Moving From XML Based Config -> Java Based Config -> Spring Boot

Got it 👍
You're essentially **explaining + demonstrating how Spring Core concepts carry over to Spring Boot**, with focus on **component scanning, dependency injection, @Primary, @Qualifier, @Value, and setter injection**.
Here's a **clean, simplified explanation** you can use (notes / revision-ready).

1️⃣ What we've done so far (Big Picture)

- We already learned **Spring Core** using:
  - XML config
  - Java-based config
  - Annotations like @Component, @Autowired, @Primary, @Qualifier, @Scope, @Value
- Now we're doing **the same things in Spring Boot**, but:

👉 **without writing any explicit configuration**

2️⃣ Why Spring Boot works without configuration

The magic comes from this annotation:

@SpringBootApplication

This annotation:

- Starts the **Spring container**
- Enables **component scanning**
- Finds all @Component classes in the same package (and sub-packages)
- Auto-configures beans in an **opinionated way**

So everything we manually configured in Spring Core is happening **behind the scenes** in Spring Boot.

3️⃣ Dependency Injection using Interface (Best Practice)

Step 1: Create an interface

```
public interface Computer {
   void compile();
}
```

Step 2: Multiple implementations

```
@Component
public class Laptop implements Computer {
   public void compile() {
      System.out.println("Compiling in Laptop...");
   }
}

@Primary
@Component
public class Desktop implements Computer {
   public void compile() {
      System.out.println("Compiling in Desktop...");
   }
}
```

- Now Spring finds **two beans** of type Computer
- This causes **ambiguity**

———

4️⃣ Resolving ambiguity: @Primary vs @Qualifier

Rule:

- @Primary → default bean
- @Qualifier → **overrides Primary**

Alien class

```
@Component
public class Alien {

   @Value("28")
   private int age;

   private Computer com;

   @Autowired
```

```
    @Qualifier("laptop") // overrides @Primary
    public void setCom(Computer com) {
        this.com = com;
    }

    public void code() {
        com.compile();
    }

    public int getAge() {
        return age;
    }
}
```

✅ Even though Desktop is @Primary, Spring injects **Laptop** because of @Qualifier.

———

5️⃣ Setter Injection (instead of field injection)

You used:

```
@Autowired
public void setCom(Computer com)
```

Why this is good:

- Better for **testing**
- Follows **clean design**
- Preferred in real projects

———

6️⃣ Injecting simple values using @Value

```
@Value("28")
private int age;
```

- This avoids hardcoding values directly in code
- Later, this value can come from:
  - application.properties
  - application.yml
  - environment variables

7️⃣ Running the application

```
@SpringBootApplication
public class SpringbootdemoApplication {

    public static void main(String[] args) {
        ApplicationContext context =
                SpringApplication.run(SpringbootdemoApplication.class, args);

        Alien obj = context.getBean(Alien.class);
        System.out.println(obj.getAge());
        obj.code();
    }
}
```

Output:

```
28
Compiling in Laptop...
```

8️⃣ Key Takeaways (Very Important)

✅ All Spring Core annotations work **exactly the same** in Spring Boot
✅ Spring Boot = Spring + Auto Configuration
✅ @Qualifier overrides @Primary
✅ Interface-based injection is best practice
✅ Less configuration, same power

1️⃣ Why @Component is not the only stereotype annotation

So far, we've used:

@Component

Meaning:

> "Hey Spring, manage this class as a bean."
>
> ✔️ Works
> ❌ But not expressive
>
> In **real-world applications**, just knowing *"this is a bean"* is not enough.
> We also want to know **what role this class plays** in the application.
>
> That's where **layer-based stereotype annotations** come in.

———

2️⃣ Typical Layered Architecture (High Level)

Even before Spring MVC, almost every application follows this flow:

Client
↓
Controller
↓
Service
↓
Repository (DAO)
↓
Database

Each layer has **one responsibility only**.

