

# Deep Dive Into Spring

Web Applications, Restful Services, and Persistence

# Contact Info

Ken Kousen

Kousen IT, Inc.

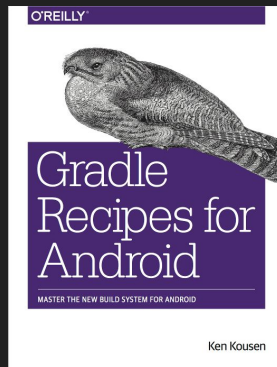
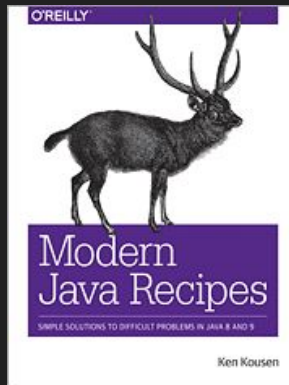
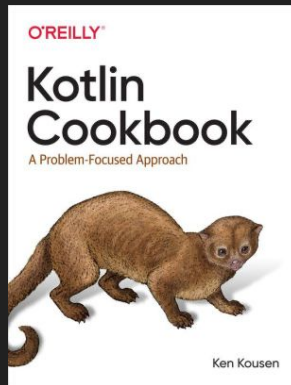
[ken.kousen@kousenit.com](mailto:ken.kousen@kousenit.com)

<http://www.kousenit.com>

<http://kousenit.org> (blog)

[@kenkousen](#) (twitter)

<https://kenkousen.substack.com> (newsletter)



# GitHub repositories

<https://github.com/kousen/spring-and-spring-boot>

[https://github.com/kousen/shopping\\_rest](https://github.com/kousen/shopping_rest)

# Spring

Project infrastructure

# Spring

Lifecycle management of "beans"

Any POJO with getters/setters

# Spring

Provides "services"

transactions, security, persistence, ...

# Spring

Library of beans available

transaction managers

rest clients

DB connection pools

testing mechanisms

# Spring

Need "metadata"

Tells Spring what to instantiate and configure

XML → old style

Annotations → better

JavaConfig → preferred

All still supported



# Spring

## Application Context

Collection of managed beans

the "lightweight" Spring container

# Spring Boot

Easy **creation and configuration** for Spring apps

Many "starters"

Gradle or Maven based

Automatic configuration based on classpath

If you add JDBC driver, it adds DataSource bean

# Spring Initializr

Website for creating new Spring (Boot) apps

<http://start.spring.io>

Incorporated into major IDEs

Select features you want

Download zip containing build file

# Spring MVC Starters

Add either **web** and/or **webflux** starter

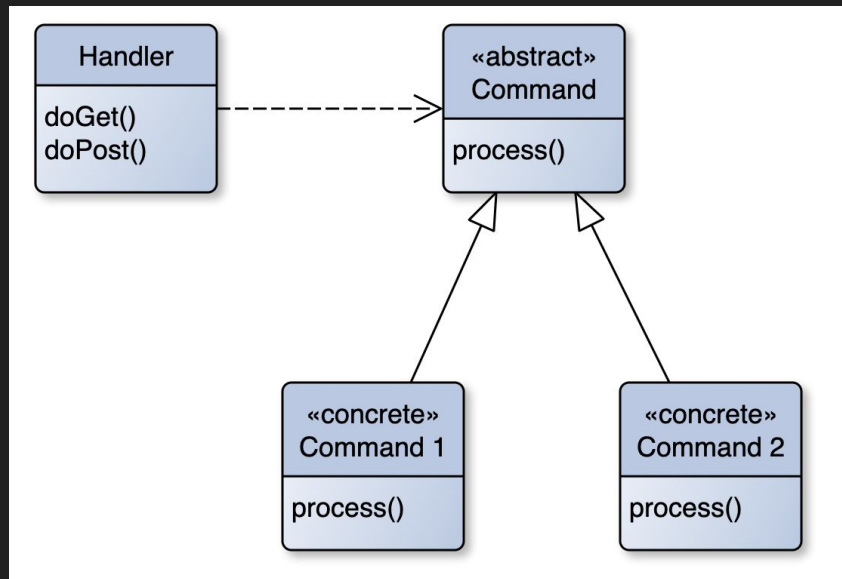
**spring-boot-starter-web** → Spring MVC

