Spring Boot

Getting Java / Spring Applications
up and running quickly

Module Outline

- Spring Boot
- Web Applications with Boot
  - Thymeleaf, JSP
- Spring Data JPA
- Spring Data REST

What is Spring Boot

- Radically faster getting started experience

- "Opinionated" approach to configuration / defaults
  - Intelligent defaults
  - Gets out of the way quickly

- What does it involve?
  - Easier dependency management
  - Automatic configuration / reasonable defaults
  - Different build / deployment options.

What Spring Boot is NOT

- Plugins for IDEs
  - Use Boot with any IDE (or none at all)
- Code generation

Demonstration – Spring Boot

Creating a new, bare-bones Spring application

Spring Boot – What Just Happened?

- Boilerplate project structure created
  - Mostly folder structure
  - “Application” class + test
  - Maven POM (or Gradle if desired)

- Dependency Management

Demonstration –
Running a Spring Boot Project

Running the newly created project

Running Spring Boot –
What Just Happened?

- SpringApplication
  - Created Spring Application Context

- @SpringBootApplication
  - Combination of @Configuration
    - Marks a configuration file
    - Java equivalent of <beans > file
  - ...And @ComponentScan
    - Looks for @Components (none at the moment)
  - ...And @EnableAutoConfiguration
    - Master runtime switch for Spring Boot
    - Examines ApplicationContext & classpath
    - Creates missing beans based on intelligent defaults

Demonstration –
Adding Web Capability

- Adding spring-boot-starter-web dependency
- Adding HelloController

Adding Web – What Just Happened?

- spring-boot-starter-web Dependency
  - Adds spring-web, spring-mvc jars
  - Adds embedded Tomcat jars

- When application starts...
  - Your beans are created
  - @EnableAutoConfiguration looks for 'missing' beans
    - Based on your beans + classpath
    - Notices @Controller / Spring MVC jars
  - Automatically creates MVC beans
    - DispatcherServlet, HandlerMappings, Adapters, ViewResolvers
  - Launches embedded Tomcat instance.

But wait, I want a WAR...

- To Convert from JAR to WAR:
  - Change POM packaging
  - Extend SpringBootServletInitializer

```
public class Application extends SpringBootServletInitializer {

    public static void main(String[] args) {
        SpringApplication.run(Config.class, args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Config.class);
    }

}
```

  - Deploy to app server
    - URL becomes http://localhost:8080/<app>/

Demonstration – WAR Deployment

- WAR Packaging
- SpringBootServletInitializer

What about Web Pages?

- Spring MVC supports a wide range of view options
- Easy to use JSP, Freemarker, Velocity, Thymeleaf
- Boot automatically establishes defaults
  - InternalResourceViewResolver for JSPs
  - ThymeleafViewResolver
    - IF Thymeleaf is on the classpath
- spring-boot-starter-thymeleaf

Demonstration –
Thymeleaf web pages

- spring-boot-starter-thymeleaf
- /templates folder
- Controller adjustments
- Web page

What Just Happened?

- spring-boot-starter-thymeleaf
  - Brought in required jars
  - Automatically configured ThymeleafViewResolver
- Controller returned a 'logical view name'
- ViewResolver found a matching template
- Render

But wait, I want JSPs...

- Thymeleaf and other templating approaches are way too advanced for my organization!
  - Besides, we have lots of existing JSPs

- No Problem!

- Just as easy to use JSPs!
  - Place JSPs in desired web-relative location
  - Set spring.mvc.view.prefix / spring.mvc.view.suffix as needed.
  - (remove thymeleaf starter pom)

Demonstration – JSP Web Pages

- Place JSP in desired folder
- Set spring.mvc.view.prefix / spring.mvc.view.suffix
- Exclude spring-boot-starter-tomcat

What Just Happened?

- No ThymeleafViewResolver configured
- Controller returned a 'logical view name'
- InternalResourceViewResolver forwarded to JSP
- Render

Spring & REST

- REST capability is built in to Spring MVC
  - Simply use domain objects as parameters / return values.
  - Mark with @RequestBody / @ResponseBody
  - Spring MVC automatically handles XML / JSON conversion
    - Based on converters available in classpath.

