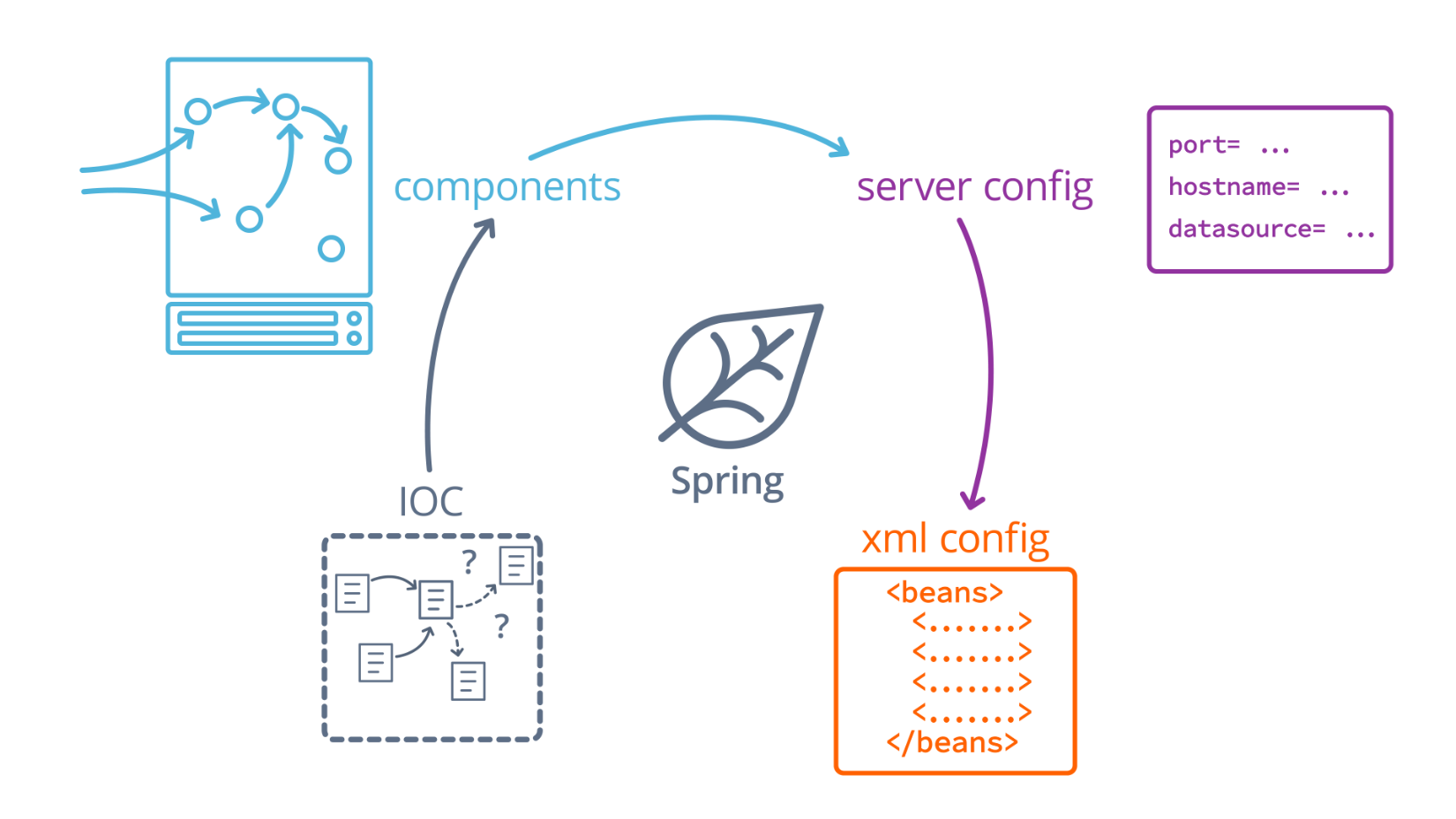
## Spring Boot Basics for Web Development

