**Benefits/Features of Spring Framework**

* Encourage POJOs without extending Framework-specific class or interface, non-invasive programming model.
* Simplify enterprise Java development through DI, AOP:
  + Manage POJOs/Beans through DI: applications are made up of multiple classes/POJOs that collaborate with each others. By using DI, objects are given/injected their dependencies at run time, instead of creating dependencies on their own, which leads to loose coupled code, so that each class can focus on its own things, and doesn’t need to mind all the matters (such as creation or configuration) related with its dependencies. Often the dependencies are interface, so classes are not coupled to specific implementation, so mock implementation can be swapped in during testing. All the POJOs/Beans are configured by either XML, annotation or Java-based, and their creation/wiring/lifecycle are fully managed by Container (such as applicationContext). Container takes the POJOs/Beans configuration, creates associations between collaborating components, and uses DI to manage them.
  + AOP:
* Beyond core, Spring is an entire ecosystem that builds on the core framework, extending Spring into many areas such as web, REST, data. Also, Spring offers integration points with several other frameworks.
* Spring Boot: a project to simplify Spring itself. Employs heavy automatic configuration techniques that can eliminate most Spring configuration.

**Defining and Configuring Beans**

Implicit bean discovery and automatic wiring

* Use @Component to mark the class as discoverable bean. Components are not turned on by default. Component ID is class name with un-capitalized first letter by default, but can also be explicitly configured.
* Use @ComponentScan on a @Configuration class to scan all components, and register them as beans. By default, scanning same package and its sub packages as the configuration class. Multiple base packages to be scanned can be explicitly configured.
* @Autowired: a means of letting Spring automatically satisfy a bean’s dependencies by finding other beans in the application context that are a match to the bean’s needs. Actually @Autowired can be applied on any method on the class apart from constructor or setters.

Explicit Java-based bean configuration

* Use @Configuration to mark the class as configuration class, and it is expected to contain details on beans (either through @ComponentScan or explicit bean factory methods) that are to be created in the Spring application context, and register them as beans.
* Use @Bean to mark bean factory methods. It tells Spring that this method will return an object that should be registered as a bean. By default, the bean ID will be the same as method’s name.
  + If a bean factory method (or constructor?) is called (e.g. singleton scope bean), Spring will intercept the call and ensure that if a bean already exists, it will return this bean instead of creating a new one.

Configuring bean profile

* Environment-specific bean, run-time decision
* Use @Profile(“sampleProfileName”) on the bean class.
* Activating profiles: use spring.profiles.active properties, or @ActiveProfiles.

Configuring conditional bean

* Use @Conditional(XXX.class) on the class. Can be given any class that implements the Condition interface which has only one matches() method. If matches() returns true, then bean is created.

Configuring injected values with given values or from properties (placeholders)

* Use @Value to assign default field values
  + Given values: @Value(“Some String Values”)
  + From properties: @Value(“${some.property}”) or (“$(property:defaultValue)”)
* If use placeholders, PropertySourcesPlaceholderConfigurer bean needs to be registered.

Configuring bean scope

* Basic scope: singleton (default), prototype, session, request.
* Working with session and request scope:
  + When autowiring session/request scope beans, use proxy, because beans may have not been created. This proxy will expose the same methods as the bean, so that for all its parent object knows, it is the dependency bean. But when parent object calls methods on the bean, the proxy will lazily resolve it and delegate the call to the actual session-scoped bean.

Addressing autowiring ambiguity

* Happens when there are more matching beans. The default qualifier is ID.
* Way 1: designate @Primary bean. Spring will choose the primary bean over any other candidate beans.
* Way 2: use @Qualifier on the @Bean or @Component, and use @Qualifier when autowiring.
  + Qualifier on @Bean or @Component: by default, it is bean ID implicitly. Can be specified explicitly, such as @Qualifier(“AName”).
  + Qualifier when autowiring: use @Qualifier on the Autowired parameter with the desired qualifier.
* Way 3: define custom qualifier annotation. One benefit is custom qualifier annotations can be applied multiplely. See Spring in Action P79.

**Configuring and using external properties**

Default property file for Spring Boot

* application.properties on the search path (classpath:, classpath:/config,file:, file:config/)
* Default file name can be changed.

Adding external property files

* Use @PropertySource on the config class in conjunction with @Configuration. It adds the property file to the Spring’s Environment.
* Access properties: Environment env; env.getProperty(String key, String optionalDefault);

**Spring Boot (Including Spring MVC):**

**Common application properties**

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

**POM**

Starters

* spring-boot-starter-parent (as Parent POM) is a special starter that provides useful Maven defaults. It also provides a dependency-management section so that you can omit version tags for “blessed” dependencies.
* Other “Starters” simply provide dependencies.

Plugins

* spring-boot-maven-plugin: spring-boot-starter-parent POM includes <executions> configuration to bind the repackage goal to create a self-contained Executable JAR.

