

Part I

Tomcat
Spring Boot
Actuator
@RestController
Proxies

Responsibilities of an Application Backend

Database Management:

- Store, retrieve, update, and delete data.
- Ensure data integrity and consistency.

Application Logic:

- Core business rules and processing.
- Controls how data is created and changed.

Authentication and Authorization:

- Login, permissions, and secure access control.

API Management:

- Handles HTTP routing, request/response, and API security.

Performance Optimization:

- Efficient memory and data handling, load balancing.

Responsibilities of an Application Backend

Security Measures:

- Protect against threats like SQL injection, XSS.

Data Integration:

- Connects to databases, APIs, and external services.

Session Management:

- Maintains user state between requests.

Error Handling and Logging:

- Detect, log, and respond to application errors.

Backup and Recovery:

- Ensures data can be restored after failure.

Notification Services:

- Manages emails, SMS, or system alerts.

Scalability Management:

- Adjusts resources to meet demand efficiently.



What Do We Need to Build a Backend?

A server that listens for client requests:

- Examples: Tomcat, Jetty, Undertow.
- Tomcat: a web server and servlet container.
- Can be embedded or run standalone.



Web Server vs Application Server



Web Server:

- Handles HTTP requests.
- Serves static files (HTML, CSS, JS).
- Examples: Apache HTTPD, Nginx.



Application Server:

- Runs dynamic app logic (Java, PHP, etc.).
- Handles servlets, database access, business logic.
- Examples: Tomcat, Wildfly, WebSphere.



Spring Boot includes an embedded application server (Tomcat).

What is a Servlet?

✚ **A Servlet is a Java class that handles HTTP requests.**

- Uses `HttpServletRequest` and `HttpServletResponse`.
- Lives inside a servlet container (like Tomcat).
- Spring Boot wraps this with `DispatcherServlet` and annotations.

Example:

```
@WebServlet("/hello")
```

```
public class HelloServlet extends HttpServlet {
```

```
    protected void doGet(HttpServletRequest req, HttpServletResponse res) throws  
    IOException {
```

```
        res.getWriter().write("Hello world");
```

```
    }
```

```
}
```



HTTP Request and Response

HTTP Request:

- Method: GET, POST, PUT, DELETE.
- URL, headers, and optional body (JSON, form).

HTTP Response:

- Status code (200, 404, 500).
- Headers: Content-Type, etc.
- Body: plain text, JSON, HTML.

 **Tools like Postman help visualize this.**

Frameworks : Why We Use Them

Servlet API is powerful but verbose.

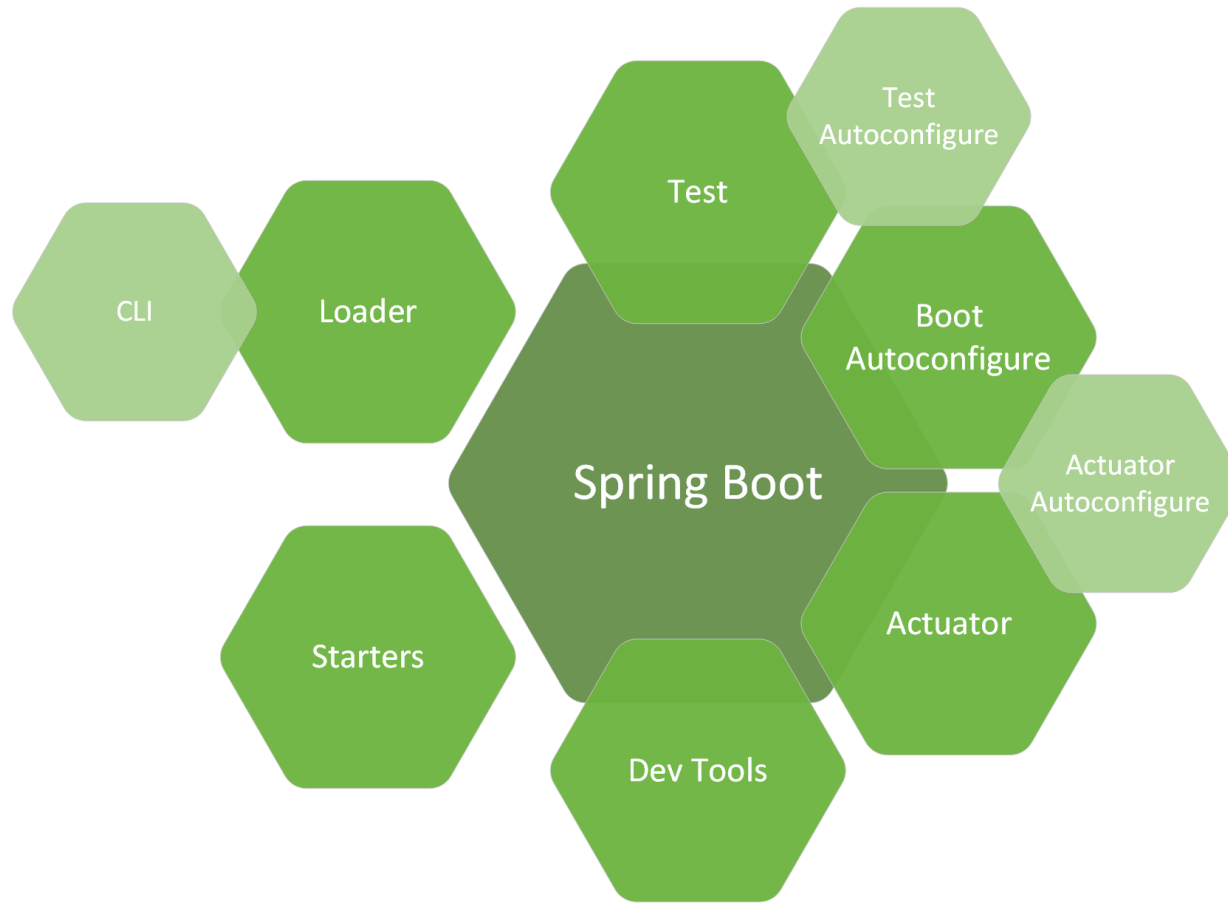
Spring Framework simplifies web development:

- Dependency Injection (DI): loose coupling and modularity.

- Aspect-Oriented Programming (AOP): cross-cutting concerns like logging.

- Removes repetitive boilerplate code.

Spring Boot



Introduction to Spring Boot



What is Spring Boot?

- A framework to build production-ready Spring applications quickly.
- Eliminates boilerplate code and configuration.
- Comes with embedded servers like Tomcat.
- Convention over configuration.
- Opinionated defaults to simplify setup.

Why Use Spring Boot?



Advantages:

-
- Rapid development with minimal configuration.
-
- Embedded server (no need to deploy WAR files).
-
- Huge ecosystem: Spring Data, Spring Security, Spring Cloud.
-
- Built-in production features: metrics, health checks.
-
- Easy integration with modern tools (Docker, Kubernetes, AWS).

Core Features of Spring Boot



Core Features:

-
- Auto-Configuration: Automatically configures application based on dependencies.
-
- Spring Boot Starters: Pre-configured starter dependencies for common use cases.
-
- Spring Boot CLI: Command-line tool to run and test apps quickly.
-
- Spring Boot Actuator: Production-ready monitoring and management tools.

Build Tool: Maven

Maven automates project build and dependency management.

Project Object Model (POM):

- pom.xml defines project config and dependencies.

Dependencies and Repositories:

- Automatically fetches and manages libraries.

Build Lifecycle:

- Phases: compile → test → package.
- Automates full build process.

Plugins and Goals:

- Extendable via plugins for tasks like compile or deploy.



Spring Boot Starters



What are Starters?

- Starters are dependency descriptors.
- They simplify Maven/Gradle config for common use cases.
- Each starter includes a curated set of dependencies.
- Examples: spring-boot-starter-web, spring-boot-starter-data-jpa.
- Helps avoid version conflicts and speeds up setup.

Common Spring Boot Starters



Popular Starters:

- spring-boot-starter-web → for web apps and REST APIs.
- spring-boot-starter-data-jpa → JPA + Hibernate + DB connectivity.
- spring-boot-starter-security → authentication and authorization.
- spring-boot-starter-test → testing tools and frameworks.
- spring-boot-starter-thymeleaf → HTML templating engine.

Spring Boot Actuator Overview

Monitoring and managing your
Spring Boot application with built-in
production-ready features.

Spring Boot Actuator



What is Actuator?

- Adds production-ready features to Spring Boot apps.
- Provides monitoring and management endpoints.
- Helps inspect running app: health, metrics, beans, env, etc.
- Endpoints are exposed as REST APIs.

Useful Actuator Endpoints

Common Endpoints:

- /actuator/health → Application health status.
- /actuator/info → App metadata (name, version).
- /actuator/metrics → Performance metrics.
- /actuator/env → Current environment properties.
- /actuator/beans → List of Spring beans.

Health Endpoint – Works Out of the Box



- `/actuator/health` is available by default.



- Shows basic application health (e.g. UP, DOWN).



- Requires no configuration for basic status.



- Add Spring dependencies (like DB, Redis) to auto-extend health checks.

