Agile plan for 2 days

Boot & MVC

Test & Data & Transactions

AOP & SPEL

Optional:

Domain Driven Design

Approaches to testing

Spring boot

Agile plan for today

Boot events and bootstrap

Setting up and autoconfiguration

Tuning autoconfiguration

Turning off autoconfiguration

Component scan gotchas

JavaConfig vs @Autwired

Properties, Profiles, Conditions

SSH Commands in your app

Who are you

What's your name

What do you do @Allegro

What's your experience in Java

What's your experience in Spock

What's your experience in Spring

What's your experience in Spring MVC

What's your experience in Spring Cloud

What do you expect from this workshop

Demo time

Create a new app from the scratch.
How long does it take?
Why it doesn’t work?

1. **get a build.gradle from http://start.spring.io/**

2. **create the controller**

   gedit src/main/groovy/eu/solidcraft/SampleController.groovy

```
@Controller @EnableAutoConfiguration
public class SampleController {
  @RequestMapping("/") @ResponseBody
  String home() {
    return "Hello World!"
  }
  public static void main(String[] args) throws Exception {
    SpringApplication.run(SampleController.class, args)
  }
}
```

3. **run**

   gradle run

Get a deployable version java -jar jar

```
gradle build
```

```
├── build
│   ├── libs
│       ├── spring-boot-workshop-0.0.1-SNAPSHOT.jar
├── build.gradle
└── src
    └── main
        └── groovy
            └── eu
                └── solidcraft
                    └── SampleController.groovy
```

```
java -jar build/libs/....jar
```

compare .jar with .jar.original

What is spring-boot?

Kind of twitter-bootstrap for your Spring projects.

‘Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.’

How does this work?

Gradle or Maven plugin

“The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, allowing you to package **executable jar** or war archives, run Spring Boot applications and **omit version information** from your build.gradle file for “blessed” dependencies.”

Fat jar

This jar is handy because it **includes all the other dependencies** and things like your web server inside the archive. You can give anybody this one .jar and they can run your entire Spring application with no fuss: no build tool required, no setup, no web server configuration, etc: just

java -jar ...your.jar.

Embedded server

gradle dependencies | grep tomcat

How is it configured?

@EnableAutoConfiguration

Enable auto-configuration of the Spring Application Context, attempting to **guess and configure beans that you are likely to need** [...] based on your **classpath** and what beans you have defined.

For example, If you have tomat-embedded.jar on your classpath you are likely to want a TomcatEmbeddedServletContainerFactory (unless you have defined your own EmbeddedServletContainerFactory bean).

Auto-configuration tries to be as intelligent as possible and will **back-away as you define** more of your own configuration. You can always manually exclude() any configuration that you never want to apply.

It is generally recommended that you place@EnableAutoConfiguration in a **root package** so that all sub-packages and classes can be searched (or you can specify @ComponentScan(basePackages = {**"..."**})).

Condition out of the box

@ConditionalOnResource

@ConditionalOnExperssion(SpEL)

@ConditionalOnProperty

@ConditionalOnJava

@ConditionalOnClass

@ConditionalOnMissingBean

Dependencies: gradle plugin

```
buildscript {
   dependencies {
       classpath("org.springframework.boot:spring-boot-gradle-plugin:1.2.3.RELEASE")
       classpath("io.spring.gradle:dependency-management-plugin:0.4.1.RELEASE")
   }
}

dependencies {
   compile("org.springframework.boot:spring-boot-starter-actuator")
   compile("org.springframework.boot:spring-boot-starter-aop")
   compile("org.springframework.boot:spring-boot-starter-data-jpa")
   compile("org.springframework.boot:spring-boot-starter-jdbc")
   compile("org.springframework.boot:spring-boot-starter-remote-shell")
   compile("org.springframework.boot:spring-boot-starter-web")
}
```

The main method

```
@EnableAutoConfiguration
@Configuration
@ComponentScan //or @Import to load selectively
public class MyApplicationConfiguration {
  public static void main(String[] args) throws Exception {
    SpringApplication.run(MyApplicationConfiguration.class, args)
  }
}
```

The main method

```
public static void main(String[] args) throws Exception {
    SpringApplication application = new SpringApplication(SampleController.class) //I haz an app
    application.logStartupInfo = false //I configure smth
    //Also I do whatever I like, hell yeah!
    application.run(args) //I run the app
}
```

The main method: builder

```
public static void main(String[] args) throws Exception {
   new SpringApplicationBuilder()
           .showBanner(false)
           .sources(Parent.class)
           .child(Application.class)
           .run(args)
}
```

maven/gradle plugin != spring-boot

You don’t need fat jars or plugins for spring-boot.

You can add dependencies manually, and use only the parts you like.

No magic included.

Starters

Setting up dependencies for you

Because: http://en.wikipedia.org/wiki/Dependency_hell

Available starters

spring-boot-starter The core starter, including auto-configuration support, logging and YAML.
spring-boot-starter-actuator Production ready features to help you monitor and manage application.
spring-boot-starter-amqp Support for the “Advanced Message Queuing Protocol” via spring-rabbit.
spring-boot-starter-aop Support for aspect-oriented programming including spring-aop and AspectJ.
spring-boot-starter-batch Support for “Spring Batch” including HSQLDB database.
spring-boot-starter-data-elasticsearch Support for the Elasticsearch search and analytics engine
spring-boot-starter-data-gemfire Support for the GemFire distributed data store including spring-data-gemfire.
spring-boot-starter-data-jpa Support for the “Java Persistence API” including spring-data-jpa, spring-orm and Hibernate.
spring-boot-starter-data-mongodb Support for the MongoDB NoSQL Database, including spring-data-mongodb.
spring-boot-starter-data-rest Support for exposing Spring Data repositories over REST via spring-data-rest-webmvc.
spring-boot-starter-data-solr Support for the Apache Solr search platform, including spring-data-solr.
spring-boot-starter-freemarker Support for the FreeMarker templating engine
spring-boot-starter-groovy-templates Support for the Groovy templating engine
spring-boot-starter-hornetq Support for “Java Message Service API” via HornetQ.

Available starters

spring-boot-starter-integration Support for common spring-integration modules.
spring-boot-starter-jdbc Support for JDBC databases.
spring-boot-starter-jta-atomikos Support for JTA distributed transactions via Atomikos.
spring-boot-starter-jta-bitronix Support for JTA distributed transactions via Bitronix.
spring-boot-starter-mobile Support for spring-mobile
spring-boot-starter-redis Support for the REDIS key-value data store, including spring-redis.
spring-boot-starter-remote-shell Support for CRaSH.
spring-boot-starter-security Support for spring-security.
spring-boot-starter-social-facebook Support for spring-social-facebook.
spring-boot-starter-social-linkedin Support for spring-social-linkedin.
spring-boot-starter-social-twitter Support for spring-social-twitter.
spring-boot-starter-test JUnit, Hamcrest and Mockito along with the spring-test module.
spring-boot-starter-thymeleaf Support for the Thymeleaf templating engine
spring-boot-starter-velocity Support for the Velocity templating engine
spring-boot-starter-web Support for full-stack web development, including Tomcat and webmvc
spring-boot-starter-websocket Support for WebSocket development.
spring-boot-starter-ws Support for Spring Web Services
spring-boot-starter-actuator Adds production ready features such as metrics and monitoring.
spring-boot-starter-remote-shell Adds remote ssh shell support.

Available starters

spring-boot-starter-jetty Imports the Jetty HTTP engine (to be used as an alternative to Tomcat)
spring-boot-starter-log4j Support the Log4J logging framework
spring-boot-starter-logging Import Spring Boot’s default logging framework (Logback).
spring-boot-starter-tomcat Import Spring Boot’s default HTTP engine (Tomcat).
spring-boot-starter-undertow Imports the Undertow HTTP engine (alternative to Tomcat)

...back away as you define...

Add HSQLDB or H2

Spring-boot will configure in-memory db

Define your own DataSource

No more in-memory db

You can also exclude manually

- add spring-data-jpa

  ```
  compile("org.springframework.boot:spring-boot-starter-data-jpa")
  ```

- gradle run - will not work (no datasource, and no db lib on classpath)
- exclude all autoconfig requiring datasource

  ```
  @EnableAutoConfiguration(exclude=[
          DataSourceAutoConfiguration, JpaRepositoriesAutoConfiguration, HibernateJpaAutoConfiguration])
  ```

- gradle run - will work

There is a shortcut

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

But remember: component scan is cool for micro only.

No modules. No shared whatever. No medium/big projects.