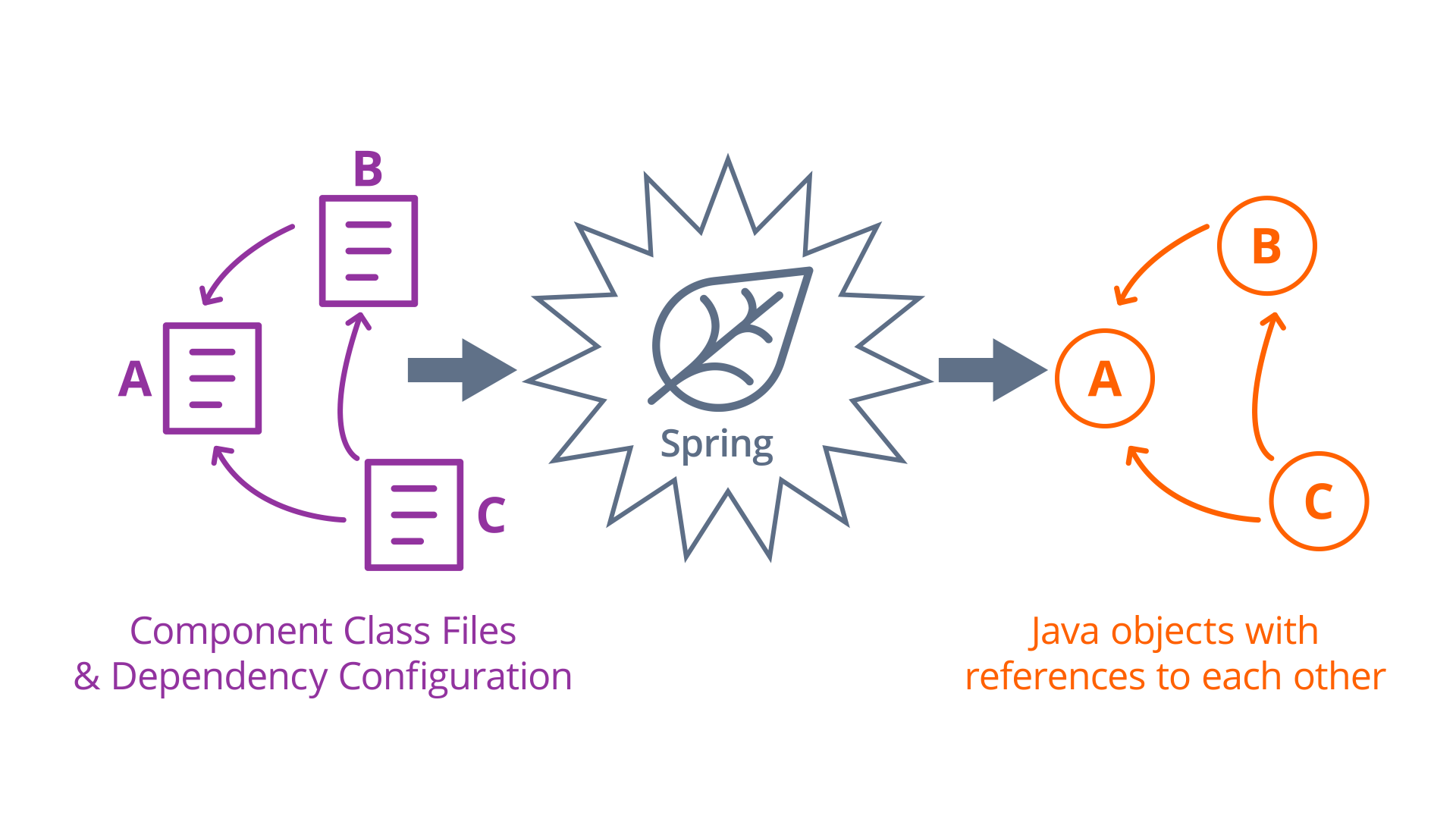


**Spring Boot's Building Blocks**

#### Lesson Outline

* **Spring IoC Configuration**: We discuss the basic mechanism at the core of Spring, Inversion of Control, and how to configure it for our applications.
* **Spring Components and Services**: We introduce a mental model for developing with Spring, built around classes that we define called components and services.
* **Server-Wide Configuration**: We learn how to configure certain server-wide properties of a Spring Boot application.
* **XML-Based Configuration**: We discuss an older form of Spring configuration, still in use, that uses XML files to define dependencies, server-wide options, and more.

# **The Spring Framework**



**The Spring Transformation**

The figure above shows that Spring takes the component class files and dependency configuration from the developer and instantiates a system of Java objects with references to each other.

Of all the tools we'll be using in this course, Spring is the most important because it defines our entire style of application development. Spring is a framework for Inversion of Control, which means that to use it, we have to package our code into individual component classes, telling Spring which components need each other to function. Spring takes the component classes we define and the dependencies we define between them and instantiates a system of matching Java objects with references to each other. This frees us from having to write so-called "glue code" to instantiate, connect, and manage components manually, and allows us to instead focus on writing so-called business logic, or code that concerns itself exclusively on the conceptual model of the application.

