Summary

Contents

Container

Annotations (can be used in which level, used for what, etc)

@Autowired

- Can be used in field, methods (not only setters), constructors
- 'required' is an attribute of it to override, default is true
- Map is also supported type, only if the key is string

@ComponentScan

- This is used in the configuration class
- 'basePackages' 'basePackagesClass' 'includeFiters' 'excludeFilters' are the attribute

@Lazy

- This is used in method level when that method is with @Bean and class level annotated with @Configuration or other stereotypes
- This is used when we want to change the default scope for a bean
- This can be used with just the annotation, or 'value' as attribute with Boolean.

@Bean

- This is used only in method level.
- This is used for: configure autowiring whether by name or type; init and destroy method for that bean; specify name (attribute) and alias (attribute as 'value') for that bean. The default bean name is the name/id of the method if doesn't specify.
- @Bean{"b1", "b2"} can have multiple names

@PropertySource

- This is used only in class annotated with @Configuration
- This is used to get system environment properties. For example:

JVM System.getProperties()

System properties System.getenv()

Properties in JNDI env

Servlet configuration init parameters

Servlet context init parameters

Property files

@PostConstruct

- This is used in bean definition method

@PreDestory

- This is used in bean definition method

@Qualifier

- This can be used in dependency injection the injection point, bean definitions and annotation definitions. So in method, class and method parameters, fields.
- This is used when there are multiple beans with same name/id

@Profile

- This can be used in class level of configuration file, and any other class file with @Component, and methods with @Bean, and the customized annotation.
- ! is avalible, but no 'exclude'
- If no profile is specified, a default profile with name "default" will be created by Spring, if a bean is not associated with any profile, it is available everywhere.

@Value

- This can be used in fields, methods, method parameters and definition of annotations.
- This is used to inject literal value to beans.

- # is SpEL, $ is property name in the application environment.

@Import

- This can be used in class level with @Configuration
- This is used to group the configuration file.

Stereotype

@Component @Controller @Service @Repository @Indexed (@RestController)

Spring Bean life cycle including BeanFactoryPostProcessor and BeanPostProcessor

- Bean configuration is read, BeanDefination object created
- BeanFactoryPostProcessor can modify the bean definition here, eg the properties
- For each bean:
  - An instance is created
  - Property and dependencies are set (constructor -> setter -> xml)
  - BeanPostProcessor (has two methods: xxxbeforeInitalizationand xxxafterInitizalization) can modify the bean instance here:
    - @PostConstruct
    - InitialingBean afterPropertySet()
    - @Bean(initMethod=xxx)
    - The bean is ready to use
- When Spring context is ready to shut down, for each bean, the callback order is:
  - @PreDestory
  - DisposableBean destroy()
  - @Bean(destoryMethod=xxx)

Dependency Injection advantage – Inversion of Control

- Reduce coupling
- Increase cohesion
- Increase test ability
- Better design
- Increase reusability
- Increase maintainability
- Make it standardize
- Reduce boilerplate codes

ApplicationContext

- ApplicationContext is an interface, any application-context is a JAVA object which implement this interface and is responsible for:
  - Instantiating beans in application context
  - Configuring beans
  - Assembling beans
  - Managing life cycle of beans
- So application-context is:
  - A bean factory
  - A hierarchical bean facotory
  - Resource loader
  - Event publisher
  - Message source
  - An envirment

6 Scopes for Spring Beans

- Singleton (default): instance per container
- Prototype: instance by request
- Session: per http session
- Request: per http request
- Application: per ServeletContext
- Websocket: per Websocket

Limitations of JDK Dynamic Proxy and CGLB

- JDK Dynamic Proxy
  - At least one interface
  - Only methods in the interface will be available
  - Must be referenced using an interface type
  - No self-invocations
- CGLB
  - Class need to be non-final
  - Method need to be non-final
  - No self-invocations
  - Requires third party library

Some other questions

- getBeans(): from BeanFactory, get beans by name or required type and name or required type.
- FactoryBean: use to create beans besides the normal beans. Output of the getObject() method can be used as bean.
- Inner beans: will be passed as parameter/constructor, does not have id, cannot be accessed from outside, nor from the ApplicationContext.
- SpEL can be used in: static method; system environment variables

- @Bean with static: static allows a method to be called without creating its configuration class; it's necessary when beans need to be early initialized in life cycle; it will never be intercepted by the container, even the @configuration class; it avoids trigger other parts of the configuration at that point.

