List Of Annotations:

1. Core Annotations

- @SpringBootApplication
- @ComponentScan
- @Confi
- guration
- @Bean
- @Value
- @PropertySource

2. Dependency Injection

- @Autowired
- @Qualifier
- @Primary
- @Inject
- @Resource

3. Stereotype Annotations

- @Component
- @Service
- @Repository
- @Controller
- @RestController

4. Web Annotations (Spring MVC)

- @RequestMapping
- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping
- @PatchMapping
- @PathVariable
- @RequestParam
- @RequestBody
- @ResponseBody
- @ModelAttribute
- @CrossOrigin

5. Validation

- @Valid
- @NotNull
- @Size
- @Min
- @Max
- @Pattern

6. JPA / Database

- @Entity
- @Table
- @Id
- @GeneratedValue
- @Column
- @Repository
- @EnableJpaRepositories
- @Transactional
- @Modifying
- @Query

7. Spring Boot Features

- @EnableAutoConfiguration
- @EnableScheduling
- @Scheduled
- @EnableAsync
- @Async
- @EnableCaching
- @Cacheable
- @CachePut
- @CacheEvict
- @ConditionalOnProperty
- @ConditionalOnClass

8. Spring Security

- @EnableWebSecurity
- @PreAuthorize
- @PostAuthorize
- @Secured
- @WithMockUser

9. Testing

- @SpringBootTest
- @WebMvcTest
- @DataJpaTest
- @MockBean
- @TestConfiguration

10. Miscellaneous

- @Profile
- @Scope
- @Import
- @EnableConfigurationProperties
- @ConfigurationProperties

Core Annotations

1. @SpringBootApplication

What:A convenience annotation that combines three annotations:@Configuration + @EnableAutoConfiguration + @ComponentScan.

Why: It simplifies the configuration of a Spring Boot application by enabling auto-configuration, component scanning, and allowing beans definition.

When: Used on the main class of a Spring Boot application.

Where: At the entry point (main class) of a Spring Boot project.

Uses:

- Bootstraps the application
- Automatically configures Spring context based on dependencies
- Scans components in the package and subpackages

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

```
@SpringBootApplication

public class MyApp {

    public static void main(String[] args) {

        SpringApplication.run(MyApp.class, args);

    }

}
```

2. @ComponentScan

What: Tells Spring where to look for components, configurations, and services.

Why: To specify packages to scan for Spring-managed components like @Component, @Service, @Repository, @Controller.

When: When your components are not in the default package or you want to scan specific packages.

Where: On a configuration class, often the main application class or a separate config class.

Uses:

- Defines base packages for scanning
- Controls component scanning scope
- Helps modularize Spring apps

```
import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;


@Configuration

@ComponentScan(basePackages = {"com.example.services", "com.example.controllers"})

public class AppConfig {

}
```

3.@Configuration

What: Indicates that the class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions.

Why: To define beans via Java code instead of XML.

When: When you want to configure beans programmatically.

Where: On classes that contain bean definitions.

Uses:

- Defines application beans
- Can import other configurations
- Alternative to XML-based configuration

```java
import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Bean;

@Configuration

public class AppConfig {


    @Bean

    public MyService myService() {

        return new MyService();

    }

}
```

4. @Bean

What: Marks a method as producing a bean to be managed by the Spring container.

Why: To explicitly declare a single bean in a @Configuration class.

When: When you want fine control over bean creation or instantiate beans from third-party classes.

Where: Inside a @Configuration-annotated class.

Uses:

- Custom bean creation
- Dependency injection of third-party or complex beans
- Define prototype or singleton beans

```java
@Bean

public MyRepository myRepository() {

    return new MyRepositoryImpl();

}
```

5. @Value

What: Injects values into fields, constructor arguments, or methods from property files or environment variables.

Why: To externalize configuration and make apps configurable without code changes.

When: When you want to inject property values into Spring-managed beans.

Where: On fields, setters, or constructor parameters.

Uses:

- Inject values from application.properties or application.yml
- Inject environment variables or system properties
- Customize configuration at runtime

```java
import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;
```

```
@Component

public class AppProperties {


    @Value("${app.name}")

    private String appName;


    public String getAppName() {

        return appName;

    }

}
```

application.properties

app.name=My Spring Boot App

6. @PropertySource

What: Specifies the property file location to be loaded into Spring's Environment.

Why: To add external .properties or .yml files besides the default ones.

When: When you want to load additional configuration files.

Where: On a @Configuration class.

Uses:

- Load custom property files
- Support multiple config files for different profiles/environments
- Manage environment-specific settings


```
import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

@Configuration

@PropertySource("classpath:custom.properties")

public class PropertyConfig {

}
```

Dependency Injection Annotations in Spring Boot

1. @Autowired

- **What:**Automatically injects a bean by type.
- **Why:**To tell Spring to resolve and inject collaborating beans into your class.
- **When:**When you want Spring to inject dependencies automatically, e.g., in fields, constructors, or setter methods.

