**Immutable:**

An object is considered immutable if its state can't be modified in any way after the object is constructed. (In this case, the object's state means the fields or variables it holds.)

An immutable object never exposes its state to the outside world and it doesn't provide any sort of behavior that could modify its state. All wrapper classes (such as `Integer`, `Float`, and `Long`) are immutable by nature. Other immutable classes include `String`, `java.util.UUID`, and `java.net.URL`.

public final class ImmutableCat {

private final String name;

private final Owner owner;

public ImmutableCat(String name, Owner owner) {

this.name = name;

this.owner = owner;

}

public String getName() {

return name;

}

public Owner getOwner() {

return owner;

}

public ImmutableCat changeOwner(Owner newOwner) {

return new ImmutableCat(this.name, newOwner);

}

}

**What is REST?**

REST (Representational State Transfer) was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. REST is an architectural style for designing distributed systems. It is not a standard, but rather a set of constraints for applications, such as being stateless, having a client/server relationship, and offering a uniform interface. These constraints encourage good performance and scalability. While REST is not restricted to HTTP, it's the protocol most commonly associated with REST.

### Principles of REST

To be RESTful, applications should:

\* Expose resources as URIs that provide an easily-understood directory structure.

\* Send messages explicitly using HTTP methods (such as GET, POST, PUT, and DELETE).

\* Transfer representations of data objects using standard formats such as JSON or XML.

\* Keep interactions stateless by not storing client context on the server between requests. (This is because state dependencies limit and restrict scalability. Instead, the client should hold session state.)

**## Designing RESTful APIs**

Now that you've gotten some experience with REST, here are some important things to keep in mind when you're designing RESTful APIs.

\*\*Consider the API consumer\*\*

Make sure to consider the client who will use the service. What are their needs? Will the service URI make sense to them? Will the response format make sense as well?

\*\*Think in nouns (resources) and not in verbs (actions)\*\*

URIs should represent resources. In addition, URIs should be hierarchical and as self-descriptive as possible. Finally, it's good practice to use plurals for URIs, so \*\*/resources\*\* rather than \*\*/resource\*\*.

\*\*Always use HTTP Methods\*\*

\* \*\*GET\*\* - Does not update anything. Should be idempotent (same result from multiple calls). \_Possible return codes: 200 (OK), 404 (NOT FOUND), 400 (BAD REQUEST)\_

\* \*\*POST\*\* - Create a new resource. Ideally, should return JSON with a link to the newly created resource. \_Possible return codes: 200 (OK), 404 (NOT FOUND), 400 (BAD REQUEST), 201 (CREATED)\_

\* \*\*PUT\*\* - Update a known resource, for example, update client details. Possible return codes: \_200 (OK)\_

\* \*\*DELETE\*\* - Delete a resource.

**Types of Spring Dependency Injection**

Spring can inject beans into your class in various ways:

Field Based Injection - The @Autowire annotation is placed on the variable you want to be injected. Spring sets the value directly into the variable.

Constructor based Injection - The @Autowire annotation is placed above the constructor, and Spring sends the value to the constructor.

Setter based Injection - The @Autowire annotation is placed above the setter of the variable to inject, and Spring sends the value to the setter.

In OrdersController, you used constructor based injection, because Spring sends the order service instance that it created into the orderService variable of the OrdersController class.

Constructor based injection is the recommended option for several reasons:

It allows Spring to use the same mechanism that other clients (like tests) would use to create your class

The dependencies are clearly identified. There is no way to forget one when instantiating the object not through Spring

The dependencies can be final, and the objects can be made immutable.

**The @Component Annotation**

Another way to let Spring know which instance to create is to use the @Component annotation. This annotation is placed directly on the class we want Spring to instantiate and inject.

Here is how you would have created the orderService using the @Component annotation, instead of using the @Bean annotation

//...

import org.springframework.stereotype.Component;

@Component

public class OrderService {

private final List<Order> orders = new ArrayList<>();

//... rest of class

}

Differences between the @Bean and @Component

@Component is placed on the class, @Bean on the function

@Bean requires a function in a @Configuration class

@Bean lets you control how the instance is created and make changes before Spring injects it

With @Bean and bean names you can create several instances of the same class

You can only use @Component for classes where you can modify the class source code..

**Spring Profiles**

Spring profiles let you set up different configurations for different use cases. For example, you can set up one configuration for testing and one for production, or one configuration for one of your customers, and a different configuration for another customer.

Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any @Component or @Configuration can be marked with @Profile to limit when it is loaded

You mark a @Configuration class as belonging to a specific profile by adding a @Profile annotation on it, with the name of the profile you want for that configuration.

When you run your Spring application, you need to specify which @Configuration to use, by specifying which profile is active. You can do this by adding an active profile property to the application.properties file.

Component annotations

As you recall, Spring can instantiate objects for you, and inject them into your code. This makes your code more configurable, and also allows Spring to provide extra services such as lazy instantiation.

To mark a class as suitable for injection or auto-wiring we need to mark it as a Spring Bean. Spring Beans are classes whose instances are created by the Spring framework and are under its control.

You saw two different ways of marking a class as a Spring Bean:

Mark the class with @Component

Put a @Bean function that creates the object in a @Configuration class

Spring provides various specializations of the @Component annotation:

@Service - this is the same as @Component, but is used on service layer classes mainly to indicate that the class contains business logic

@Controller - A component in the web handling layer

@RestController - A web controller that is used for REST API calls, rather than serve HTML pages

@Repository - A data access component in the persistence layer.

These annotations can often be used interchangeably with only minor differences in the results. The main reason they are used is for readability, to indicate the layer and purpose of the class that they annotate. If a reader looks at code marked with @Service, they know that it is part of the business logic.

However, there are some differences in behavior between an annotation and its specializations. For example:

@Repository is equivalent to @Component, but adds the ability to catch proprietary database SQL exceptions and rethrow them as standard Spring exceptions.

@RestController is simply a @Controller but with @ResponseBody added to every function in the class. @ResponseBody causes the result of the function to be serialized into JSON before being passed back into the HttpResponse object. @RestController classes are used for handling REST web service calls, instead of handling HTML UI.

**Scaling:**

The process of improving applications to better handle heavy loads and large amounts of data is called scaling. Scalability is the capability of the system to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

Horizontal and Vertical Scaling

There are two different ways a system can grow.

In vertical scaling the size of existing resources is increased. For example, you upgrade the memory from 4GB to 8GB, upgrade the disk size from 1TB to 4TB, increase the speed of the processor, or go from using 4 CPU cores to 8 CPU cores.

In horizontal scaling you add more of the same resources you already have but in different locations, and split the usage between them. For example you run the same service on two different machines and send each one half of the requests, or add another hard drive and store half of the data on each drive.

Here is a definition from Wikipedia:

To scale horizontally (or scale out/in) means to add more nodes to (or remove nodes from) a system, such as adding a new computer to a distributed software application

To scale vertically (or scale up/down) means to add resources to (or remove resources from) a single node in a system, typically involving the addition of CPUs or memory to a single computer.

Each method has its own advantages and disadvantages. Vertical scaling is simpler, horizontal scaling usually requires distributed computing and distributed data leading to complex systems. However, vertical scaling is usually limited. There is only so much memory you can add before you get to the maximum possible for a machine, or get to the fastest CPU available today. These top end components are also very expensive. Horizontal scaling tends to be less limited, as you can always add another node.

**Spring Cloud:**

Spring Cloud is a Spring framework for simplifying building Java microservices applications by providing commonly needed infrastructure.

such as Discovery, Circuit Breaking, Load Balancing and others.

Ribbon for load balancing, Eureka for service discovery, and Hystrix for fault tolerance. All of these components were available as open source libraries.

In 2015, a community effort copied these libraries and unified them under the Spring Cloud namespace. In some projects, developers can choose between using the Netflix or the Spring Cloud library with minor differences.

**Load Balancing**

A load balancer splits the work between several components providing the same service to improve service performance. Generally, it tries to split the work evenly, so that each instance gets an equal share of the work.

Client-Side and Server-Side Load Balancing

In server-side load balancing the load balancer is its own stand-alone component, which receives all requests from the clients, and decides where to send each request.

**Server Side Load Balancing**

In client-side load balancing it is the client's responsibility to choose which of the instances that provide the service it will call with each request. The load balancer runs as an embedded component within the client. With client-side load balancing each client keeps a list of all available end points providing the same service, and uses some algorithm such as round-robin to decide which one to call each time.

**Client Side Load Balancing**

According to Netflix (creators of ribbon):

Ribbon is a client side IPC library that is battle-tested in cloud. It provides the following features

Load balancing

Fault tolerance

Multiple protocol (HTTP, TCP, UDP) support in an asynchronous and reactive model

Caching and batching