Subscribing to events

new SpringApplicationBuilder().listeners(**/*here be listeners*/**).run(args);

Basic context

ContextClosedEvent, ContextRefreshedEvent, ContextStartedEvent, ContextStoppedEvent,

Servlet

PortletRequestHandledEvent, ServletRequestHandledEvent, BrokerAvailabilityEvent

Websockets:

SessionConnectedEvent, SessionConnectEvent, SessionDisconnectEvent, SessionSubscribeEvent, SessionUnsubscribeEvent

Spring-boot specific:

ApplicationStartedEvent, ApplicationEnvironmentPreparedEvent, ApplicationPreparedEvent,

ApplicationFailedEvent

When JVM says bye bye

Last chance to react:

implement DisposableBean interface

add @PreDestroy to a bean method

implement ExitCodeGenerator interface

Logging

Logging

Commons Logging for all internal logging

Default configurations are provided for Java Util Logging,Log4J and Logback.

By default Logback

To configure

Logback - logback.xml

Log4j - log4j.properties or log4j.xml

JDK (Java Util Logging) - logging.properties

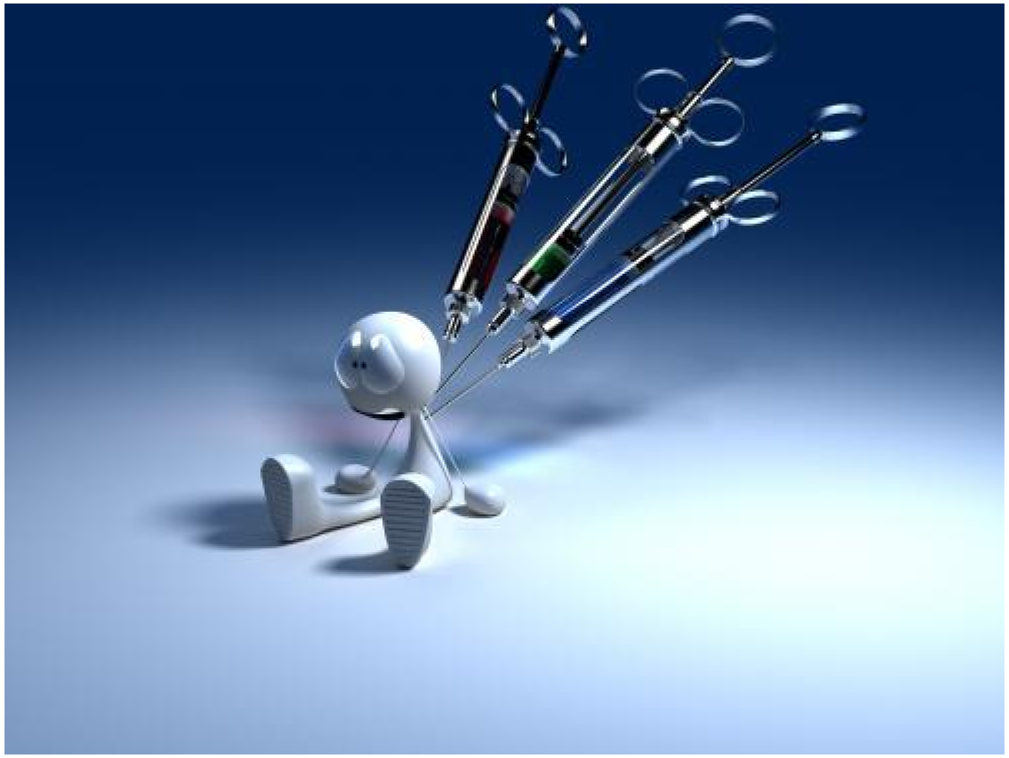
IoC Container

What do we keep in a container?

Services, controllers, all other objects

… except what we keep in database
(stateful objects: entities, a.k.a. model)

Dependency Injection
Inversion of control



No inversion

```
class SomeController {
    private SomeEntityRepository someEntityRepository = new SomeEntityRepository()

    public Map mine() {
        List<SomeEntity> entities = someEntityRepository.findByUsername("some")
        return ["entities": entities]
    }
```

Problem?

You cannot change someEntityRepository to anything else without recompilation.

You cannot test on a mock or on a different repository implementation.

SOLID (object-oriented design)

http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Initial	Stands for (acronym)	Concept
S	SRP	**Single responsibility principle** a class should have only a single responsibility.
O	OCP	**Open/closed principle** “software entities … should be open for extension, but closed for modification”.
L	LSP	**Liskov substitution principle** “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”. See also design by contract.
I	ISP	**Interface segregation principle** “many client-specific interfaces are better than one general-purpose interface.”[5]
D	**DIP**	**Dependency inversion principle** one should “Depend upon Abstractions. Do not depend upon concretions.”[5] Dependency injection is one method of following this principle.

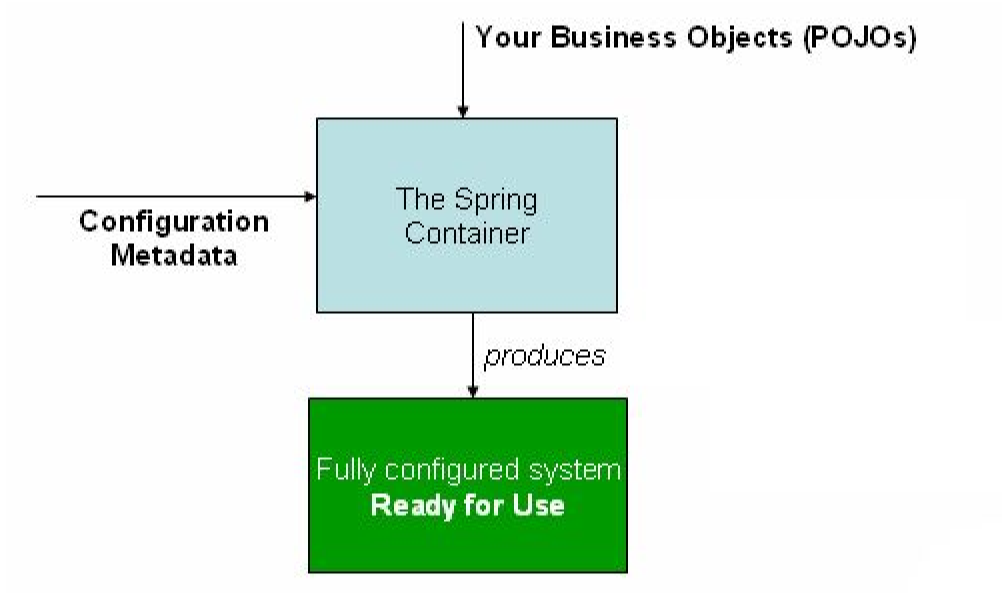
Solution

“a software design pattern that allows the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time.”

Dependency Injection

```
class SomeController {
    private final SomeEntityRepository someEntityRepository

    public SomeController(SomeEntityRepository someEntityRepository) {
        this.someEntityRepository = someEntityRepository
    }

    public Map mine() {
        List<SomeEntity> entities = someEntityRepository.findByUsername("some");
        return ["entities": entities]
    }
```