AOP

The meaning of the related concepts

- Join Point: The join point is the execution of a program which additional behaviors can be inserted into using AOP. Eg, method invocation, findxxx()
- Poincut: A pointcut selects one or more join join points out of the set. Eg. Execution(Entity *Repository.find*(…))
- Advice: Advice is the additional behavior, typically a cross concern, that is to be executed at a certain point of the program.
- Aspect: An aspect brings one or more pointcuts with one or more advice. Typically one aspect encapsulates one cross concern.
- Weaving: Weaving is the process by which aspects and application code is combined as to enable execution of cross cutting concerns. Can occurs
  - Compile time weaving, byte code of classes at compilation time. AspectJ use this
  - Load time weaving, byte code od classes is modified at loading time.
  - Runtime weaving, Proxy objects are crated at runtime. Spring AOP use this.

The designators. Which can be omitted, which are mandatory

Eg execution (public String se.nl.exaple.MyClass.*(String))

[method visibility] [return type] [package].[class].[method]([parameters] [throws exceptions])

- Can omitted: method visibility, package, class,
- Mandatory: return type, name(method), parameters
- Can use !: method visibility, return type, parameters, exceptions
- Can use wildcard * or ..: return type can use *, class can use *, method can use *, package can use .. when is the last in package name and * with the package names, parameters can use .. .
- Supported designators: within, this, target, args, @target, @args, @within, @annotation, bean

Advice, which can throw exceptions, which can catch exceptions

- Before: Executed before a join point, cannot prevent proceeding to the join point unless the advice itself is **throwing expectation**.
- After returning: Executed after a join point is completed without throwing exception.
- After throwing: Executed after a join point is resulting an exception.
- After: Executed after a join point regardless of exception throw or not.
- Around: The most powerful one, can decide to execute before or after the join point. Can choose whether or not to execute this join point. Can change return value as well, can **throw exceptions and try catch exception.**

How to enable AOP

- @Configuration with @EnableAspectJAutoProxy, then enable advice class with @Aspect together with @Configuration

Cross cutting concerns

- Logging
- Caching
- Security
- Monitoring
- Transaction management
- Error handling
- Data validation
- Customizing business value

AOP advantage

- Avoid code duplication
- Avoid mixing of, business logic and cross cutting concerns

Some other questions

- Spring AOP (Java based):
  - Only support public method invocations join points. (AspectJ can do more)
  - Only support method execution, does not support interception of constructors nor the static initialization code.
  - Can only apply to spring beans
  - The AOP proxy is created by BeanPostProcessor

JDBC & Transaction Management & JPA

Annotations (can be used in which level, used for what, meaning)

@Transactional

- It can be used in method and class level
- It is used to state the transaction management
- It has attribute: rollBackFor, rollBackForClassName, noRollBackFor, noRollBackForClassName, isolation, propagation, readOnly, timeOut, transactionManager

@EnableTransactionManagement

- It can be used in class level with @Cofiguration
- It is used to enable the annotation driven transaction management.

@EnableJapRepository

- It can be used in the class level with with @Configuration
- It is used to enable Spring data JPA.

@Query

- It can be sued in method level which is a finder
- It is used to specify the customized query which can use Spring data repository methods
- It supports JPQL, Naïve SQL

How to configure a datasource in Spring, how about SpringBoot

- Spring
  - Depends on the standalone application or the web application. We can create a dataSource bean.
  - The setDriverClassName, setUrl, setUserName, setPassword is needed.

- SpringBoot
  - It is the same as Spring that we need a dataSource bean
  - But SpringBoot has an in-Memeory h2 database which can be used as well
  - The bean can be set in configuration class or in the application.properties.

The three JdbcTemplate callback interface

- ResultSetExtractor
  - Suitable for multiple rows return to an Object.
  - The extractData method from this interface return to a JAVA object.
- RowCallBackHandler
  - Suitable for accumulating some type of results.
  - The processData method from this interface has void return.
- RowMapper
  - Suitable from each of the row return to domain object
  - The mapRow method from this interface return to JAVA object.