Ex 00

- Generate a new Spring Boot project with version 3.4.5, Java 17 and Maven.
- Add spring-boot-starter-actuator and spring-boot-starter-web dependencies.
- Check actuator/health endpoint works



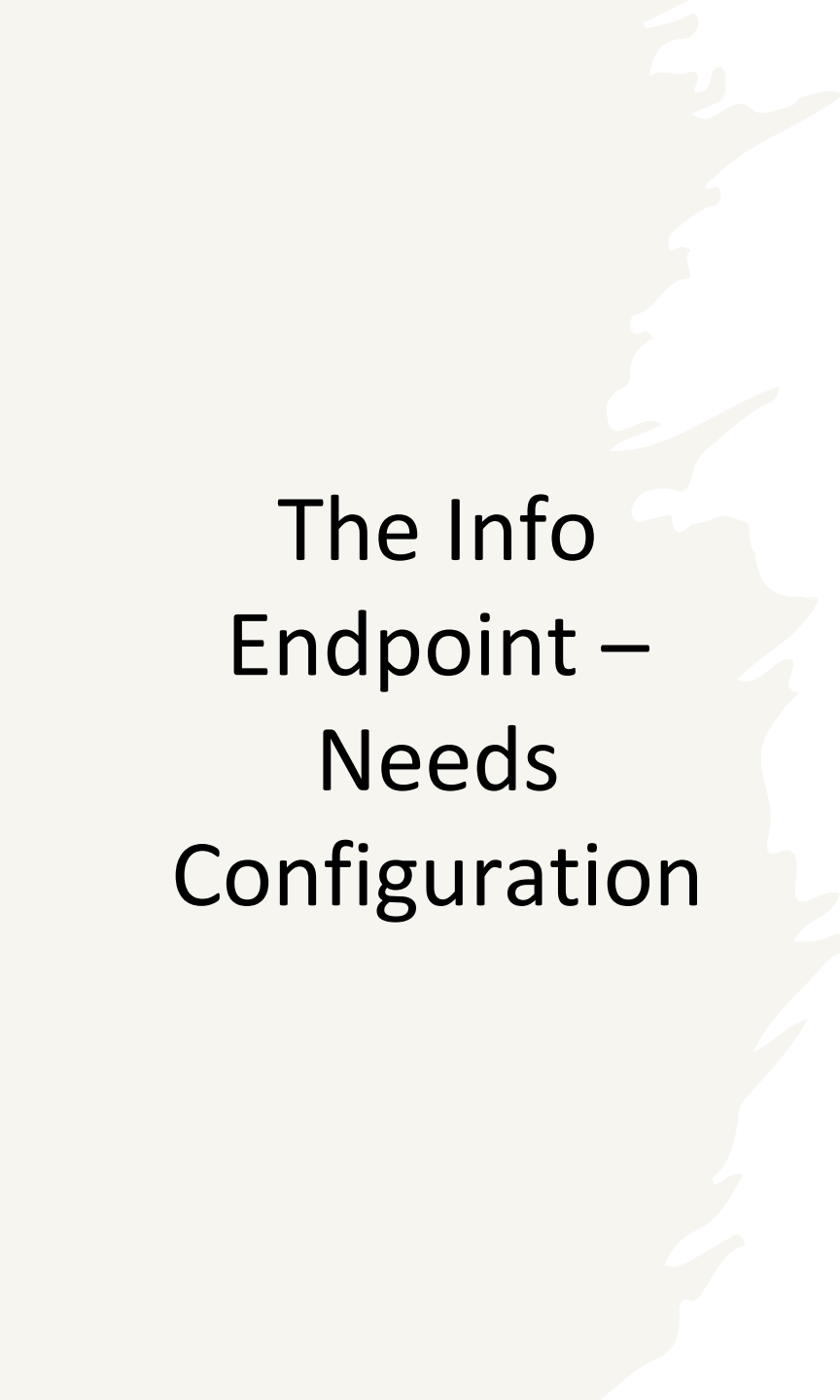
Enabling Other Endpoints

- By default, only /actuator/health is exposed over HTTP.
- To expose others, use:

```
management.endpoints.web.exposure.include=info,health,metrics
```

- To expose everything (not recommended for prod):

```
management.endpoints.web.exposure.include=*
```



