SPRING BOOT

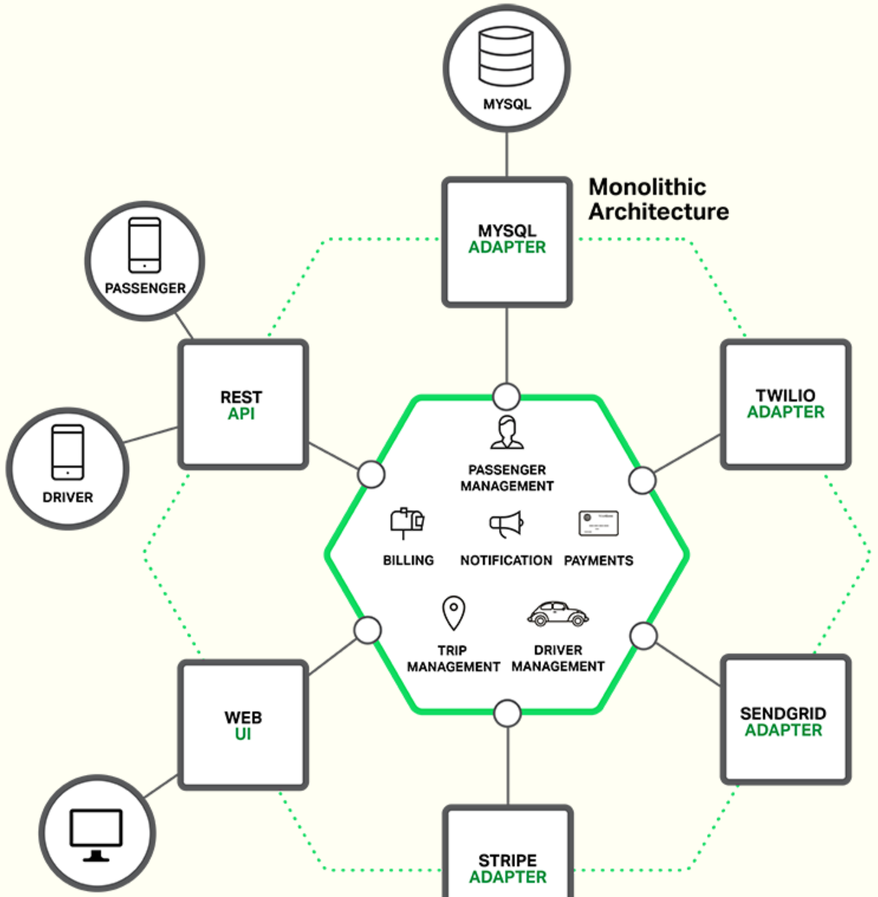
banuprakashc@yahoo.co.in

Monolithic Architecture

- Logically different modules
- But packaged and deployed as a Single Unit
- Initial Phases of Project
  - Simple to deploy
  - Vertical scaling
- Later on
  - Difficult to manage and scale
  - Longer start-up times
  - Slow down development
  - CI becomes challenging

Building Monolithic Applications

- To build a brand new taxi-hailing application intended to compete with Uber and Ola
  - At the core of the application is the business logic, which is implemented by modules that define services, domain objects, and events.
  - Surrounding the core are adapters that interface with the external world. Examples of adapters include database access components, messaging components that produce and consume messages, and web components that either expose APIs or implement a UI.



Building Monolithic Applications

- Despite having a logically modular architecture, the application is packaged and deployed as a monolith.
- The actual format depends on the application’s language and framework.
- For example, many Java applications are packaged as WAR files and deployed on application servers such as Tomcat or Jetty.
- Other Java applications are packaged as self-contained executable JARs.
- Similarly, Rails and Node.js applications are packaged as a directory hierarchy.
- These kinds of applications are also simple to test. You can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
- Monolithic applications are also simple to deploy. You just have to copy the packaged application to a server.
- You can also scale the application by running multiple copies behind a load balancer.
- In the early stages of the project it works well

Marching Towards Monolithic Hell

- Successful applications have a habit of growing over time and eventually becoming huge.
- During each sprint, your development team implements a few more stories, which, of course, means adding many lines of code.
- After a few years, your small, simple application will have grown into a monstrous monolith
- One major problem is that the application is overwhelmingly complex.
- It’s simply too large for any single developer to fully understand.
- As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming.
- What’s more, this tends to be a downwards spiral.

Marching Towards Monolithic Hell

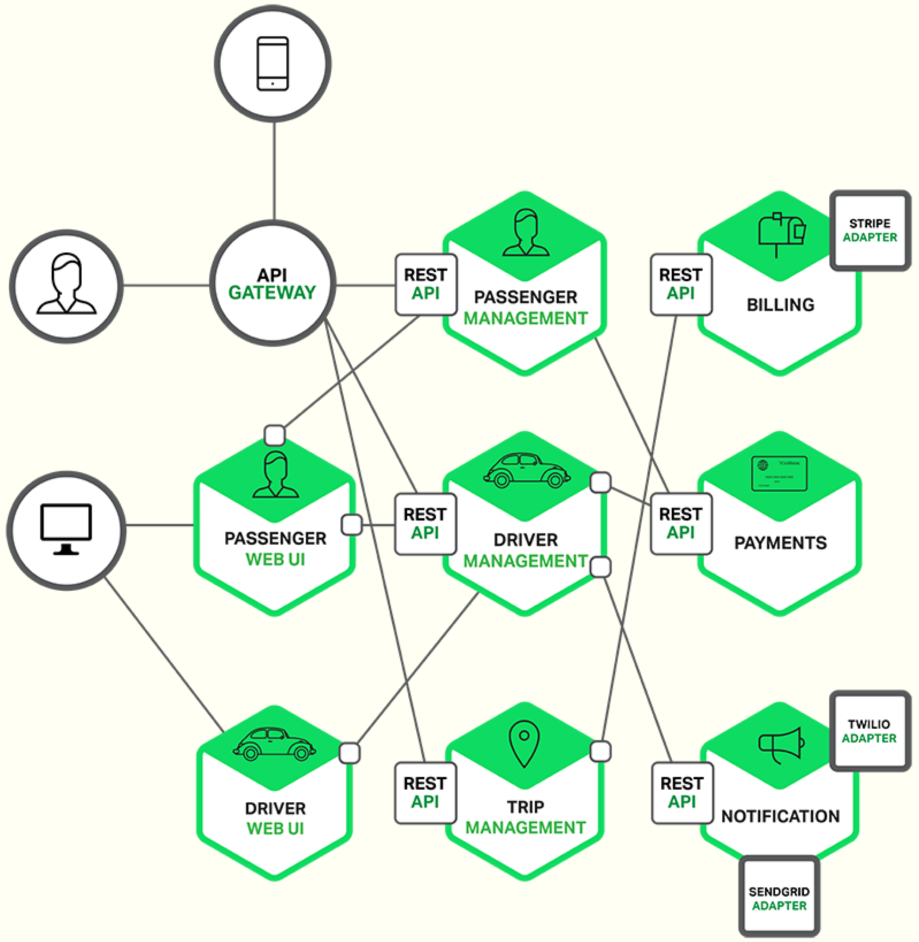
- Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements.
- For example, one module might implement CPU-intensive image processing logic and would ideally be deployed in Amazon EC2 Compute Optimized instances.
- Another module might be an in-memory database and best suited for EC2 Memory-optimized instances.
- However, because these modules are deployed together you have to compromise on the choice of hardware

Microservices – Tackling the Complexity

- Many organizations, such as Amazon, eBay, and Netflix, have solved this problem by adopting what is now known as the Microservices Architecture pattern.
- Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services.
- A service typically implements a set of distinct features or functionality, such as order management, customer management, etc.
- Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters.
- Some microservices would expose an API that's consumed by other microservices or by the application's clients.
- Other microservices might implement a web UI.
- At runtime, each instance is often a cloud VM or a Docker container.

Microservices – Tackling the Complexity

- Each functional area of the application is now implemented by its own microservice.
- Moreover, the web application is split into a set of simpler web applications (such as one for passengers and one for drivers in our taxi-hailing example).
- This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

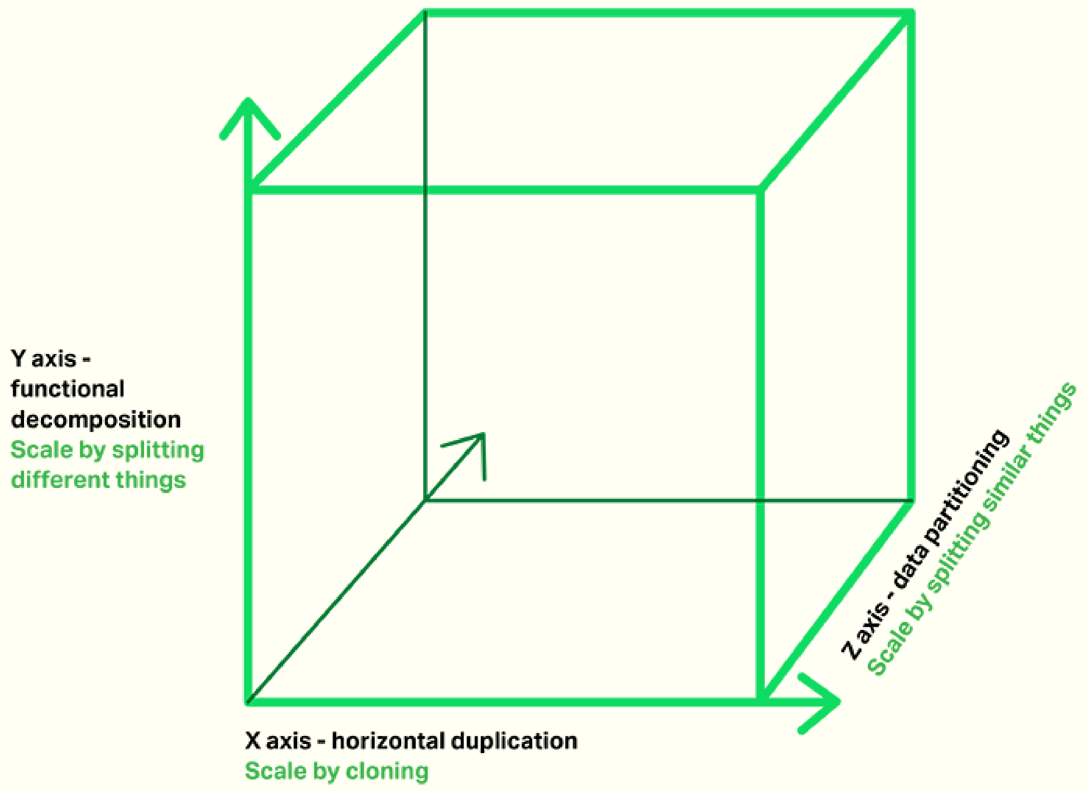


Microservices – Tackling the Complexity

- Each backend service exposes a REST API and most services consume APIs provided by other services.
- For example, Driver Management uses the Notification server to tell an available driver about a potential trip.
- The UI services invoke the other services in order to render web pages. Services might also use asynchronous, message-based communication.