### **Key Terms**

* **Inversion of Control (IoC)**: A design pattern in which the developer creates independent application components and uses a framework to connect them, rather than writing the integration code themselves
* **Business Logic**: Code that relates exclusively to an application's conceptual model and featureset. Contrast with utility code like database access, HTTP request parsing, etc.
* **Persistent Components**: Java Objects that Spring maintains over time; they're persistent because they're created and maintained for as long as the application needs them to be.
* **Dependency Injection**: A mechanism by which IoC may be implemented. Components are configured with dependencies on other components, which are injected at runtime. Injection is quite literal - a component's dependencies are usually expressed as annotated fields on the component class, and Spring will populate those fields with the dependencies at runtime.

# **What is Spring**

When designing Spring applications, the most important principle to keep in mind is separation of concerns. What that really means is that for every problem your application has to solve has a home in a component class that is easy to find and flexible to use. By building your application out of small but focused components, you'll make the most of Spring's boilerplate-crushing power, and when it's time to add new features, you'll know exactly where to put them. On a larger team, this means greater coordination and less time spent trying to find and eliminate redundancies and conflicts.

But in order to allow your components to communicate effectively with one another, you'll need a system of data types to share between them. These are simple classes designed to store structured data, and little else. They're useful as conceptual representations of application data, like user profiles, or shipment invoice details, but made to be used and manipulated by the components of an application. A good rule of thumb to decide which is which is the new keyword test. A component should never be manually instantiated with the new keyword - components are managed by Spring, which means we need to rely on dependency injection to get a component reference. Data types, on the other hand, are no more special than Java's collections, and we can manually instantiate them with the new keyword whenever we'd like. Of course, we can use Spring (and its related libraries) to instantiate them for us as well, and in the coming lessons we sometimes will.

### **Key Terms:**

* **Separation of Concerns**: A code organization philosophy that emphasizes single-purpose components. In Java and Spring, this means keeping all methods related to a specific problem domain in the same class, for the sake of maintainability and reducing code reuse.
* **Data Types**: Sometimes called POJOs (plain-old-java-objects), Data Types are classes in application designed purely to hold structured application data, like users, profiles, or anything else an application might manage. These objects are helpful for us to maintain a good conceptual model of an application, and are created and accessed frequently during execution.
* **Components**: Components are persistent class instances managed by Spring in an application. They usually resemble libraries more than typical objects, with methods that must be called to perform specific actions. Components are only created when Spring is configured to create them, usually at server startup.

# **Spring Boot IoC Configuration**

Under the hood, Spring is just a Java application itself - and it responds to our configuration in a predictable way. When a Spring application starts, it scans your code base for specially-marked class files and configuration options. It uses that information to instantiate your application components as Java objects, and it stores them in a special data structure called the **application context**. This context is ultimately very similar to a Map or a python dictionary, and it can be queried at runtime to find specific components when needed. This is a closed system, so components instantiated outside of Spring won't automatically be injected with dependencies like those instantiated by Spring. Mind the new keyword!