The Info Endpoint – Needs Configuration

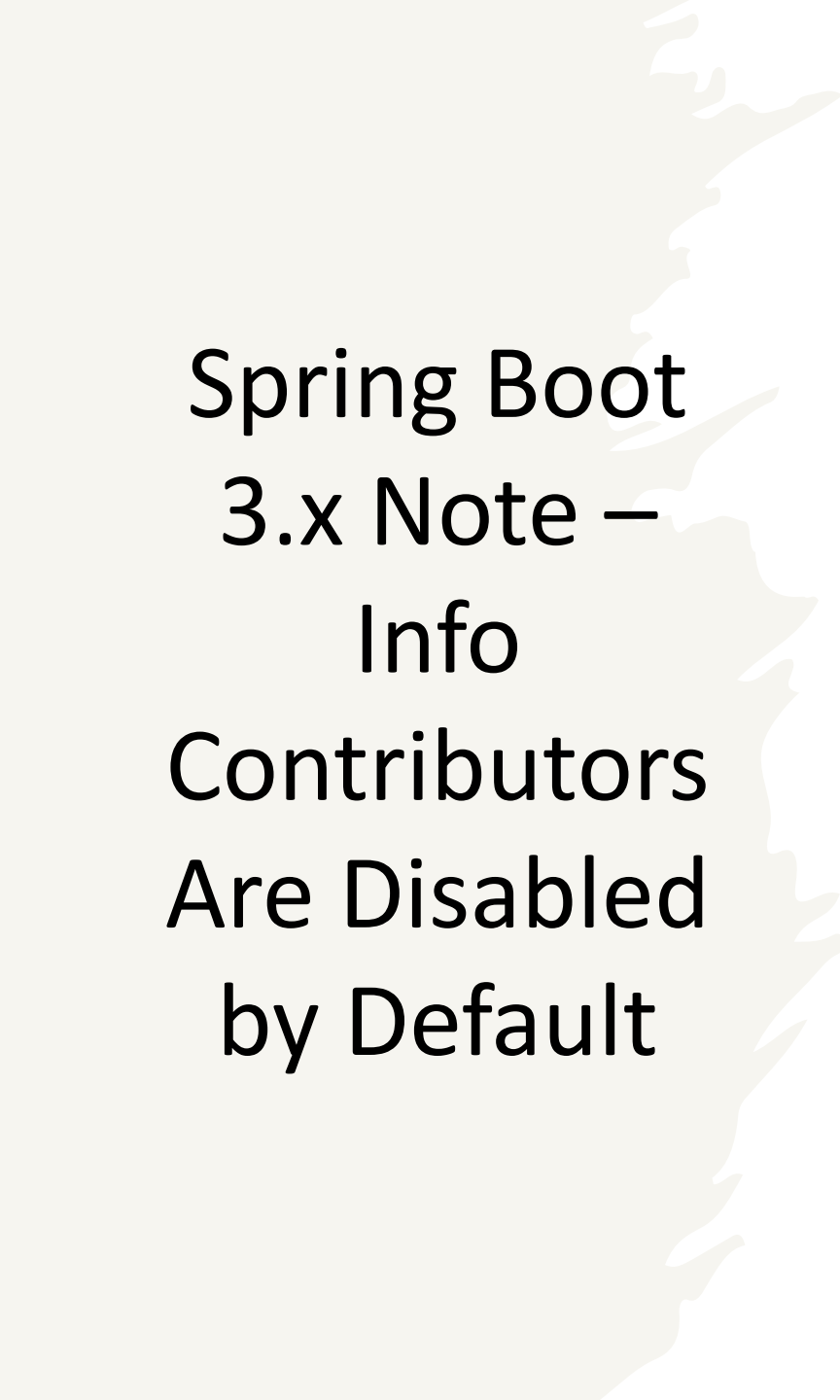
- `/actuator/info` returns `{}` by default.
- It requires `info.*` properties in `application.properties` or YAML.

Configuring the Info Endpoint

```
application.properties:  
info.app.name=My App  
info.app.version=1.0.0  
info.feature.experimental=true
```

→ Result in /actuator/info:

```
{  
  "app": { "name": "My App", "version":  
    "1.0.0" },  
  "feature": { "experimental": true }  
}
```