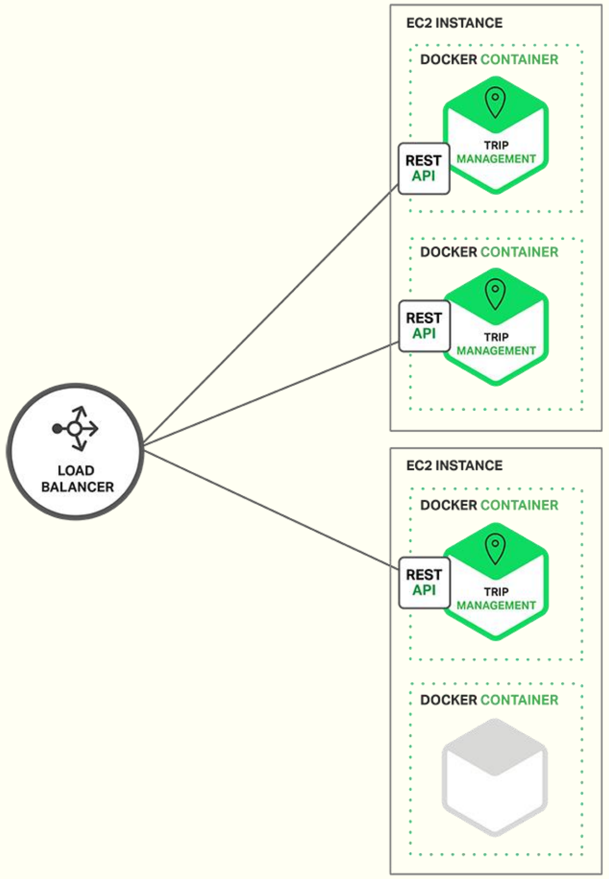
The Microservices Architecture pattern 3D model

- The Microservices Architecture pattern corresponds to the Y-axis scaling
- X axis for cloning
- Z axis for portioning


Y axis - functional decomposition
Scale by splitting different things
Z axis - data partitioning
Scale by splitting similar things
X axis - horizontal duplication
Scale by cloning

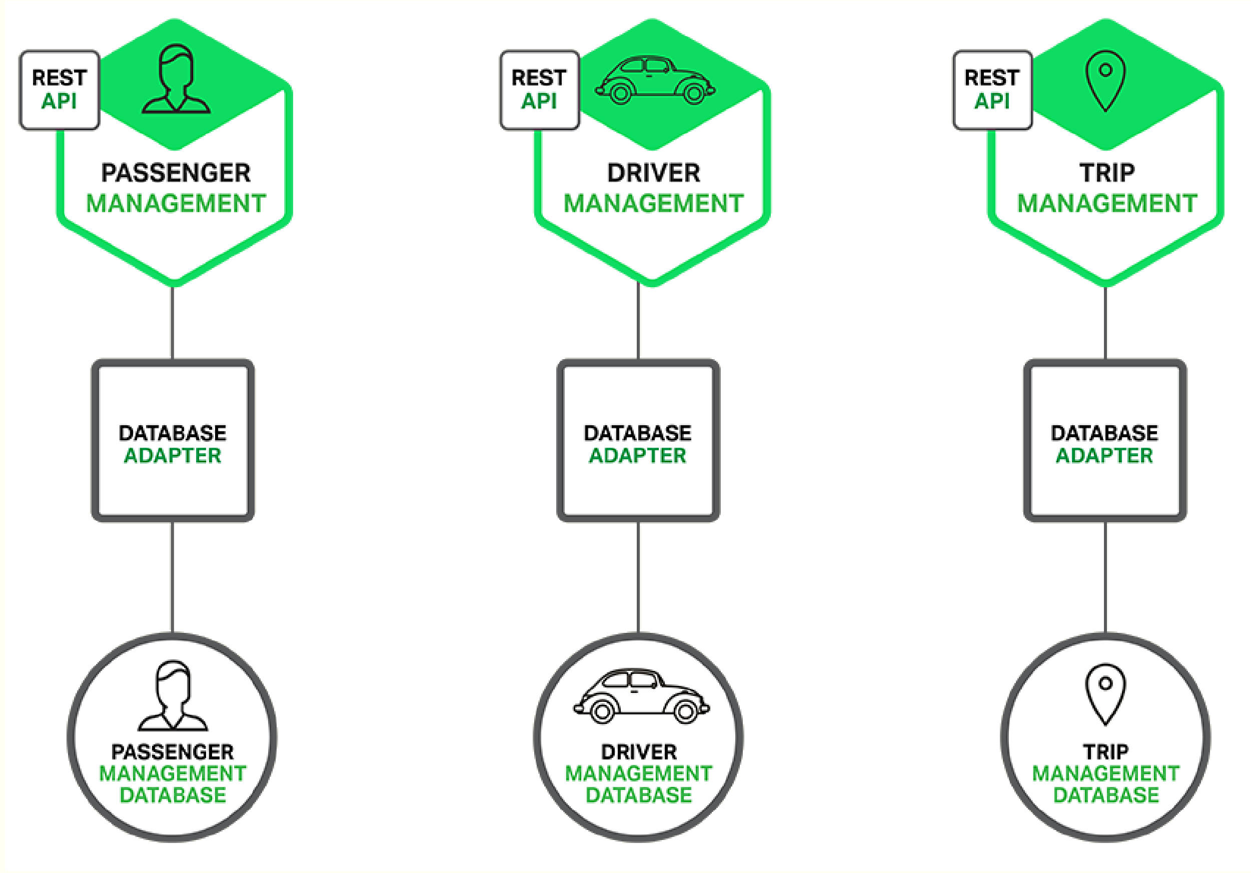
X-Axis scaling

- At runtime, the Trip Management service consists of multiple service instances.
- Each service instance is a Docker container. In order to be highly available, the containers are running on multiple Cloud VMs.
- In front of the service instances is a load balancer such as NGINX that distributes requests across the instances.
- The load balancer might also handle other concerns such as caching, access control,API metering, and monitoring.

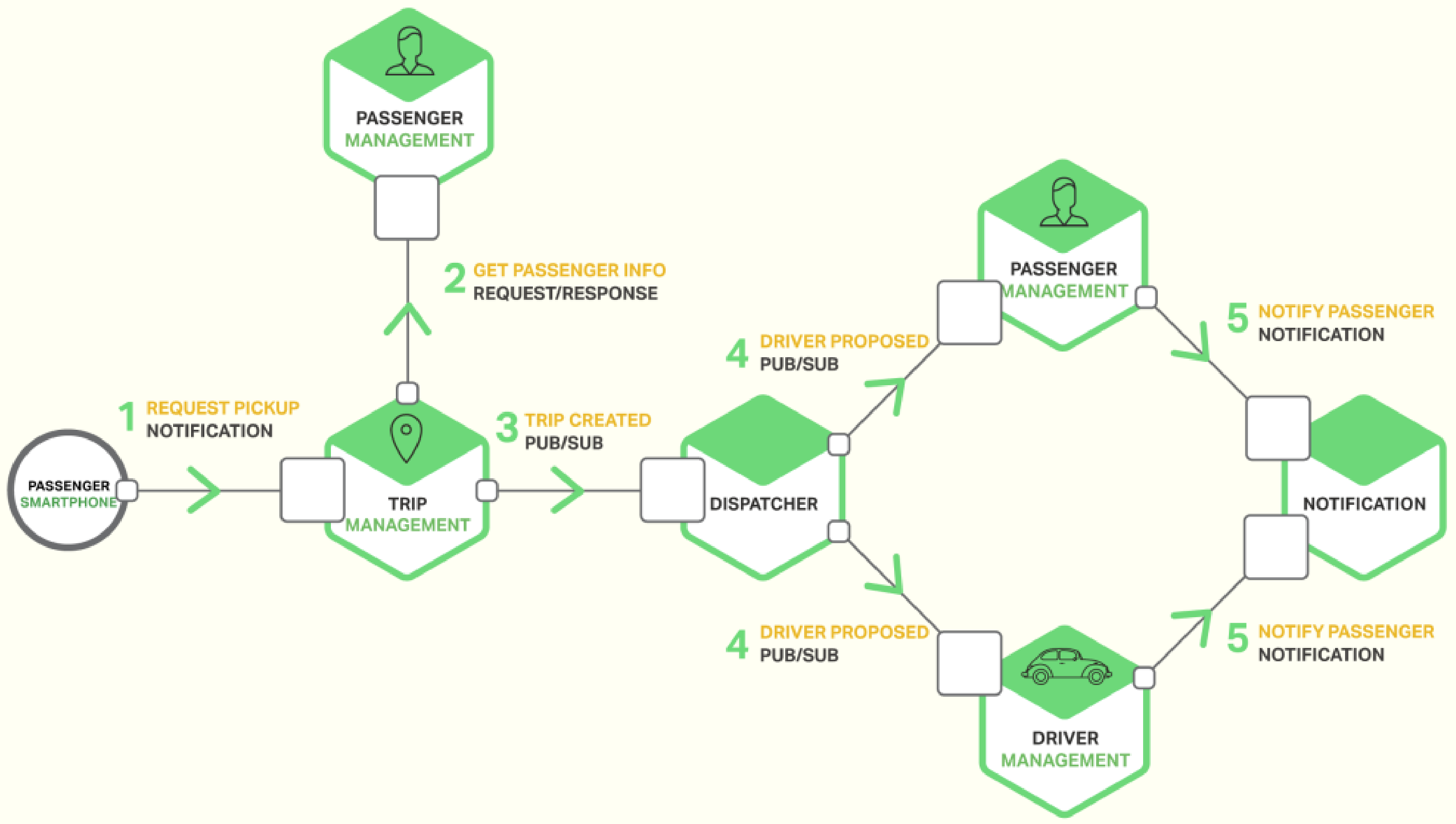


Microservices – Relationship between app and database

- Each of the services has its own database.
- Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture.
- For example, Driver Management, which finds drivers close to a potential passenger, must use a database that supports efficient geo-queries.



Services Communication

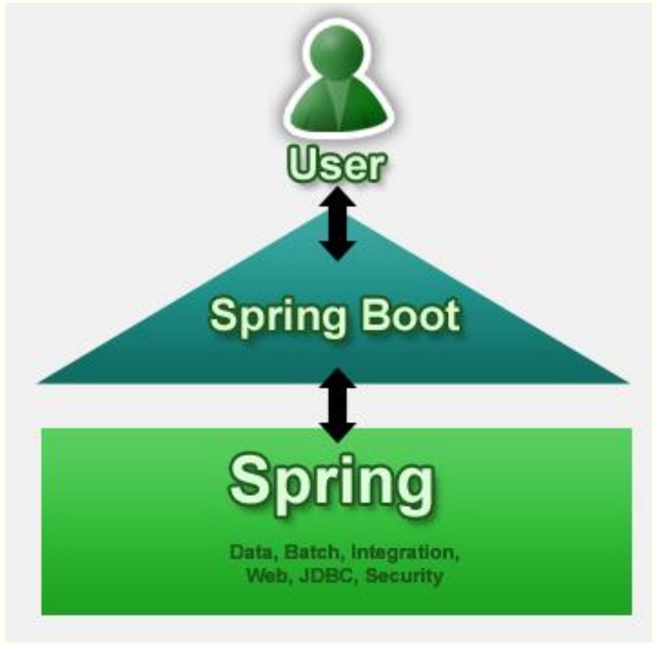


Services Communication

- The services use a combination of notifications, request/response, and publish/subscribe.
- For example, the passenger’s smartphone sends a notification to the Trip Management service to request a pickup.
- The Trip Management service verifies that the passenger’s account is active by using request/response to invoke the Passenger Service.
- The Trip Management service then creates the trip and uses publish/subscribe to notify other services including the Dispatcher, which locates an available driver.

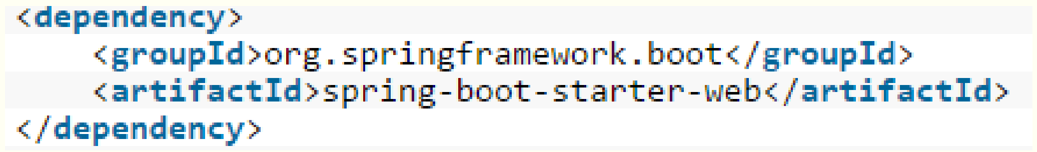
Spring Boot

- Spring Boot:
  - A tool for getting started very quickly with spring
  - To provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration)
  - To be opinionated out of the box
  - Gets out of the way quickly if you want to change defaults
  - Single Responsibility: Focuses attention at a single point (as opposed to large collection of spring-* projects)
  - Spring Boot does *not* generate code and there is absolutely **no** requirement for XML configuration.

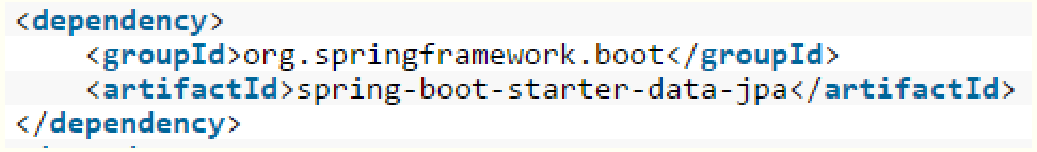


Why Spring Boot?

1. **Easy dependency Management**
   - **spring-boot-starter-web** dependency by default it will pull all the commonly used libraries while developing Spring MVC applications such as **spring-webmvc, jackson-json, validation-api** and **tomcat**
   - **spring-boot-starter-data-jpa** dependency pulls all the **spring-data-jpa** dependencies and also adds **Hibernate** libraries because majority of the applications use Hibernate as JPA implementation



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

2. Auto Configuration
   - Not only the **spring-boot-starter-web** adds all these libraries but also configures the commonly registered beans like **DispatcherServlet, ResourceHandlers, MessageSource** etc beans with sensible defaults. **DataSource, EntityManagerFactory,TransactionManager** etc beans but they are automatically gets created

Why Spring Boot?