**Run**

* mvn spring-boot:run (spring-boot-starter-parent POM includes the useful run goal)
* java –jar xxx.jar

**Structure**

* Main Class
  + recommend in root package along with @ComponentScan.
  + @ComponentScan: @Component, @Service, @Repository, @Controller and @Configuration will be automatically scanned and registered as Spring Beans.
  + @EnableAutoConfiguration: automatically configure Spring application based on the jar dependencies.
  + @SpringBootApplication: same as @Configuration @EnableAutoConfiguration @ComponentScan.
* Directories
  + controller
  + domain
  + dao
  + service
* Resource
  + application.properties (needs to be in classpath. In Maven, just place in resources directory) (default search path: classpath:, classpath:/config,file:, file:config/) (file name can be changed, see 24.3)

**Bootstrap**

SpringApplication Class

* use static run(sourceObj, args)
  + create appropriate ApplicationContext instance
  + register CommandLinePropertySource to expose command line arguments
  + loading all singleton beans
  + trigger any CommandLineRunner beans
* or use instance run(args)
  + creating SpringApplication class first, customize app setting, then run()

CommandLineRunner Interface

* used to mark a bean that should run. Multiple beans can be ordered through @order.
* offer a single run method which will be called just before SpringApplication.run(…​) completes.

**Persistence**

DataAccessException

* Spring takes the stance that many exceptions are the result of problems that can’t be addressed in a catch block. Instead of forcing developers to write catch blocks (which are often left empty), Spring promotes the use of unchecked exceptions.

Data Source

* DataSource is an interface as the preferred means of getting a connection to a data source. Its implementations usually provide pooling and distributed transactions. All of the connections produced by instances of that DataSource class will automatically be pooled connections.
* With auto-configuration, Spring Boot will create DataSourceProperties (base class for configuration of a data source, default ConfigurationProperties is spring.datasource), then create DataSource implementation based on DataSourceProperties.
* So with auto-configuration, just put database configuration (spring.datasource.\*) in application.properties.
* How a pooling DataSource implementation is selected:
  + Auto-configuration algorithm: Tomcat pooling DataSource is always chosen if available; otherwise, if HikariCP is available we will use it.
  + spring-boot-starter-data-jpa ‘starters’ will automatically get a dependency to tomcat-jdbc (so that Tomcat pooling DataSource is always selected by auto-configuration)
* Auto-configuration will not occur if DataSource bean is explicitly defined.
* Different ways to define pooled DataSource:
  + From JNDI: define JndiObjectFactoryBean
  + From Spring auto or manual configuration: see above
* The data source will be used anywhere needed. If multiple, the primary data source will be used by others by default.

Entity (Entity Definition see **Hibernate Note**)

* Traditionally, JPA ‘Entity’ classes are specified in a persistence.xml file. With Spring Boot, xml is not necessary, as with auto configuration, any classes annotated with @Entity, @embeddable or @MappedSuperClass will be scanned. (all packages below main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) will be searched)

Database Creating/Dropping by Hibernate

* With auto-configuration, it is controlled by spring.jpa.hibernate.ddl-auto=create-drop in application.properties. If = create, then tables will remain upon Hibernate termination.

Spring and JPA

* JPA spec defines two kinds of entity managers: Application based and container based. To define one of them, just define and configure LocalEntityManagerFactoryBean (Application based) or LocalContainerEntityManagerFactoryBean (Container based).
* Define jpaVendorAdapter to specify which JPA implementation to use. E.g. HibernateJpaVendorAdapter. Dialect, database and other properties can be set in the vendor adaptor.
* Write a JPA-based repository (See Spring in Action 11.2.2)
  + Repository class is annotated with @Transactional to indicate all methods are involved in a transactional context.
  + Repository class is annotated with @Repository to specify exceptions should be translated into Spring’s unified data-access exceptions. PersistenceExceptionTranslationPostProcessor needs to be defined to do the actual translation. If without the translation processor, the native JPA-specific or Hibernate-specific exceptions will be thrown without any translation.
  + EntityManager annotated with @PersistenceContext needs to be wired as instance variable. @PersistenceContext doesn’t inject an actual EntityManager, instead a proxy to a real EntityManager, to make sure working with EntityManager in a thread-safe way. PersistenceAnnotationBeanPostProcessor needs to be defined to work with @PersistenceContext JPA annotation.
* Spring Data Automatic JPA Repository (usage see **Spring Data JPA Repository Usage**)
  + Interfaces that can be defined as DAO to access/manipulate data. JPA queries can be created automatically from method names.
  + Extend from CrudRepository (such as JpaRepository or MongoRepository) interface.
  + With auto configuration, repositories will be searched from the package containing your main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) down.
  + Repository instance will be created by Spring with the configuration. Repository instance can be autowired directly.

Custom/Multiple Data Sources

* Follow the default DataSource auto-configuration logic: create DataSourceProperties (provide custom ConfigurationProperties), then create DataSource.
* There must be one Primary DataSource/EntityManagerFactory/ PlatformTransactionManager if multiple.
* DataSource can also be programmingly configured/overridden.
* If multiple DataSource, set up EntityManagerFactory also for each DataSource.
* If multiple EntityManagerFactory, set up PlatformTransactionManager also for each EntityManagerFactory.
* If multiple EntityManagerFactory/ PlatformTransactionManager, set up Repository configuration for each EntityManagerFactory/ PlatformTransactionManager.