Spring Boot 3.x Note – Info Contributors Are Disabled by Default

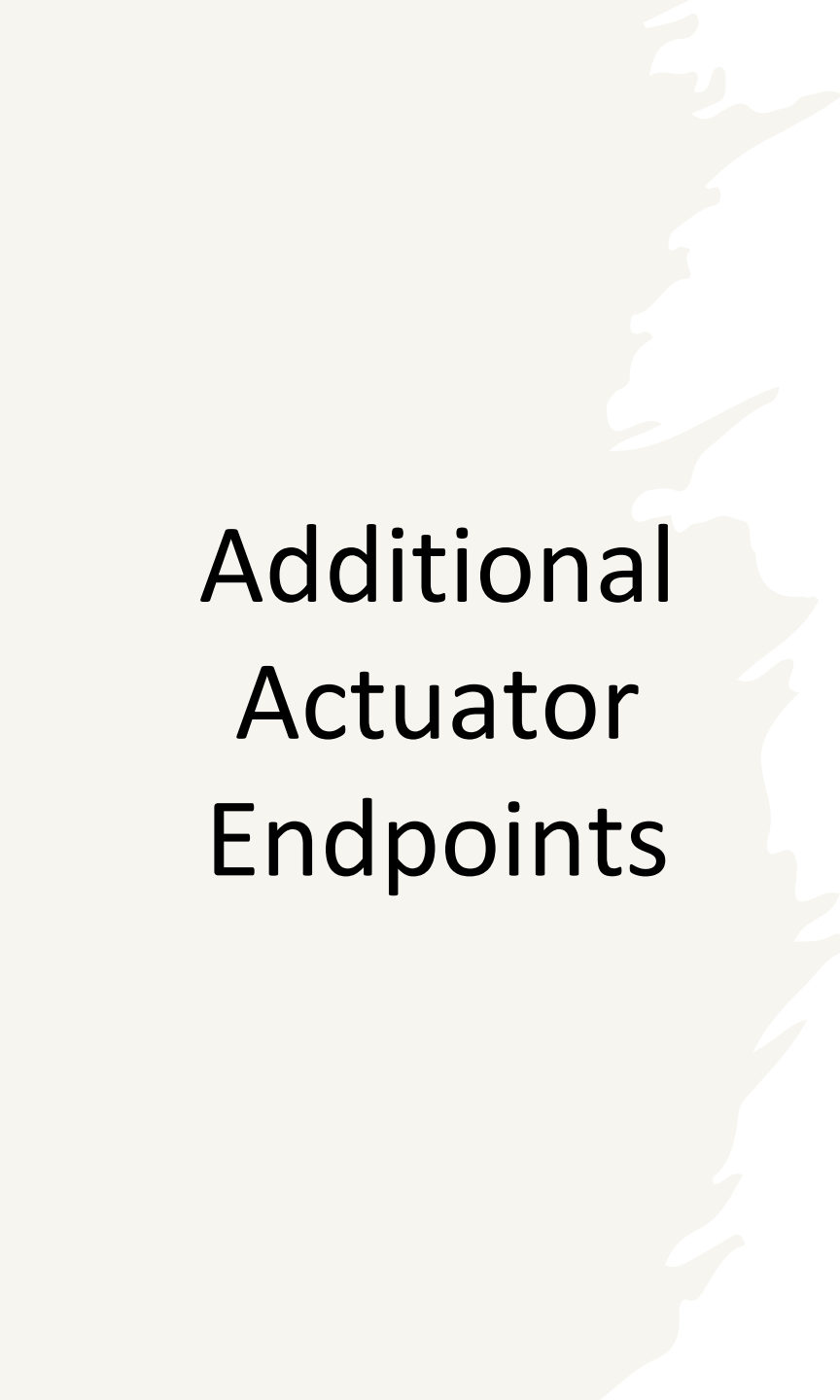
- You must explicitly enable info contributors in Spring Boot 3.x:
`management.info.defaults.enabled=true`
`management.info.env.enabled=true`
- Optional: enable build or git info contributors
`management.info.build.enabled=true`
`management.info.git.enabled=true`

Custom InfoContributor (Optional)

- For full control over /actuator/info:

@Component

```
public class MyInfoContributor implements  
InfoContributor {  
    public void contribute(Info.Builder builder) {  
        builder.withDetail("my", Map.of("key", "value"));  
    }  
}
```



Additional Actuator Endpoints

- `/actuator/metrics` – application and JVM metrics (Micrometer)
- `/actuator/env` – all environment properties
- `/actuator/loggers` – view and change logging levels
- `/actuator/httptrace` – recent HTTP request traces (requires httptrace dependency)
- `/actuator/threaddump` – thread dump snapshot
- `/actuator/heapdump` – JVM heap dump
- `/actuator/prometheus` – Prometheus scrape endpoint (with Prometheus registry)

Summary

- /actuator/health works out of the box.
- /actuator/info requires configuration.
- Spring Boot 3.x: enable contributors explicitly.
- You can extend /info with custom InfoContributors.

Ex 01

- Enable actuator/info endpoint.
- Add to application.properties details about the app (name,description,version) and on a different tag, details about the developer (name, role).
- Debug InfoEndpointAutoConfiguration with and without info endpoint enabled

5. Spring MVC

A powerful module for building web applications.