- **Where:** On a field, constructor, or setter method inside Spring-managed beans.

@Component

 public class UserService{

@Autowired

private UserRepository userRepository;

@Autowired

 public UserService(UserRepository userRepository){

this.userRepository = userRepository;

} }

2. @Qualifier

- **What:** Used along with @Autowired to specify which bean to inject when multiple candidates exist.
- **Why:** To disambiguate injection when multiple beans of the same type exist.
- **When:** When you have multiple implementations of an interface or multiple beans of the same type.
- **Where:** On the injection point (field, constructor, or setter).

@Component

public class UserService

{

 @Autowired

@Qualifier("mysqlUserRepository")

private UserRepository userRepository; }

@Component("mysqlUserRepository")

 public class MySqlUserRepositoryimplements UserRepository{ // implementation}

 @Component("mongoUserRepository")

public class MongoUser Repository implements UserRepository{ // implementation}

3. @Primary

- **What:** Marks a bean as the default choice for autowiring when multiple candidates are present.
- **Why:** To avoid using @Qualifier everywhere by designating one bean as the primary.
- **When:** When multiple beans exist, but one should be the default.
- **Where:** On the bean class or bean method.
- **Example:**

@Component

@Primary

public class MySqlUserRepository implements UserRepository{ // default implementation}

 @Component

public class MongoUserRepository implements UserRepository{ // alternative implementation}

4. @Inject (from javax.inject)

- **What:** Standard Java CDI annotation equivalent to @Autowired.
- **Why:** To follow Java Dependency Injection standards.
- **When:** Can be used instead of @Autowired for injection.
- **Where:** On fields, constructors, or setters.
- **Example:**

import javax.inject.Inject;

@Component

public class UserService{

@Inject

private UserRepository userRepository; }

Works similarly to @Autowired.

5. @Resource (from javax.annotation)

- **What:** Injects by bean **name** by default, unlike @Autowired which injects by type.
- **Why:** Useful when you want to inject a specific bean by its name.
- **When:** When injection by name is preferred or required.
- **Where:** On fields or setter methods.

import javax.annotation.Resource;

@Component

 public class UserService{

@Resource(name = "mysqlUserRepository")

private UserRepository userRepository; }

Stereotype Annotations

1. @Component

what: Generic stereotype for any Spring-managed component.

Why: To declare a class as a Spring bean when no more specific stereotype (@Service, @Repository, @Controller) applies.

When/Where: Use on "utility" or "helper" classes, or any component that doesn't fit the other stereotypes.

Uses:

- Automatic detection via component scanning
- Base for custom stereotypes (metaannotation)

import org.springframework.stereotype.Component;

@Component

public class AuditLogger{ public void log(String msg){ System.out.println("[AUDIT] "+ msg); } }

2. @Service

What: Specialized @Component for the service layer.

Why: To semantically mark businesslogic classes and allow for any servicespecific processing (e.g., AOP).

When/Where: On classes that implement business rules, orchestrate calls to repositories or external APIs.

Uses:

- Clarifies intent in your codebase
- Can be targeted by pointcuts (e.g., logging, transactions)

import org.springframework.stereotype.Service;

@Service

 public class OrderService{ private final OrderRepository repo;

public OrderService(OrderRepository repo){ this.repo = repo; }

public void placeOrder(Order o){ // business logic, validation, etc.repo.save(o); } }

3. @Repository

What: Specialized @Component for the persistence (DAO) layer.

Why: Enables translation of persistence exceptions into Spring's DataAccessException hierarchy.

When/Where: On classes that interact with the database or external data sources.

Uses:

- Exception translation
- Clear separation of dataaccess logic

import org.springframework.stereotype.Repository; import org.springframework.data.jpa.repository.JpaRepository;

@Repository

public interface OrderRepository extends JpaRepository<Order, Long> { // Spring Data JPA will autoimplement CRUD methods}

4. @Controller

What: Specialized @Component for MVC controllers.

Why: To mark classes that handle web requests (returning views).

When/Where: On Spring MVC controller classes that produce HTML (Thymeleaf, JSP, etc.).

Uses:

- Map HTTP routes to handler methods
- Return view names (templates)

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

 import org.springframework.web.bind.annotation.GetMapping;

```
@Controller

public class WebController

{

 @GetMapping("/welcome")

public String welcomePage(Model model)

{

model.addAttribute("msg", "Welcome!");

return" welcome"; // resolves to src/main/resources/templates/welcome.html} }
```

5. @RestController

What: Combines @Controller + @ResponseBody.

Why: To simplify creation of RESTful web services by automatically serializing return values as JSON/XML.

When/Where: On classes that expose REST endpoints.