———

3️⃣ Responsibilities of Each Layer

🔷 Client

- Sends request
- Receives response
- Could be:
  - Browser
  - Mobile app
  - Another service
  - Console app (like your current example)

🔷 **Controller (we'll cover later)**

- Accepts request
- Sends response
- ❌ No business logic
- ❌ No database calls

Annotation:

@Controller
@RestController

🔷 **Service Layer (Business Logic)**

What happens here?

- Processing
- Calculations
- Decision-making
- Filtering
- Sorting
- Validation

Examples:

- Find best laptops between $10–$100
- Calculate ratings
- Apply discount logic
- Choose best stock

📌 Service does **NOT** talk directly to client or database.

Annotation:

@Service

🔷 **Repository / DAO Layer (Data Access)**

What happens here?

- Interacts with database
- Fetches data
- Saves data
- Updates data

Examples:

- Get laptop data
- Save laptop info
- Fetch laptops by price range

📌 Repository does **NO business logic**

Annotation:

@Repository

4️⃣ Why not just use @Component everywhere?

Technically, you *can* do this:

```
@Component
public class LaptopService { }
```

But Spring provides **semantic meaning** with stereotypes:

Annotation	Meaning
@Component	Generic bean
@Service	Business logic
@Repository	Database access
@Controller	Request handling

Benefits:

✅ Better readability
✅ Clear separation of concerns
✅ Easier debugging
✅ Better tooling support
✅ Spring adds **extra behavior** (especially for @Repository)

5️⃣ Important Hidden Advantage (Very Interview-Friendly)

@Repository does more than @Component

It:

- Translates database exceptions into Spring's unchecked exceptions
- Helps with consistent error handling

This is something **only @Repository does**, not @Component.

6️⃣ Applying this to your Laptop Example (Conceptually)

Right now you have:

Client (main method)
↓
Laptop

But in a layered design, it should be:

Client
↓
LaptopService (@Service)
↓
LaptopRepository (@Repository)
↓
Database

Each layer talks **only to the next layer**, never skipping.

7️⃣ Which annotation should I use when creating classes?

Class Type	Annotation
Utility / Generic	@Component
Business Logic	@Service
Database Layer	@Repository
Request Handler	@Controller / @RestController

Introduction to Layered Architecture in Spring Applications

- Applications are typically divided into **multiple layers** for better structure and responsibility separation.
- Even though the current project is **console-based**, the same layered concept applies to full applications.
- Common layers discussed:
  - **Client**
  - **Service**
  - **Repository**
- Each layer has a **specific responsibility** and should not overlap with others.

Title: Importance of Using Separate Packages for Each Layer

- It is recommended to use **different packages** for different layers.
- While not mandatory for compilation, separate packages help with:
  - Better code organization
  - Easier maintenance
  - Clear understanding of responsibilities
- Packages introduced:
  - model
  - service
  - (repository to be introduced later)

Title: Model Layer for Entity Representation

- Objects such as **Laptop**, **Alien**, and **Desktop** represent actual data entities.
- These objects are passed between different layers.
- They represent data that may eventually be stored in a **database**.
- Such classes should be placed in a **model package**.
- The model layer contains **pure data representation**, not logic.

Title: Creating the Service Layer

- The **Service layer** is responsible for:
  - Processing data
  - Applying business logic
  - Making decisions based on object configurations
- It does **not** interact directly with the database.
- Example responsibilities:
  - Checking if a laptop is good for coding
  - Performing operations on laptop objects

Title: Using @Service as a Stereotype Annotation

- To allow Spring to manage the Service class, a stereotype annotation is required.
- Instead of @Component, the **@Service annotation** is used.
- Key points:
  - @Service internally uses @Component
  - Functionally, both behave the same
  - @Service provides better **semantic meaning**
- Using @Service helps identify the class as a **business logic layer**.

```
@Service
public class LaptopService {
   public void addLaptop(Laptop lap) {
      // Processing logic
   }

   public boolean isGoodForCoding() {
      return true;
   }
}
```

Title: Service Layer Responsibilities vs Database Operations

- The service layer **should not contain database connection logic**.
- Although JDBC code can technically be written inside a service class, it is **not a good practice**.
- Responsibilities of the Service layer:
  - Process data
  - Coordinate with repository
- Database interaction should be handled in a **separate repository layer**.