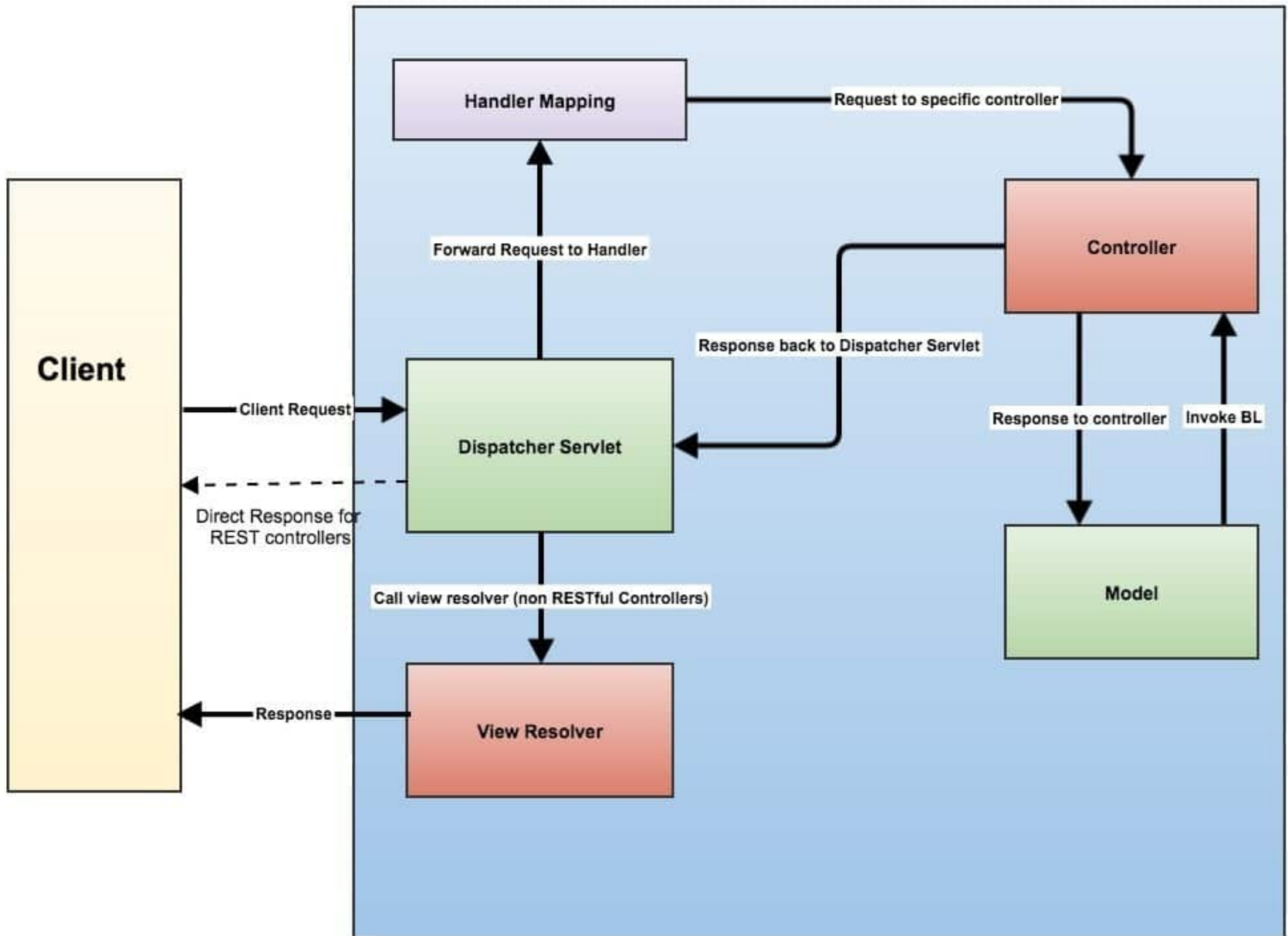
- Based on Model-View-Controller pattern.

DispatcherServlet:

- Front controller for routing requests.
- Delegates to appropriate controller handlers.

MVC Breakdown:

- Model: business logic and data.
- View: renders output (HTML, XML, JSON).
- Controller: processes input and coordinates model + view.



Spring MVC Architecture

Spring Boot HTTP Request Lifecycle

What happens when a request hits Spring Boot:

1. Browser/client sends HTTP request to server (e.g. localhost:8080).
2. Embedded Tomcat receives the request.
3. DispatcherServlet handles the request.
4. Spring matches the URL to a `@RestController` method.
5. The method runs and returns a response object.
6. Response is serialized (e.g. JSON) and sent back over HTTP.

Introduction to Spring Web & REST

Spring Web provides tools to build web and RESTful applications.

- REST: Representational State Transfer — a design pattern for APIs.
- Spring MVC handles HTTP requests and responses via annotations.
- RESTful APIs follow HTTP methods to expose resources (data).
- Controller classes define endpoints for the web or APIs.