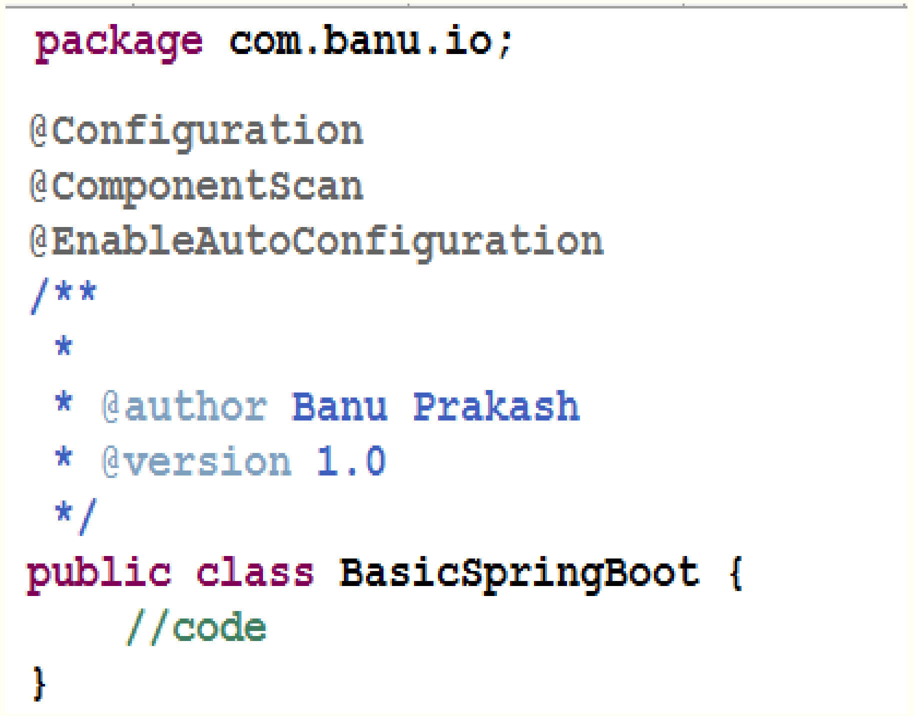
- **Faster**
  - Generate project scaffolding using Initializr and your IDE
  - Spring Boot start-up from 10 seconds to 2 during development
- **Smarter**
  - Tune or disable to your needs using @EnableAutoConfiguration
- **Easier**
  - Integrate with your IDE
- **Cloudier**
  - Deploy your Spring Boot apps to the cloud

Reducing Code with Custom Auto Configurations

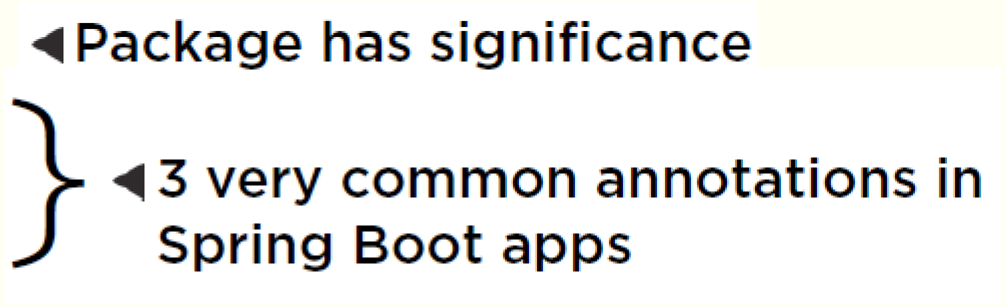
- Spring Boot is highly configurable
  - Web?
  - Front end?
  - Data access?
  - Security?

Reducing Code with Custom Auto Configurations

- @EnableAutoConfiguration is like having your own opinionated event planner. It enables features and configures functionality



```
package com.banu.io;

@Configuration
@ComponentScan
@EnableAutoConfiguration
/**
 *
 * @author Banu Prakash
 * @version 1.0
 */
public class BasicSpringBoot {
    //code
}
```



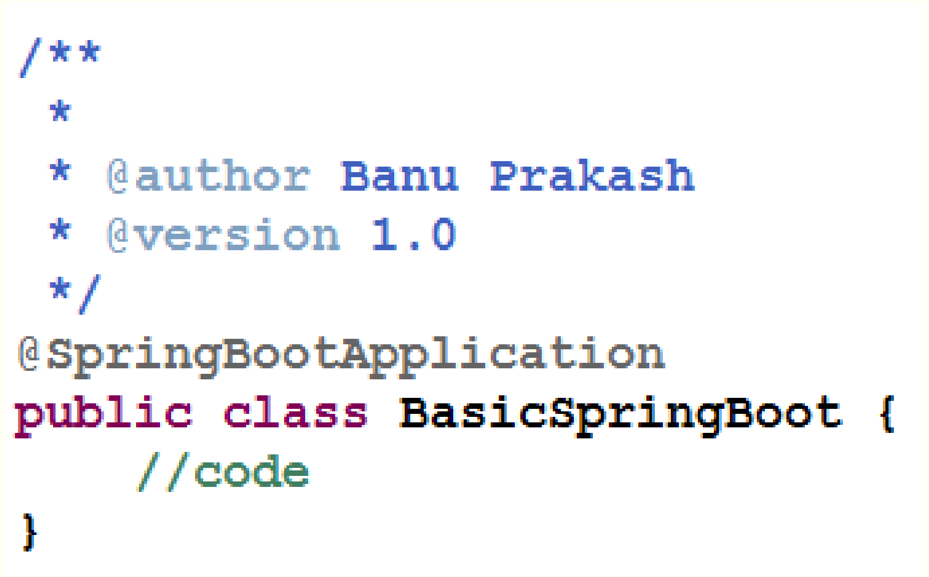
◀ Package has significance

◀ 3 very common annotations in Spring Boot apps

Reducing Code with Custom Auto Configurations

- @Configuration tags the class as a source of bean definitions for the application context.
- @EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.
- @ComponentScan tells Spring to look for other components, configurations, and services in the package package/sub package.

Reducing Code with Custom Auto Configurations

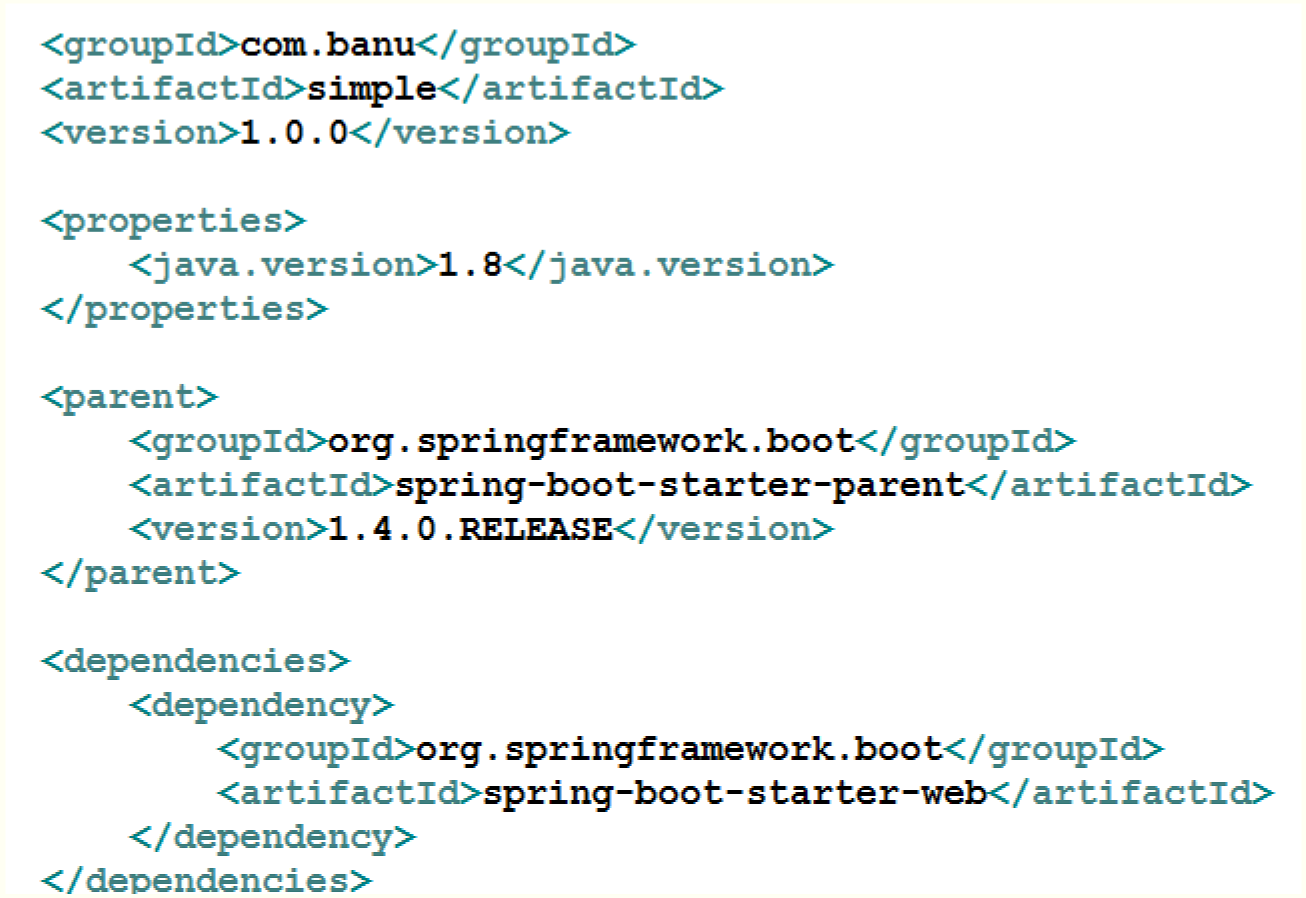


```
/**
 *
 * @author Banu Prakash
 * @version 1.0
 */
@SpringBootApplication
public class BasicSpringBoot {
    //code
}
```

- @SpringBootApplication was introduced in Spring Boot1.2.
- Its 3 annotations in one:
  - @Configuration
  - @ComponentScan
  - @EnableAutoConfiguration