Uses:

- Build APIs that return data payloads
- No need to annotate each method with @ResponseBody

```
import org.springframework.web.bind.annotation.RestController;

 import org.springframework.web.bind.annotation.GetMapping;

 @RestController

 public class Api Controller{

@GetMapping("/api/orders")

public List<Order> getAllOrders()

{ returnList.of(newOrder(1, "Coffee"), newOrder(2, "Tea")); } }
```

Web Annotations (Spring MVC)

1. @RequestMapping

What: General-purpose annotation to map HTTP requests to handler methods or classes.

Why: To define the URL path (and optionally HTTP methods, headers, params) that a controller or method should respond to.

When/Where: At the class level (to set a base path) or method level (to handle specific endpoints).

Uses:

- Grouping related handlers under a common path
- Handling multiple HTTP methods on one method

```
@RestController

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

```java
public class UserController{

@RequestMapping(value = "/all", method = {RequestMethod.GET, RequestMethod.POST})

publicList<User> allUsers(){ // handles GET and POST on /api/users/allreturnuserService.findAll(); } }
```

2. @GetMapping

What: Shorthand for @RequestMapping(method = RequestMethod.GET).

Why: To handle HTTP GET requests more concisely.

When/Where: On methods that should respond to GET.

Uses:

- Fetching resources
- Returning pages or JSON data

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

public User getUser(@PathVariable Long id) {

returnuserService.findById(id); }
```

3. @PostMapping

What: Shorthand for @RequestMapping(method = RequestMethod.POST).

Why: To handle HTTP POST requests succinctly.

When/Where: On methods that create resources or accept form/data submission.

Uses:

- Creating new entities
- Processing form submissions

```java
@PostMapping("/users")

public User createUser(@RequestBody User newUser) {

return userService.save(newUser); }
```

4. @PutMapping

What: Shorthand for @RequestMapping(method = RequestMethod.PUT).

Why: For idempotent updates of a resource.

When/Where: On methods that fully update an existing resource.

Uses:

- Replacing an entity's state

```java
@PutMapping("/users/{id}")

public User replaceUser(@PathVariable Long id, @RequestBody User user) { user.setId(id);

 return userService.update(user); }
```

5. @DeleteMapping

What: Shorthand for @RequestMapping(method = RequestMethod.DELETE).

Why: To handle HTTP DELETE requests clearly.

When/Where: On methods that delete resources.

Uses:

- Removing entities

```
@DeleteMapping("/users/{id}")

public void delete User(

@PathVariable

Long id) { userService.delete(id); }
```

6. @PatchMapping

What: Shorthand for @RequestMapping(method = RequestMethod.PATCH).

Why: For partial updates to a resource.

When/Where: On methods that apply partial modifications.

Uses:

- Patching a subset of fields

```
@PatchMapping("/users/{id}")

public User updateUserEmail(@PathVariable Long id, @RequestBody Map<String, Object> updates) {

 return userService.patch(id, updates); }
```

7. @PathVariable

What: Binds a URI template variable to a method parameter.

Why: To capture dynamic values from the URL path.

When/Where: On method parameters.

Uses:

- Identifying which resource to act upon

```
@GetMapping("/orders/{orderId}/items/{itemId}")

public Item getItem(@PathVariable Long orderId, @PathVariable Long itemId) {
returnorderService.getItem(orderId, itemId); }
```

8. @RequestParam

What: Binds a query parameter or form field to a method parameter.

Why: To extract request parameters into variables.

When/Where: On method parameters.

Uses:

- Filtering, pagination, optional flags

```
@GetMapping("/products")
```

```java
public List<Product> find( @RequestParam(required = false)
String category,
@RequestParam(defaultValue = "0")int page )
{ return
productService.search(category, page); }
```

9. @RequestBody

What: Binds the HTTP request body (e.g., JSON) to a Java object.

Why: To deserialize incoming payloads into domain objects.

When/Where: On method parameters.

Uses:

- ○ Accepting JSON/XML payloads for create/update

```java
@PostMapping("/login")
public Token login(@RequestBodyLoginForm form) { return authService.authenticate(form); }
```

10. @ResponseBody

What: Indicates the return value should be written directly to the response body (not a view).

Why: To serialize return values (JSON/XML) for REST endpoints.

When/Where: On methods or at the class level (alternative to @RestController).

Uses:

- ○ Building RESTful APIs without @RestController

```java
@Controller
 public class LegacyApiController{
@GetMapping("/legacy/data")
@ResponseBody
public LegacyData getData(){
return legacyService.fetch(); } }
```

11. @ModelAttribute

What: Binds request parameters to a model object, and makes it available to a view.

Why: To populate form-backing beans or add common model attributes.

When/Where: On method parameters or on methods in a controller.

Uses:

- ○ Form handling in MVC
- ○ Preloading reference data

@PostMapping("/users")

public String saveUser(@ModelAttribute UserForm form, Model model) {

userService.save(form);

return "redirect:/users"; }

@ModelAttribute("roles")

publicList<Role> populateRoles(){

 return roleService.findAll(); }

12. @CrossOrigin

What: Enables CrossOrigin Resource Sharing (CORS) on controller or method.