IoC Container



IoC / DI

IoC: Inversion of Control

DI: Dependency Injection

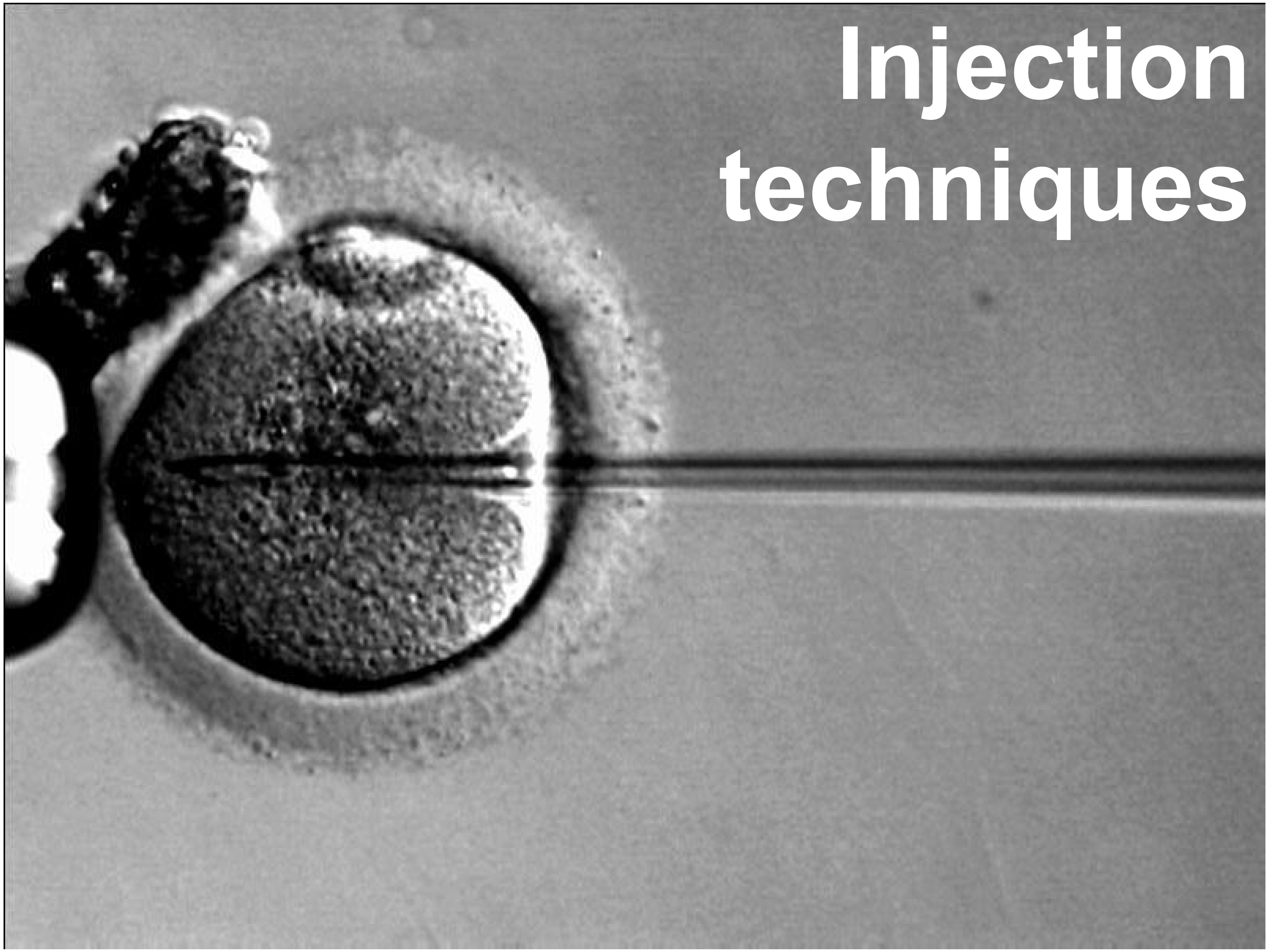
“As a result I think we need a more specific name for this pattern. Inversion of Control is too generic a term, and thus people find it confusing. As a result with a lot of discussion with various IoC advocates we settled on the name Dependency Injection.”

http://martinfowler.com/articles/injection.html 2004

IoC container

org.springframework.context.ApplicationContext

represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans



Injection techniques

Injection via constructor

```
public class WelcomeController {
   private final WorkshopRepository workshopRepository;
   private final LoggedUserRepository loggedUserRepository;
   private final UserRepository userRepository;

   public WelcomeController(
               WorkshopRepository workshopRepository,
               LoggedUserRepository loggedUserRepository,
               UserRepository userRepository) {
      this.workshopRepository = workshopRepository;
      this.loggedUserRepository = loggedUserRepository;
      this.userRepository = userRepository;
   }
```

Injection via setters

```
public class WelcomeController {
    private WorkshopRepository workshopRepository;
    private LoggedUserRepository loggedUserRepository;
    private UserRepository userRepository;

public void setWorkshopRepository(WorkshopRepository workshopRepository) {
    this.workshopRepository = workshopRepository;
}

public void setLoggedUserRepository(LoggedUserRepository loggedUserRepository) {
    this.loggedUserRepository = loggedUserRepository;
}

public void setUserRepository(UserRepository userRepository) {
    this.userRepository = userRepository;
}
```

Injection by magic

```
public class WelcomeController {
   private WorkshopRepository workshopRepository;
   private LoggedUserRepository loggedUserRepository;
   private UserRepository userRepository;

  //BADUM! Tssssss...
```

Configuring container metadata

or how to register beans in the container

@Annotations

because we are lazy

How to turn it on

```
@ComponentScan(basePackages = {"eu.solidcraft.forf"})
public class Application {...}


@Component
class SomeService {
    private final SomeEntityRepository someEntityRepository

    @Autowired
    SomeService(SomeEntityRepository someEntityRepository) {
        this.someEntityRepository = someEntityRepository
    }
```

Autowiring by

name

using id, name or alias

type

assuming you have only one object with that type

Now we have a few options

@Autowired - applies to methods, constructors and fields

@Required - applies to bean property setter methods

@Resource - from JSR 250 (setters and fields)

@Inject and @Named - from JSR 330

Which one to use?
Rule of a thumb.

If in doubt, use @Autowired

If you want to be able to get rid of Spring,
use @Inject

@Autowired and pointing to id

```
public class MovieRecommender {
    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;
…
}
```

@Autowired and pointing to id

```
public class MovieRecommender {
    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;
…
}

@Component(“main”)
class MovieCatalog {
}
```

@Autowired: required by default

```
@Autowired(required=false)
public void setMovieFinder(MovieFinder movieFinder) {
	this.movieFinder = movieFinder;
}
```

@Autowired: constructors

“Only *one annotated constructor per-class* can be marked as *required*, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the *greediest* constructor whose dependencies can be satisfied, that is the constructor that has the largest number of arguments.”

Registering a bean with annotations

@Component(“someName”)

@Service(“someName”)

@Repository(“someName”)

@Controller(“someName”)

@RestController(“someName”)

JSR 330 annotations

```
@Named("movieListener")
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(
                @Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Spring vs JEE annotations

Table 4.6. Spring annotations vs. standard annotations

Spring	javax.inject.*	javax.inject restrictions / comments
@Autowired	@Inject	@Inject has no *required* attribute
@Component	@Named	-
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring’s prototype. However, in order to keep it consistent with Spring’s general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring’s @Scope annotation. javax.inject also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.
@Qualifier	@Named	-
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent

@PostConstruct & @PreDestroy

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }

}
```

@Lazy

```java
@Component
@Lazy
public class SomeLazyBastard {
}
```

Indicates whether a bean is to be lazily initialized. If not present: initialization on startup

If present: when referenced

Has a value (true|false) but why would you?

@Lazy

```
@Autowired
public SomeLazyBastard(@Lazy UserService userService) {
  this.userService = userService;
}
```

In addition to its role for component initialization, this annotation may also be placed on injection points marked with @Autowired or @Inject.

In that context, it leads to the creation of a lazy-resolution proxy for all affected dependencies

Annotation method is limited

Register two beans of the same class, under dirrefent id, with different configuration.

Java config

power for real projects

@Configuration & @Bean

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

Because I know you will look at slides instead of documentation :)

The @Bean annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container.

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions. Furthermore, @Configuration classes allow inter-bean dependencies to be defined by simply calling other @Bean methods in the same class.

Changing id

```
@Configuration
public class AppConfig {

    @Bean(name=”someService”)
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

Aliasing

```
@Configuration
public class AppConfig {

    @Bean(name={”someService”, “some2”})
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

Importing, or keeping it simple

```
@Configuration
public class ConfigA {
     @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {
    @Bean
    public B b() {
        return new B();
    }
}
```

Dependencies

```
@Configuration
public class AppConfig {

    @Bean
    public Foo foo() {
        return new Foo(bar());
    }

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

Dependencies

```
@Configuration
public class AppConfig {

    @Bean
    public Foo foo() {
        return new Foo(bar());
    }

    @Bean
    public Bar bar() {  //how many times will this be called?
        return new Bar();
    }
}
```

Dependencies

```
@Configuration
public class AppConfig {

    @Bean
    public Foo foo(Bar bar) {
        return new Foo(bar);
    }

    @Bean
    public Bar bar() {  //how many times will this be called?
        return new Bar();
    }
}
```

Because I know you will look at slides instead of documentation :)

The @Bean methods in a Spring component are processed differently than their counterparts inside a Spring @Configuration class. The difference is that@Component classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within @Configuration classes @Bean methods create bean metadata references to collaborating objects. Methods are *not* invoked with normal Java semantics. In contrast, calling a method or field within a @Component classes @Bean method *has* standard Java semantics.

Dependencies from other files

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }

}
```

@Lazy

```
@Lazy @Bean
public SomeLazyBastard someLazyBastard() {
  return new SomeLazyBastard();
}
```

Works on @Bean just the same.

If Lazy is present on a Configuration, all @Bean methods within that @Configuration should be lazily initialized.

Can be overridden on a method level.

@Annotations advantages

Close to your code... but we are already swimming in annotations

Close to your code, so you see it right away

Just one language (if you don’t like XML…)

...but limited (you will need Java/XML anyway)

Java Config advantages

Just one language (if you don’t like XML)

Setup logic can be easily written in pure Java

Much more powerful than @Annotations, especially in big projects

Setup logic can be advanced… but why do you need it?

The JVM world kind of drifts into this direction (Dagger1, Dagger2, Guice)

Scopes

Scopes

Singleton (default) - one per application

Prototype - created new for each reference

Request

Session

Global Session - portlets only

you can have your own

(instance per user, for example?)