The methods that JdcbTemplate supported (8)

- Query
- Execute
- batchUpdate
- update
- queryForObject
- queryForMap
- queryForList
- queryForRowSet

Transaction ACID principle

- Atomic
  - All or Nothing
- Consistent
  - Any integrity constraints, for instance of a database, are not violate.
- Isolation
  - Transactions are isolated from each other and do not affect each other.
- Durability
  - Charges applied as the results of successfully completed transaction are durable.

Local & Global transaction and the related transaction manager, what is PlatformTransactionManager

- Local
  - When transactions are associated with only one resource. for example a database or a message queue.
  - JpaTransacationManager is recommended here.
- Global
  - When transactions are spanned with multiple resources, for example a database and a queue, then this is global transaction
  - JtaTransactionManager is recommended here.
- PlatformTransactionManager
  - This is the base interface for all transaction managers.
  - Void commit (TransactionStatus)
  - Void rollback (TransactionStatus)
  - TransactionStatus getTransaction(TransactionDefination)

Transaction levels

- Serializable
  - The highest isolation level
  - Rrange block with read and write. Example: A read id=35 records, A update, until A commit, B cannot write id = 35.
- Repeatable reads
  - Where phantom reads can occur.
  - A select age between 20- 50, A start transaction, B inserts age 35. A doesn't see the age 35 record. But A cannot write age 35 anymore. After A submit, If A select again, he sees the 35 record.
- Read committed
  - A select a Person, A start transaction, B updates this person and committed, A query again already see this change.
- Read uncommitted
  - Where dirty reads can occur
  - A select a Person, B updates and doesn't need to committed.

Transaction propagation and the famous require_new problem

- There are 7 kinds, nested, never, mandatory, require, requires_new, not_supported, supports
- Require is the default. Method is executed in the current transaction, if no transaction then create a new one
- Require_new. Create a new transaction in the current method. Suspend any existing transaction.
- If method1() call method2() in the same class. It is still the self-invocation problem. So any the transaction way of the method2 will not come to any effect, it will be bypassed.

In Spring how to use Spring JPA, how about SpringBoot

- Spring

- Dependencies, typically a ORM
- Entity classes @Entity
- Define EntityManager Bean
- Define DataSource
- Define TransactionManager bean
- Implement repositories.

- SpringBoot
  - Spring-boot-starter-data-jpa
  - Automatically create a datasource, EntityManagerFactory Bean, JpaTransactionManager.

Naming convention for finder methods in a Repository interface.

Find(First[count]By[xx][less than/between/like][desc])

- eg, findFirstPersonByNameLikeJackOrderByLastiModifiedDesc

Security

Annotations (can be used in which level, used for what, meaning)

@EnableWebSecurity

- Use in class level with @Configuration
- Used to enable Spring web security, be attention this is optional for SpringBoot

@EnableGlobalMethodSecurity

- Use in class level with @Configuration
- Used to enable regular Spring method security

@EnableReactiveMethodSecuirty

- Use in class level with @Configuration
- Used to enable reactive Spring method security

@PreAuthorize

- Can be used in class and method level, support SpEL
- Must @EnableGlobalMethodSecurity(prePostEnabled=true)

@PostAuthorize

- Can be used in class and method level, support SpEL
- Must @EnableGlobalMethodSecurity(prePostEnabled=true)

@PreFilter

- Can be used in class and method level, support SpEL
- Must @EnableGlobalMethodSecurity(prePostEnabled=true)

@PostFilter

- Can be used in class and method level, support SpEL
- Must @EnableGlobalMethodSecurity(prePostEnabled=true)

@RolesAllowed

- Can be used in class and method level, do not support SpEL.
- Must @EnableGlobalMethodSecurity(jsr250Enabled=true)

@Secured

- Can be used in class and method level, do not support SpEL.
- Must @EnableGlobalMethodSecurity(securedEabled=true)

How Spring implement Security and the setup

Implementation

- Method level Spring is using Spring AOP
- Web infrastructure, Spring is entirely based on servlet filter.

