# Spring Overview

[Spring Framework Documentation](https://docs.spring.io/spring-framework/docs/current/reference/html/)

[Spring IoC Container and Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-introduction)

* + - Spring IoC Container implements ApplicationContext interface which is a sub-interface of BeanFactory
  + BeanFactory provides the configuration framework and basic functionality
  + ApplicationContext adds more enterprise-specific functionality
* A bean is an object that is instantiated, assembled and managed by the Spring IoC container
* Several implementations of the ApplicationContext are supplied with Spring:
  + ClassPathXmlApplicationContext
  + FileSystemXmlApplicationContext
  + AnnotationConfigApplicationContext
  + AnnotationConfigWebApplicationContext
  + XmlWebApplicationContext
* Configuration
  + Java configuration (Explicit bean definition)
    - Uses @Configuration on your configuration class and @Bean on methods in the configuration class
    - To avoid a @Configuration class from getting too big, use multiple config files combined with @Import
    - Best Practice – keep application config and infrastructure config separate
    - Spring has an Environment abstraction that loads config properties from different property sources: JVM properties, environment variables, properties files, etc.
    - Can inject Environment just like any other bean then call its getProperty method
    - Add @PropertySource to your config class to tell Spring to load values into Environment (JVM System Properties and Environment variables are always loaded automatically)
      * @PropertySource supports resource prefixes: classpath:, file:, http:
    - Use @Value to access environment bean: @Value(“${db.driver}”)
  + Annotation configuration (Implicit bean definition)
    - Uses @Configuration and @ComponentsScan on your configuration class and @Component on your beans and @Autowired where you need to inject
    - @Component stereotypes: @Service, @Repository, @Controller, @RestController
    - @Autowired dependencies are required by default but can use @Autowired(required=false) or Optional<AccountService>
    - Autowiring and disambiguation:
      * Autowire resolution rules:
        1. Find unique bean of the required type
        2. Use @Qualifier if supplied

@Component(“jpaAccountRepository”)

@Qualifier(“jpaAccountRepository”) where you’re injecting

* + - * 1. Try to find a matching bean by name
    - @Lazy beans are created first time used
    - @Scope(“prototype”)
  + Profiles
    - Beans with no profile are always included
    - Add @Profile to your configuration class then everything in that configuration class belongs to that profile
    - Add @Profile to @Bean methods
    - Can use ! in profile names
    - Profiles must be activated at runtime:
      * -Dspring.profiles.active=embedded,jpa
      * System.setProperty(“spring.profiles.active”, “embedded,jpa”)
      * Integration tests only: @ActiveProfiles
  + SpEL
    - Can access other beans - @Value(“#{strategyBean.keyGenerator}”)
    - Can access environment properties - @Value(“#{environment[‘daily.limit’]}”)
    - Properties are always strings so you may have to cast
    - Fallback values:
      * @Value(“${daily.limit : 100000}”)
      * @Value(“#{environment[‘daily.limit’] ?: 100000)
* Bean Scopes:
  + Singleton – (default) scopes 1 bean instance for each Spring IoC container
  + Prototype – new instance each time the type is requested – unlike other scopes, Spring does not manage the complete lifecycle of a prototype bean because it has no other record of the bean once it returns it to the requesting client
  + Request – scopes 1 bean instance to the lifecycle of a single HTTP request (must be web-aware ApplicationContext)
  + Session – scopes 1 bean instance to the lifecycle of an HTTP session (must be web-aware ApplicationContext)
  + Application – scopes 1 bean instance to the lifecycle of a ServletContext (must be web-aware ApplicationContext)
  + WebSocket – scopes 1 bean instance to the lifecycle of a WebSocket (must be web-aware ApplicationContext)
* Lifecycle Callbacks:
  + @PostConstruct – attribute to identify the bean initialization method (can also be done with xml config init-method attribute or Java config initMethod attribute of @Bean)
    - Constructor injection 🡪 Setter Injection 🡪 @PostConstruct method
  + @PreDestroy – attribute to identify the bean dispose method that gets called when the ConfigurableApplicationContext is closed (can also be done with xml config destroy-method attribute or destroyMethod attribute of @Bean)
  + Use @PostConstruct/@PreDestroy for your own classes
  + Use initMethod/destroyMethod attributes of @Bean for classes you didn’t write and can’t annotate
  + You can configure the Spring container to look for named initialization and destroy callback methods on every bean without having to configure them for each bean definition.