@RestController and @RequestMapping

@RestController:

- Combines @Controller + @ResponseBody.
- Tells Spring to return data directly (JSON by default).

@RequestMapping:

- Base annotation to map web requests to handler methods.
- Can be applied at class or method level.
- Can define method, path, headers, params, etc.

HTTP Method-Specific Mappings

@GetMapping("/items"):

- Maps to HTTP GET for retrieving resources.

@PostMapping("/items"):

- Maps to HTTP POST for creating new resources.

@PutMapping("/items/{id}"):

- Maps to HTTP PUT for updating existing resources.

@DeleteMapping("/items/{id}"):

- Maps to HTTP DELETE for deleting resources.

@PatchMapping("/items/{id}"):

- Maps to HTTP PATCH for partial updates.

Interpreting Request Data

@PathVariable:

- Extracts value from URI path. Example: /items/{id}.

@RequestParam:

- Extracts query parameter. Example: /search?keyword=book.

@RequestBody:

- Maps the HTTP request body to a Java object (usually JSON).

@RequestHeader:

- Reads HTTP header value from request.

@ModelAttribute:

- Binds form fields to a model object (mainly for web forms).

Example REST Controller

```
@RestController
```

```
@RequestMapping("/api/items")
```

```
public class ItemController {
```

```
    @GetMapping("/{id}")
```

```
    public Item getItem(@PathVariable Long id) { ... }
```

```
    @PostMapping
```

```
    public Item createItem(@RequestBody Item item) { ... }
```

```
}
```


```
@GetMapping("/filter")
```

```
public Item getItem(@RequestParam Long idc) { ... }
```



Ex 02

- Create a PizzaController with:
 - - an endpoint that returns a list of Pizzas ("Salami", "Prosciutto", "Capriciosa", etc)
 - - an endpoint that uses a path variable to return one Pizza by name
 - - an endpoint that uses a request parameter to filter Pizzas that contain a sequence of characters in their name

Intro to Spring: Beans & Dependency Injection



What is a Bean?



-
- A Spring `**bean**` is an object managed by the Spring IoC container.
-
- Created, initialized, and injected automatically by Spring.
-
- Typically defined using annotations like `@Component`, `@Service`, `@Repository`, or `@Bean`.

Creating a Bean

@Component

```
public class Engine {}
```

@Service

```
public class CarService {
```

@Autowired

```
private final Engine engine;
```

```
public CarService() {
```

```
}
```

```
}
```


What is Dependency Injection?

- Technique where one object provides dependencies of another object.
- You don't create the object manually; Spring injects it.

Benefits:

- ✓ Loose coupling
- ✓ Easier testing
- ✓ Clean, readable code

Dependency Injection Types

- **Constructor Injection** (recommended): Ensures dependencies are final and not null.
- **Field Injection**: Uses `@Autowired` on fields (less testable).
- **Setter Injection**: Optional dependencies, mutable objects.

Constructor Injection Example

@Component

```
public class Car {  
    private final Engine engine;
```

@Autowired

```
public Car(Engine engine) {  
    this.engine = engine;  
}  
}
```

Spring Bean Scopes

- ****Singleton**** (default): One shared instance.
- ****Prototype****: New instance every time it's requested.
- ****Request**** (web): One per HTTP request.
- ****Session**** (web): One per HTTP session.

Use `@Scope("prototype")` to set.

Bean Scope Example

```
@Component
```

```
@Scope("prototype")
```

```
public class TaskProcessor {  
    // new instance every time  
}
```

@Bean vs @Component

- `@Component`: Automatically detected via classpath scanning.
- `@Bean`: Declares a bean in a `@Configuration` class manually.

Use `@Bean` when you don't control the class source or need customization.

Manual Bean Definition Example

@Configuration

```
public class AppConfig {
```

@Bean

```
public Engine engine() {  
    return new Engine("V8");  
}
```

```
}
```

Summary

- Beans are the backbone of Spring apps.
- DI allows Spring to inject what you need automatically.
- Choose the right bean scope for your use case.
- Use `@Component` for automatic, `@Bean` for manual definitions.