Building an Application with Spring Boot

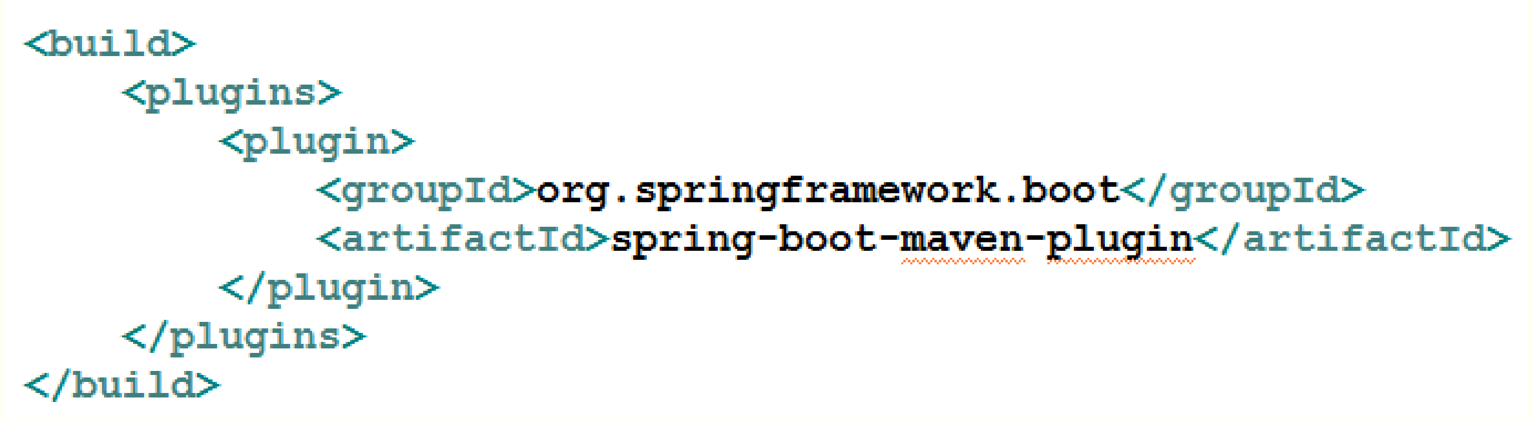
- Build with Maven [pom.xml]



```
<groupId>com.banu</groupId>
<artifactId>simple</artifactId>
<version>1.0.0</version>

<properties>
    <java.version>1.8</java.version>
</properties>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Building an Application with Spring Boot

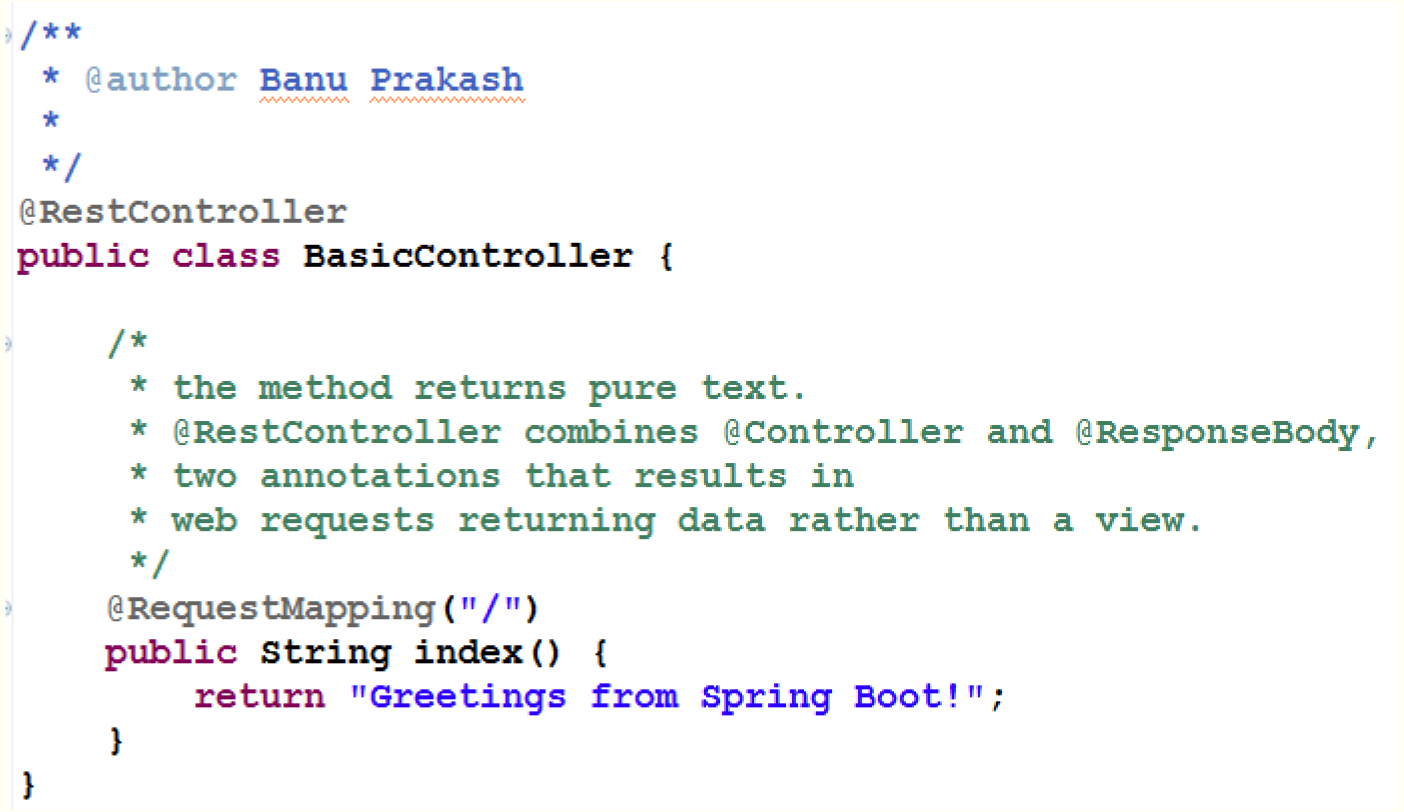
- Build with Maven [pom.xml]
- The Spring Boot Maven plugin provides many convenient features:
- It collects all the jars on the classpath and builds a single, runnable "jar", which makes it more convenient to execute and transport your service.
- It searches for the public static void main() method to flag as a runnable class.



```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Building an Application with Spring Boot

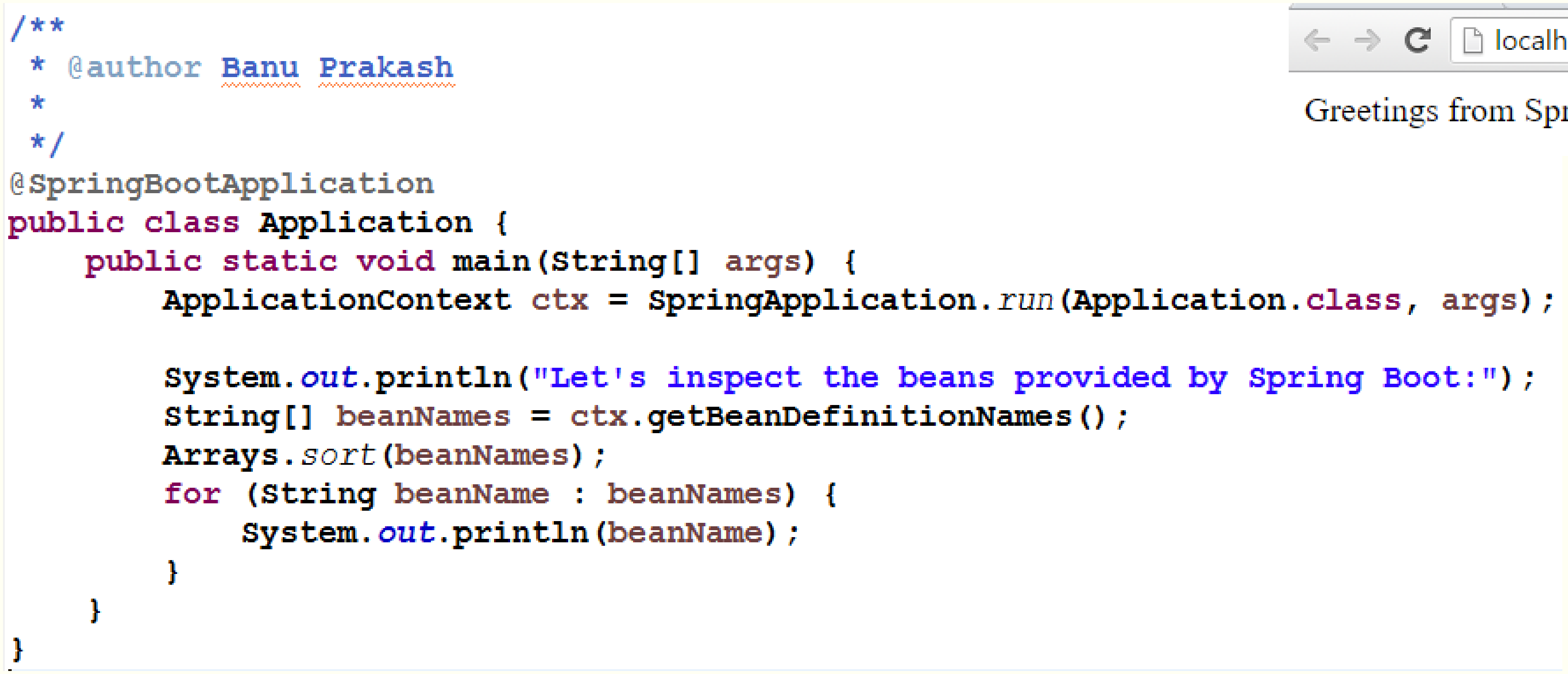
- RestController



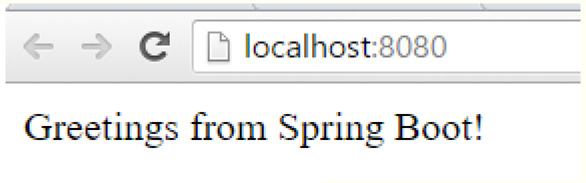
```
/**
 * @author Banu Prakash
 *
 */
@RestController
public class BasicController {

    /*
     * the method returns pure text.
     * @RestController combines @Controller and @ResponseBody,
     * two annotations that results in
     * web requests returning data rather than a view.
     */
    @RequestMapping("/")
    public String index() {
        return "Greetings from Spring Boot!";
    }
}
```

Create an Application class

- The main() method uses Spring Boot's SpringApplication.run() method to launch an application.
- There isn't a single line of XML No web.xml file either. This web application is 100% pure Java and you didn't have to deal with configuring any plumbing or infrastructure.



```java
/**
 * @author Banu Prakash
 *
 */
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Application.class, args);

        System.out.println("Let's inspect the beans provided by Spring Boot:");
        String[] beanNames = ctx.getBeanDefinitionNames();
        Arrays.sort(beanNames);
        for (String beanName : beanNames) {
            System.out.println(beanName);
        }
    }
}
```



Greetings from Spring Boot!

Tuning your Auto Configuration

- Intelligent Decision Making Based on Conditions

Presence/
Absence of Jar

Presence/
Absence of Bean

Presence/Absence
of Property

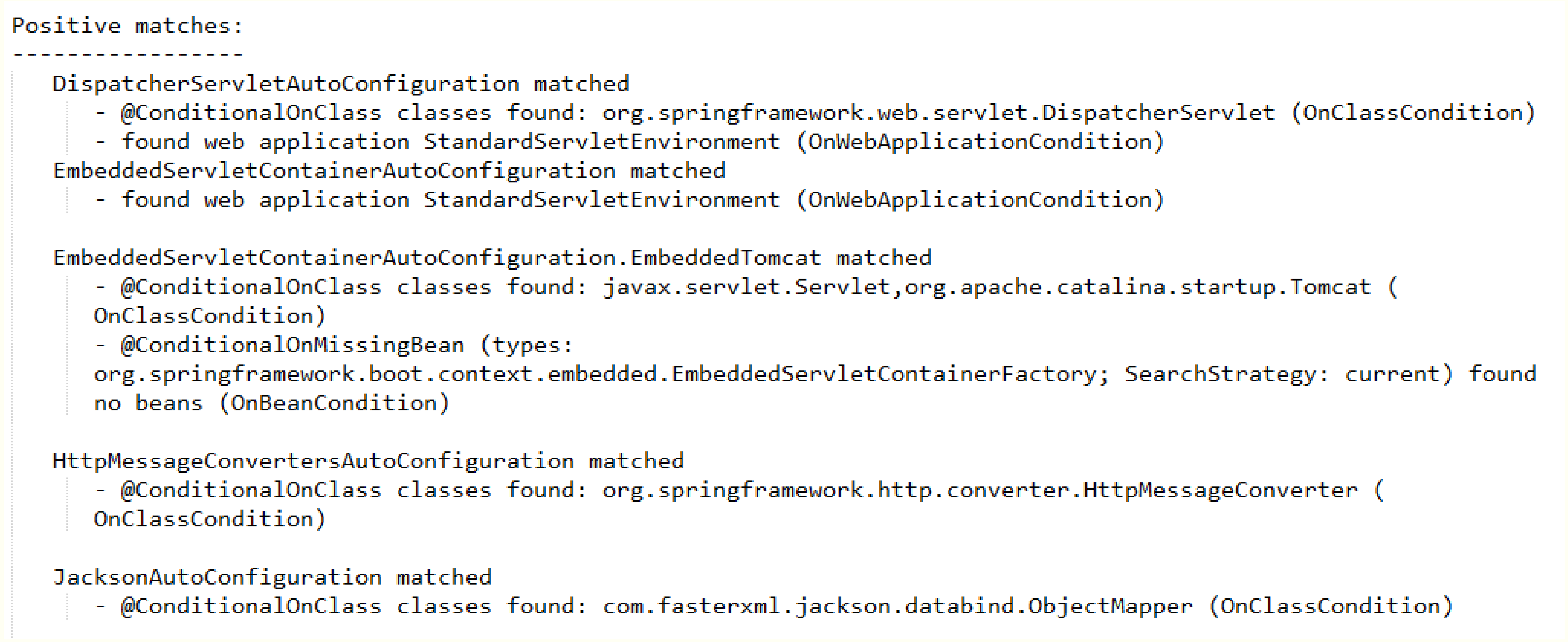
Many More!