Setup

- Setup Filter chain (SpringBoot no need)
  - Requires a DelegatingFilterProxy which must be called springSecurityFilterChain
  - Set up the filter chain using two options:
    - Subclass AbstractSecurityWebApplicationInitializer
    - Declare as a <filter> in web.xml
- Configure security(authorization) rules
  - Use @EnableWebSecurity in configuration file, optional for SpringBoot
- Setup Web Authentication
  - In-Memory Authentication Manager

- Http Authentication
- Implementing a custom UserDetailsService; LDAP; Single-sign-on

DelegatingFilterProxy, FilterChainProxy, SecurityFilterChain

DelegatingFilterProxy

- It is defined in web.xml and delegates to a FilterChainProxy
- It implements the Filter interface, so it is a servlet filter.

FilterChainProxy

- It is defined as a spring bean and takes one or more SecurityFilterChain instance as constructor parameters.

SecurityFilterChain

- It associates a request URL pattern with a list of security filters.

Spring security core components

SecurityContextHolder

- Contains and provide access to SecurityContext,
- Default behavior is connect the current thread with the SecurityContext.

SecurityContext

- Holds the Authentication object
- May also includes the special request info.

Authentication

- Represents the token of the authentication requests or the authenticated principle after the request has been granted.
- Also contains authorities in the application that an authenticated principle that has be granted.

GrantedAuthority

- Represents authority granted to a principle

UserDetails

- Holds user info, Such as user-name, password.
- The principle is often but not always a UserDetails object

UserDetaisService

- To retrieve information of the user in a UserDetails object, act as a DAO to load user info from example database or in-memory db.

The two requestMatcher

MvcRequestMatcher

- Matches more than ant, eg, http.authorizeRequest().mvcMatchers("/admin").hasRole("ADMIN"). This matches /admin, /admin/, /admin.html.
- Uses the same matching rules as used by @RequestMapping and it is newer.

AntPathRequestMatcher

**** Patern**

- * Matches any path on the level
- ** Matches any path on the level and below

Password Hashing, Salting

Password Hashing

In Spring is it PasswordEncoder

Salting

In Spring it is supported, it is used to avoid always store the same hash value of a word

Spring MVC & REST

Annotations (can be used in which level, used for what, meaning)

@EnableWebMvc

- Used in class level with @Cofiguration
- It is used to enable Spring MVC

@RequestMapping

- Used in method level and class level (means the root)
- It is used to map web request to Spring controller methods
- It has “value”, “headers”, “method”, “produces”(specify the produce file type, json or xml) attribute

@GetMapping …

- Used in method level.
- It’s the same function as @RequestMapping with method given
- GET, PUT, POST, DELETE, PATCH

@RequestParam

- Used in method parameters
- It is used to bind request parameters to method parameters, it will do automatic type convention, map as String. Eg

  @GetMapping(“/accounts”)

  Public list<Account> list(@RequestParam long userId) {}

- If there are multiple parameters, eg http://localhost:8080/greeting?firstName=xx&lastName=xx

@PathVariable

- Used in method parameters
- Has the same function as @RequestPara, It map part of the URL to handler methods. Eg

  @GetMapping("/accounts/{userId}")

  Public list<Account> list(@PathVariable long userId) {}
- If there are multiple parameters, eg
  http://localhost:8080/greeting/firstName/xx/lastName/xx

@RestController

- It is a composed annotation, equal to @Controller and @ResponseBoday
- So it used in class level, as the same with @Controller

@RequestBody

- It is used in method parameters of a controller
- It is used to bind the request body to a controller method argument.

@ResponseBody

- Can be used in class level of a controller or the method level
- It is used to serialize the result of the controller results processed by a **HttpMessageConverter，** and if a controller class is annotated with @RestController then we don't need it.

@ResponseStatus

- Can be used in class level which is exception class, and the method level to override the original response status.

What is Spring MVC, can we declare MVC controllers without using annotation

- Spring MVC is a design pattern which provides sets of guidelines for developing web application.
- We can declare MVC controllers without using annotations, do something in the web.xml.

DispatcherServlet, what it is and where we define it

- It is a servlet that implement the front controller design pattern.
- Receives requests and delegates to handlers, handlers send back logical view name, view resolver give a view to servlet and it send the model to a view, the view return html and it gives user response. Also responsible to render the error view if exception throw.
- For REST service, it receives requests and delegates to handlers, handlers send back data, servlet and it send the data to HttpMessageConverter, it converts the data to json or xml send to the servlet, then it gives user response.
- Can be defined in web.xml or register them with the ServletContext as Container, SpringBoot does this.