Why: To allow AJAX requests from different domains (e.g., frontend on localhost:3000).

When/Where: On controller classes or individual handler methods.

Uses:

- ○ Configuring allowed origins, methods, headers

@RestController

@RequestMapping("/api")

@CrossOrigin(origins = "http://localhost:3000")

public class ApiController{

@GetMapping("/data")

public Data getData(){

return data Service.get(); } }

Validation

 1. @Valid

What: Triggers validation on an object or method parameter annotated with JSR303 constraint annotations.

Why: To instruct Spring (and the underlying Validator) to check the constraints on the target object.

When/Where: On a controller method parameter (e.g., a requestbody DTO) or on a method/constructor for AOPstyle validation.

Uses:

- Validating incoming JSON/form data in REST or MVC controllers
- Enforcing constraints on service method arguments

```
@RestController

public class UserController{

@PostMapping("/users")

public ResponseEntity<?> createUser(

@Valid

@RequestBody

User Dto userDto, BindingResult result) { if (result.hasErrors()) { // handle validation errors

 return ResponseEntity.badRequest().body(result.getAllErrors()); }

userService.save(userDto);

return ResponseEntity.ok().build(); } }
```

2. @NotNull

What: Asserts that the annotated element (field, method parameter, etc.) must not be null.

Why: To ensure mandatory values are provided.

When/Where: On fields or parameters that are required.

Uses:

- Mandatory form fields
- Required JSON properties

```
public class UserDto{

@NotNull(message = "Username must be provided")

private String username; // getters/setters}
```

3. @Size

What: Asserts that the annotated element's size (String length, Collection size, array length) is within specified bounds.

Why: To constrain the length or count of values.

When/Where: On String, Collection, Map, or array fields/parameters.

Uses:

- Limiting username lengths
- Restricting list sizes

```
public class UserDto{

@Size(min = 3, max = 20, message = "Username must be 3–20 characters")

private String username;

 @Size(max = 5, message = "At most 5 roles allowed")

privateList<String> roles; // getters/setters}
```

4. @Min

What: Asserts that the annotated numeric value must be at least the specified minimum.

Why: To enforce lower bounds on numeric inputs.

When/Where: On numeric fields (int, long, BigDecimal, etc.) or their wrappers.

Uses:

- Age must be ≥ 18
- Price must be ≥ 0

```
public class ProductDto{

@Min(value = 0, message = "Price must be zero or positive")

private BigDecimal price;

 @Min(value = 1, message = "Quantity must be at least 1")

private int quantity; // getters/setters}
```

5. @Max

What: Asserts that the annotated numeric value must be at most the specified maximum.

Why: To enforce upper bounds on numeric inputs.

When/Where: On numeric fields or parameters.

Uses:

- Rating must be ≤ 5
- Quantity must not exceed stock

```
public class ReviewDto{

@Min(1)

@Max(value = 5, message = "Rating must be between 1 and 5")

private int rating; // getters/setters}
```

6. @Pattern

What: Asserts that the annotated String matches the specified regular expression.

Why: To validate format patterns (email, phone, custom syntax).

When/Where: On String fields or parameters.

Uses:

- Email address format
- Phone number or ZIP code pattern

```
public class ContactDto

{ @Pattern(regexp = "^(.+)@(.+)$", message = "Email must be valid")

private String email;

@Pattern(regexp = "\\+?[0-9]{10,15}", message = "Phone must be 10–15 digits, optional +")

private String phone; // getters/setters}
```

JPA / Database

1. @Entity

What: Marks a Java class as a JPA entity (mapped to a database table).

Why: So that the JPA provider knows to manage persistence for instances of this class.

When/Where: On any domain/model class you want to persist.

Uses:

- Enables ORM mapping
- Allows CRUD operations via EntityManager or Spring Data JPA

```
import javax.persistence.Entity;

 import javax.persistence.Id;

@Entity

 public classUser{

@Id

private Long id;

private String username; // getters/setters, constructors...}
```

2. @Table

What: Specifies the database table name (and schema, catalog) to which the entity maps.

Why: To customize table name or namespace when it differs from the class name.

When/Where: On an @Entity class.

Uses:

- ○ Mapping to existing tables
- ○ Using different naming conventions

import javax.persistence.Entity;

import javax.persistence.Table;

@Entity

@Table(name = "app_user", schema = "public")

public class User{ ... }

3. @Id

What: Marks a field as the primary key of the entity.

Why: To uniquely identify each row and allow JPA to track entity identity.

When/Where: On one field (or getter) in each @Entity.

Uses:

- ○ Required for every entity
- ○ Combined with @GeneratedValue for auto IDs

@Entity

public class Order{

@Id

private Long orderId; // ...}

4. @GeneratedValue

What: Configures automatic generation of primarykey values.

Why: To let the database or JPA provider generate unique IDs.