Prototype Scope

```
@Component
@Scope(value = BeanDefinition.SCOPE_PROTOTYPE)
class MySpecialBean {
}

@Autowired
private MySpecialBean myVeryOwnBean;
```

Session Scope

```
@Component
@Scope(value= WebApplicationContext.SCOPE_SESSION)
public class LoggedUserInSessionRepository {
    private User loggedInUser;

    public void login(User user) {
        this.loggedInUser = user;
    }

    public User getLoggedUser() {
        verifyUserExists();
        return loggedInUser;
    }

    private void verifyUserExists() {
        if(loggedInUser == null) {
            throw new RuntimeException("No user logged in");
        }
    }
}
```

Injecting smaller scope into a larger

“If you want to inject (for example) an HTTP request scoped bean into another bean, you must inject an AOP proxy in place of the scoped bean.“

Types of proxies

CGLIB-based Class proxy

proxy-target-class="true"

Interface-based proxy

proxy-target-class="false"

Define proxy per bean

```
@Component
@Scope(value= WebApplicationContext.SCOPE_SESSION,
            proxyMode= ScopedProxyMode.TARGET_CLASS)
public class LoggedUserInSessionRepository
      implements LoggedUserRepository {

   private User loggedInUser;

}
```

Remember JEE vs Spring differences?

Table 4.6. Spring annotations vs. standard annotations

Spring	javax.inject.*	javax.inject restrictions / comments
@Autowired	@Inject	@Inject has no *required* attribute
@Component	@Named	-
@Scope("singleton")	**@Singleton**	**The JSR-330 default scope is like Spring's prototype. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring's @Scope annotation. javax.inject also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.**
@Qualifier	@Named	-
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent

Profiles

Profiles with annotations

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

Profiles work with tests

```
@Transactional
@TransactionConfiguration(defaultRollback = true)
@ContextConfiguration(locations = ["classpath:/spring/test.ioc.xml"])
@ActiveProfiles(profiles = ['pricing.test'])
abstract class IntegrationSpec extends Specification {
```

@Conditional inside

The @Conditional annotation indicates specific org.springframework.context.annotation.Condition implementations that should be consulted before a @Bean is registered.

```
...
@Conditional(ProfileCondition.class)
public @interface Profile {
```

Profile Condition

```
class ProfileCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        if (context.getEnvironment() != null) {
            MultiValueMap<String, Object> attrs =
                        metadata.getAllAnnotationAttributes(Profile.class.getName());
            if (attrs != null) {
                for (Object value : attrs.get("value")) {
                    if (context.getEnvironment().acceptsProfiles(((String[]) value))) {
                        return true;
                    }
                }
                return false;
            }
        }
        return true;
    }
}
```

Profiles are NOT exclusive

```
@Configuration
@EnableWebSecurity
@Profile([Profiles.DEVELOPMENT, Profiles.TEST])
class HttpBasicSecurityConfig {
```

How to turn on profiles

//JVM param or system environment variable

```
-Dspring.profiles.active="pricing.development"
```

//servlet context parameter

```
<servlet>
      <servlet-name>dispatcher</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <init-param>
          <param-name>spring.profiles.active</param-name>
          <param-value>production</param-value>
      </init-param>
  </servlet>
```

//in code

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

Special profile

If no profile is specified, it is equal to “**default**” profile. You can set it up yourself:

Command line:

```
-Dspring.profiles.default=production
```

Env variable:

```
export spring_profiles_default=production
```

Property:

```
spring.profiles.default=dev
```

Properties

good old properties

Properties

Set up defaults (for example in application.properties)

Give other on production via:

- /config/application.properties subdir in current dir
- /application.properties in current dir
- application-{profile}.properties or YAML files
- OS environment variables.
- Java System properties
- Command line arguments (--server.port=9000)
- ...more

Don't forget about Profiles.

You can often keep properties for staging/testing/dev in the repo, under different profiles. Production properties, on the other hand, not really.

Property priorities

Higher override lower. These are popular, the full list is much longer.

Command line arguments.

Java System properties (System.getProperties())

OS environment variables.

application-{profile}.properties outside of your packaged jar

application-{profile}.properties inside your jar

application.properties outside of your packaged jar

application.properties inside your jar

@PropertySource annotations on your @Configuration classes.

Loading properties

```
@PropertySource("classpath:com/foo/foo.properties")
```

http://java.dzone.com/articles/properties-spring

Resource syntax

Prefix	Example	Explanation
classpath:	classpath:com/myapp/config.xml	Loaded from the classpath.
file:	file:/data/config.xml	Loaded as a URL, from the filesystem. [1]
http:	http://myserver/logo.png	Loaded as a URL.
(none)	/data/config.xml	Depends on the underlying ApplicationContext.

Accessing properties

```
//default after semicolon
@Value( "${jdbc.url:localhost}" ) private String jdbcUrl;


@Autowired private Environment env;
...
dataSource.setUrl(env.getProperty("jdbc.url"));
```

http://java.dzone.com/articles/properties-spring

Accessing properties

${...} the property placeholder syntax, only to dereference properties

#{...} SpEL syntax, which is far more complex, also handle property placeholders

http://java.dzone.com/articles/properties-spring

You can use YAML instead of properties

```
connection:
    username: admin
    remoteAddress: 192.168.1.1
```

You can have typesafe properties

```
public class ConnectionSettings {
  private String username;
  private InetAddress remoteAddress;

@Configuration
@EnableConfigurationProperties
public class PropertiesConfiguration {

        @Bean
        @ConfigurationProperties(prefix="connection")
        public ConnectionSettings connectionSettings() {
          return new ConnectionSettings();
        }
...
```

You can generate random values

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

In tests

To have default properties in tests, in JUnit use

@SpringApplicationConfiguration

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SpringExercisesApplication.class)
@WebAppConfiguration
public abstract class IntegrationTest {
```

or setup config file initializer

initializers = ConfigFileApplicationContextInitializer.**class**

```
@WebAppConfiguration
@ContextConfiguration(classes = SpringExercisesApplication, initializers =
                                        ConfigFileApplicationContextInitializer.class)
class IntegrationSpec extends Specification {
```

Properties in tests

requires: SmartContextLoader

```
@ContextConfiguration
@TestPropertySource("classpath:test.properties")
public class MyIntegrationTests {...}
```

```
@ContextConfiguration

//relative to the package in which the test class is defined

@TestPropertySource("/test.properties")

public class MyIntegrationTests {...}
```

Properties in tests

also can be inlined

```
@ContextConfiguration
@TestPropertySource(properties={"timezone = GMT", "port: 4242"})
public class MyIntegrationTests {...}
```

we can inline them in many ways:

"key=value"

"key:value"

"key value"

Properties in tests: the default

If @TestPropertySource is declared as an empty annotation (i.e., without explicit values for the locations or properties attributes), an attempt will be made to detect a default properties file relative to the class that declared the annotation.

```
@ContextConfiguration
@TestPropertySource
public class MyTest {...}
```

"classpath:com/example/MyTest.properties"

Properties in tests: precedence

Test property sources have higher precedence than those loaded from the operating system’s environment or Java system properties as well as property sources added by the application declaratively via @PropertySource or programmatically.

Inlined properties override other.

MVC

Embedded server

Use prepared module:
spring-boot-starter-jetty Imports the Jetty HTTP engine (to be used as an alternative to Tomcat)
spring-boot-starter-tomcat Import Spring Boot’s default HTTP engine (Tomcat).
spring-boot-starter-undertow Imports the Undertow HTTP engine (alternative to Tomcat)

Or just add it manually to the classpath (Boot checks the classpath for it).

Embedded server: customization

application.properties

server.port — The listen port for incoming HTTP requests.

server.address — The interface address to bind to.

server.sessionTimeout — A session timeout.

//and more

Or:

```
@Component
public class CustomizationBean implements EmbeddedServletContainerCustomizer {
   @Override
   public void customize(ConfigurableEmbeddedServletContainer container) {
      container.setPort(9000);
   }
}
```

Embedded server: customization

If you need more options:

```
@Bean
public EmbeddedServletContainerFactory servletContainer() {
    TomcatEmbeddedServletContainerFactory factory = new
        TomcatEmbeddedServletContainerFactory();
    factory.setPort(9000);
    factory.setSessionTimeout(10, TimeUnit.MINUTES);
    factory.addErrorPages(new ErrorPage(HttpStatus.404, "/notfound.html");
    return factory;
}
```

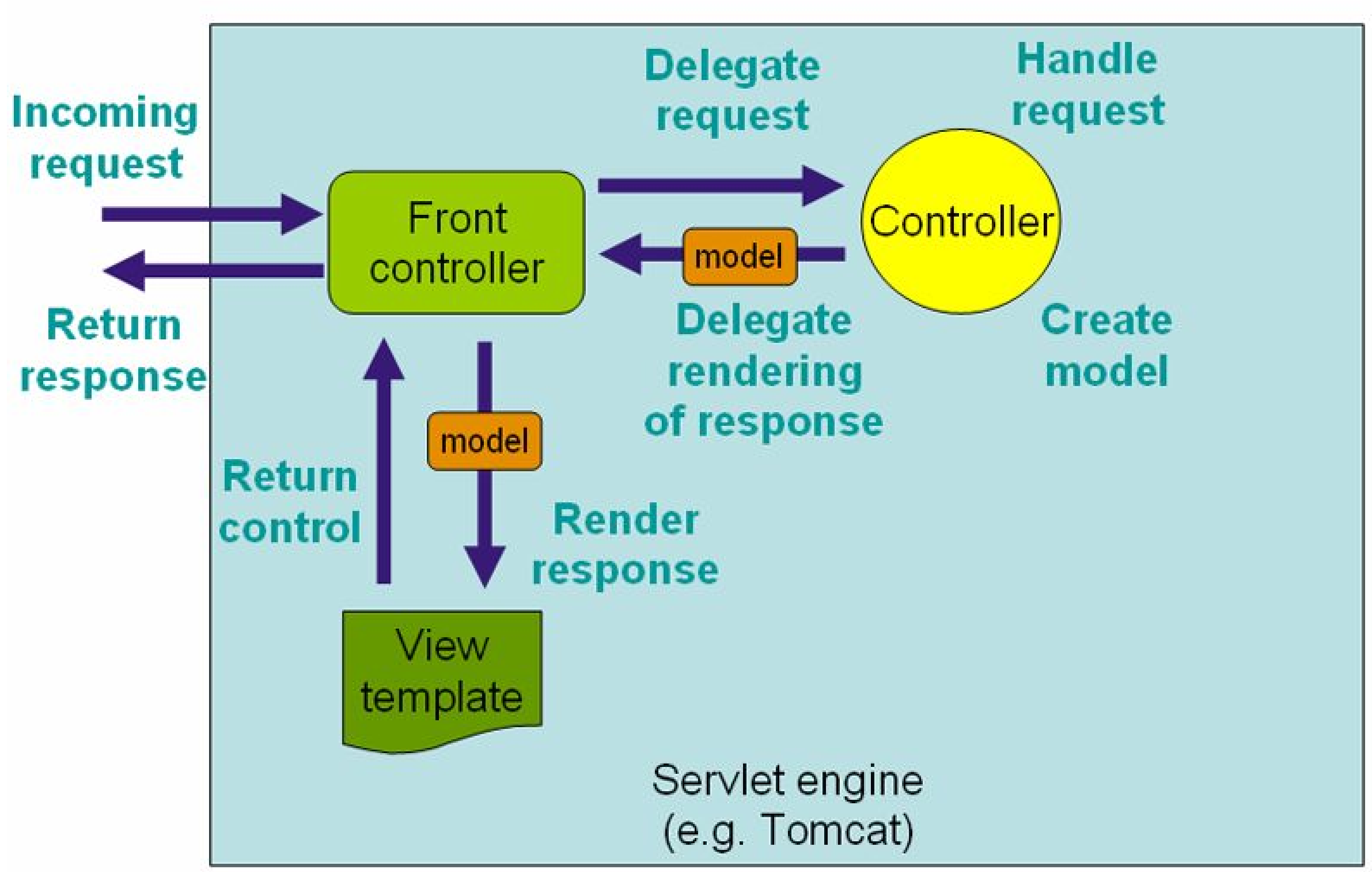
Out of the box:

TomcatEmbeddedServletContainerFactory,

JettyEmbeddedServletContainerFactory

UndertowEmbeddedServletContainerFactory

DispatcherServlet a.k.a Front Controller



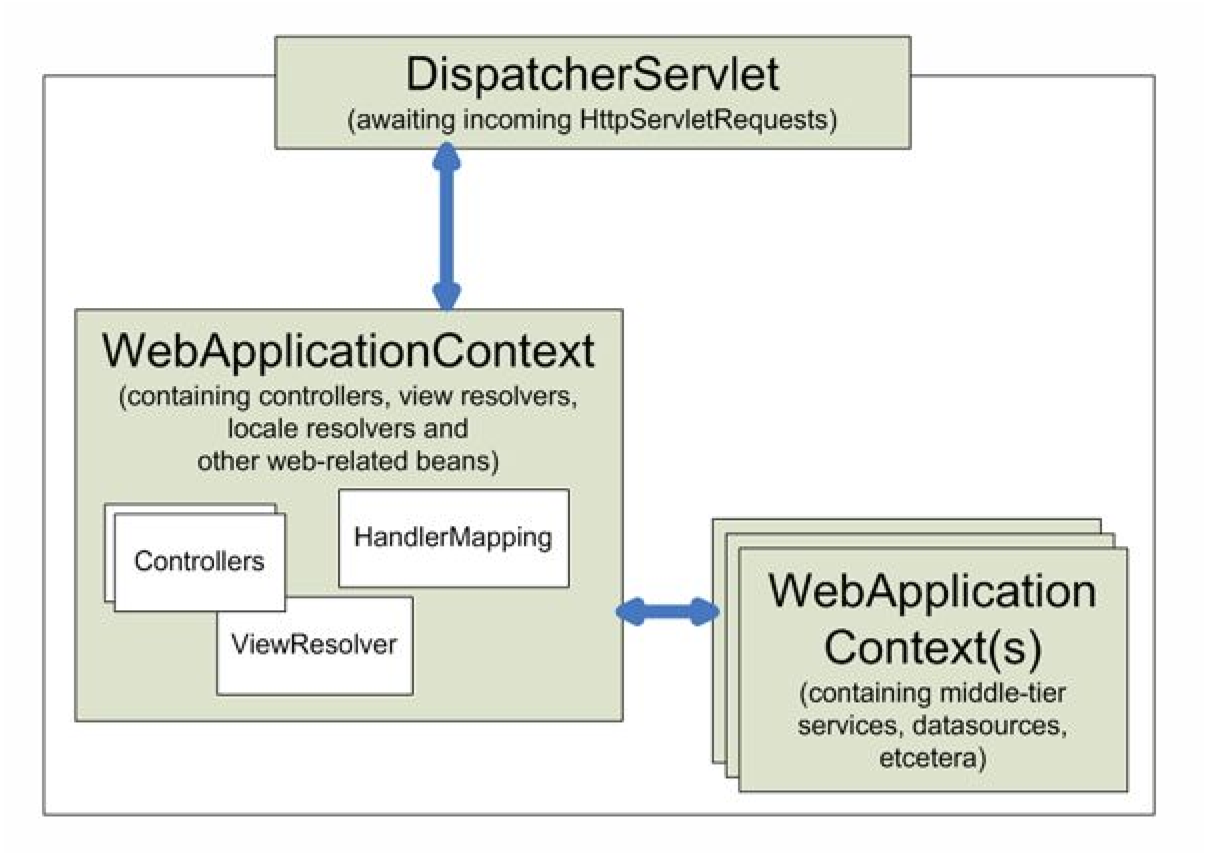
Setting up in web.xml

```
<servlet>
    <servlet-name>testkata</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/webmvc.ioc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>testkata</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Setting up without web.xml

```
class WebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return createWebApplicationContext("classpath:spring/main.ioc.xml")
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        return createWebApplicationContext("classpath:spring/webmvc.ioc.xml")
    }

    private WebApplicationContext createWebApplicationContext(String configLocation) {
        XmlWebApplicationContext webApplicationContext = new XmlWebApplicationContext()
        webApplicationContext.setConfigLocation(configLocation)
        return webApplicationContext
    }
```

Context over context


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context(s)
(containing middle-tier services, datasources, etcetera)

Controllers

```
@Controller
public class ProductController {

    @Autowired
    private ProductService productService;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    String index() {
        productService.doSomething(...)
        return "redirect:/index";
    }
```

Controller method examples

```
@RequestMapping(value = "/index", method = RequestMethod.GET)
public Map list() {
    List<Product> products = productRepository.findAll();
    return ImmutableMap.of("products", products);
}

@RequestMapping(value = "/some/mine", produces="application/json")
public Map mine() {
   List<SomeEntity> entities = someEntityRepository.findByUsername("some");
   return ["entities": entities]
}
```

@RequestMapping with path params

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

@RequestMapping with regex, or ant

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]}")
   public void handle(@PathVariable String version, @PathVariable String extension) {
      // please, kill me
   }
}
```

Or

```
/owners/*/pets/{petId}
/myPath/*.do
```

What if URL matches multiple patterns?

Let’s find out the most specific match

1. Lower count of URI variables and * = more specific
2. Longer is more specific
3. Fewer wildcards is more specific
4. default (/**) is less specific than anything else
5. prefil pattern (/public/**) is less specific than any other

You keep it complicated = you will have bad time debugging

Matrix patterns

