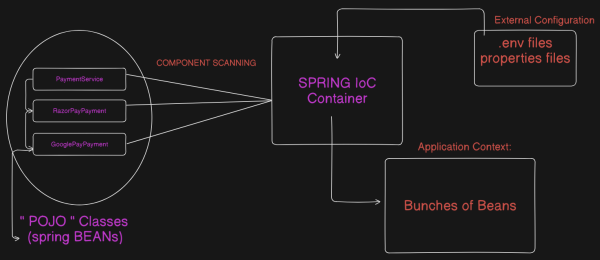
Intro To SPRINGBOOT Application:



Key Responsibilities of `ApplicationContext`

The `ApplicationContext` goes far beyond simply creating and managing beans. It provides a rich set of features that are essential for building robust applications.

- **Bean Lifecycle Management:** It's responsible for the complete lifecycle of beans. This includes instantiating them, populating their properties, calling initialization methods (`@PostConstruct`), and managing their destruction (`@PreDestroy`) when the application shuts down.
- **Dependency Injection:** This is its most well-known function. It automatically **wires** the dependencies between beans, as you saw in the diagram. When `UserService` needs a reference to `UserRepository`, the `ApplicationContext` finds and injects the correct `UserRepository` bean for you.
- **Event Propagation:** It provides a framework for publishing and listening to application events, allowing for a decoupled communication model between components.
- **Internationalization (i18N):** It can resolve text messages from a properties file, allowing you to easily support different languages in your application.
- **Resource Loading:** It can load resources (like files, URLs, or classpath resources) in a generic way, simplifying access to external files.

`ApplicationContext` **vs.** `BeanFactory`

You might also hear about the `BeanFactory`, which is an older, more basic interface. The `ApplicationContext` is an advanced version of the `BeanFactory` that includes all of the features listed above.

- `BeanFactory`: A basic factory. It instantiates beans only when they are requested (lazy loading). This is suitable for simple applications where memory is a concern.
- `ApplicationContext`: The modern, feature-rich container. It pre-instantiates all singleton beans at startup (eager loading), ensuring dependencies are available immediately. This is the preferred choice for almost all enterprise applications because of its extensive capabilities.

In short, think of the `ApplicationContext` as the fully-featured, ready-to-go IoC container that makes the magic of the Spring Framework happen. It's the central hub for all the components you see in the diagram.

JavaBeans & POJO Classes:

A POJO, or **Plain Old Java Object**, is a simple, ordinary Java class that encapsulates data without being tied to any specific framework. The name was coined by Martin Fowler and others in 2000 to advocate for using basic, clean Java objects instead of complex, framework-specific ones.

Core characteristics of a POJO

- **No framework dependencies:** A POJO does not extend special base classes, implement special interfaces, or use framework-specific annotations. This makes it portable across any Java environment.
- **Encapsulation:** Data fields are typically declared as `private`. Access to these fields is managed through `public` getter and setter methods, which helps maintain data integrity.
- **Simple structure:** POJOs are easy to write, understand, and reuse. They contain only the necessary logic for their purpose as data containers, separating them from complex business logic.

```
public class Student{

    // Private fields for data encapsulation
    private int id;
    private String name;
    private int age;

    // Optional: A default, no-argument constructor
    public Student() {
    }

    // Optional: A parameterized constructor for convenience
    public Student(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    // Public getter methods to access the private fields
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Public setter methods to modify the private fields
    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    // Optional: Override toString() for better debugging and logging
    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

A JavaBean is **a special type of Plain Old Java Object (POJO) that adheres to a specific set of conventions, allowing it to be used and manipulated by development tools and frameworks.** The primary purpose of JavaBeans is to create reusable software components.

The JavaBeans technology was introduced in 1996 to promote the creation of reusable, platform-independent components. Its standardized format allows development environments (IDEs) to "introspect" or automatically inspect a bean's capabilities, such as its properties and methods.

The core conventions of a JavaBean

A Java class must follow these conventions to be considered a JavaBean:

- **Public no-argument constructor:** The class must have a public constructor that takes no arguments. This allows frameworks and tools to easily instantiate the bean.
- **Serializable:** The class must implement the `java.io.Serializable` interface. This allows the bean's state to be saved (persisted) and restored.
- **Encapsulated properties:** All instance variables (or properties) must be `private`.
- **Public accessor (getter) and mutator (setter) methods:** To access and modify the private properties, the class must provide public getter and setter methods that follow a strict naming convention.
  - **Getters:** For a property named `propertyName`, the method must be `getPropertyName()`. For a `boolean` property, the getter can be `isPropertyName()`.
  - **Setters:** For a property named `propertyName`, the method must be `setPropertyName(propertyType value)`.