Http verbs, codes and RestTemplate supported

Htttp verbs (all RestTemplate supported)

- GET – safe and idempotent
- PUT – idempotent
- POST
- DELETE – idempotent
- HEAD
- OPTIONS
- PATCH

HttpCodes

- 1xx Informational
- 2xx Success
  - 200 OK
  - 201 Created
  - 202 Accepted
  - 203 Non-Authoritative information
  - 204 No content
  - 205 Reset content
  - 206 Partial content
- 3xx redirection
- 4xx client error
- 5xx server error

What are the parameter types can be used in controller method, and annotations can be used as controller method arguments, and valid return types of a controller method?

Parameter types can be used as controller argument

- WebRequest …
- ServletRequest…
- Principle HttpMethod HttpEntityLocale SessionStatus UriComponentBuilder TimeZone
- IO related
- Errors

Annotations can be used as arguments

- @PathVaraible @MatrixVariable

- @RequestParam @RequestHeader
- @CookieValue
- @RequestBody @RequestPart
- @ModelAttribute @SessionAttribute @RequesAttribute

Valid return types

- Http related, eg entity, header
- String: name of the view to be rendered
- View Model ModelAndView
- Void: used when controller method handles response by writing to an output stream or if the method is annotated with @ResposeStatus. Used in REST methods that are not return a response body, used in HTML controller methods selecting the default view name. Return null from a controller method produces the same results.

Spring mvc dependency, how about in SpringBoot

Spring

Spring-web

SpringBoot

Spring-boot-starter-web

- Set up a DispatcherServlet
- Set up internal configuration to support controllers (@EnableWebMvc optional)
- Set up default resource locations
- Set up default message converters

Testing & SpringBoot Testing

Annotations (can be used in which level, used for what, meaning)

@ContextConfiguration

- It can be used in the class level with integration tests
- It's used to load a Spring application context, and the ApplicationContext instantiated only once.
- It has attribute classes = {xxx.class, xxx.class}, locations = {"xxx.xml", "xxx.xml"}, name and value(alias of the location) or directly use string to specify the config xml file.

@SpringJunitConfig

- It's a composed annotation, @ExtendWith (SpringExtention.class) + @ContextConfiguration

@TestPropertySource

- It can be used in test class level
- To customize the properties just for trsting
- It has attribute properties = {"username=xx", "password=ww"}, locations = {"classpath:/xxx"}

@ActiveProfile

- It can be used in test class level
- In configuration file we must define related beans with @Profile

@Sql

- It can be used in class (can define errorMode) and method level for testing.
- It is used for integration testing against SQL database.
- Input is the sql script location. Or the scripts attribute, or the config to specify the errorMode.

@DirtiesContext

- It can be used in test method level and class level.
- Forces context to be closed at the end of the method. Allow to test the @Predestory, next tests get a new ApplicationContext.

@SpringBootTest

- Used for integration test, The *@SpringBootTest* annotation is useful when we need to bootstrap the entire container
- Used in class level
- Autoconfigres a TestRestemplate

@WebMvcTest

- Disable full auto-configuration and apply only MVC related
- Eg @WebMvcTest(AccountController.class)

@DataJpaTest

- Can be used when tests only focus on JPA components
- Auto configs TestEntityManager
- Used an in memory database

@MockBean @Mock

- @Mock used when Spring context is not needed
- @MockBean used when Spring context needed

Spring Junit 4 support (also 5)

- Packaged as separate module – spring-test.jar
- Consists of several Junit test support classes

- Central support is SpringJunit4ClassRunner

Slice Unit testing

- Performs isolated testing within a slice of an application
  - Web slice
  - Service slice
  - Repository slice
  - Caching slice
- Dependencies need to mock

Test dependencies with spring-boot-starer-test (7)

- Junit
- Spring & SpringBootTest
- AssertJ
- Hamcrest
- Mockito
- JSONassert
- JsonPath

SpringBoot Intro & SpringBoot Autoconfig & SpringBoot Actuator

Annotations (can be used in which level, used for what, meaning)

@Coditional[xxx]