```
// GET /owners/42;q=11/pets/21;q=22

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
public void findPet(
        @MatrixVariable(value="q", pathVar="ownerId") int q1,
        @MatrixVariable(value="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22
}
```

You have to set it up explicite, though. Won’t work out of the box.

Only if a param or header exist

```
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET,
params="myParam=myValue")
public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
    // implementation omitted
}
```

```
@RequestMapping(value = "/pets", method = RequestMethod.GET,
headers="myHeader=myValue")
public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
    // implementation omitted
}
```

Works with: "myParam", "!myParam", or "myParam=myValue"

@RequestMapping on class

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {
    ...
    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @RequestMapping(value="/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(
            @PathVariable @DateTimeFormat(iso=ISO.DATE) Date day,
            Model model) {

        return appointmentBook.getAppointmentsForDay(day);
    }
```

Combine on class with params

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping("/pets/{petId}")
    public void findPet(
              @PathVariable String ownerId,
              @PathVariable String petId, Model model) {
        // implementation omitted
    }

}
```

Consuming and producing

```
@Controller
@RequestMapping(value = "/pets", method = RequestMethod.POST,
consumes="application/json")
public void addPet(@RequestBody Pet pet) {
```

if the *Content-Type* request header matches the specified media type

```
@Controller
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET,
produces="application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
```

if the *Accept* request header matches the specified media type

ensures the actual content type used to generate the response respects the media types specified in the *produces*

Arguments for controller methods

@PathVariable

@RequestParam

@RequestHeader

@RequestBody //the whole body

@RequestPart //for multipart/form-data

Map, Model, ModelMap

Arguments for controller methods

RedirectAttributes

Command or form objects

Errors, BindingResult

ServletRequest, HttpServletRequest, WebRequest

HttpSession

Locale

InputStream, Reader, OutputStream, Writer

Principal

RequestEntity

and more...

Arguments for controller methods

```
@RequestMapping("/handle")
 public void handle(RequestEntity<String> request) {
   HttpMethod method = request.getMethod();
   URI url = request.getUrl();
   String body = request.getBody();
 }
```

Since Spring 4.1

Returned types

ModelAndView, Model, Map
+ @ModelAttribute annotated reference data accessor methods.

View

String view name or redirect

void

@ResponseBody (converted via HttpMessageConverter)
any type of your own (assumes it’s the only thing in the model)

HttpEntity<T> and ResponseEntity<T>
and more...

Return examples

```
@RequestMapping(value = "/", method = RequestMethod.GET)
String index() {
    return "redirect:/index";
}

@RequestMapping(value = "/index", method = RequestMethod.GET)
public Map list() {
    List<Product> products = productRepository.findAll();
    return ImmutableMap.of("products", products);
}

@RequestMapping(value = "/loans/mine", produces="application/json")
public List<Loan> myLoans() {
    String username = loggedUserRepository.getLoggedUserName();
    return loanRepository.findByUsername(username);
}
```

Return examples

```
public HttpEntity(T body, MultiValueMap<String, String> headers) ...
public ResponseEntity(T body, MultiValueMap<String, String> headers, HttpStatus statusCode)  ...
```

Builders:

```
return ResponseEntity.
            created(location).
            header("MyResponseHeader", "MyValue").
            body("Hello World")
```

Since Spring 4.1

Helper methods

Annotation that identifies methods which initialize the WebDataBinder which will be used for populating command and form object arguments of annotated handler methods.

```
@InitBinder
public void initBinder(WebDataBinder binder) {
   SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
   dateFormat.setLenient(false);
   binder.registerCustomEditor(Date.class,
         new CustomDateEditor(dateFormat, false));
}
```

```
//signature
registerCustomEditor(Class<?> requiredType, PropertyEditor propertyEditor);
```

Helper methods

@ModelAttribute methods are used to populate the model with commonly needed attributes for example to fill a drop-down with states or with pet types, or to retrieve a command object like Account in order to use it to represent the data on an HTML form.

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
   return accountManager.findAccount(number);
}
```

@ModelAttribute and data binding

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit",
method = RequestMethod.POST)
public String processSubmit(@ModelAttribute Pet pet) { }
```

An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model

Once present in the model, the argument's fields should be populated from all request parameters that have matching names.

@ModelAttribute and data binding

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit",
method = RequestMethod.POST)
public String processSubmit(@ModelAttribute Pet pet) { }
```

Where does a pet come from?

- already in the model due to use of @SessionAttributes
- already in the model due to an @ModelAttribute method in the same controller
- retrieved based on a URI template variable and type converter
- instantiated using its default constructor

And then it is populated from request (a.k.a. data binding)

Storing model between requests

1. Keep it in Session

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

2. Keep it in FlashMap and redirect

`FlashMap` is used to hold flash attributes while `FlashMapManager` is used to store, retrieve, and manage `FlashMap` instances.

Does have its concurrency problems, so is stored with the path and query parameters of the target redirect URL.

Storing model between requests

```
@RequestMapping(value = "/accounts", method = RequestMethod.POST)
 public String handle(Account account, BindingResult result, RedirectAttributes redirectAttrs) {
     if (result.hasErrors()) {
       return "accounts/new";
     }
     // Save account ...
     redirectAttrs.addAttribute("id", account.getId())
     redirectAttrs.addFlashAttribute("message", "Account created!");
     return "redirect:/accounts/{id}";
 }
```

A RedirectAttributes model is empty when the method is called and is never used unless the method returns a redirect view name or a RedirectView.

After the redirect, flash attributes are automatically added to the model of the controller that serves the target URL.

Building URIs

```
UriComponents uriComponents =
UriComponentsBuilder.fromUriString("http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();
```

Typical usage involves:

- Create a UriComponentsBuilder with one of the static factory methods (such as fromPath(String) or fromUri(URI))
- Set the various URI components through the respective methods (scheme(String), userInfo(String), host(String), port(int), path(String), pathSegment(String...), queryParam(String, Object...), and fragment(String).
- Build the UriComponents instance with the build() method.

Building URIs

ServletUriComponentsBuilder - reusing servlet information

```
// Re-use host, scheme, port, path and query string
// Replace the "accountId" query param

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}").build()
        .expand("123")
        .encode();

// Re-use host, port and context path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts").build()
```

Building URIs

MvcUriComponentsBuilder - helps to build URIs to Spring MVC controllers and methods from their request mappings.

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {
   @RequestMapping("/bookings/{booking}")
   public String getBooking(@PathVariable Long booking) {
…

UriComponents uriComponents = MvcUriComponentsBuilder
   .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

//hotel = 42, booking = 21
```

RESTafaranism

`@RestController` = `@ResponseBody` + `@Controller`

Controller Advices

Classes annotated with @ControllerAdvice can contain:

- @ExceptionHandler
- @InitBinder
- @ModelAttribute

annotated methods, and these methods will apply to @RequestMapping methods **across all controller hierarchies** as opposed to the controller hierarchy within which they are declared.

Controller Advices

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class AnnotationAdvice {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class BasePackageAdvice {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(
   assignableTypes = {ControllerInterface.class, AbstractController.class})
public class AssignableTypesAdvice {}
```

Views

methods in the Spring Web MVC controllers must resolve to a logical view name, either explicitly (e.g., by returning a String, View, or ModelAndView) or implicitly (i.e., based on conventions)

ViewResolver

AbstractCachingViewResolver

XmlViewResolver

ResourceBundleViewResolver

UrlBasedViewResolver

InternalResourceViewResolver

VelocityViewResolver

FreeMarkerViewResolver

ContentNegotiatingViewResolver

UrlBasedViewResolver

Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition.

This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

ContentNegotiatingViewResolver

Implementation of the ViewResolver interface that resolves a view based on the **request file name** or **Accept header** (media type).

Once the requested media type has been determined, this resolver queries each delegate view resolver for a View and determines if the requested media type is MediaType compatible with the view's View content type. The most compatible view is returned.