```
import java.io.Serializable;

public class Person implements Serializable {
    // 1. Implements Serializable | Public classes
    private static final long serialVersionUID = 1L;

    // Private properties
    private String firstName;
    private String lastName;
    private int age;

    // 2. Public no-argument constructor
    public Person() {
    }

    // Optional: A convenient parameterized constructor is also allowed
    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    // 3. Public accessor methods (getters)
    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }

    // 3. Public mutator methods (setters)
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Spring beans and Dependency injection

Spring beans are the foundational objects managed by the Spring Inversion of Control (IoC) container, while dependency injection is the process the container uses to "inject" these beans and their dependencies into other objects. This system promotes loose coupling and simplifies application development.

What is a Spring bean?

In the Spring Framework, a bean is a Java object that is instantiated, assembled, and managed by the Spring IoC container. The container is responsible for its entire lifecycle, from creation to destruction. Instead of manually creating objects with the `new` keyword, you define the structure, and Spring handles the rest.

How to define beans

- **Stereotype annotations:** The most common approach uses class-level annotations. When Spring's component scanning is enabled, it automatically detects and registers classes with these annotations as beans.
  - `@Component`: A generic annotation for any Spring-managed component. It is the parent for more specialized stereotypes.
  - `@Service`: For classes that hold business logic.
  - `@Repository`: For classes in the data access layer that interact with a database.
  - `@Controller` and `@RestController`: For classes that handle web requests in a Spring MVC application.
- **`@Bean` annotation:** Used on methods within a class annotated with `@Configuration`. The method's return value is registered as a bean in the container. This is useful for integrating third-party libraries where you cannot edit the source code to add a stereotype annotation.

What is dependency injection?

Dependency injection (DI) is a design pattern that implements the Inversion of Control (IoC) principle. Instead of an object being responsible for creating its own dependencies, the Spring container creates the dependent objects (beans) and "injects" them into the objects that need them.

Benefits of dependency injection

- **Loose coupling:** Components are no longer tightly bound to their specific implementations. For example, a `PaymentService` can depend on a `PaymentGateway` interface, and the Spring container can inject a specific implementation without the `PaymentService` needing to know about it.
- **Improved testability:** Because components are loosely coupled, it is easy to swap out real dependencies for mock objects during unit testing.
- **Simplified code:** DI eliminates much of the boilerplate code for manually instantiating and wiring objects.

Types Of Injections:

- **Constructor injection:** The container supplies dependencies as constructor arguments. This is the Spring team's preferred method because it ensures required dependencies are present when an object is created and can be declared `final`, making the object immutable.

java

```
@Service
public class MyService {
    private final MyRepository myRepository;

    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

- **Setter injection:** The container calls setter methods on the bean after construction. This is best for optional dependencies that can be changed after the object is created.

java

```
@Service
public class MyService {
    private MyRepository myRepository;

    @Autowired
    public void setMyRepository(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

- **Field injection:** The container injects dependencies directly into a class's fields using the `@Autowired` annotation. While convenient and concise, this approach is generally discouraged because it makes the class harder to test and obscures its dependencies.

java

```
@Service
public class MyService {
    @Autowired
    private MyRepository myRepository;
}
```