Demonstration –
REST Controllers in Spring MVC

- Additional domain objects
- Automatic HTTP Message Conversion

What Just Happened?

- Controller returned a domain object
  - NOT a logical view name (page)

- Spring MVC noticed @ResponseBody
  - Or @RestController

- Invoked correct HttpMessageConverter
  - Based on
    - Requested format
    - JARS on classpath

What if I want XML?

- No Problem!

- Annotate domain classes with JAXB annotations
  - JAXB already part of Java SE

- When App Starts
  - Spring creates HttpMessageConverter for JAXB
    - Based on classpath contents

- XML or JSON returned
  - based on requested format

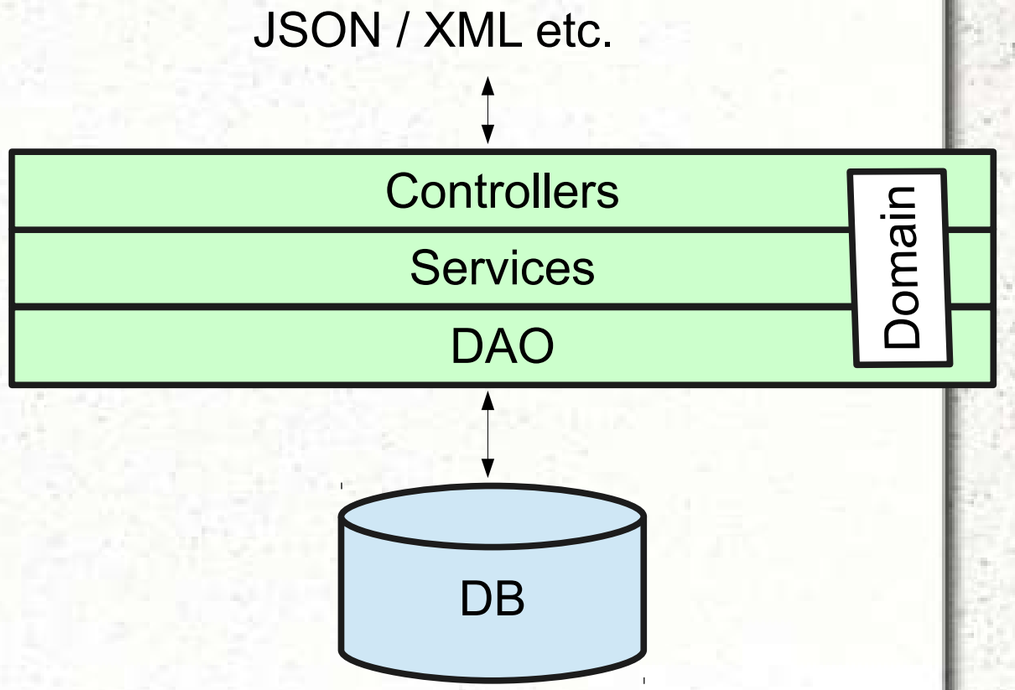
Adding JPA Capability

- Adding the spring-boot-starter-data-jpa Dependency
  - Adds Spring JDBC / Transaction Management
  - Adds Spring ORM
  - Adds Hibernate / entity manager
  - Adds Spring Data JPA subproject
    - (explained later)

- Does NOT add a Database Driver
  - Add one manually (HSQL)