* Container Extension Points
  + Customize beans with BeanPostProcessor interface - defines callback methods that you can implement to provide your own (or override the container’s defaults) instantiation logic, etc.
    - Provides postProcessBeforeInitialization and postProcessAfterInitialization methods
    - Both return Object so you can return a totally different bean instance if you want
  + Customize configuration metadata with BeanFactoryPostProcessor interface – lets you read configuration metadata and potentially change it before the container instantiates any beans other than BeanFactoryPostProcessor instances
    - Recommended to use a static @Bean method to define BeanFactoryPostProcessor beans (they need to run before any beans are created)
* AOP
  + Concepts
    - Join Point – a point in the execution of a program such as a method call or exception thrown
    - Pointcut – an expression that selects one or more Join Points
    - Advice – Code to be executed at each selected Join Point (e.g. @Before, @AfterReturning, @AfterThrowing, etc)
    - Aspect – A module that encapsulates pointcuts and advice
    - Weaving – Technique by which aspects are combined with main code
  + AOP Proxy – an “enhanced” class that stands in place of your original with extra behavior (Aspect) added (woven) into it
  + Enable AOP by adding @EnableAspectJAutoProxy to your configuration class then add @Aspect on your aspect beans
    - Within your @Aspect class add @Before to a method then add JoinPoint as a parameter or (ProceedingJoinPoint in the case of @Around)
  + Types of Advice:
    - @Before – runs before target execution and target is not called if advice throws an exception
    - @AfterReturning with the returning attribute - add a parameter that accepts the returning value
    - @AfterThrowing with the throwing attribute – invokes advice if the right exception is thrown – doesn’t stop the exception from propagating but can throw a different type of exception
    - @After – called regardless of whether an exception has been thrown by the target or not
    - @Around – intercepts call and allows you to choose to bypass it or proceed with it by calling the proceed method on the ProceedingJoinPoint arg
  + Limitations:
    - Can only advise non-private methods
    - Can only apply aspects to Spring Beans
    - Advice only called when method is called from outside (i.e. calls into proxy)
* Spring Testing
  + @ExtendWith(SpringExtension.class) on test classes – replaces Junit 4’s @RunWith. Also add @ContextConfiguration(classes={SystemTestConfig.class}).
  + @SpringJUnitConfig(SystemTestConfig.class) combines:
    - @ExntedWith(SpringExtension.class)
    - @ContextConfiguration(classes={SpringTestConfig.class})
  + Can use @Autowired on method arguments (ONLY in tests)
  + Classes with @SpringJUnitConfig will automatically look for a nested configuration class
  + @DirtiesContext on a test method forces the ApplicationContext to be closed at the end of the test method. Allows testing @PreDestroy behavior.
  + @TestPropertySource allows you to specify location of one or more properties files to load for the test (defaults to looking for [classname].properties)
  + @ActiveProfiles can be applied to a test class to only instantiate beans in those profiles or beans that have no profile
  + Testing databases:
    - Can use @Sql(“/testfiles/test-data.sql”) on a test method to run a script before a that test method executes (executionPhase attribute can be used to run it after the test method). Could also apply this to a test class so that it applies to all methods in the class except those with their own @Sql.
* Spring JDBC
  + JdbcTemplate handles all the boilerplate code for you then provides callbacks that allow you to implement your logic. It catches vendor-specific exceptions then wraps them in more general runtime exceptions.
  + JdbcTemplate is thread safe so create it once and reuse it.