**MongoDB**

com.mongodb.MongoClient

* A MongoClient has internal connection pooling. Consider it as a DataSource.

com.mongodb.DB

* A thread-safe client view of a logical database, can be obtained from MongoClient.
* DB db = mongoClient.getDB("<db name>");

Spring process of configuring MongoDB

* MongoClientFactoryBean with configuration (host, port, credentials)
* MongoClient can be obtained from MongoClientFactoryBean
* MongoDbFactory (SimpleMongoDbFactory) with configuration (mongoClient, databaseName)
* MongoDbFactory will be used by MongoTemplate/MongoRepository to create pooled DB instance

**Spring Data JPA Repository Usage**

Saving entities

* repository.save(…): persist entities by using the underlying JPA EntityManager.persist() if entity hasn’t been persisted; otherwise merge() will be called (like upsert).

Query method creation

* See Spring JPA Reference 4.4.2 & 5.3.2

Using @Query on method name

* @Query(“A JPQL Query”)

**Spring MVC**

@Controller

* The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations.

@RestController

* @RestController is a stereotype annotation that combines @ResponseBody and @Controller.

@RequestMapping

* The class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method or an HTTP request parameter condition.
  + Without class level @RequestMapping, all paths are simply absolute, not relative
  + @RequestMapping maps all HTTP methods by default.
  + Can narrow the primary mapping by specifying a list of consumable/producible media types.
  + e.g. @RequestMapping(path = "/new", method = RequestMethod.GET, consumes = "application/json")

@PathVariable

* Use the @PathVariable annotation on a method argument to bind it to the value of a URI template variable.
  + Spring automatically converts to the appropriate type or throws a TypeMismatchException.

@RequestParam

* Use the @RequestParam annotation to bind request parameters to a method parameter in controller.
  + Type conversion is applied automatically if the target method parameter type is not String.
  + Parameters are required by default, but can be changed by @RequestParam(name="id", required=false)

@RequestBody

* @RequestBody indicates that a method parameter should be bound to the value of the HTTP request body.
* HttpMessageConverter is responsible for converting from the HTTP request message to an object and converting from an object to the HTTP response body. The RequestMappingHandlerAdapter supports the @RequestBody annotation with the default HttpMessageConverters.
* HttpMessageConverter will look at Content-Type header to determine delegated converter.

@ResponseBody

* Spring converts the returned object to a response body by using an HttpMessageConverter, instead of any viewResolver.
* HttpMessageConverter will look at Accept header to determine delegated converter.

@RequestHeader

* @RequestHeader annotation allows a method parameter to be bound to a request header.

ResponseEntity

* Extension of HttpEntity that adds a HttpStatus status code. Used in RestTemplate as well @Controller methods.
* If return ResponseEntity, no need to annotate @ResponseBody.
* ResponseEntity.ok()/status().body()

**Logging**

Spring Boot uses Commons Logging for all internal logging, but leaves the underlying log implementation open (default is Logback).

Apache Commons Logging

* Java-based logging utility and a programming model
* Six log level: fatal, error, warn, info, debug, trace
* Two basic abstractions: Log and Logfactory
* Usage: Log log = LogFactory.getLog(XXX.class); log.info(“…”);

Configuration

* Set log levels
  + logging.level.\*=LEVEL (TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF)
  + logging.level.org.hibernate=DEBUG
* File output
  + logging.file or logging.path properties
* Logging patter
  + logging.pattern.console/file properties
* Force using a particular logging system
  + org.springframework.boot.logging.LoggingSystem properties

**Security**

To be done.

**Actuator**

Provides production-ready features. Add a dependency to the spring-boot-starter-actuator ‘Starter’ to enable.

Endpoints

Actuator endpoints allow you to monitor and interact with your application. The ID of the endpoint is mapped to a URL.

* autoconfig: Displays an auto-configuration report showing all auto-configuration candidates and the reason why they ‘were’ or ‘were not’ applied.
* beans: Displays a complete list of all the Spring beans in your application.
* flyway: Shows any Flyway database migrations that have been applied.
* health: Shows application health information.
* mappings: Displays a collated list of all @RequestMapping paths.
* trace: Displays trace information (by default the last 100 HTTP requests).

Customizing endpoints

* endpoints.NAME.ATTR=XXX in application.properties

CORS Support

CORS is disabled by default and is only enabled once the endpoints.cors.allowed-origins is set. E.g.

endpoints.cors.allowed-origins=http://example.com

endpoints.cors.allowed-methods=GET,POST

Accessing Sensitive Endpoint

By default all sensitive HTTP endpoints are secured such that only users that have an ACTUATOR role may access them.

* Disable authentication check: management.security.enabled=false
* Set username, password and role:
  + security.user.name=admin
  + security.user.password=secret
  + management.security.roles=SUPERUSER