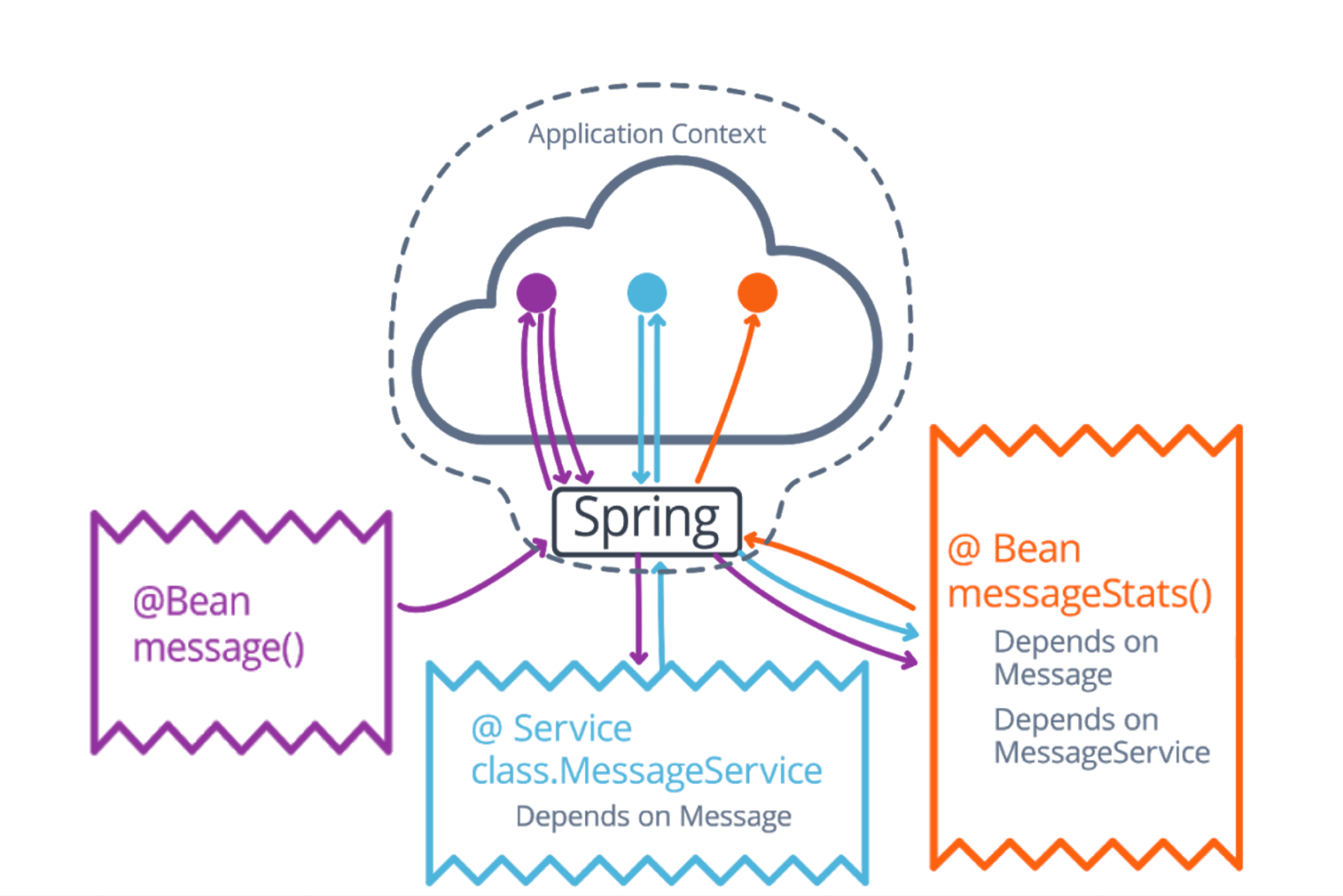
## Annotations to begin the Setup process

In a Spring Boot application, the basic setup of the Spring application context has already been done for us. The following annotations are the starting point of an application:

1. **@SpringBootApplication** - In a generated project starter, you're always given a main application class with the @SpringBootApplication annotation. This annotation is actually equivalent to three other annotations from Spring: @Configuration, @EnableAutoConfiguration, and @ComponentScan. The @SpringBootApplication configures Spring's component scanning and auto-configuration.
2. **@Configuration** - It tells Spring that the annotated class includes component definitions for Spring to process. These take the form of @Bean-annotated method whose return values are included as components in the application context. These methods can also take parameters, which act like the dependencies of the components returned by the methods.
3. **@EnableAutoConfiguration** - It tells Spring that it's okay to try to match dependencies to components automatically. Usually, that means dependencies are filled based on type, so in the example above, we have the compoundMessage method which depends on the basicMessage implicitly - the only String bean that Spring is aware of when constructing the compoundMessage is the basicMessage, so it uses that.

When there are multiple beans that satisfy a specific dependency, Spring's auto-configuration needs some help to decide which to use. A common solution is to mark a bean with the @Primary annotation to indicate a universally-preferred bean for its type. Or, you can use pairs of @Qualifier annotations on beans and the dependencies that reference them to gain a finer level of control.

@ComponentScan provides another layer of automation - automatic component scanning throughout your entire code base. But that's a topic for the next video!



**How Spring Processes an IoC Configuration.**

The figure above shows an example of how Spring processes an IoC configuration. The general steps are:

* A bean without dependencies is initialized first and placed within the application context.
* A service is instantiated by Spring, and the first bean is retrieved from the app context to be injected as a dependency, after which Spring places the service in the application context.
* Finally, another bean is initialized by Spring, which retrieves the previous two components to be injected as dependencies, after which the new bean is placed in the app context, and the application is fully initialized.