Enabling the Auto Configuration Report

- Command line argument [--debug]
- VM arguments [-Ddebug]
- Environment variables [export DEBUG=true]
- application.properties [debug=true]
- application.properties [logging.level.=debug]
- Using actuator endpoint and many more …

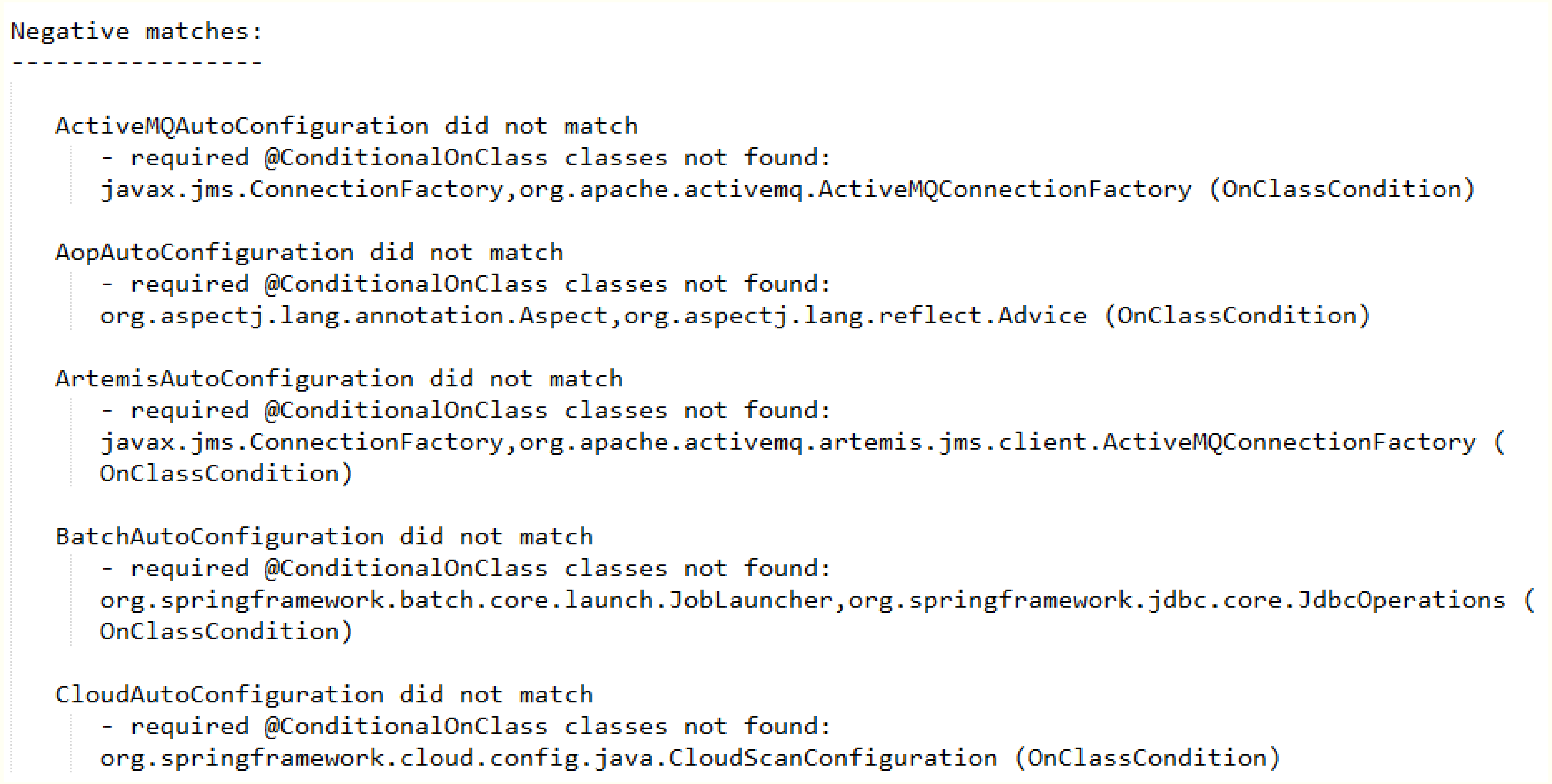
```
=============================
AUTO-CONFIGURATION REPORT
=============================
```

AUTO-CONFIGURATION REPORT [Positive matches]



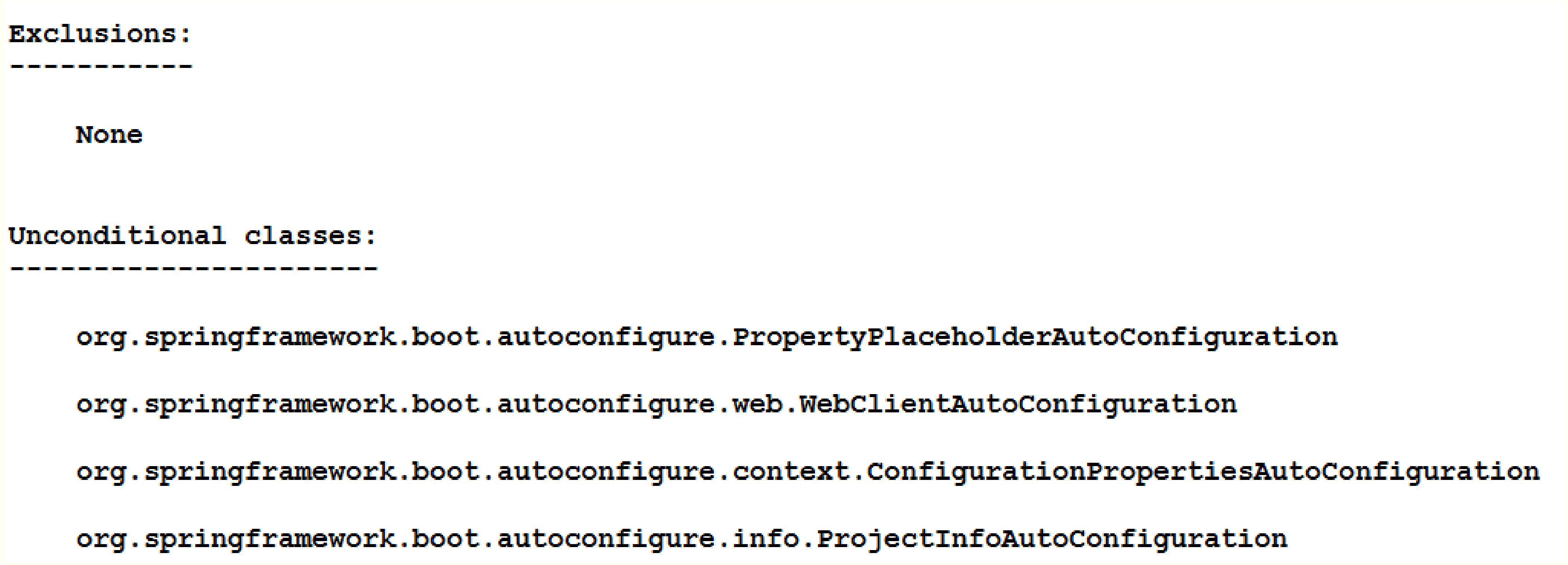
```
Positive matches:
-----------------
   DispatcherServletAutoConfiguration matched
      - @ConditionalOnClass classes found: org.springframework.web.servlet.DispatcherServlet (OnClassCondition)
      - found web application StandardServletEnvironment (OnWebApplicationCondition)
   EmbeddedServletContainerAutoConfiguration matched
      - found web application StandardServletEnvironment (OnWebApplicationCondition)

   EmbeddedServletContainerAutoConfiguration.EmbeddedTomcat matched
      - @ConditionalOnClass classes found: javax.servlet.Servlet,org.apache.catalina.startup.Tomcat (
      OnClassCondition)
      - @ConditionalOnMissingBean (types:
      org.springframework.boot.context.embedded.EmbeddedServletContainerFactory; SearchStrategy: current) found
      no beans (OnBeanCondition)

   HttpMessageConvertersAutoConfiguration matched
      - @ConditionalOnClass classes found: org.springframework.http.converter.HttpMessageConverter (
      OnClassCondition)

   JacksonAutoConfiguration matched
      - @ConditionalOnClass classes found: com.fasterxml.jackson.databind.ObjectMapper (OnClassCondition)
```

AUTO-CONFIGURATION REPORT [Negative matches]



```
Negative matches:
-----------------

   ActiveMQAutoConfiguration did not match
      - required @ConditionalOnClass classes not found:
      javax.jms.ConnectionFactory,org.apache.activemq.ActiveMQConnectionFactory (OnClassCondition)

   AopAutoConfiguration did not match
      - required @ConditionalOnClass classes not found:
      org.aspectj.lang.annotation.Aspect,org.aspectj.lang.reflect.Advice (OnClassCondition)

   ArtemisAutoConfiguration did not match
      - required @ConditionalOnClass classes not found:
      javax.jms.ConnectionFactory,org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory (
      OnClassCondition)

   BatchAutoConfiguration did not match
      - required @ConditionalOnClass classes not found:
      org.springframework.batch.core.launch.JobLauncher,org.springframework.jdbc.core.JdbcOperations (
      OnClassCondition)

   CloudAutoConfiguration did not match
      - required @ConditionalOnClass classes not found:
      org.springframework.cloud.config.java.CloudScanConfiguration (OnClassCondition)
```

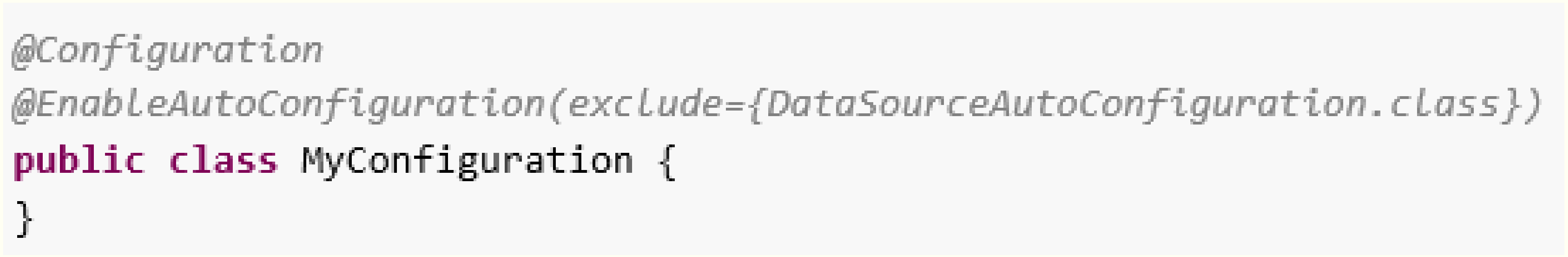
AUTO-CONFIGURATION REPORT



```
Exclusions:
-----------

    None


Unconditional classes:
----------------------

    org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration

    org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration

    org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration

    org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration
```

Excluding Unnecessary or Misconfigured Auto Configurations via Annotations

- If you find that specific auto-configure classes are being applied that you don’t want, you can use the exclude parameter of annotation as a comma separated list of classes attribute of @EnableAutoConfiguration to disable them