When/Where: On the same field as @Id.

Uses:

- ○ strategy = GenerationType.IDENTITY for autoincrement columns
- ○ SEQUENCE, TABLE, or AUTO strategies as needed

@Entity public class Order{

@Id

```
@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id; // ...}
```

5. @Column

What: Specifies mapping between a field and a database column.

Why: To customize column name, length, nullability, uniqueness, etc.

When/Where: On entity fields (or getters).

Uses:

- Renaming columns
- Defining column constraints

```
@Entity

 public class Product

{

@Id

@GeneratedValue

private Long id;

@Column(name = "prod_name", nullable = false, length = 100)

private String name;

 @Column(precision = 10, scale = 2)

private BigDecimal price; // ...}
```

6. @Repository

What: Stereotype for persistence-layer beans; also enables exception translation.

Why: To mark interfaces or classes as Spring Data repositories and translate JPA exceptions to Spring's DataAccessException.

When/Where: On interfaces extending JpaRepository or custom DAO classes.

Uses:

- Define CRUD/repository interfaces
- Autoimplement query methods

```
import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository
```

```java
public interface UserRepository extends JpaRepository<User, Long> { // query methods, e.g.
findByUsername(...)}
```

7. @EnableJpaRepositories

What: Enables scanning for Spring Data JPA repositories.

Why: To bootstrap detection and implementation of repository interfaces.

When/Where: On a configuration class (often your main @SpringBootApplication).

Uses:

- ○ Point Spring Boot to custom package locations
- ○ Configure repository settings

```java
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication

@EnableJpaRepositories(basePackages = "com.example.repo")

public class MyApp{ ... }
```

8. @Transactional

What: Declares transactional boundaries around methods or classes.

Why: To ensure ACID semantics: begin, commit, or rollback a transaction.

When/Where: On service-layer methods or classes; can also apply to repository methods.

Uses:

- ○ Group multiple repository calls in one transaction
- ○ Roll back on exceptions automatically

```java
import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public classOrderService{

@Transactional

public void placeOrder(OrderDto dto){ // multiple DB operations here} }
```

9. @Modifying

What: Indicates that a repository query method is an update/delete operation.

Why: To allow execution of modifying JPQL/SQL queries via @Query.

When/Where: On repository interface methods paired with @Query.

Uses:

- Bulk updates or deletes
- Custom DML queries

```
@Repository

public interface UserRepository extends JpaRepository<User, Long> {

@Modifying

@Query("UPDATE User u SET u.active = false WHERE u.lastLogin < :cutoff")

int deactivate Inactive(@Param("cutoff")LocalDate cutoff); }
```

Note: Methods annotated with @Modifying should also run in a transaction.

10. @Query

What: Defines a custom JPQL or native SQL query for a repository method.

Why: To express queries that cannot be derived from method names.

When/Where: On methods in a Spring Data JPA repository.

Uses:

- Complex joins, groupings, filters
- Native queries via nativeQuery=true

```
@Repository

 public interface OrderRepository extends JpaRepository<Order, Long> {

@Query("SELECT o FROM Order o JOIN FETCH o.items WHERE o.id = :id")

Optional<Order> findWithItems(

@Param("id")

Long id);

@Query(value = "SELECT * FROM orders WHERE status = ?1", nativeQuery = true)

List<Order> findByStatusNative(String status); }
```


Spring Boot Features

1. @EnableAutoConfiguration

What: Tells Spring Boot to guess and configure beans you're likely to need based on classpath settings, other beans, and various property settings.

Why: To reduce boilerplate by automatically configuring Spring features (e.g., DataSource, MVC, Jackson) when relevant dependencies are present.

When/Where: Placed on your main @Configuration class (often via @SpringBootApplication).

Uses:

- Auto-configure web server, JPA, security, etc.
- Customize or exclude auto-configurations with exclude attribute

@SpringBootApplication// includes

@EnableAutoConfiguration

public class MyApp{

public static void main(String[] args){

SpringApplication.run(MyApp.class, args); } }

Or explicitly:

@Configuration

@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)

public class AppConfig{ }

2. @EnableScheduling

What: Enables Spring's scheduled task execution capability.

Why: To allow methods annotated with @Scheduled to run on a cron, fixeddelay, or fixedrate schedule.

When/Where: On a @Configuration class in any Boot application.

Uses:

- Periodic cleanup
- Reporting jobs
- Polling external systems

@Configuration

@EnableScheduling

public class SchedulingConfig{ }

@Component

public class ReportTask{

@Scheduled(cron = "0 0 6 * * *")// every day at 6ampublicvoidgenerateDailyReport(){ // generate report} }

3. @Scheduled

What: Marks a method to be executed on a schedule.

Why: To define timing (cron, fixedDelay, fixedRate) for background tasks.

When/Where: On methods of a Spring-managed bean when @EnableScheduling is active.