Spring Bean Lifecycle Phases

Key phases:

Bean Instantiation

Populate Properties (Dependency Injection)

Aware Interfaces Callback

BeanPostProcessor Pre-Initialization

@PostConstruct

InitializingBean.afterPropertiesSet()

Custom init-method

BeanPostProcessor Post-Initialization

Bean Destruction: @PreDestroy,

DisposableBean.destroy(), custom destroy-method

@PostConstruct Annotation

Definition and Usage

Part of javax.annotation package (JSR-250)

Executed after dependency injection and before Spring init callbacks

Applicable to methods with no arguments

Runs only once per bean instance

Example: @PostConstruct

Example in a Spring component:

```
@Component
public class MyBean {
    @PostConstruct
    public void init() {
        // initialization logic
    }
}
```

Ex 03

- Create A Pizza class with a **Long id** and **String name** fields
- Create a Pizza Service class and implement Pizza creation, search and filtering logic into the Service
- Inject Pizza Service (field or constructor injection) into Pizza controller

Spring Proxies

Interface-based vs Class-based (CGLIB)

Understanding how Spring creates
dynamic proxies

Why Proxies?

- Intercept method calls for AOP (transactions, security, caching)
- Add functionality before/after method execution
- Decouple cross-cutting concerns from business logic

JDK Dynamic Proxies

- Used when bean implements at least one interface and proxyTargetClass=false
- Creates a proxy implementing the specified interfaces
- Delegates calls to the target bean
- ****Limitation:**** Cannot proxy concrete classes or non-interface methods

Example:

```
ProxyFactory factory = new ProxyFactory(targetBean);  
factory.setInterfaces(MyService.class);  
MyService proxy = (MyService) factory.getProxy();
```

CGLIB Proxies

- Used when bean has no interfaces or proxyTargetClass=true
- Creates a subclass of the target class at runtime
- Can proxy classes and methods directly
- ****Limitation:**** Final classes/methods cannot be proxied

Example:

```
Enhancer enhancer = new Enhancer();  
enhancer.setSuperclass(MyClass.class);  
enhancer.setCallback(advice);  
MyClass proxy = (MyClass) enhancer.create();
```


Configuring Proxy Type

- Annotation:

`@EnableAspectJAutoProxy(proxyTargetClass = true/false)`

- XML:

`<aop:config proxy-target-class="true|false"/>`

Determining Proxy Strategy

1. If proxyTargetClass=true → CGLIB proxy always
2. Else if bean implements interfaces → JDK Dynamic Proxy
3. Else → CGLIB proxy

Pros & Cons

- ****JDK Proxies:****
 - ✓ Interface enforcement
 - ✗ Cannot proxy classes
- ****CGLIB Proxies:****
 - ✓ Class proxying
 - ✗ Cannot proxy final classes/methods
 - Slightly more complex and heavier

Summary

- Spring uses JDK proxies by default when interfaces are present
- Use `proxyTargetClass` to force CGLIB
- Choose based on your bean design and AOP needs
- Understanding proxies helps debug AOP and transaction behavior

Spring Annotation Essentials

Overview of key annotations for
bean selection, ordering, and
conditional configuration.

@Primary

- Marks a bean as the default when multiple candidates exist.
- Ideal for choosing one implementation.

Example:

```
@Component
```

```
@Primary
```

```
public class DefaultPaymentService implements  
PaymentService {}
```

@Qualifier

- Disambiguates beans by name or custom qualifier.
- Use when multiple beans implement the same interface.

Example:

```
public class OrderService {  
    @Autowired  
    @Qualifier("creditCardService")  
    private PaymentService paymentService;  
}
```

@Order

- Specifies ordering for collections or aspects.
- Lower values have higher priority.

Example:

```
@Component
```

```
@Order(1)
```

```
public class FirstFilter implements Filter {}
```


@ConditionalOnBean

- Registers a bean only if another bean exists.
- Often used in auto-configuration.

Example:

```
@Bean
```

```
@ConditionalOnBean(DataSource.class)
```

```
public JdbcTemplate jdbcTemplate(DataSource ds) {  
    return new JdbcTemplate(ds);  
}
```

@ConditionalOnProperty

- Registers a bean based on a property value.
- Enables feature toggles via config.

Example:

```
@Bean
```

```
@ConditionalOnProperty(name =  
"feature.x.enabled", havingValue = "true")
```

```
public FeatureX featureX() {  
    return new FeatureX();  
}
```

@ConditionalOnClass & @ConditionalOnMissingBean

- @ConditionalOnClass: only if a class is on the classpath.
- @ConditionalOnMissingBean: only if a bean is NOT present.

Example:

```
@Configuration
```

```
@ConditionalOnClass(name =  
"com.google.gson.Gson")
```

```
public class GsonConfig {}
```

Ex 04

- Create a Pizza Service interface with multiple implementations
- Inject (using constructor injection) all implementations into the Pizza Controller and refactor methods to take all Pizza Services into account.
- Inject exactly one Pizza Service as a “fast pizza service”. Use a new endpoint to retrieve a pizza only from this service
- All generated spring proxies should be JDK proxies