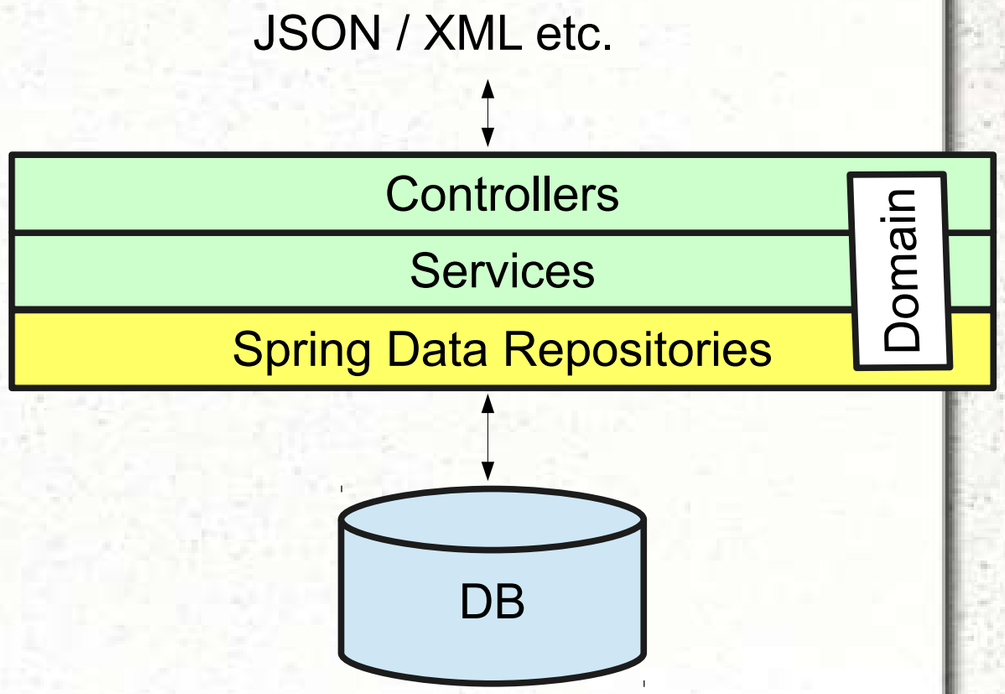
Spring Data JPA

- Typical web application architecture
- REST Controllers provide CRUD interface to clients
- DAO provide CRUD interface to DB



Spring Data – Instant Repositories

- Spring Data provides *dynamic repositories*
- You provide the interface, Spring Data dynamically implements.
  - JPA, MongoDB, GemFire, etc.
- Service Layer / Controllers have almost no logic.



Demonstration –
Adding Spring Data JPA

- spring-boot-starter-data-jpa
- org.hsqldb / hsqldb
- Annotate domain objects with JPA
- Extend CrudRepository

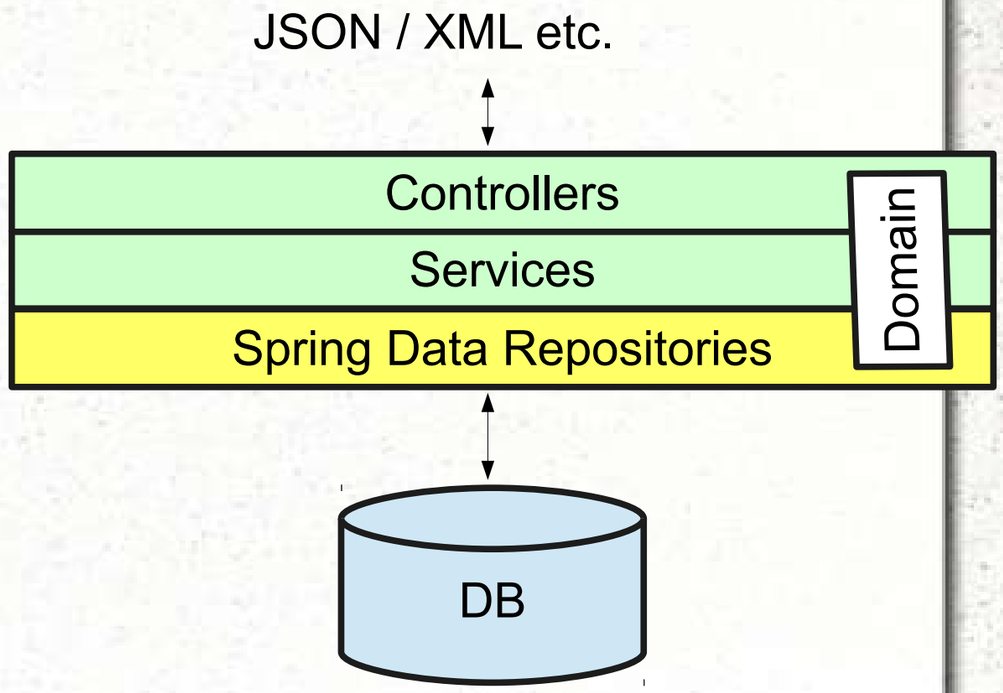
Adding Spring Data JPA
– What Just Happened?

