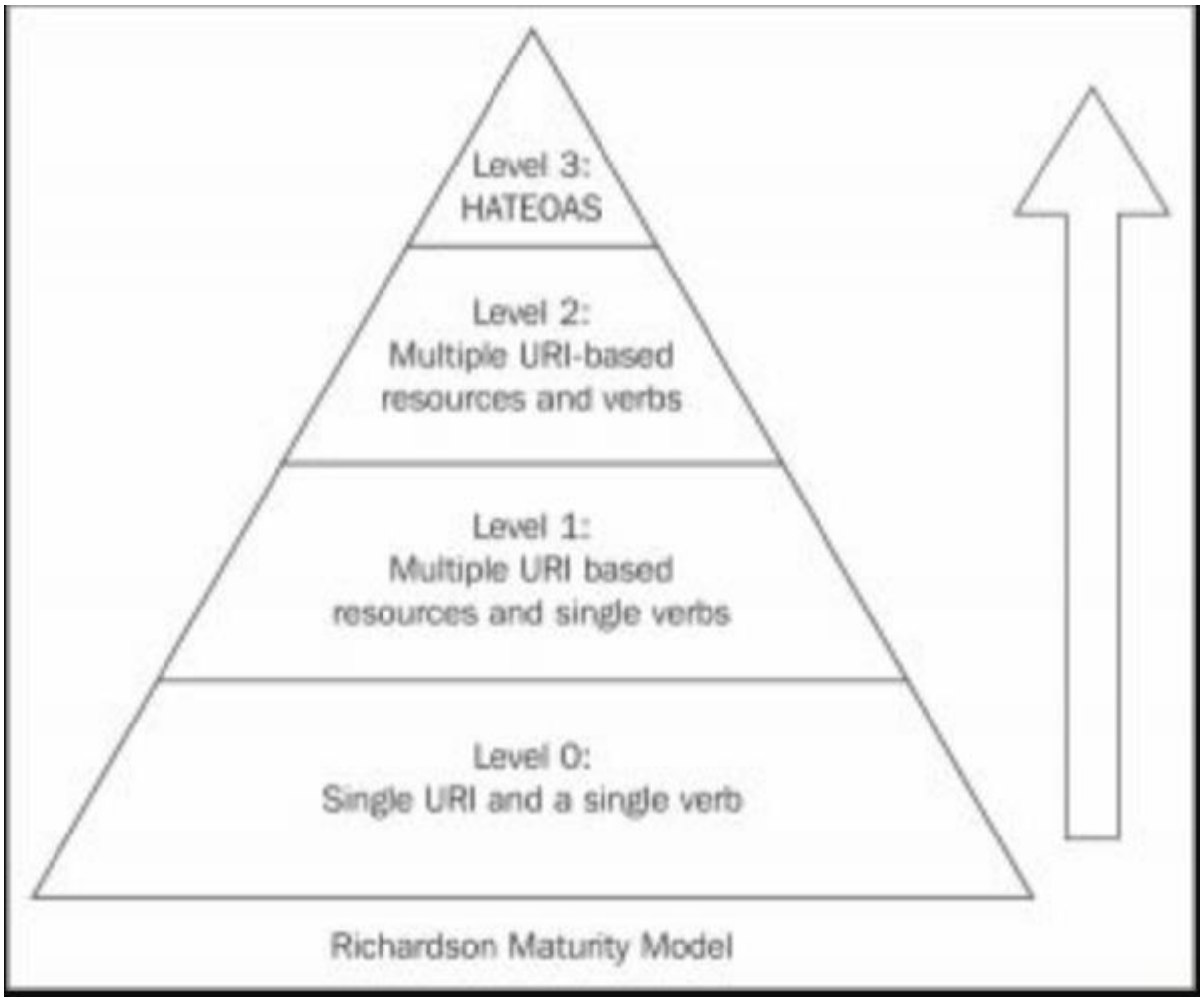
Liskov Substitution Principle: Every derived class should be substitutable for its parent class.

Interface Segregation Principle: We should have more smaller interfaces rather than a few big one.

Dependency Inversion Principle: High level modules(or classes) should not depend upon low level modules or low level classes. Both of them should depend on abstractions.

7) For other developers can use our API, our API should be documented in order to know the endpoints and operations supported by that endpoints, what authentication methods to use etc. That's why it is important to document the our API in order to make our API be used. Swagger is the way to document API specifying that what exactly API can do. It is mainly designed for REST API.

8) It was name after Leonard Richardson. He basically divided web service designs into four categories and used three factors to decide maturity level of a service which are URI, HTTP methods and Hypermedia.



Level 0 = The services at 0 maturity level have a single URI and use a single HTTP method.

Level 1 = The services at maturity level 1 use many URIs but only a single HTTP verb.

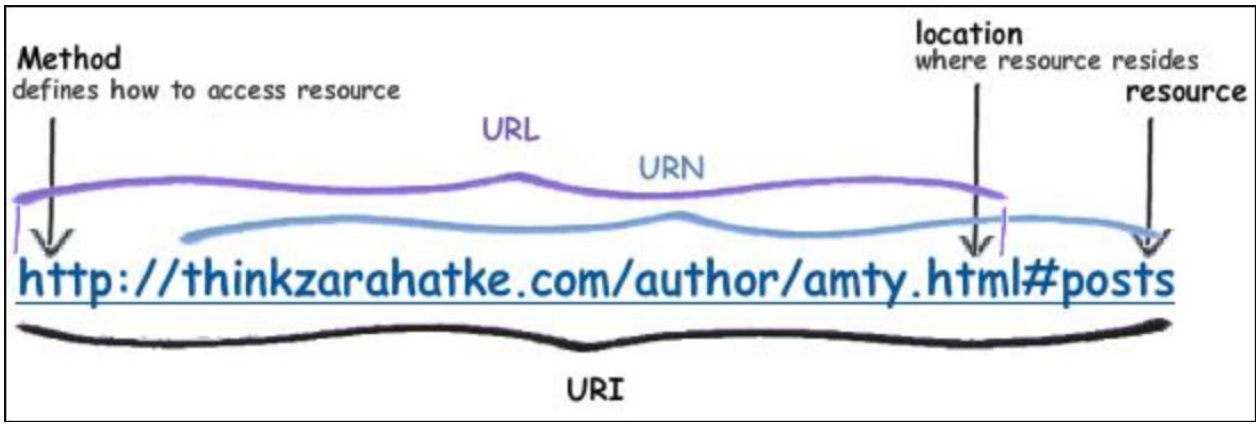
Level 2 = The level 2 services generally host addressable resources.

Level 3 = The level 3 makes it easy for the responses to be self descriptive by using HATEOAS.

9) URI (Uniform Resource Identifier) = Identifies a resource.

URL (Uniform Resource Locator) = Subset of the URIs that include a network location

URN (Uniform Resource Name) = Subset of URIs that include a name within a given space, but no location



10) In terms of a RESTful service, for an operation to be idempotence, clients should be able to repeatedly multiple identical requests while it is producing the same result with single request.

DELETE (in first request user gets **204**, but after that he/she gets **404** because content has already deleted. But the result is the same anyway), **GET**, **HEAD**, **OPTIONS**, **PUT** and **TRACE** methods are idempotent.

11) RFC is document with standard numbers used to identify TCP/IP. HTTP/1.0 is published in 1945. Specification of the protocol referred to as HTTP/1.1 is in RFC 2068.