Uses:

- Cleanup stale data
- Send email reminders
- Refresh caches

@Component

public class CacheRefresher{

@Scheduled(fixedRate = 300000)// every 5 minutespublicvoidrefreshCache(){ cacheService.refreshAll(); } }

4. @EnableAsync

What: Enables Spring's asynchronous method execution capability.

Why: To let methods annotated with @Async run in a separate thread pool.

When/Where: On a @Configuration class.

Uses:

- Fire-and-forget tasks
- Parallel processing
- Non-blocking I/O operations

@Configuration

@EnableAsync

public class AsyncConfig{ }

5. @Async

What: Indicates that a method should execute asynchronously.

Why: To free up the caller thread and run timeconsuming tasks in the background.

When/Where: On public methods of Spring-managed beans when @EnableAsync is active.

Uses:

- Sending emails
- Long-running computations
- Calling remote services

@Service

public class NotificationService{

@Async

public   CompletableFuture<Void> sendEmail(String to, String msg){ mailClient.send(to, msg);

return

CompletableFuture.completedFuture(null); } }

6. @EnableCaching

What: Enables Spring's annotation-driven cache management capability.

Why: To allow methods annotated with @Cacheable, @CachePut, or @CacheEvict to interact with a cache abstraction.

When/Where: On a @Configuration class.

Uses:

- ○ Improve performance by caching expensive method results

@Configuration

@EnableCaching

public class CacheConfig{ }

7. @Cacheable

What: Indicates that the result of a method should be stored in a cache.

Why: To avoid recomputing or reloading data on subsequent calls with the same parameters.

When/Where: On methods of Spring-managed beans when caching is enabled.

Uses:

- ○ Caching DB queries
- ○ Caching remote API calls

@Service

public class ProductService{

@Cacheable("products")

public Product findById(Long id){

returnrepository.findById(id).orElse(null); } }

8. @CachePut

What: Updates (or populates) the cache without interfering with the method execution.

Why: To refresh the cache entry with the method's latest return value.

When/Where: On methods that should always run and then update the cache.

Uses:

- Save or update operations

```
@Service

 public class ProductService{

@CachePut(value = "products", key = "#product.id")

public Product update(Product product){ return

repository.save(product); } }
```

9. @CacheEvict

What: Removes one or more entries from the cache.

Why: To keep the cache in sync with the underlying data when it changes.

When/Where: On methods that perform deletes or bulk updates.

Uses:

- Deleting stale cache after removal
- Clearing all entries on major updates

```
@Service

public class ProductService{

 @CacheEvict(value = "products", key = "#id")

public void deleteById(Long id){ repository.deleteById(id); }

@CacheEvict(value = "products", allEntries = true)

public void clearAllCache(){ } }
```

10. @ConditionalOnProperty

What: Enables a bean only if a specified property has a particular value (or is defined).

Why: To conditionally include features based on configuration.

When/Where: On @Configuration classes or @Bean methods.

Uses:

- Enable/disable auto-configuration
- Feature toggles

```
@Configuration

public class FeatureConfig{
```

```
@Bean

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

public FeatureXService featureXService(){ return newFeatureXService(); } }
```

11. @ConditionalOnClass

What: Enables a bean only if a certain class is present on the classpath.

Why: To auto-configure optional features only when their dependencies are available.

When/Where: On @Configuration classes or @Bean methods.

Uses:

- Integrate with thirdparty libraries if present
- Guard against missing dependencies

```
@Configuration

public class OptionalConfig{

@Bean

@ConditionalOnClass(

name = "com.example.ExternalClient")

public  ExternalClientService externalClientService(){

returnnewExternalClientService(); } }
```

Spring Security

1. @EnableWebSecurity

What: Enables Spring Security's web security support and provides the Spring MVC integration.

Why: To activate the WebSecurityConfigurerAdapter (or SecurityFilterChain bean) that you define in your configuration.

When/Where: On a @Configuration class in your application.

Uses:

- Hook into HTTP security setup
- Customize authentication, authorization, CORS, CSRF, session management, etc.
-

```
importorg.springframework.context.annotation.Configuration;
importorg.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
importorg.springframework.security.config.annotation.web.builders.HttpSecurity;
importorg.springframework.security.web.SecurityFilterChain;
importorg.springframework.context.annotation.Bean;

 @Configuration

@EnableWebSecurity
```

```
public class SecurityConfig{

 @Bean

public SecurityFilterChain filterChain(HttpSecurity http)throwsException { http .authorizeHttpRequests(auth ->
auth .antMatchers("/admin/**").hasRole("ADMIN") .anyRequest().authenticated() ) .formLogin();
returnhttp.build(); } }
```

2. @PreAuthorize

What: Evaluates a SpEL expression **before** a method is invoked to decide if access is allowed.

Why: To apply finegrained, methodlevel security based on roles, permissions, or properties of method arguments.

When/Where: On service or controller methods.