ContentNegotiatingViewResolver

```
@TypeChecked
@Configuration
@EnableWebMvc
class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void configureViewResolvers(ViewResolverRegistry registry) {
       registry.enableContentNegotiation(true, new MappingJackson2JsonView())
       registry.viewResolver(createThymeleafViewResolver())
  }

  @Override
  public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
     configurer.
          mediaTypes(['json': MediaType.APPLICATION_JSON, 'html':MediaType.TEXT_HTML]).
          defaultContentType(MediaType.TEXT_HTML)

  }
```

Redirecting

```
@RequestMapping(value = "/", method = RequestMethod.GET)
String index() {
    return "redirect:/index";
}
```

Using model in view: example

```
<!DOCTYPE html SYSTEM
"http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-spring3-3.dtd">
<html xmlns:th="http://www.thymeleaf.org" lang="en">

  [...]

  <div class="container">
    <h1>Products</h1>
    <div th:each="product : ${products}">
      <h4>Name: <b th:text="${product.name}" /></h4>
      <p>Description: <span th:text="${product.description}" /> </p>
      <p>SKUs:
        <div th:each="sku : ${products.skus}">
          <b th:text="${sku.name}" />
        </div>
      </p>
    </div>
  </div>
```

ModelAndView

```
public ModelAndView handleRequest() {
    ModelAndView mav = new ModelAndView("displayShoppingCart");
    mav.addObject(cartItems);
    mav.addObject(user);
    return mav;
}
```

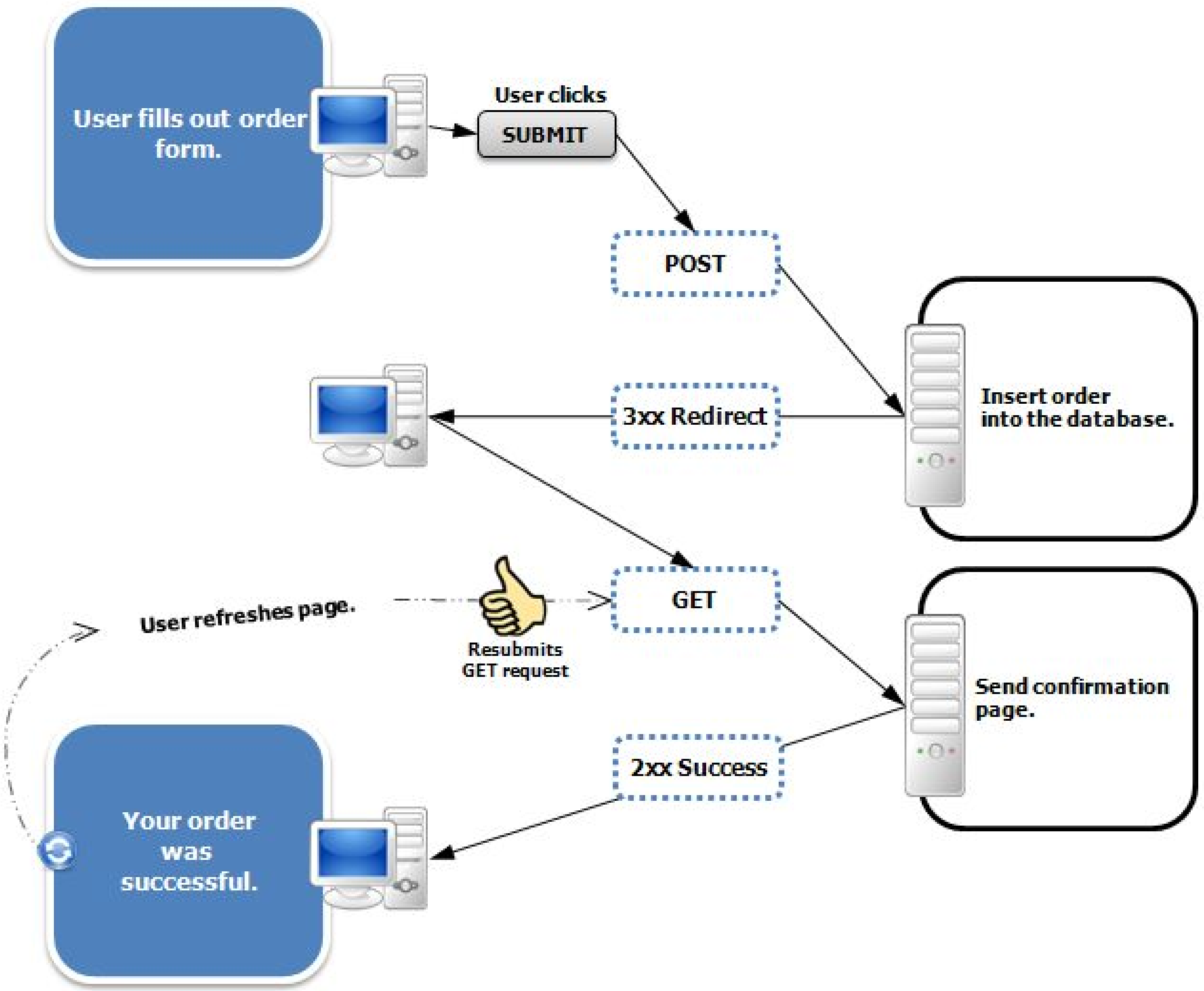
Serving static resources

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
      registry.
          addResourceHandler("/resources/**").addResourceLocations("/public-resources/");
    }

}
```

Serving view without controller

```
@TypeChecked
@Configuration
@EnableWebMvc
class WebConfig extends WebMvcConfigurerAdapter {
        @Override
        void addViewControllers(ViewControllerRegistry registry) {
          registry.addViewController('/error').setViewName('error')
        }
}
```

Post-redirect-get pattern



http://en.wikipedia.org/wiki/File:PostRedirectGet_DoubleSubmitSolution.png

Locales

RequestContext //the class you add to controller method params

requestContext.getLocale()

requestContext.getTimeZone() //may or may not return something

AcceptHeaderLocaleResolver

CookieLocaleResolver

SessionLocaleResolver

LocaleChangeInterceptor will change locale if siteLanguage is found, i.e.:

http://www.sf.net/home.view?**siteLanguage=pl**

Handling exceptions

1. Implement **HandlerExceptionResolver**
   resolveException(Exception, Handler)

   or

2. Use **SimpleMappingExceptionResolver**
   (map class to view name)

   or

3. Create **@ExceptionHandler** method
   on @Controller or @ControllerAdvice

DefaultHandlerExceptionResolver

Translates Spring MVC exceptions to specific error status codes:

BindException - 400 (Bad Request)

ConversionNotSupportedException - 500 (Internal Server Error)

HttpMediaTypeNotAcceptableException - 406 (Not Acceptable)

HttpMediaTypeNotSupportedException - 415 (Unsupported Media Type)

HttpMessageNotReadableException - 400 (Bad Request)

HttpMessageNotWritableException - 500 (Internal Server Error)

HttpRequestMethodNotSupportedException - 405 (Method Not Allowed)

MethodArgumentNotValidException - 400 (Bad Request)

MissingServletRequestParameterException - 400 (Bad Request)

MissingServletRequestPartException - 400 (Bad Request)

NoHandlerFoundException - 404 (Not Found)

NoSuchRequestHandlingMethodException - 404 (Not Found)

TypeMismatchException - 400 (Bad Request)

Handling exceptions

In one @Controller:

```
@ExceptionHandler(IOException.class)
public ResponseEntity<String> handleIOException(IOException ex) {
    // prepare responseEntity
    return responseEntity;
}
```

Or globally in a @ControllerAdvice

Extending ResponseEntityExceptionHandler is a convenient way.

Handling exceptions

You can also annotate the Exception, but you will bind it with the view.

```
@ResponseStatus(HttpStatus.FORBIDDEN)
class AccessRefusedException extends RuntimeException {
...
}
```

Marks a **method** or **exception** class with the status code and reason that should be returned. The status code is applied to the HTTP response when the handler method is invoked, or whenever said exception is thrown.

Errors

Apart from standard stuff (@ExceptionHandler in Spring MVC), boot registers /error

And you can customize it:

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer(){
    return new MyCustomizer();
}

private static class MyCustomizer implements EmbeddedServletContainerCustomizer {
    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    }
}
```

What WebApplicationContext provides

```
@WebAppConfiguration
@ContextConfiguration
public class WacTests {

    @Autowired WebApplicationContext wac; // cached

    @Autowired MockServletContext servletContext; // cached

    @Autowired MockHttpSession session;

    @Autowired MockHttpServletRequest request;

    @Autowired MockHttpServletResponse response;

    @Autowired ServletWebRequest webRequest;

}
```

Session/Request scoped in tests

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

    @Test
    public void requestScope() {
        //given
        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        //when
        LoginResults results = userService.loginUser();
```

Spring MVC testing: Base

```
@WebAppConfiguration
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=[Application.class])
abstract class MvcIntegrationTest  {
    @Autowired private WebApplicationContext webApplicationContext
    protected MockMvc mockMvc

    @Before
    void setupMockMvc() {
        mockMvc =
           MockMvcBuilders.
                   webAppContextSetup(webApplicationContext).
                   build()
    }
}
```

Spring MVC testing

```
class JsonVieResolverMvcTest extends MvcIntegrationTest {
    String JSON_MEDIA_TYPE = "application/json"
    @Autowired AuthenticationManager authenticationManager
    @Autowired private Environment environment

    @Test
    void "view resolver should render JSON"() {
        //given
        userIsLoggedIn()

        //when then
        mockMvc.perform(
                    post('/loans/mine')
                ).
                andExpect(status().isOk()).
                andExpect(content().contentType(JSON_MEDIA_TYPE))
    }
```

Spring MVC testing: more options

```
@Test
void "should return successful json to customer on payment success"() {
    //expect
    mockMvc.perform(post('/pay').
            param('billedCode', BillerFactory.CODE).
            param('msisdn', ExternalBillerServiceMock.CORRECT_MSISDN).
            param('productCode', BillerFactory.PRODUCT_CODE).
            accept(MediaType.parseMediaType(JSON_MEDIA_TYPE))).
            andExpect(status().isOk()).
            andExpect(content().contentType(JSON_MEDIA_TYPE)).
            andExpect(content().string('{"status":"ok"}'))
}
```

File uploads? R U kidding me?