- Can be used in class or method level
- Allow conditional bean creation. Eg ConditionalOnBean , ConditionalOnMissingBean, ConditionalOnClassxxx

@SpringBootApplication

- Used in class level of application
- Composed annotation of @Configuration @EnableAutoConfiguration @ComponentScan

@EnbaleAutoConfiguration

- Used in class level with @Configuration
- It reads spring-boot-autoconfig/META/INF/spring.factories

Review, How Spring simplify transaction, JPA. Mvc

Transaction & JPA – spring-boot-starter-data-jpa

- Automatically create a datasource, EntityManagerFactory Bean, JpaTransactionManager.
- For JPA, Override using @EntityScan
- Simplifies repositories, only need interface to extends CRUDxxx

MVC - Spring-boot-starter-web

- Set up a DispatcherServlet
- Set up internal configuration to support controllers (@EnableWebMvc optional)
- Set up default resource locations
- Set up default message converters

SpringBoot dependencies

- Spring-boot-starter-parent
- Spring-boot-starter-autoconfigure
- Spring-boot-starter-data-jpa
- Spring-boot-starter-web
- Spring-boot-starter-test
- Spring-boot-actuator
- Spring-boot-devtools

Where does SpringBoot look to config

- Auto-configuration
  - Mechanism to detect dependencies through classpath, beans and config properties
  - Creating Spring beans optioned
  - Can be customized

Spring.factories, how to customize the auto configuration

- It contains a list of auto-configuration classes
- Functions:
  - Register application event listeners of how SpringBoot application is created
  - Locate autoconfiguration candidates in
  - Customized the env or application context prior to the SpringBoot application
  - Register a filter to limit the auto-configuration class considered.
- We can customized the autoconfiguration by put this to property: spring.main.alllow-bean-defination.overriding=true

How are properties defined, what's the default?

- By default it looks for application.properties located in /config, the working directory, -config in the classpath, root classpath
- Creates a PropertySource based on these files
- Also support yaml files, the difference is:
  - User.name=salad
  - User:

    Name:salad

What embedded containers it supports

- Tomcat
- Jetty
- Undertow

Jar, Fat jar, War

- War needs to deployed by container like Tomcat
- Jar included an embedded container so we can directly use it Myapp.jar.original
- Fat jar, contains also all the dependencies of the application. Use gradle assemble or mvn package. Myapp.jar

Two ways of runner

- CommandLine runner, has run() method, handle arguments as array
- ApplicationRunner, has run() method as well, handle arguments as ApplicationArguments

SpringBoot logging

- By default, it includes SLF4J and Logback, other framworks supported Java Util Logging, Log4J, Log4J2
- By default it writes logs to console, can set in propterties, logging.path=/var/log/xx
- As default, ERR0R, WARN, INFO levels will be write to log. To enable DEBUG and TRACE, use the –debug or –trace or set in property files
  - Logging.level.root=WARN Logging.level.se.salad=DEBUG
- Also we can customized the log color
  - Logging.pattern.console=%clr()

Actuators, default value, default enabled or not, how to customized

How to access actuator

- JMX: this is by default all exposed
  - Management.endpoints.jmx.exposure.include=*
- HTTP endpoints: only info and health
  - Management.endpoints.web.exposure.include=*

Type

- Info: general data, build info of the last commit
- Health: application health status
  - Default output is minimal, only basic info status
    - Management.endpoints.health.show-details=always
  - Can be customized by create a class implements HealthIndicator and override health()
  - Build in status: DOWN OUT_OF_SERVICE UNKOWN UP
    - Serverity order can be overriding: management.endpoints.health.status.order= FATAL, DOWN, OUT_OF_SERVICE, UNKOWN, UP
- Metrics: list of generic and custom metrics measured by the application

- Default is disabled
- Can customized it with Counter, Timer… Create or register a MeterRegistry, metrics are listed /actuator/metrics/[salad]

  Public ReportController(MeteRegistry meter) {

  Summary=DistriubtionSummary.builder(“salad”).regsiter(meter)

  }

- Beans
- Conditional
- Env
- Loggers
- Mapping
- Session

How to access an endpoint with tag, how to create metrics 2 ways?

- Hierarchical http.method.get.status.200
- Dimensional http?tag=method:get&status:200

3rd-party external monitoring system benefits

- Gather data
- Show graphic