Spring and Spring Boot

December 2020

Ken Kousen
ken.kousen@kousenit.com
http://www.kousenit.com (home page)
https://kousenit.org (blog)
https://kenkousen.substack.com (newsletter)
@kenkousen

Slides:
https://www.dropbox.com/s/2cil0gwr06klkmr/Deep%20Dive%20Into%20Spring.pdf?dl=0

Exercises:
http://www.kousenit.com/springboot/

This document:
https://docs.google.com/document/d/1AWxNt1F_5UesPCxkPoWcZyoOgvXk3MKtQuOCvJiyLxY/edit?usp=sharing
or
https://tinyurl.com/ybrqode5 (shortened)

Anyone with the link can view this document. No data is being collected.

GitHub repositories:
https://github.com/kousen/spring-and-spring-boot
https://github.com/kousen/shopping_rest

Declarative vs Programmatic services
- Transactions
- Security
- Resource pooling
- Java Mail
- Database connection pools
- …

Programmatic → API
- UserTransaction: begin, commit, rollback
- Write the code to create and commit your transactions

Declarative → Metadata
- XML
- Annotations

JavaConfig → Java class that is read by Spring as metadata

Java is on a six-month lifecycle for major versions
Every March and every September there is a new major version
They established Long Term Support versions (LTS) that are good for three years
Java 8 was retroactively considered an LTS version
The current LTS version is Java 11
The next one will be Java 17, scheduled for September 2021
Nearly 70% of the Java industry is still on Java 1.8
About 20 - 25% have moved to Java 11

Open source vs Java directly from Oracle
JDK download from Oracle → Oracle allows you to use it for free in dev and test, but not prod
vs OpenJDK project → always free
→ both are (almost) entirely the same source code

The current version of Java is 15, but will become 16 in March 2021

Java Is Still Free
https://medium.com/@javachampions/java-is-still-free-2-0-0-6b9aa8d6d244

Testing
- Unit tests → test a class in isolation
    - In Spring, this means no actual server, and Spring itself is not involved
    - Unit tests are normally very fast and automated
- Integration tests → used as part of an application, in whole or in part
    - Spring is involved, but (probably) no actual server
    - Takes more effort to set up and deploy, but faster than driving a browser
    - Spring provides mock objects for most infrastructure
- Functional tests
    - Start a test server, deploy the app, test it programmatically
    - Most realistic, but requires an actual server
    - Spring will start up a test server on a random port and deploy app for you

MockMVC example with a MockBean
https://docs.spring.io/spring-boot/docs/2.4.1/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-mvc-tests

REST principles:
(REST == Representational State Transfer)
1. Addressable resources → every object has its own URL
2. Content negotiation → client specifies the form of the response, usually through an Accept header in the Http request

3. Uniform interface → only the HTTP verbs are allowed (GET, POST, PUT, DELETE, PATCH, OPTIONS, HEAD, …)
4. Hypermedia As The Engine Of Application State (HATEOAS) → Each response includes the links to move to the next step

Functional features in Java 8+

Java was always an Object Oriented language
All methods must be in a class
Makes for odd compromises, like the Math class

```
Math.abs()
Math.max(...)
Math.random()
// all the methods in Math are static; you never instantiate the Math class
```

The functional approach lets you consider methods as first-class objects
Lambda expression is the implementation of a method in an interface
    But in Java, you are restricted to interfaces with only a single abstract method

The functional interfaces in the java.util.function package fall into four categories:

- Consumer → takes a single generic argument and returns nothing
- Supplier → takes no arguments and returns a generic response
- Predicate → takes a single generic argument and returns a boolean
- Function → takes a single generic argument and returns a generic response

For all of these, there are Int, Long, and Double variations to avoid autoboxing
For several of them, there are binary versions
    BiConsumer takes two args and returns nothing
    BiPredicate takes two args and returns a boolean
    BiFunction takes two inputs and returns a single output

A Function where the input and output are the same type is called a UnaryOperator
A BiFunction where both inputs and the output are all the same type is called a BinaryOperator

Before picture:

```
package com.nfjs;

import java.util.*;

public class LoopsSortsAndIfs {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("this", "is", "a",
             "list", "of", "strings");

        // Find even-length strings
        List<String> evens = new ArrayList<>();
        for (String s : strings) {
            if (s.length() % 2 == 0) {
                evens.add(s);
            }
        }

        // Sort them by length
        Collections.sort(evens, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        });

        // Print them out one by one
        for (String s : evens) {
            System.out.println(s);
        }
    }
}
```

After picture:

```
package com.nfjs;

import java.util.stream.Stream;

import static java.util.Comparator.comparingInt;
import static java.util.Comparator.naturalOrder;

public class LoopsSortsAndIfsAfter {
    public static void main(String[] args) {
        Stream.of("this", "is", "a", "list", "of", "strings")
                .filter(s -> s.length() % 2 == 0)
                .sorted(comparingInt(String::length).thenComparing(naturalOrder()))
                .forEach(System.out::println);
    }
}
```

Layered architecture used by all Java web apps since roughly 2000

Presentation layer (views and controllers)
||
Service layer (transactions and business logic)
||
Persistence layer (transform objects to tables and back)

Recently, in the Android world, they defined a Repository as a front end on both a database and a network layer, but in Spring, Repository == Database layer

Spring defines annotations based on @Component:
    @Controller, @RestController → receive HTTP requests and return responses
    @Service → transaction boundaries and business logic
    @Repository → connect to database
    @Configuration → add beans to application context

All exceptions in Spring are RuntimeExceptions (i.e., unchecked)
For repositories, all SqlExceptions are caught and rethrown as one of a family based on DataAccessException

To do the translation, Spring needs a bean of type
PersistenceExceptionTranslationPostProcessor

You autowire on a constructor if the class needs the argument in all cases
You autowire on a method if the class only needs the property optionally

With Hibernate, you had a Session → interact with the database
and a SessionFactory → compiled the SQL, created connection pool, etc.

There should be only one SessionFactory → Let Spring manage that

Along came JPA → Java Persistence API
  Uses a "provider" → the most common provider is Hibernate

Session → EntityManager
SessionFactory → EntityManagerFactory
save → persist
delete → remove

The Hibernate developers say, use the JPA classes and names wherever possible

What you used to do was to autowire an EntityManagerFactory into your DAO, and then for each interaction with the database, you needed a transaction and an EntityManager

Let Spring manage the transactions using the @Transactional annotation on a service

Hibernate also established a concept called the "persistence context"
  configuration of the database
  management of all the entities

https://docs.spring.io/spring-boot/docs/2.4.1/reference/htmlsingle/#howto-set-active-spring-profiles