  + Methods:
    - Querying for simple Java types:  
      queryForObject(  
       “select count(\*) from t where Value > ?”,  
       Long.class,  
       10)
    - Writes (for ANY non-SELECT SQL – insert, update, delete):  
      update(“insert into T(Name) values(?)”, name)
    - Generic query for a single row:  
      queryForMap(…) returns a map/dictionary of column name/column value pairs
    - Generic query for multiple rows:  
      queryForList(…) returns a list of maps/dictionaries
    - RowMapper<T> Interface (or Lambda) allows mapping of each row to a domain object.  
      T mapRow(ResultSet rs, int rowNum) throws SQLException
    - ResultSetExtractor<T> Interface (or Lambda) allows processing of an entire ResultSet.  
      T extractData(ResultSet rs) throws SQLException, DataAccessException
    - Query for single domain object:  
      queryForObject(  
       “select Name from T where Id = ?”,  
       (rs, rowNum) -> new Person(rs.getString(“Name”)),  
       10)
    - Query for multiple domain objects:  
      query(  
       “select Name from T”,  
       (rs, rowNum) -> new Person(rs.getString(“Name”)))
    - Spring Transactions
    - To use:
    - Declare a PlatformTransactionManager bean
    - Add @EnableTransactionManagement to a configuration class
    - Declare transactional methods using @Transactional or programmatically
    - Spring provides several PlatformTransactionManager implementations:
    - DataSourceTransactionManager
    - JmsTransactionManager
    - JpaTransactionManager
    - JtaTransactionManager
    - WebLogicJtaTransactionManager
    - WebSphereUowTransactionManager
    - And many more
    - @Transactional creates a proxy that uses an @Around advice to:
    - Start a transaction before the method is called
    - Commits the transaction after the method is called
    - Rolls back the transaction if a RuntimeException is thrown
    - Rollback rules can be specified to change the rollback logic:  
      @Transactional(rollbackFor=MyCheckedException.class, noRollbackFor={JmxException.class, MailException.class})
    - DataSourceUtils.getConnection(dataSource) will get you the transaction context of the current thread
    - There are 7 levels of transaction propagation with the default being REQUIRED meaning it joins an existing transaction or creates a new transaction if one does not exist. Can be set with:  
      @Transactional(propagation=Propagation.REQUIRES\_NEW)
    - @Transactional in an integration test will roll back your transaction after each test method completes. Can add @Commit to a test method to override this behavior.
    - Spring Boot
    - Takes an “opinionated” view of the Spring platform and third-party libraries but defaults can be easily overridden
    - Provides lots of non-functional features that are common to a large set of projects
    - POM file:
    - spring-boot-starter-parent defines Maven plugins, sets up Java version, defines properties for dependencies like ${spring-framework.version}
    - spring-boot-starter brings in multiple coordinated dependencies (16 JARs!) including transitive dependencies (version is not needed because it’s defined by spring-boot-starter-parent)
    - The spring-boot-maven-plugin will produce a “fat JAR” that contains all dependencies for your web app including Tomcat so it can be run directly using the java -jar command
    - YourApp-1.0.0.jar.original contains only your code (a traditional JAR)
    - YourApp-1.0.0.jar contains your code and all dependencies (a fat JAR)
    - Properties:
    - Spring boot looks for application.properties in these locations (in this order) then it creates a PropertySource based on these files:
    - /config sub-directory of the working directory
    - The working directory
    - Config package in the classpath
    - Classpath root
    - It also looks for profile-specific properties in files following the application-{profile}.properties naming convention
    - Also supports yaml file extensions (Spring framework and @PropertySource do not support YAML config files)
    - Can add multiple profile-specific properties in a single file, separating each with a “---” line
    - Instead of using string expressions in @Value we can bind properties to a class:
    - Add @Component and @ConfigurationProperties(prefix=”some.app”) to the class that will contain the properties
    - Add @EnableConfigurationProperties(MySettings.class) and @ConfigurationPropertiesScan(prefix=”some.app”) to your application class
    - Auto-configuration
    - Allows Spring Boot to automatically define beans it thinks you need based on some conditions
    - Enabled by adding @EnableAutoConfiguration to your configuration classes or, more commonly, @SpringBootApplication is used instead because it combines:
    - @SpringBootConfiguration
    - @EnableAutoConfiguration
    - @ComponentScan
    - Makes use of Spring-provided auto-configuration classes that are just @Configuration classes with conditions like “does classpath include specific classes, are certain properties set, are certain beans already configured or not configured, etc”
    - To make a bean conditional you add one of the @Conditional annotations:
    - @ConditionalOnBean(DataSource.class)
    - @ConditionalOnMissingMissingBean
    - @ConditionalOnClass
    - @ConditionalOnMissingClass
    - @ConditionalOnProperty, etc.