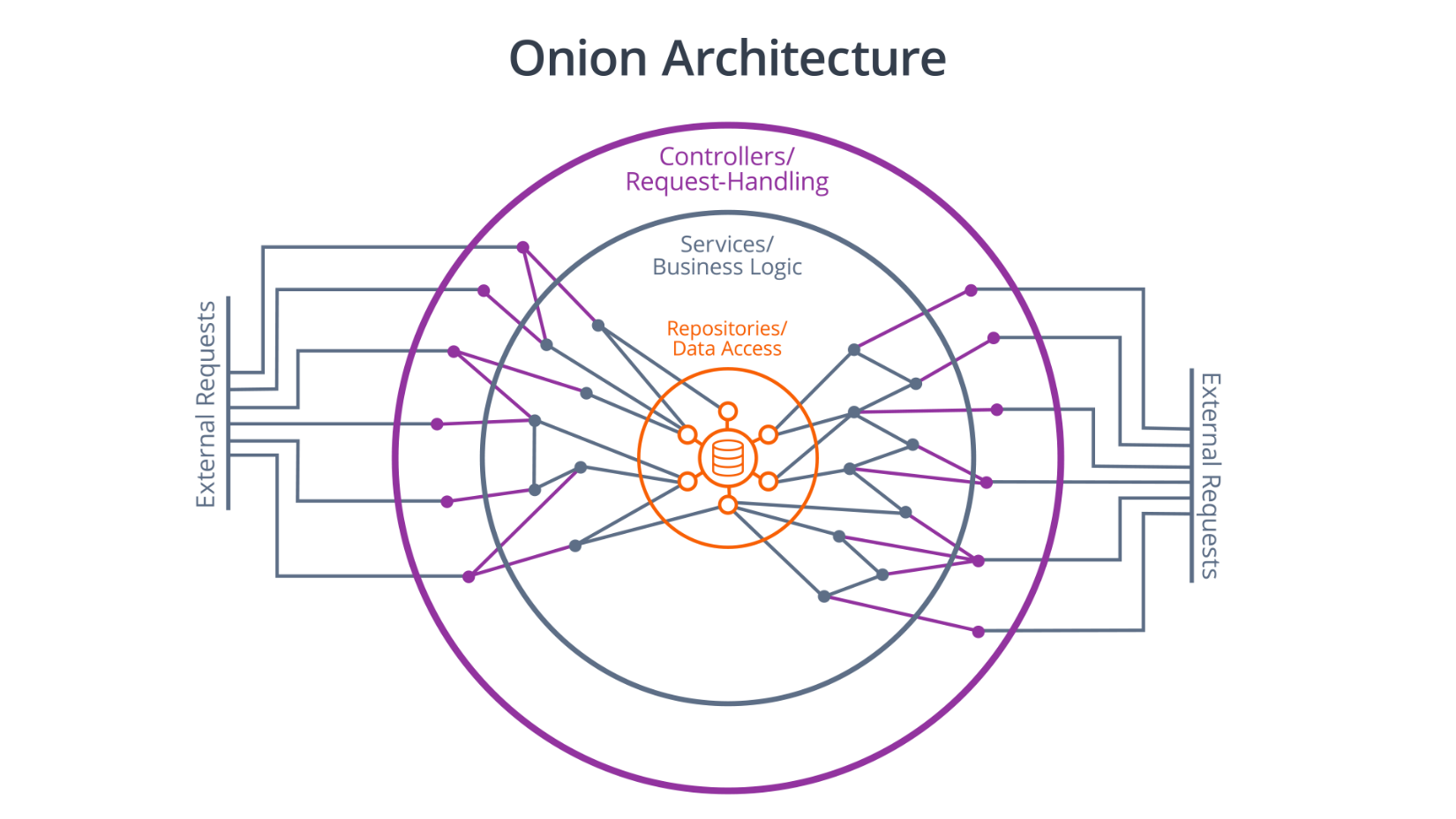
### **Key Terms**

* **Configuration Files**: Project files that configure some part of Spring's operation. Some are embedded in Java classes, like we just discussed, and others are .properties, .yaml, and .xml files that we'll discuss later this lesson. Some of them configure the IoC context, like the ones we just discussed, and others configure more abstract pieces of Spring's system.
* **Component Annotations**: Component annotations are annotations that identify application components for Spring to manage. @Bean and @Configuration are examples from the most recent videos, and in the next section we'll discuss @Component and @Service as well.
* **Application Context**: Spring's application context is just a giant data structure that holds all application component instances. It can be queried to gain access to a specified component at runtime, and it's what Spring uses to resolve dependencies.
* **Beans**: "Beans" are Spring's name for generic application components, and include any value Spring has stored in the application context. A bean is always either an object or primitive value.
* **Closed System**: Spring's application context is a closed system, which means that it manages all of the components stored within. It is not possible to instantiate a component manually and still link it fully with Spring - it will never be aware of the components inside of Spring's application context, and vice versa.
* **@SpringBootApplication**: An annotation put on the main application class of a Spring Boot project. It serves as an alias of three other annotations, @Configuration, @EnableAutoConfiguration, and @ComponentScan
* **@Configuration**: A class annotated with @Configuration is instantiated and managed by Spring as a component, but also as a bean factory. Any methods of the configuration class that are annotated with @Bean are used by Spring to create new beans to add to the application context.
* **@Bean**: A method annotated with @Bean inside of a configuration class will be used by Spring to generate a bean of the method's return type. This means that the developer can manually configure beans to be included in the application context.
* **@EnableAutoConfiguration**: A class annotated with @EnableAutoConfiguration tells Spring to try to automatically match beans to dependencies based primarily on type. This reduces the need for boilerplate code explicitly identifying individual beans as dependencies.
* **@Primary**: This annotation distinguishes the annotated bean method as the default dependency of its type. This is used to resolve conflicts that arise from having multiple bean definitions of the same type when auto configuration is enabled.
* **@Qualifier**: This annotation distinguishes the annotated bean method or dependency declaration as a qualified bean or dependency. Qualified beans are considered for unqualified dependencies, but only matching qualified beans are considered for qualified dependencies. You can read more about it [**here**](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-autowired-annotation-qualifiers).