- What I did:
  - Added dependencies for spring-boot-starter-data-jpa and hsqldb
  - Annotated Domain objects with plain JPA annotations
  - Added an interface for Spring Data JPA
  - Dependency injected into controller

- When application starts...
  - Spring Data dynamically implements repositories
    - find*(), delete(), save() methods implemented.
  - DataSource, Transaction Management, all handled.

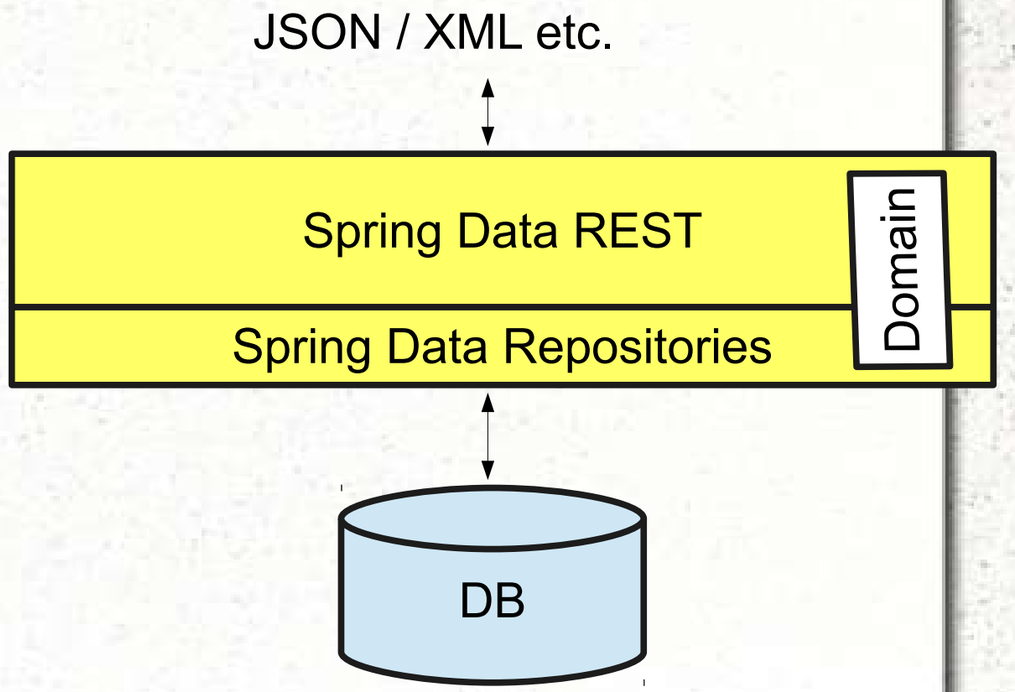
Spring Data – REST

- Often, applications simply expose DAO methods as REST resources
- Spring Data REST handles this automatically...



Adding Spring Data REST

- Plugs into dynamic repositories
- Generates RESTful interface
  - GET, PUT, POST, DELETE
- Code needed only to override defaults.



Demonstration –
Adding Spring Data REST

- spring-boot-starter-data-rest
- @RestResource(path=”teams”, rel=”team”)

Adding Spring Data REST
– What Just Happened?

- When application starts...
  - @RestResource annotations interpreted
  - @Controllers beans created
  - @RequestMappings created

Adding HATEOAS

- Spring Data Rest simply returns RESTful resources
  - Conversion handled by Jackson, or JAXB
- Underlying Data Relationships used to build Links
  - IF matching repositories exist
- Consider the Team → Player relationship

- Player Repository needed to force link creation

Demonstration – Adding HATEOAS Links

- Creating a Player DAO

HATEOAS
– What Just Happened?

- Spring Data REST noticed two repositories
  - The relationship between entities is know via JPA annotations.
- Spring automatically represents the children as links
  - @RestResource determines names of links

Summary

- Spring Boot makes it easy to start projects
  - And easy to add feature sets to projects
  - Opinionated approach
  - Run as JAR or WAR
  - Web Applications (JSP, Thymeleaf, others)
- REST
  - Automatic resource conversion
- Spring Data JPA
  - Automatic repository implementation
- Spring Data REST
  - Automatic REST controllers

Exercise

Create a Spring Boot Application
Utilizing Spring Data JPA and
Spring Data REST