```
mockMvc.
    perform(fileUpload("/doc").
    file("a1", "ABC".getBytes("UTF-8")));
```

Manual Transactions - since 4.1 (more on that later)

```
@ContextConfiguration(classes = TestConfig.class)
public class ProgTransactionMngTests extends AbstractTransactionalJUnit4SpringContextTests {
    @Test
    public void transactionalTest() {
        // assert initial state in test database:
        assertNumUsers(2);
        deleteFromTables("user");

        // changes to the database will be committed!
        TestTransaction.flagForCommit();
        TestTransaction.end();
        assertFalse(TestTransaction.isActive());
        assertNumUsers(0);

        TestTransaction.start();
        // perform other actions against the database that will  be rolled back after the test completes...
    }
```

Around transactions... but it’s usualy a very bad idea

```
@BeforeTransaction
public void verifyInitialDatabaseState() {
    // logic to verify the initial state before a transaction is started
}
@Before
public void setUpTestDataWithinTransaction() {
    // set up test data within the transaction
}
@After
public void tearDownWithinTransaction() {
    // execute "tear down" logic within the transaction
}
@AfterTransaction
public void verifyFinalDatabaseState() {
    // logic to verify the final state after transaction has rolled back
}
```

Populating the database

```
@Test
public void databaseTest {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScripts(
        new ClassPathResource("test-schema.sql"),
        new ClassPathResource("test-data.sql"));
    populator.setSeparator("@@");
    populator.execute(this.dataSource);
    // execute code that uses the test schema and data
}
```

If you really have to… because the speed is going to be painful.

Manual SQL setup

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@Sql("/test-schema.sql")
public class DatabaseTests {

    @Test
    public void emptySchemaTest {
        // execute code that uses the test schema without any test data
    }

    @Test
    @Sql({"/test-schema.sql", "/test-user-data.sql"})
    public void userTest {
        // execute code that uses the test schema and test data
    }
}
```

Please… don’t.

Testing

@SpringApplicationConfiguration

class-level annotation that is used to determine how to load and configure an ApplicationContext for integration tests. Similar to the standard ContextConfiguration but uses Spring Boot's SpringApplicationContextLoader.

But it doesn't work with Spock!

In Spock (+spock-spring) use:

```
@ContextConfiguration(loader = SpringApplicationContextLoader.class)
```

But do you really need a running embedded server working on a port, to test your app? Spring MVC testing could be enough for the blackbox.
If you don't need running embedded server, use:

```
@ContextConfiguration(classes=[Application.class])
```

Testing

@IntegrationTest

Test class annotation signifying that the tests are integration tests (and therefore require an application to startup "fully leaded" and listening on its normal ports).

```
@WebAppConfiguration
@SpringApplicationConfiguration(classes = MyApplication.class)
@IntegrationTest({"server.port=0", "management.port=0"})
public class SomeIntegrationTests { }
```

Testing: discover the port

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SampleDataJpaApplication.class)
@WebAppConfiguration
@IntegrationTest("server.port:0") //RANDOM
public class CityRepositoryIntegrationTests {

    @Autowired
    EmbeddedWebApplicationContext server;

    @Value("${local.server.port}")
    int port;
}
```

Then you can consume/test the app with

```
  RestTemplate restTemplate = new TestRestTemplate();
```

Spring Data Web support

Requires SpringMVC turned on.

In Java config:

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport //Spring HATEOAS stuff registered if on classpath
class WebConfiguration { }
```

Domain class converter

```
@Controller
@RequestMapping("/users")
public class UserController {

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") User user, Model model) {

    model.addAttribute("user", user);
    return "userForm";
  }
}
```

The instance can be resolved by letting Spring MVC **convert the path variable** into the id type of the domain class first and eventually **access the instance** through calling findOne(…) on the repository instance registered for the domain type.

Domain class converter

Be careful like hell, because: SECURITY.

Also you can register our own converters.

HandlerMethodArgumentResolver

```
@Controller
@RequestMapping("/users")
public class UserController {
  @Autowired UserRepository repository;

  @RequestMapping
  public String showUsers(Model model, Pageable pageable) {
    model.addAttribute("users", repository.findAll(pageable));
    return "users";
  }
}
```

Request parameters going into Pageable:

page - number of the page you want to retrieve

size - size of the page you want to retrieve

sort - ?sort=firstname&sort=lastname,asc

HandlerMethodArgumentResolver

What if I have several tables?

```
public String showUsers(Model model,
    @Qualifier("foo") Pageable first,
    @Qualifier("bar") Pageable second) { … }
```

Request parameters going into Pageable:

foo_page - number of the page you want to retrieve

bar_page - number of the page you want to retrieve

Also possible: paged resources - HATEOAS

```
{ "links" : [ { "rel" : "next",
               "href" : "http://localhost:8080/persons?page=1&size=20 }
  ],
  "content" : [
     … // 20 Person instances rendered here
  ],
  "pageMetadata" : {
    "size" : 20,
    "totalElements" : 30,
    "totalPages" : 2,
    "number" : 0
  }
}
```

Also possible: paged resources - HATEOAS

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

  @RequestMapping(value = "/persons", method = RequestMethod.GET)
  HttpEntity<PagedResources<Person>> persons(Pageable pageable,
                                              PagedResourcesAssembler assembler) {

    Page<Person> persons = repository.findAll(pageable);
    return new ResponseEntity<>(assembler.toResources(persons), HttpStatus.OK);
  }
}
```

Actuator

Actuator

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

Actuator

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.

configprops

Displays a collated list of all @ConfigurationProperties.

dump

Performs a thread dump.

env

Exposes properties from Spring’s ConfigurableEnvironment.

health

Shows application health information (defaulting to a simple ‘OK’ message).

Actuator

info

Displays arbitrary application info.

metrics

Shows ‘metrics’ information for the current application.

mappings

Displays a collated list of all @RequestMapping paths.

shutdown

Allows the application to be gracefully shutdown (not enabled by default).

trace

Displays trace information (by default the last few HTTP requests).

Actuator

To customize endpoints

In application.properties, edit endpoints + . + name

For example:

```
endpoints.beans.id=springbeans
endpoints.beans.sensitive=false
endpoints.shutdown.enabled=true
```

Depending on how an endpoint is exposed, the **sensitive** parameter may be used as a security hint. For example, sensitive endpoints will require a username/password when they are accessed over HTTP (or simply disabled if web security is not enabled).

Actuator

You can easily customize health and info endpoint.

gradle-git plugin will create git.properties with git.branch and git.commit, that will be show in the info endpoint

So you know what’s running on production.

So far I’ve been doing it myself with git hooks or bash scripts.

Not cool to make it all public, right?

Add Spring Security

You will have basic auth from the box with username user and a generated password (look into the logs):

Or you can set it up in application.properties

```
security.user.name=admin
security.user.password=secret
management.security.role=SUPERUSER
```

Move all endpoints to sub-path

In application.properties:

```
management.context-path=/manage
```

and instead of /info you have /manage/info

You can also customize port:

```
management.port=8081
```

(you may not need Spring Security then, since you can protect the port on the network level)

Or you can turn them all off

```
management.port=-1
```

All the endpoint can also be exposed via JMX

SSH

```
ssh user@localhost -p 2000
```

(password in logs)

dashboard

SSH custom credentials

In application.properties:

```
shell.auth.simple.user.name=user
shell.auth.simple.user.password=passwd
```

Could also work with Spring Security configuration

SSH commands

help

dashboard

jvm heap

metrics

thread ls/top/dump

beans

….

Your own commands

Create command in groovy or java. Put it in:

classpath*:/commands/**

classpath*:/crash/commands/**

Example

src/main/resources/commands/hello.groovy

```
import org.crsh.cli.Usage
import org.crsh.cli.Command
class hello {
  @Usage("Say Hello")
  @Command
  def main(InvocationContext context) {
    return "Hello"
  }
}
```

Your own commands

Your commands are run by
http://www.crashub.org/

The @Command annotation declares the main method as a command

The @Usage annotation describes the usage of the command and its parameters