    - Auto-configuration classes are processed after explicitly created beans are defined
    - To override auto-configuration you have several options:
    - Set Spring Boot properties like:  
      spring.autoconfigure.exclude=\ org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
    - Explicitly define beans yourself so Spring Boot won’t
    - Explicitly disable some auto-configuration like:  
      @EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)
    - Change dependencies or their versions
    - App Entry Points
    - CommandLineRunner and ApplicationRunner are special beans that are detected by Spring Boot and invoked before returning from SpringApplication.run() but after all beans are configured
    - They offer Spring-style entry points so you don’t have to have business logic in the main() method
    - JPA - Jakarta Persistence API (formerly Java Persistence API)
    - Add spring-boot-starter-data-jpa to your POM file
    - If JPA is on classpath, Spring Boot automatically auto-configures:
    - a DataSource
    - a LocalContainerEntityManagerFactoryBean
    - A JpaTransactionManager
    - By default Spring Boot will look for entities in the same package (and sub-packages) of the class annotated with @EnableAutoConfiguration but you can override with @EntityScan(“rewards.internal”)
    - By default Spring Boot will look for repositories in the same package (and sub-packages) of the class annotated with @EnableAutoConfiguration but you can override with @EnableJpaRepositories(basePackages=”rewards.internal”)
    - Spring MVC
    - Supports server-side rendering and REST
    - Add @Controller to your controller class and @ResponseBody to the return type of your methods (this turns off the view handling system)
    - Simpler alternative is to add @RestController to your controller class which combines @Controller and @ResponseBody so you don’t have to add @ResponseBody to each method
    - Spring will inject whatever web-specific arguments you need: HttpServletRequest, HttpSession, Principal, Locale, etc.
    - Argument annotations:
    - @RequestParam – extracts value from the query string
    - @PathVariable – extracts value from the url path
    - @RequestHeader – extracts value from http headers
    - To run as a WAR within a web container:
    - Your @SpringBootApplication class needs to extend the SpringBootServletInitializer class
    - Then override the configure() method
    - Change artifact type to war file
    - To support running as WAR or JAR just add a main() method that calls SpringApplication.run()
    - Add @ResponseStatus to void methods in order to return a status other than 200
    - For POST methods if you want to return a 201 with the full URI of the newly created resource then use ServletUriComponentsBuilder:  
      URI location = ServletUriComponentsBuilder  
       .fromCurrentRequestUri()  
       .path(“/{itemId}”)  
       .buildAndExpand(“item A”)  
       .toUri();  
        
      return ResponseEntity.created(location).build();
    - Integration Testing:
    - You can add @SpringBootTest(classes=Application.class) to your test classes as an alternative to @SpringJUnitConfig in order to create your ApplicationContext with auto-configuration.
    - @SpringBootTest auto-configures a TestRestTemplate that doesn’t throw exceptions and accepts relative uris. Use RestTemplateBuilder if you need customizations.
    - Starts an embedded server when @SpringBootTest’s webEnvironment attribute is set to RANDOM\_PORT or DEFINED\_PORT
    - You can exclude the classes attribute from @SpringBootTest because it will search for a @SpringBootConfiguration class provided the configuration is in a package above the test (only one @SpringBootConfiguration is allowed in a hierarchy)
    - Add @AutoConfigureMockMvc to your test class to test calls to controllers. It processes requests through the DispatcherServlet but does not require running a web container to test.