**spring-boot-starter-webflux** → Reactive Spring

Need this for **WebClient**

# Spring MVC

Designed around Front Controller design pattern



# Spring Boot

Application with `main method` created automatically

Annotated with `@SpringBootApplication`

Gradle or Maven build produces executable jar in build/libs folder

```
$ java -jar appname.jar
```

Or use gradle task `bootRun`

# @SpringBootApplication

Composite annotation, includes:

- @SpringConfiguration (@Configuration by another name)
- @EnableAutoConfiguration
- @ComponentScan

# DispatcherServlet

Central servlet that acts as front controller

Spring Boot sets up and maps automatically



# Special Beans

Spring library useful bean types

**HandlerMapping** and **HandlerAdapter** → maps URLs to bean methods

**ViewResolver** → Converts strings to views

**HandlerExceptionResolver** → Map exceptions/errors to views

# Processing Requests

## DispatcherServlet:

- Find `WebApplicationContext` and bind request
- Use locale resolver, if necessary
- Use theme resolver, if necessary
- Use multipart file resolver, if necessary
- Use `HandlerMapping` to invoke method
- Use `ViewResolver` to connect to view

# Request Processing

Everything can be configured and customized

# Spring Boot Simplifications

Spring Boot autoconfiguration provides:

- ContentNegotiatingViewResolver
- BeanNameViewResolver
- HttpMessageConverters
- Static content:
  - /static or /public or /resources directory

# Path Matching and Content Negotiation

Disables suffix pattern matching by default

Uses **Accept headers** for content negotiation

**Template Engines** for dynamic HTML content

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

All use `/src/main/resources/templates` by default

# Spring MVC

Annotation based MVC framework

`@Controller`, `@RestController` → controllers

`@GetMapping` → annotations for HTTP methods

Similar for POST, PUT, PATCH, DELETE, ...

`@RequestParam` and more for model parameters

`@PathVariable` for URI templates

# Custom Error Page

In folder `src/main/resources/public/error`

Add `404.html` (or other error code)

More general, add `5xx.html`, etc

# CORS

Cross-origin resource sharing

Easy way: `@CrossOrigin` annotation

More complex: Register a `WebMvcConfigurer` bean  
with `addCorsMappings(CorsRegistry)` method



# Testing

Spring tests automatically include special JUnit 5 extension

```
@ExtendWith(SpringExtension.class)
```

Annotate test class with `@SpringBootTest`

Annotate tests with `@Test`

Use normal asserts as usual

# Testing

Special annotations for web integration tests

```
@WebMvcTest(... controller class ...)
```

MockMvc package

MockMvcRequestBuilders

MockMvcRequestMatchers

# Testing

Integration tests:

`WebTestClient` (newer, reactive)

`TestRestTemplate` (older, synchronous)

# Mock Objects

Includes Mockito

`@MockBean`

Set expectations and verify as usual

# Component Scan

Spring detects annotated classes in the expected folders

`@Component` → Spring bean

`@Controller`, `@Service`, `@Repository`, `@Configuration`  
→ based on `@Component`

# Application properties

Two options for file name

Default folder is `src/main/resources`

`application.properties` → standard Java properties file

`application.yml` → YAML format

# Web Apps

Add `Model` parameter to controller methods

Carries data from controllers to views

Model attributes copied into each request

# Validation

Spring uses any JSR-303 implementation on classpath

Hibernate validator by default

`@Valid`

`@Min, @Max, @NotBlank, ...`



# Persistence

More conventions:

Two standard files in `src/main/resources`

`schema.sql` → create test database

`data.sql` → populate test database

Both executed on startup, using DB connection pool

`application.properties`:

`spring.datasource.schema`, `spring.datasource.data`

# JdbcTemplate

Standard practice:

- Create DAO class

- Autowire DataSource into constructor

- Instantiate JdbcTemplate from DataSource

Spring boot lets you autowire the JdbcTemplate directly

# JdbcTemplate

Use `queryForObject` to map DB row to Java class

(`query` method does the same for all rows)

In Java 7, uses inner class that implements `RowMapper<MyClass>`

In Java 8, can use lambda expression

Spring 5, Spring Boot 2 both require Java 8+

# H2 Database

- Add the H2 dependency
  - `runtime('com.h2database:h2')`
  - Automatically adds DataSource for it

If you add the web starter and the dev-tools dependency,

you also get the H2 console

<http://localhost:8080/h2-console>

DB URL (by default) is `jdbc:h2:mem:testdb`

# SimpleJdbcInsert

Specify table name and generated key columns

Create a `SqlParameterSource`

Run `executeAndReturnKey(parameters)`

# Transactions

Spring transactions configured with `@Transactional`

Spring uses `TransactionManager` to talk to resource

usually a relational DB, but other options available

# @Transactional

Each method wrapped in a REQUIRED tx by default

Propagation levels:

REQUIRED, REQUIRES\_NEW, SUPPORTS, NOT\_SUPPORTED

In tests, transactions in test methods roll back by default

Can configure isolation levels:

READ\_UNCOMMITTED, READ\_COMMITTED,

REPEATABLE\_READ, SERIALIZABLE

# JPA

## Java Persistence API

Uses a "provider" → **Hibernate** most common

Annotate entity classes

`@Entity`, `@Table`, `@Column`, `@Id`, `@GeneratedValue`

use in Spring `@Repository` → exception translation

`@PersistenceContext` → `EntityManager`



# Spring Data

Large, powerful API

Create interface that extends a given one

`CrudRepository`, `PagingAndSortingRepository`

We'll use `JpaRepository<class, serializable>`

Add your own finder method declarations

All SQL generated automatically

# Profiles

Create the same beans to be used under different situations

Either:

Multiple files with profile name in them  
application-{profilename}.properties

Or:

One YAML file with section separated by ---

# Profiles

logging:

level:

org.springframework.web: DEBUG

---

spring:

profiles: prod

datasource: ...

---

spring:

profiles: dev

datasource: ...

# Profiles

Annotate beans for specific profiles

```
@Profile("dev")  
@Profile({"dev", "prod"})  
@Profile("!test")
```

Set the active profile:

```
spring.profiles.active = prod
```

Set `SPRING_PROFILES_ACTIVE` environment variable

```
--spring.profiles.active=prod
```

 on command line

# Web.fn

Functional approach

Router function bean

maps URLs to handler methods

Kotlin has a nice DSL for it

Handler class

all methods take `ServerRequest` and return `ServerResponse`

# How Does Spring Do Its Job?

- Load bean definitions from all sources
- Post-process bean definitions
  - Substitute values at config time, like JDBC properties
  - Read values from `application.yml` or `application.properties`
- Create and configure all the beans
  - Set properties and dependencies
- Run bean post-processors
  - Generate any necessary proxies

# HAL Browser

Browser used to access RESTful web services

Executes HTTP methods

Parses JSON responses

Handles hypermedia

# References

- [Spring in Action](#), 5th edition, by Craig Walls
- For Hibernate/JPA
  - Pro JPA 2, 2nd edition (on Safari)
  - <https://thoughts-on-java.org/> has lots of Hibernate tips
  - <https://vladmihalcea.com/books/high-performance-java-persistence/>
- Online reference docs for:
  - Spring Framework
  - Spring Boot
  - Spring Data
  - Spring Security