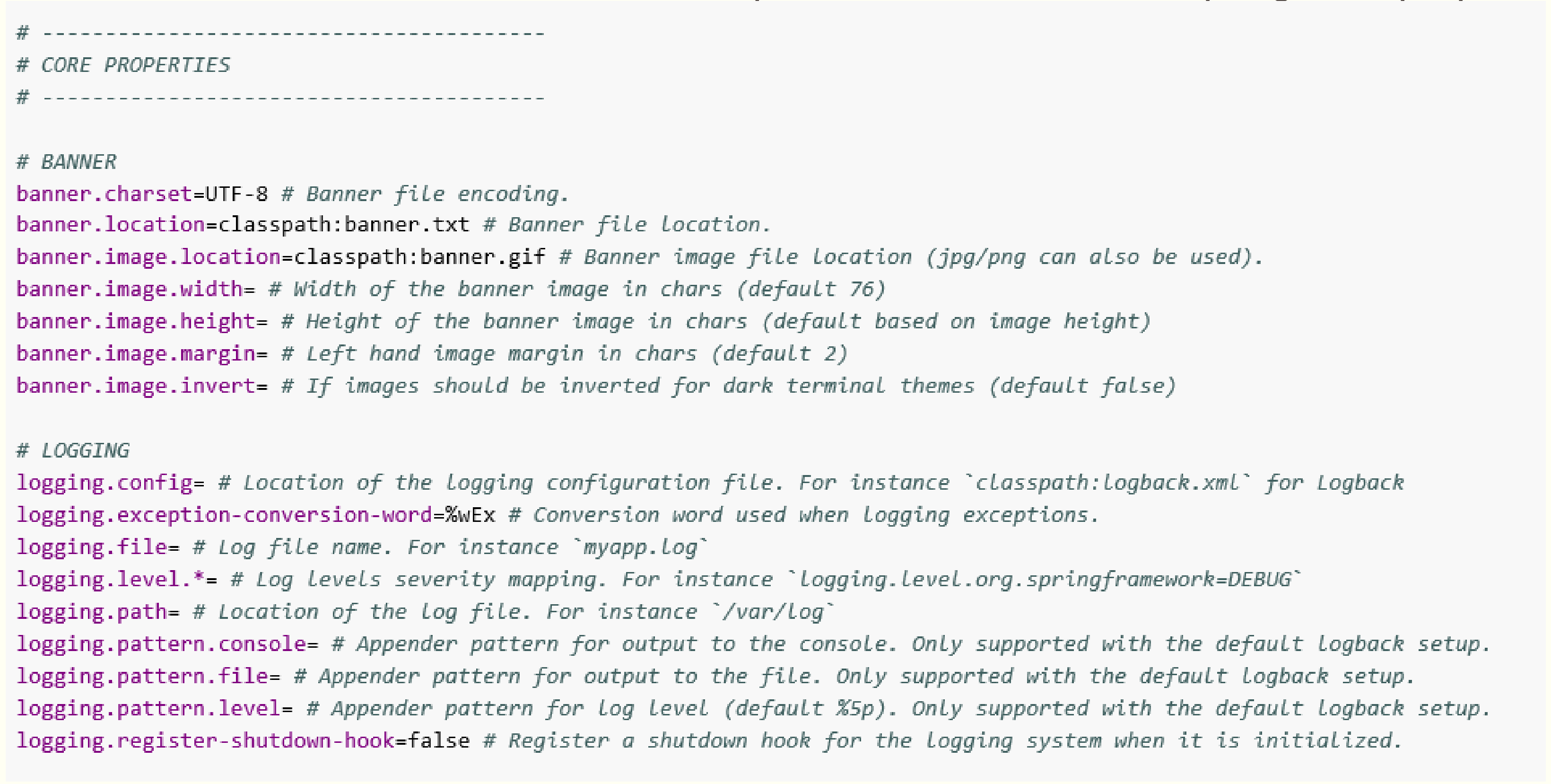


```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

- If the class is not on the classpath, you can use the excludeName attribute of the annotation and specify parameter of annotation as a comma separated list of class names. [**@EnableAutoConfiguration**(excludeName = { “a.b.SomeAutoConfig” })]
- Finally, you can also control the list of auto-configuration classes to exclude via the spring.autoconfigure.exclude property. [**spring.autoconfigure.exclude** = my.company.SomeAutoConfig]

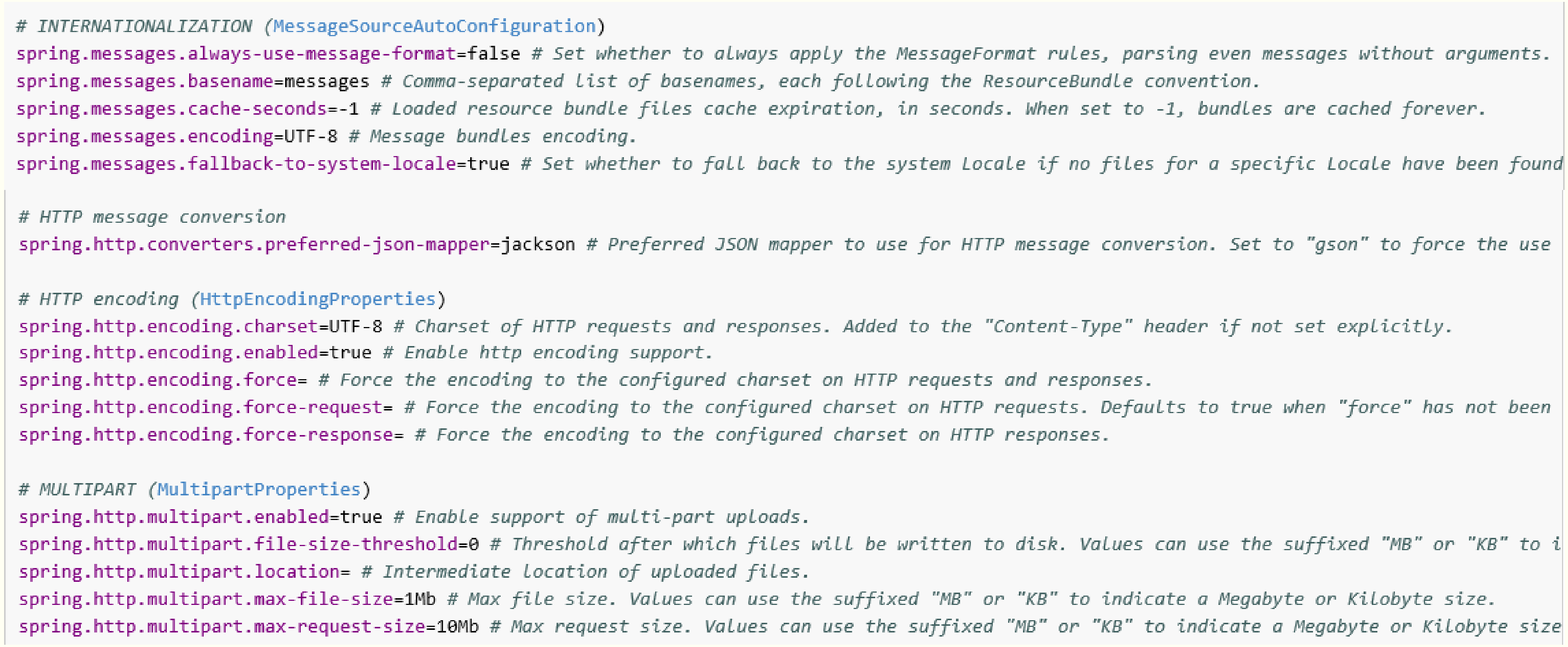
Tune auto configurations via properties in the application.properties

- **Common application properties**
  - Various properties can be specified inside your application.properties/application.yml file or as command line switches. This section provides a list common Spring Boot properties and references to



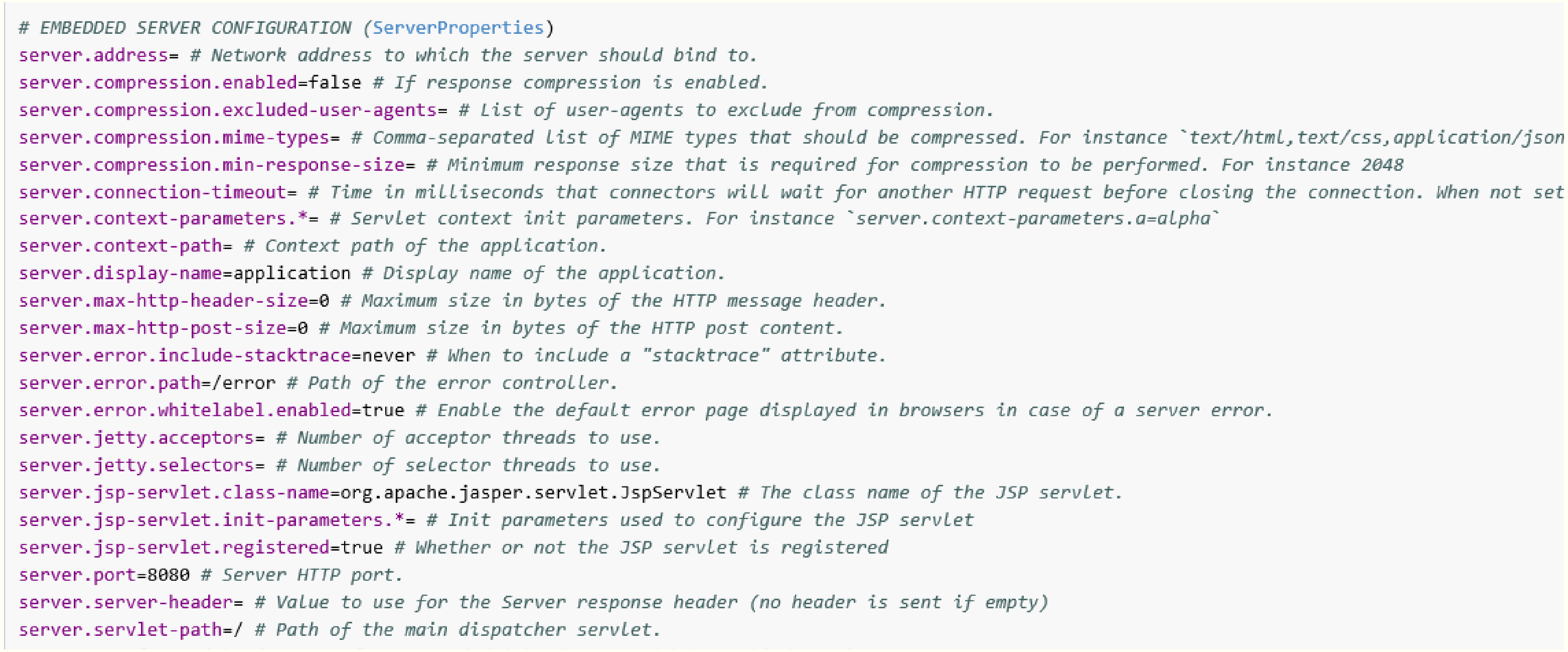
```
# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------

# BANNER
banner.charset=UTF-8 # Banner file encoding.
banner.location=classpath:banner.txt # Banner file location.
banner.image.location=classpath:banner.gif # Banner image file location (jpg/png can also be used).
banner.image.width= # Width of the banner image in chars (default 76)
banner.image.height= # Height of the banner image in chars (default based on image height)
banner.image.margin= # Left hand image margin in chars (default 2)
banner.image.invert= # If images should be inverted for dark terminal themes (default false)

# LOGGING
logging.config= # Location of the logging configuration file. For instance `classpath:logback.xml` for Logback
logging.exception-conversion-word=%wEx # Conversion word used when logging exceptions.
logging.file= # Log file name. For instance `myapp.log`
logging.level.*= # Log levels severity mapping. For instance `logging.level.org.springframework=DEBUG`
logging.path= # Location of the log file. For instance `/var/log`
logging.pattern.console= # Appender pattern for output to the console. Only supported with the default logback setup.
logging.pattern.file= # Appender pattern for output to the file. Only supported with the default logback setup.
logging.pattern.level= # Appender pattern for log level (default %5p). Only supported with the default logback setup.
logging.register-shutdown-hook=false # Register a shutdown hook for the logging system when it is initialized.
```

Tune auto configurations via properties in the application.properties



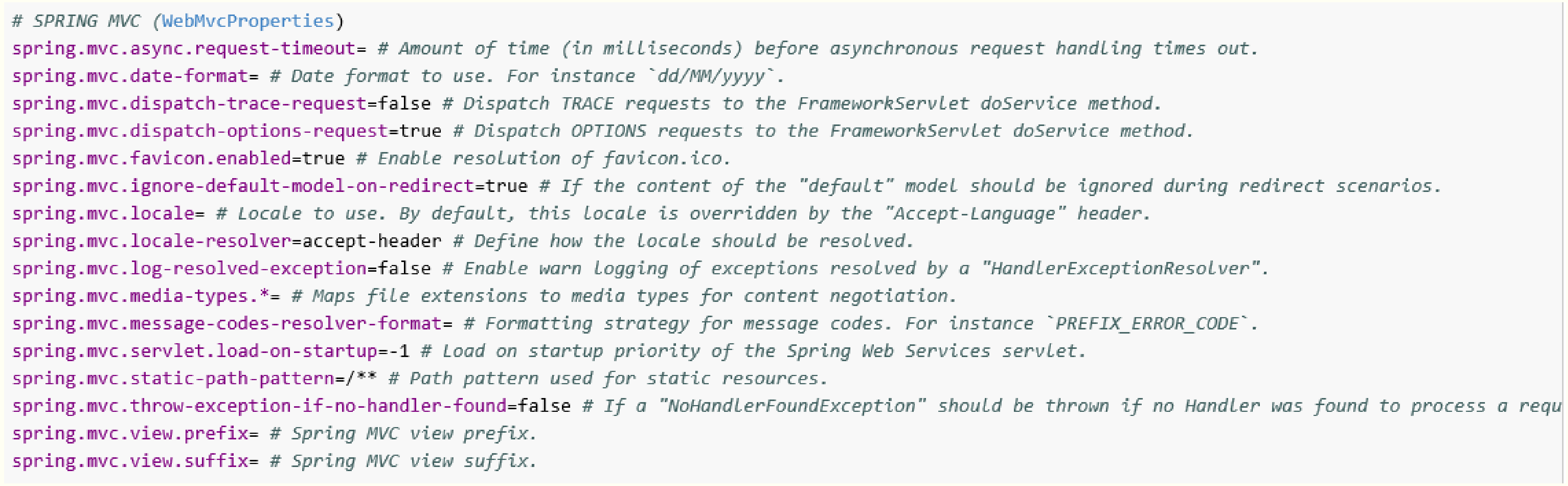
```
# INTERNATIONALIZATION (MessageSourceAutoConfiguration)
spring.messages.always-use-message-format=false # Set whether to always apply the MessageFormat rules, parsing even messages without arguments.
spring.messages.basename=messages # Comma-separated list of basenames, each following the ResourceBundle convention.
spring.messages.cache-seconds=-1 # Loaded resource bundle files cache expiration, in seconds. When set to -1, bundles are cached forever.
spring.messages.encoding=UTF-8 # Message bundles encoding.
spring.messages.fallback-to-system-locale=true # Set whether to fall back to the system Locale if no files for a specific Locale have been found

# HTTP message conversion
spring.http.converters.preferred-json-mapper=jackson # Preferred JSON mapper to use for HTTP message conversion. Set to "gson" to force the use

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses. Added to the "Content-Type" header if not set explicitly.
spring.http.encoding.enabled=true # Enable http encoding support.
spring.http.encoding.force= # Force the encoding to the configured charset on HTTP requests and responses.
spring.http.encoding.force-request= # Force the encoding to the configured charset on HTTP requests. Defaults to true when "force" has not been
spring.http.encoding.force-response= # Force the encoding to the configured charset on HTTP responses.