Uses:

- o Check user roles or authorities
- o Inspect method arguments (e.g. #id == principal.id)
- o Combine conditions (hasRole('ADMIN') and #dto.owner == principal.username)

```
import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.stereotype.Service;

 @Service

public class ReportService{

@PreAuthorize("hasRole('MANAGER')")

public Report generateReport(ReportRequest dto){ // only managers can executereturnrepository.create(dto);
} }
```

3. @PostAuthorize

What: Evaluates a SpEL expression **after** the method has been executed, allowing you to inspect the return value.

Why: To make access decisions based on the method's result (e.g., only return if the user owns the returned object).

When/Where: On service or controller methods.

Uses:

- o Filter or reject based on returned data
- o Protect data leaks (e.g., only allow users to see their own orders)

```
import org.springframework.security.access.postpostauthorize.PostAuthorize;

 import org.springframework.stereotype.Service;

@Service

public class OrderService{

 @PostAuthorize("returnObject.customer == authentication.name")
```

```java
public  Order findOrder(Long id){

return  repository.findById(id).orElseThrow(); } }
```

4. @Secured

What: Specifies a list of roles (authorities) that are allowed to invoke the method.

Why: To apply simple rolebased methodlevel security without SpEL.

When/Where: On service or controller methods (or class level).

Uses:

- Quick, declarative role checks
- Legacy applications or when SpEL isn't needed

```java
import org.springframework.security.access.annotation.Secured;

 import org.springframework.stereotype.Service;

@Service

public  class UserService{ @Secured({"ROLE_ADMIN", "ROLE_MANAGER"})

public void deleteUser(Long userId){ repository.deleteById(userId); } }
```

5. @WithMockUser

What: Sets up a mock user in the SecurityContext for testing secured methods or controllers.

Why: To simplify writing unit or integration tests for securityprotected code without needing a real authentication flow.

When/Where: On JUnit test methods or test classes.

Uses:

- Test access control rules
- Simulate users with different roles/authorities

```java
import org.junit.jupiter.api.Test; importorg.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest; import org.springframework.security.test.context.support.WithMockUser;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.beans.factory.annotation.Autowired;

 @WebMvcTest(AdminController.class)

public classAdminControllerTest{

@Autowired

private MockMvc mvc;

@Test

@WithMockUser(username = "alice", roles = {"ADMIN"})

public void adminEndpointAccessibleToAdmin()throwsException { mvc.perform(get("/admin/dashboard"))
.andExpect(status().isOk()); }
```

```
@Test

@WithMockUser(username = "bob", roles = {"USER"})

public void  adminEndpointForbiddenForUser()throwsException { mvc.perform(get("/admin/dashboard"))
.andExpect(status().isForbidden()); } }
```

Testing

1. @SpringBootTest

What: Boots the full Spring application context for integration tests.

Why: To test endtoend behaviors—including full configuration, filters, controllers, services, repositories, and any autoconfigured beans.

When/Where: On a test class that needs the complete Spring context (e.g., multilayer integration tests).

Uses:

- Verify that all beans load correctly
- Test REST endpoints with TestRestTemplate or full MockMvc setup
- Real database access (often with an embedded DB)

```
import org.springframework.boot.test.context.SpringBootTest;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.web.client.TestRestTemplate;

import staticorg.assertj.core.api.Assertions.assertThat;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)

public classApplicationIntegrationTest{

@Autowired

private TestRestTemplate restTemplate;

@Test void contextLoadsAndEndpointWorks(){ Stringbody=restTemplate.getForObject("/api/health",
String.class); assertThat(body).contains("UP"); } }
```

2. @WebMvcTest

What: Loads only Spring MVC components (controllers, controller advice, filters) for focused weblayer tests.

Why: To test controller logic in isolation, without starting the full application or loading unrelated beans.

When/Where: On a slice test class targeting one or more controllers.

Uses:

- Mocked service/repository dependencies via @MockBean
- Exercise request mappings, validation, serialization, error handling

```java
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.mock.mockito.MockBean;

import staticorg.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import staticorg.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
@WebMvcTest(UserController.class)

publicclassUserControllerTest{

@AutowiredprivateMockMvc mvc;

@MockBeanprivateUserService userService;

@Test

void getUser Returns Json()throwsException { when(userService.findById(1L)) .thenReturn(newUser(1L,
"alice"));

mvc.perform(get("/users/1")) .andExpect(status().isOk()) .andExpect(jsonPath("$.username").value("alice")); } }
```

3. @DataJpaTest

What: Bootstraps an embeddeddatabase JPA test slice: configures entities, repositories, and DataSource (no web layer).

Why: To test JPA repositories in isolation with an inmemory database.

When/Where: On test classes that focus on repository methods or JPA mappings.