# **Components and Services**

If we want to declare custom classes as Spring Components, the best way to do it is to make use of @ComponentScan, an annotation that tells Spring to search your code base for classes annotated with @Component. These classes will automatically be instantiated as Spring beans, so there's no need to define an @Bean-annotated method if you already have @Component on you classes. There are other variants of @Component that identify specific roles for each component to play. We'll see some examples of these in the coming lectures, but if you want to learn more, check out the link below.

One important thing to keep in mind is that @ComponentScan only marks the package of the class it's annotating for scanning - any classes outside of that package or its subpackages will be excluded by Spring. Here are the [**official Spring docs for @ComponentScan**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) as well as the [**official Spring docs explaining the how different stereotype annotations like @Component, @Service, @Repository, and others, function**](https://docs.spring.io/spring/docs/4.3.27.RELEASE/spring-framework-reference/htmlsingle/#beans-stereotype-annotations).



**Onion Architecture**

The figure above shows the basic structure of Onion Architecture. External requests must first pass through a layer of controllers or request handlers whose only purpose is to handle these external requests. These controllers then use the next layer of the onion, the services, to process the actions or analysis triggered by the request. The services, in turn, use each other and the next layer, repositories and data access, to persist the results of the actions triggered by the request.

### **Key Terms**

* **Onion Architecture**: An architectural pattern in which an application is separated into nested layers. In order for a request to be processed by the application, it must first travel through an outer layer of external interfaces and controllers, then through a middle layer of services and business logic, and finally through a persistence layer of data access objects. The separation of these layers emphasizes clean separation of concerns.
* **Application Component**: In Spring, this is any @Component-annotated class that is instantiated by Spring and placed in Spring's application context. Architecturally speaking, this is a logical unit of an application - a single-purpose library or object that solves a particular problem an application faces.
* **Service**: In Spring, this is any @Service-annotated class, handled identically to an @Component-annotated class. The difference between the two is semantics - a component is the most generic type of bean, and can be any kind of shared application structure. A service is specifically a collection of library methods that manage one aspect of an application's business logic. For example, a UserService would expose high-level actions related to the users of an application, and an AuthenticationService would expose actions for registering and authenticating a user. Services represent the middle layer of an onion architecture, and should contain the bulk of an application's business logic.
* **Repository**: In Spring, an @Repository-annotated class is treated identically to an @Component-annotated one, but as with @Service, the semantics are different. In an onion architecture, repositories are the inner layer - each repository should act like an interface to a specific set of stored or persistent data. For example, a UserRepository would expose an interface capable of create/read/update/delete and query operations on the users table of the database.