Title: Motivation for Repository Layer

- Database interactions require:
  - JDBC steps
  - Connection handling
  - Query execution
- Writing this code inside the service layer is discouraged.
- A **Repository class** is designed to:
  - Handle all database operations
  - Keep data access logic isolated
- The repository layer will be introduced in the next step.

Title: Accessing Service Beans Using ApplicationContext

- Service objects are retrieved using Spring's container.
- The ApplicationContext is used to fetch the service bean.

```
LaptopService lapService = context.getBean(LaptopService.class);
Laptop lap = context.getBean(Laptop.class);
lapService.addLaptop(lap);
```

- This demonstrates:
  - Dependency injection via Spring
  - Service acting as a processing layer for model objects

Title: Summary of Layer Roles Discussed

- **Model Layer**
  - Represents entities
  - Passed across layers
- **Service Layer**
  - Handles business logic
  - Uses stereotype @Service
- **Repository Layer**
  - Intended for database operations
  - Will be implemented separately

Title: Key Takeaways from This Section

- @Component is not the only stereotype annotation.
- @Service is preferred for business logic classes.
- Clear separation of layers improves maintainability.
- Database logic should not reside in service classes.
- Layered architecture prepares the application for scalability.

Introduction to the Repository Layer

- The **Repository layer** is responsible for **interacting with the database**.
- All database-related logic should be placed in this layer.
- The repository handles:
  - Creating records
  - Reading data
  - Updating records
  - Deleting records (**CRUD operations**)
- The service layer should **not** contain database connection logic.

Title: Creating the Repository Class

- A new class is created to handle database operations.
- Example class name: LaptopRepo
- This class represents the **data access layer**.
- Methods in this class will later contain **JDBC steps** (not implemented yet).

```
public void save(Laptop lap) {
   // JDBC steps will go here
}
```

- For now, the method simply prints a confirmation message to simulate saving data.

Title: Using @Repository Annotation

- To allow Spring to manage the repository object, a stereotype annotation is required.
- Instead of @Component, the **@Repository annotation** is used.
- Key points:
  - @Repository internally uses @Component

        - Functionally behaves the same as @Component
        - Provides **semantic clarity** that this class is a database layer
    - Helps developers easily identify data-access classes.

```
@Repository
public class LaptopRepo {
   public void save(Laptop lap) {
      System.out.println("Laptop saved successfully...");
   }
}
```

Title: Injecting Repository into Service Layer

- The **service layer** needs access to the repository to save data.
- This is done using **dependency injection**.
- The repository object is injected into the service class using @Autowired.

```
@Autowired
public LaptopRepo laptopRepo;
```

- This is an example of **field injection**.
- Setter injection can also be used if required.

Title: Calling Repository Methods from Service

- The service layer calls repository methods instead of handling database logic itself.
- The addLaptop() method in the service class delegates saving to the repository.

```
public void addLaptop(Laptop lap) {
   laptopRepo.save(lap);
}
```

- This maintains a **clear separation of responsibilities**:
    - Service → Processing
    - Repository → Database interaction

Title: Error When Repository Is Not Annotated

- If @Repository (or @Component) is not used:
  - Spring cannot find the repository bean
  - Application throws a **bean not found** error
- Adding the appropriate stereotype annotation resolves the issue.

Title: Organizing Repository in a Separate Package

- Repository classes should be placed in a dedicated package.
- Example package name:
  - com.springbootproject.springbootdemo.repo
- Benefits:
  - Better code organization
  - Clear architectural structure
  - Easier navigation in large projects

Title: Relationship Between Service and Repository Layers

- **Service Layer**
  - Handles processing and business logic
  - Calls repository methods
- **Repository Layer**
  - Handles database connectivity
  - Executes CRUD operations
- Service should never directly include JDBC or database logic.

Title: Key Takeaways from Repository Layer

- Repository layer is dedicated to database operations.
- @Repository is preferred over @Component for data-access classes.
- Service layer communicates with repository using dependency injection.
- JDBC logic should only exist in the repository layer.
- Proper package structure improves maintainability and clarity.