    - Use @SpringBootTest for integration testing and @ContextConfiguration for slice testing
    - @Mock vs @MockBean:
    - @Mock is from Mockito and can be used when Spring context is not needed
    - @MockBean is from Spring Boot and can be used when Spring context is needed
    - Slice testing:
    - @WebMvcTest only auto-configures beans relevant to MVC tests and MockMvc  
      @WebMvcTest(AccountController.class) would only create beans relevant to AccountController
    - @DataJpaTest can be used to only load @Repository beans and to auto-configure TestEntityManager (uses an embedded in-memory database but can be overridden with @AutoConfigureTestDatabase)
    - Securing REST Applications
    - To configure Spring Security:
    - Setup Filter chain (Spring Boot does this for you or you can subclass AbstractSecurityWebApplicationInitializer)
    - Configure security (authorization) rules
    - Setup web authentication
    - Your filter chain can use permitAll() to allow open access but requests are still processed by the Spring Security Filter chain. Use ignoring() to completely bypass the filter chain.
    - Method security:
    - Can be enabled with @EnableGlobalMethodSecurity(jsr250Enabled=true) then adding @RolesAllowed(“ROLE\_MEMBER”) to your methods (NOTE: You must include the ROLE\_ prefix with this annotation)
    - Can also use SpEL in a @PreAuthorize annotation
    - Security Testing:
    - Add @WithMockUser(roles={“ADMIN”}) to your test methods then assert expected response status codes
    - Or you can use TestRestTemplate’s withBasicAuth() method with valid and invalid credentials against a running app
    - Actuator and Health Indicators
    - Actuator is accessible via JMX and numerous HTTP endpoints:
    - /actuator/info
    - /actuator/health
    - /actuator/metrics (not exposed by default)
    - Add spring-boot-starter-actuator to your POM file
    - Enabled (endpoint is created and beans exist in the app context)
    - HTTP default = all to enabled except shutdown, httptrace and auditevents
    - JMX default = disabled but can be enabled with spring.jmx.enabled=true
    - Exposed (endpoint is accessible via JMX or HTTP)
    - HTTP default = only exposes info and health
    - JMX default = all enabled endpoints are exposed
    - HTTP Endpoints map to /actuator/xyz by default but can be changed with management.endpoints.web.base-path=/admin
    - Can override default exposure with:
    - management.endpoints.web.exposure.include=health,info
    - or expose all with management.endpoints.web.exposure.include=\*
    - You can secure the endpoints with Spring Security like:  
      http.authorizeRequests()   
       .requestMatchers(  
       EndpointRequest.to(  
       HealthEndpoint.class,   
       InfoEndpoint.class  
       )  
       ).permitAll()  
       .requestMatchers(  
       EndpointRequest.toAnyEndpoint()  
       ).hasRole(“ACTUATOR”);
    - Micrometer library is used for metrics.
    - Custom metrics can be measured using the Micrometer classes like Counter, Gauge, Timer, DistributionSummary
    - The custom metric would be registered with a MeterRegistry bean then accessed at /actuator/metrics/[custom-metric-name]
    - All metrics can be accessed in 2 ways:
      * Hierarchical like http.method.get.status.200 or http.method.get.status.\*
      * Dimensional using tags like http?tag=method:get&tag=status:200
    - Health endpoints don’t show detail by default but can be overridden with:  
      management.endpoint.health.show-details=always
    - Health indicators can get grouped with:  
      management.endpoint.health.group.<group-name>.include=<indicators>
    - Groups can be configured individually:  
      management.endpoint.health.group.<group-name>.show-details=always
    - Add custom health indicators by:
      * Implementing HealthIndicator interface and override the health() method
      * Or Extend AbstractHealthIndicator and override the doHealthCheck() method
    - You can override the severity order with:  
      management.health.status.order=FATAL, DOWN, OUT\_OF\_SERVICE, UNKNOWN, UP