# MULTIPART (MultipartProperties)
spring.http.multipart.enabled=true # Enable support of multi-part uploads.
spring.http.multipart.file-size-threshold=0 # Threshold after which files will be written to disk. Values can use the suffixed "MB" or "KB" to i
spring.http.multipart.location= # Intermediate location of uploaded files.
spring.http.multipart.max-file-size=1Mb # Max file size. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.
spring.http.multipart.max-request-size=10Mb # Max request size. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size
```

Tune auto configurations via properties in the application.properties



```
# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.address= # Network address to which the server should bind to.
server.compression.enabled=false # If response compression is enabled.
server.compression.excluded-user-agents= # List of user-agents to exclude from compression.
server.compression.mime-types= # Comma-separated list of MIME types that should be compressed. For instance `text/html,text/css,application/json
server.compression.min-response-size= # Minimum response size that is required for compression to be performed. For instance 2048
server.connection-timeout= # Time in milliseconds that connectors will wait for another HTTP request before closing the connection. When not set
server.context-parameters.*= # Servlet context init parameters. For instance `server.context-parameters.a=alpha`
server.context-path= # Context path of the application.
server.display-name=application # Display name of the application.
server.max-http-header-size=0 # Maximum size in bytes of the HTTP message header.
server.max-http-post-size=0 # Maximum size in bytes of the HTTP post content.
server.error.include-stacktrace=never # When to include a "stacktrace" attribute.
server.error.path=/error # Path of the error controller.
server.error.whitelabel.enabled=true # Enable the default error page displayed in browsers in case of a server error.
server.jetty.acceptors= # Number of acceptor threads to use.
server.jetty.selectors= # Number of selector threads to use.
server.jsp-servlet.class-name=org.apache.jasper.servlet.JspServlet # The class name of the JSP servlet.
server.jsp-servlet.init-parameters.*= # Init parameters used to configure the JSP servlet
server.jsp-servlet.registered=true # Whether or not the JSP servlet is registered
server.port=8080 # Server HTTP port.
server.server-header= # Value to use for the Server response header (no header is sent if empty)
server.servlet-path=/ # Path of the main dispatcher servlet.
```

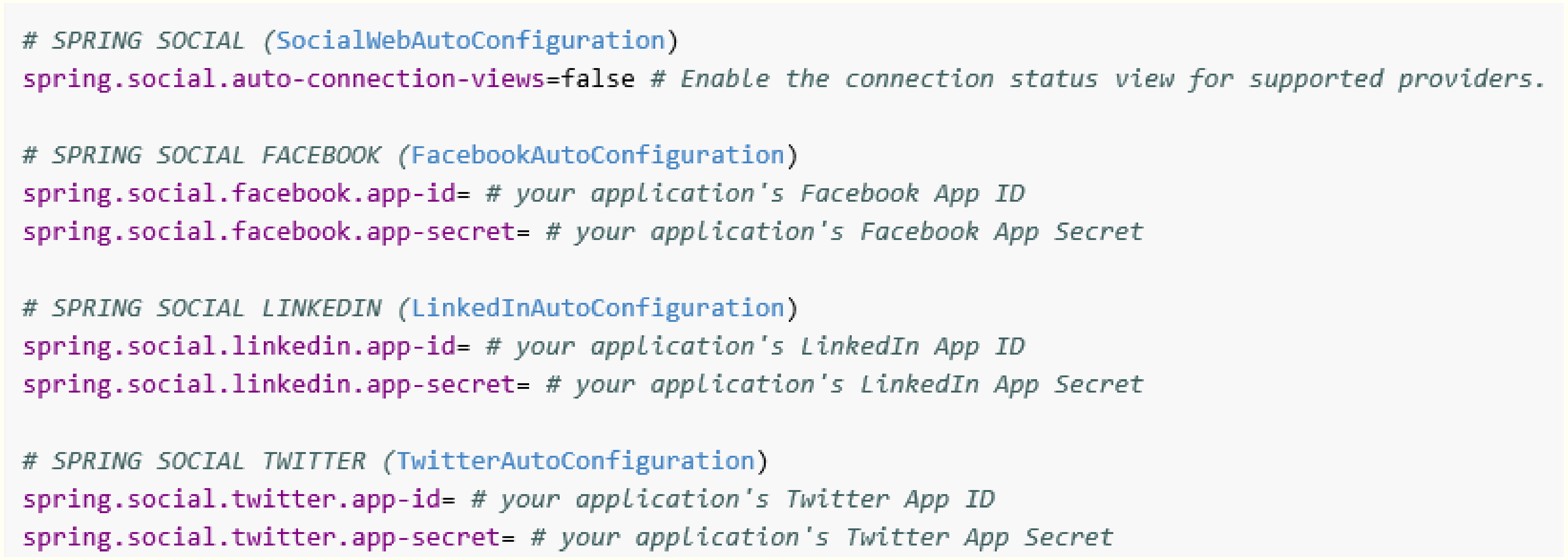
Tune auto configurations via properties in the application.properties



```
# SPRING MVC (WebMvcProperties)
spring.mvc.async.request-timeout= # Amount of time (in milliseconds) before asynchronous request handling times out.
spring.mvc.date-format= # Date format to use. For instance `dd/MM/yyyy`.
spring.mvc.dispatch-trace-request=false # Dispatch TRACE requests to the FrameworkServlet doService method.
spring.mvc.dispatch-options-request=true # Dispatch OPTIONS requests to the FrameworkServlet doService method.
spring.mvc.favicon.enabled=true # Enable resolution of favicon.ico.
spring.mvc.ignore-default-model-on-redirect=true # If the content of the "default" model should be ignored during redirect scenarios.
spring.mvc.locale= # Locale to use. By default, this locale is overridden by the "Accept-Language" header.
spring.mvc.locale-resolver=accept-header # Define how the locale should be resolved.
spring.mvc.log-resolved-exception=false # Enable warn logging of exceptions resolved by a "HandlerExceptionResolver".
spring.mvc.media-types.*= # Maps file extensions to media types for content negotiation.
spring.mvc.message-codes-resolver-format= # Formatting strategy for message codes. For instance `PREFIX_ERROR_CODE`.
spring.mvc.servlet.load-on-startup=-1 # Load on startup priority of the Spring Web Services servlet.
spring.mvc.static-path-pattern=/** # Path pattern used for static resources.
spring.mvc.throw-exception-if-no-handler-found=false # If a "NoHandlerFoundException" should be thrown if no Handler was found to process a requ
spring.mvc.view.prefix= # Spring MVC view prefix.
spring.mvc.view.suffix= # Spring MVC view suffix.
```

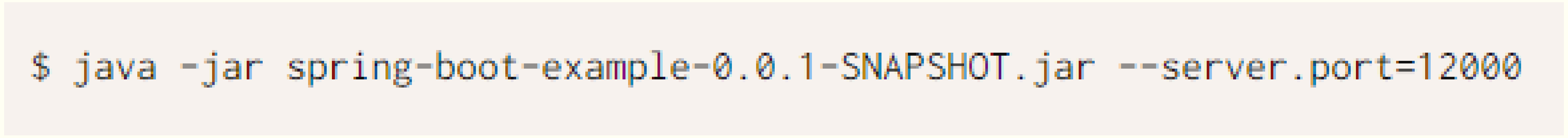
Tune auto configurations via properties in the application.properties

- Refer: http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties for complete list



```
# SPRING SOCIAL (SocialWebAutoConfiguration)
spring.social.auto-connection-views=false # Enable the connection status view for supported providers.

# SPRING SOCIAL FACEBOOK (FacebookAutoConfiguration)
spring.social.facebook.app-id= # your application's Facebook App ID
spring.social.facebook.app-secret= # your application's Facebook App Secret

# SPRING SOCIAL LINKEDIN (LinkedInAutoConfiguration)
spring.social.linkedin.app-id= # your application's LinkedIn App ID
spring.social.linkedin.app-secret= # your application's LinkedIn App Secret

# SPRING SOCIAL TWITTER (TwitterAutoConfiguration)
spring.social.twitter.app-id= # your application's Twitter App ID
spring.social.twitter.app-secret= # your application's Twitter App Secret
```

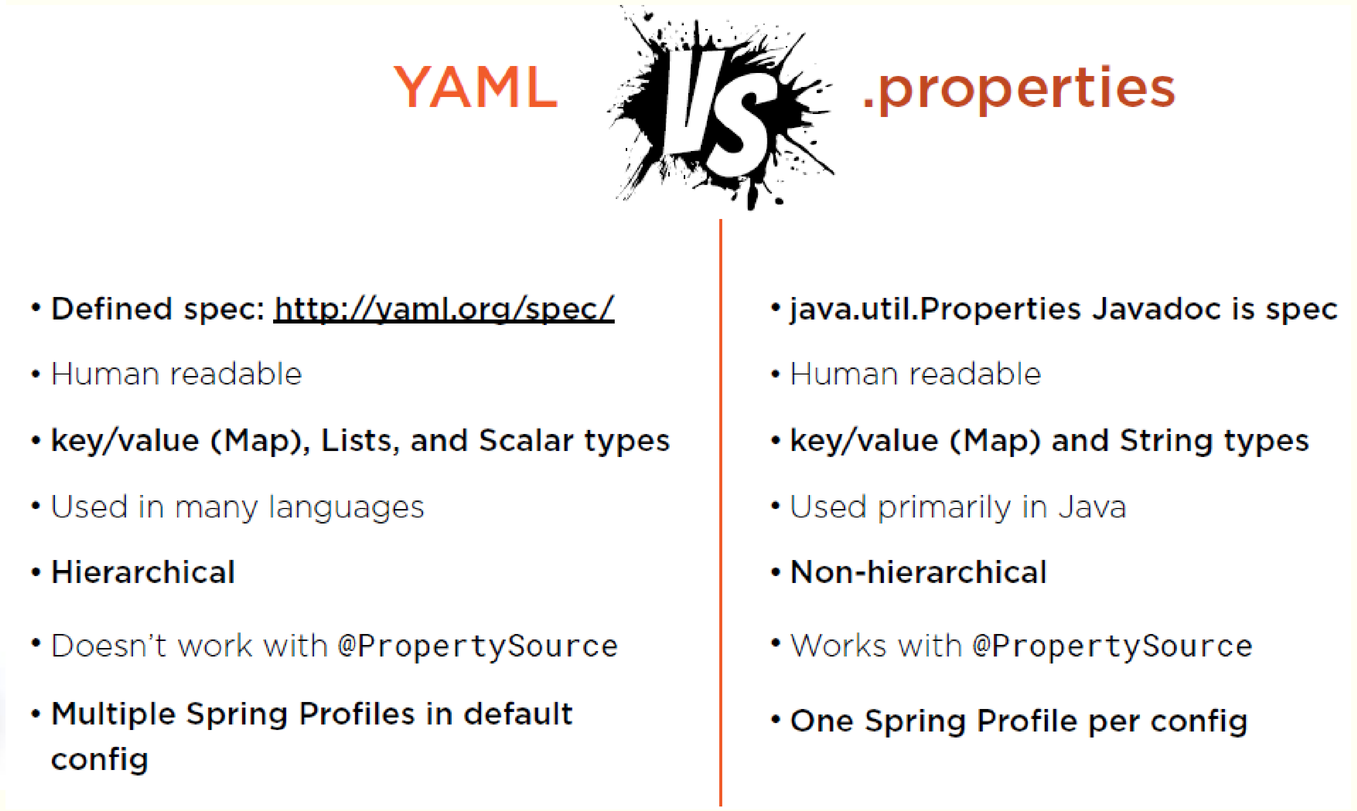
Overriding default configuration with command line arguments