Uses:

- Verify query methods
- Test custom @Query behavior
- Validate correct schema generation and constraints

```java
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import org.springframework.beans.factory.annotation.Autowired;

import staticorg.assertj.core.api.Assertions.assertThat;

@DataJpaTest

public class UserRepositoryTest{

@Autowired

private UserRepository userRepository;

@Test

void saveAndFind(){

User=new User(null, "bob");
```

```java
userRepository.save(u);

Userfound=userRepository.findByUsername("bob");

assertThat(found).isNotNull(); assertThat(found.getUsername()).isEqualTo("bob");

}}
```

4. @MockBean

What: Adds a Mockito mock of a Spring bean into the application context.

Why: To replace real beans (services, repositories, etc.) with mocks during a slice test (

@WebMvcTest, @SpringBootTest).

When/Where: On fields in test classes using any Springboot test slice.

Uses:

- Stub out dependencies
- Control interactions and verify behavior without hitting real resources

```java
@WebMvcTest(OrderController.class)

public class OrderControllerTest{


@Autowired private MockMvc mvc;

@MockBean

private OrderService orderService; // injected mock

@Test

void  getOrder()throwsException

{

when(orderService.find(1L)).thenReturn(newOrder(1L, "Book"));

mvc.perform(get("/orders/1")) .andExpect(status().isOk()) .andExpect(jsonPath("$.item").value("Book"));

}}
```

5. @TestConfiguration

What: Defines a nested configuration class for tests, whose beans are only available in the test context.

Why: To provide additional or override beans specifically for testing scenarios.

When/Where: As a static inner class inside your test, alongside your test annotations.

Uses:

- Supply testspecific beans (e.g., a stub implementation)
- Override production beans without affecting the main context

@SpringBootTest

```java
public class PaymentServiceIntegrationTest{

@TestConfiguration

static class PaymentTestConfig{

@Bean

 public ExternalPaymentClient paymentClient(){ // return a stub or fake client for tests return new StubPaymentClient(); } }

@Autowired

privatePaymentService paymentService;

@Test

void processPaymentUsesStubClient(){

booleanresult=paymentService.process(100); assertThat(result).isTrue(); } }
```

Miscellaneous

1. @Profile

What: Specifies that a component or configuration is active only for one or more named Spring profiles.

Why: To load beans selectively based on the current runtime environment (e.g., dev, test, prod).

When/Where: On @Component or @Configuration classes (or on individual @Bean methods).

Uses:

- ○ Environmentspecific beans (different DataSources, endpoints, etc.)
- ○ Clean separation of config across stages

```java
@Configuration

@Profile("dev")

public class DevDataSourceConfig{

@Bean

public DataSource dataSource(){ // embedded or inmemory DataSource for development

return new EmbeddedDatabaseBuilder() .setType(EmbeddedDatabaseType.H2) .build(); } }
```

2. @Scope

What: Defines the lifecycle scope of a Spring bean (singleton, prototype, request, session, etc.).

Why: To control how and when bean instances are created and shared.

When/Where: On @Component, @Service, @Configuration classes or @Bean methods.

Uses:

- ○ singleton (default) for one shared instance

- prototype for new instance on each injection
- Web scopes (request, session) in web apps

@Component

@Scope("prototype")

public class TaskProcessor{ // each injection yields a new TaskProcessor instance}

3. @Import

What: Imports additional configuration classes (or ImportSelector/ImportBeanDefinitionRegistrar implementations) into the current Spring context.

Why: To modularize configuration and assemble a context from multiple config classes.

When/Where: On a @Configuration class.

Uses:

- Bring in thirdparty or shared configurations
- Split large configs into focused classes

@Configuration

@Import

({DataSourceConfig.class, SecurityConfig.class})

public class AppConfig{ // beans from DataSourceConfig and SecurityConfig are now included}

4. @EnableConfigurationProperties

What: Enables support for @ConfigurationProperties–annotated beans, binding external properties to them.

Why: To activate and register @ConfigurationProperties classes without needing @Component on each.

When/Where: On a @Configuration class (often the main application class).

Uses:

- Cleanly map groups of related properties into POJOs

@Configuration

@EnableConfigurationProperties(MailProperties.class)

public class AppConfig{ // MailProperties bean is registered and populated from application.properties}

5. @ConfigurationProperties

What: Binds external configuration (properties or YAML) to a structured POJO.

Why: To group and typesafe external settings (prefixbased) into beans.

When/Where: On a POJO class, optionally with @Component (or registered via @EnableConfigurationProperties).

Uses:

- Organize settings by feature (mail, cache, API clients)
- Validate property values with JSR303 annotations

```
import org.springframework.boot.context.properties.ConfigurationProperties;

import javax.validation.constraints.NotEmpty;

@ConfigurationProperties(prefix = "mail")

public class MailProperties{

@NotEmpty

private String host;

private int port=25; // getters & setterspublicString getHost(){ returnhost; }

public void setHost(String host){ this.host = host; }

public int  getPort(){ returnport; }

 public void setPort(intport) { this.port = port; } }
```

application.properties

ini

mail.host=smtp.example.com mail.port=587