Java Springboot notes

* Easy to get started with Spring Development
* Minimizes manual configurations
* Perform Auto-configuration based on props files and JAR classpath
* Spring boot uses the same code as the spring framework, its primarily used to start up projects quickly.

Core Container

* Factory for creating beans and managing bean dependencies
  + Beans
  + Core
  + SpEL
  + Context

Infrastructure

* AOP: Aspect Oriented Programming
  + Adds functionality to objects declaratively
  + Logging security, transactions
  + AOP
  + Aspects
  + Instrumentation
  + Messaging

Data Access Layer

* Communicating with the DB
* JDBC
  + Reduces source code significantly
* ORM
  + Object relational mapping
* Transactions
* OXM
* JMS
  + Java message service
  + Send a message to a queue, asynchronous
  + Spring provides helper classes for this

Test Layer

* Unit
* Integration
* Mock

GAV

* Group ID
* Artifact ID
* Version

Do not use the webapp directory if the application is packaged as a JAR

* Only works with WAR packaging (for a maven directory)

Spring boot starters

* Curated list of Maven dependencies
* Collection of dependencies grouped together
* EX: spring-boot-starter-web

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter>

* spring-boot-devtools will automatically restart the application when your code is updated

Spring Boot Actuator

* Exposes endpoints to monitor and manage your application
* Easy get DevOps functionality
* REST endpoints are automatically updated here
* actuator/health
  + Checks status of your application
  + Used by monitoring apps to see if it is up or down
* actuator/info
  + Provides more information about your application
* /auditevents
* /beans
* /mappings
* <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints>

Command line running

* Mvnw package (to package project)
* Java – jar (appname)
* Mvnw spring-boot:run

Spring boot properties

* <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties>
* Core
* Web
* Security
* Data
* Actuator
* Integration
* Devtools
* Testing

Dependency injection

* Constructor injection
  + Use when you have require dependencies
  + Generally recommended as first choice
* Setter injection
  + Use this when you have optional dependencies
  + If not provided, can provide reasonable default logic
* Field injection
  + Generally try to avoid using this
  + It’s difficult to unit test
* Autowiring
  + Spring will look for a class that matches
  + Matches by type and uses it automatically

Component scanning

* Spring will can your components for specific tags

Qualifiers

* You can use the Qualifier annotation to specify which class to use for injection
* You can also use the @Primary annotation to specify which one
* Qualifier overrides the primary annotation
* Can only have one primary annotation class

Bean Scopes

* Default scope is singleton
* Prototype, new bean for each container request
* Request, scoped to http web request
* Session, scoped to http session
* Global-session, scope to a global http web session

Bean Lifecycle

* Bean instantiated
* Dependencies injected
* Internal spring processing
* Custom initialization method
* Bean ready for use
* Container is shutdown
* Custom destroy method
* Stop

This custom can be good for adding any startup logic you need

* DB connections etc

Prototype beans do NOT call the destroy method when they are destroyed

There is no need to use the @Lazy annotation for a prototype bean, it’s already configured as such by default

Examples for using @Bean

* When you want to use a third party class available for the Spring framework
* No access to the source code for the third party code

**Hibernate and JPA**

Hibernate is the connection between your Java application and your database

* Will get rid of the lower level SQL code

JPA

* Jakarta Persistence API
* Defines set of interfaces
* Standard API for Object to relational mapping (ORM)
* Requires an implementation to be usable
* Can switch vendor implementations without needing you alter your source code

Annotations

* @entity
  + Public or protected no argument constructor
* @id
* @column (name=”id”)
  + optional
  + but avoid this because we want to have explicit notation for DB’s
* ID Generation Strategies
  + AUTO
    - Picks appropriate strategy for the particular database
  + IDENTITY (Preferred)
    - Assign primary keys using database identity column
  + SEQUENCE
    - Assign primary keys using a DB sequence
  + TABLE
    - Assign keys using an underlying DB table to ensure uniqueness
  + You can also create a custom strategy
    - Create implementation of org.hibernate.id.IdentifierGenerator
    - Override the method: public Serializable generate(…)

DAO

* Data Access Object
* Used for communicating with the database

Spring is automatically transactional

* @Transactional to make this a transaction

Java JSON Data Binding (Mapping ex)

* Java POJO
  + Converting JSON to another format
  + Plain Old Java Object
* Spring uses Jackson Project
  + Handles the data binding between JSON and Java POJO
  + Essentially takes the JSON and turns it into the appropriate class object

Best practice advice

* Look for the most predominate noun in the request to base it off of what you should use for creating classes
* Follow the CRUD operations
* Create a service between your rest controllers and DAO
* Apply transactional boundaries at the service layer
  + Service layer handles the transactional implementations

Bad patterns

* Don’t use the actual action in the endpoint
  + Api/deleteEmployee etc

Spring Data JPA

* Helps reduce code amount
* Plug in the specific DAO and spring will supply the CRUD implementation for you
* Use JPARepository interface for these features/operations