```
$ java -jar spring-boot-example-0.0.1-SNAPSHOT.jar --server.port=12000
```

Externalized Configuration

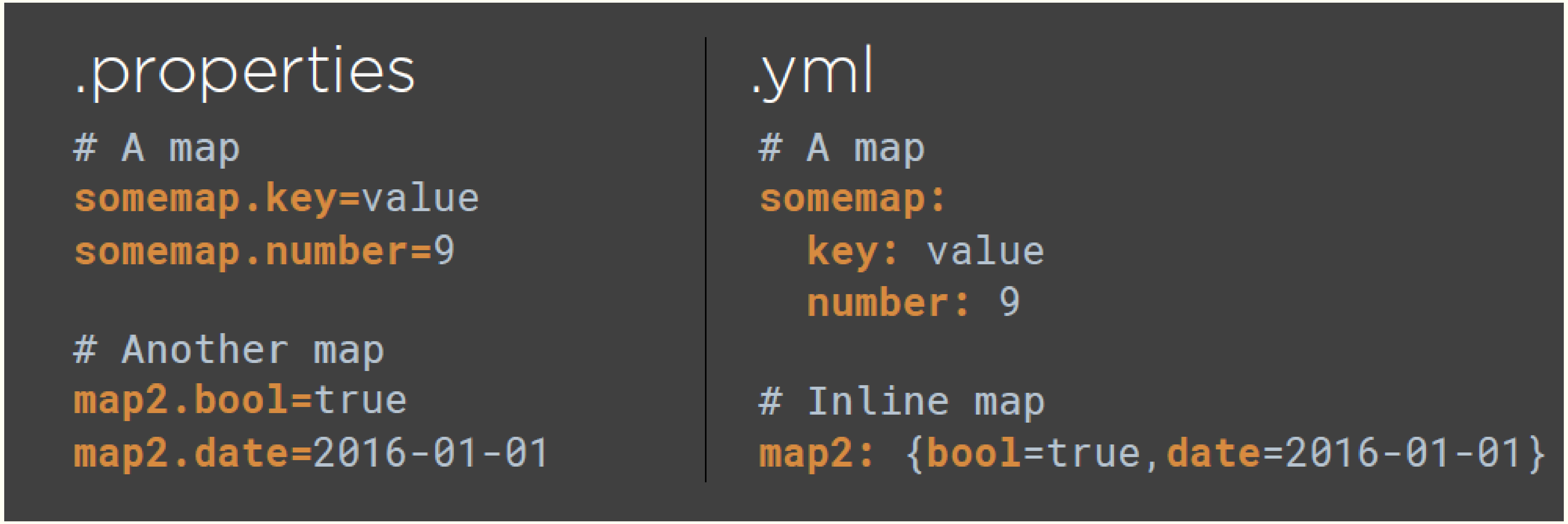
- YAML
  - A data serialization standard made for configuration files



YAML	.properties
Defined spec: http://yaml.org/spec/	**java.util.Properties Javadoc is spec**
Human readable	Human readable
key/value (Map), Lists, and Scalar types	**key/value (Map) and String types**
Used in many languages	Used primarily in Java
Hierarchical	**Non-hierarchical**
Doesn't work with `@PropertySource`	Works with `@PropertySource`
Multiple Spring Profiles in default config	**One Spring Profile per config**

YAML Maps

- .properties uses dots to denote hierarchy (a Spring convention)
- .yml uses hierarchy (consistent spaces) to create maps



.properties

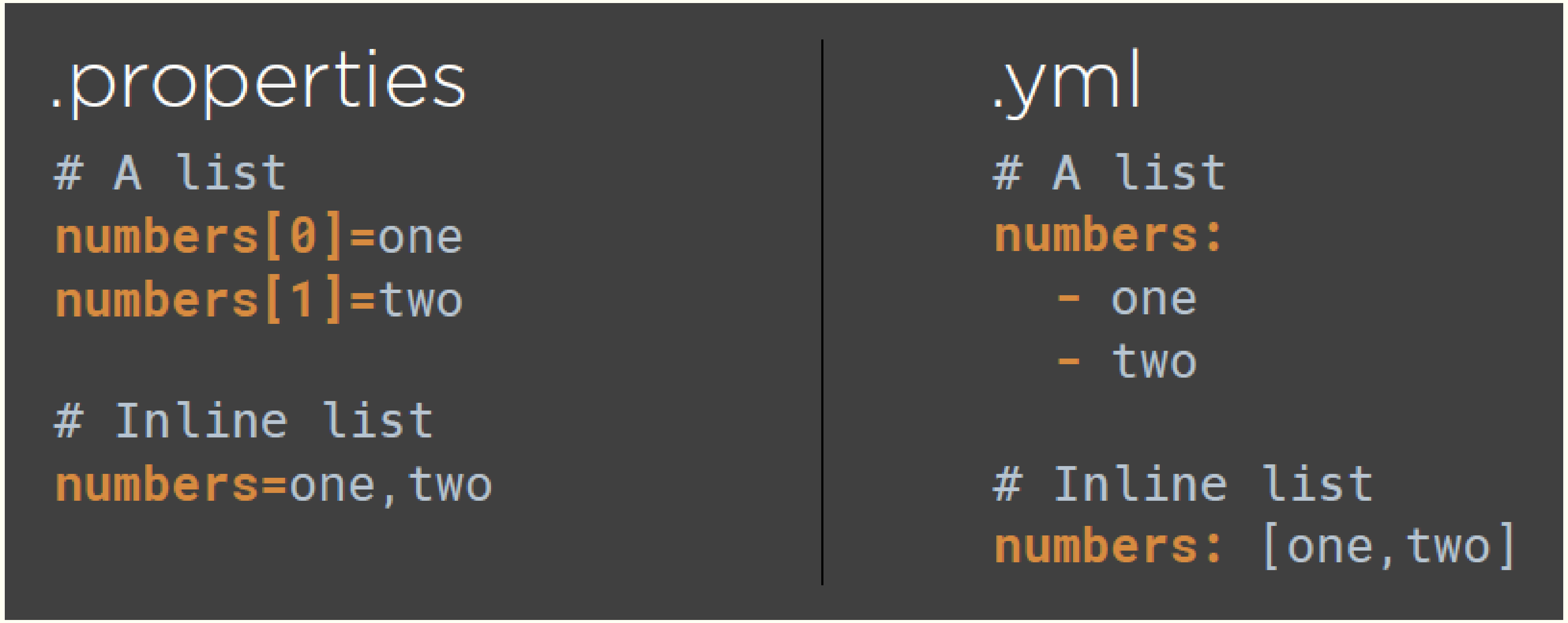
```
# A map
somemap.key=value
somemap.number=9

# Another map
map2.bool=true
map2.date=2016-01-01
```

.yml

```
# A map
somemap:
  key: value
  number: 9

# Inline map
map2: {bool=true,date=2016-01-01}
```

YAML Lists

- .properties uses prop[index] or commas for a List (a Spring convention)
- .yml uses '- value' or commas surrounded with brackets for a List



.properties

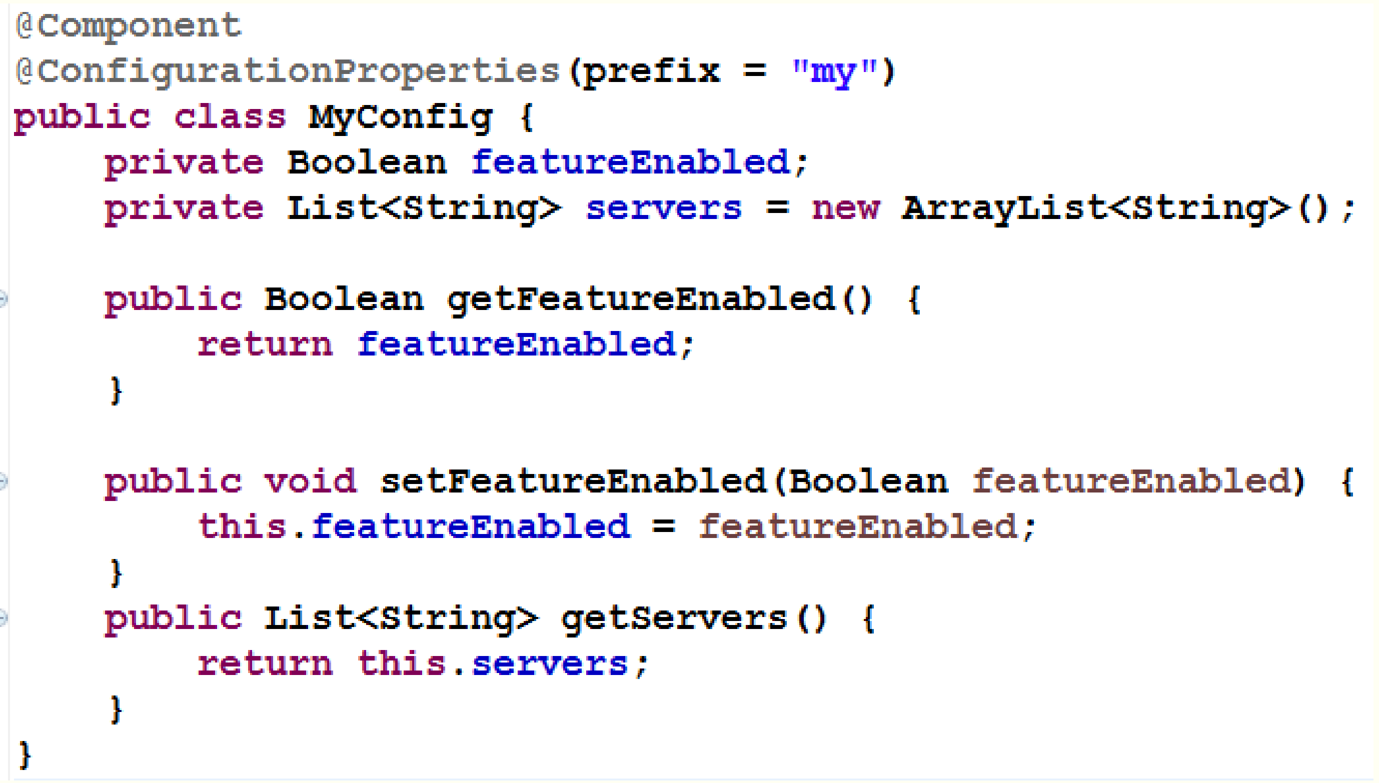
```
# A list
numbers[0]=one
numbers[1]=two

# Inline list
numbers=one,two
```

.yml

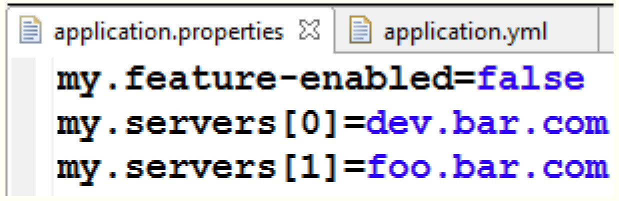
```
# A list
numbers:
  - one
  - two

# Inline list
numbers: [one,two]
```

Typesafe Configuration

- @ConfigurationProperties turns all of your application configuration into typesafe POJOs

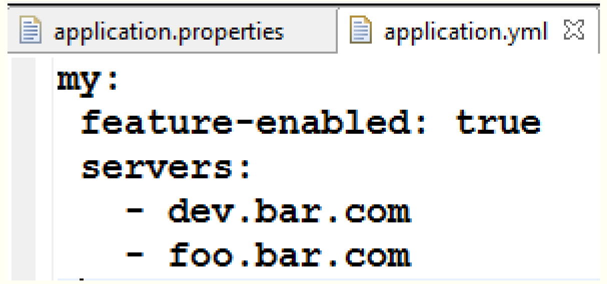


```java
@Component
@ConfigurationProperties(prefix = "my")
public class MyConfig {
    private Boolean featureEnabled;
    private List<String> servers = new ArrayList<String>();

    public Boolean getFeatureEnabled() {
        return featureEnabled;
    }

    public void setFeatureEnabled(Boolean featureEnabled) {
        this.featureEnabled = featureEnabled;
    }
    public List<String> getServers() {
        return this.servers;
    }
}
```



application.properties application.yml

```
my.feature-enabled=false
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```

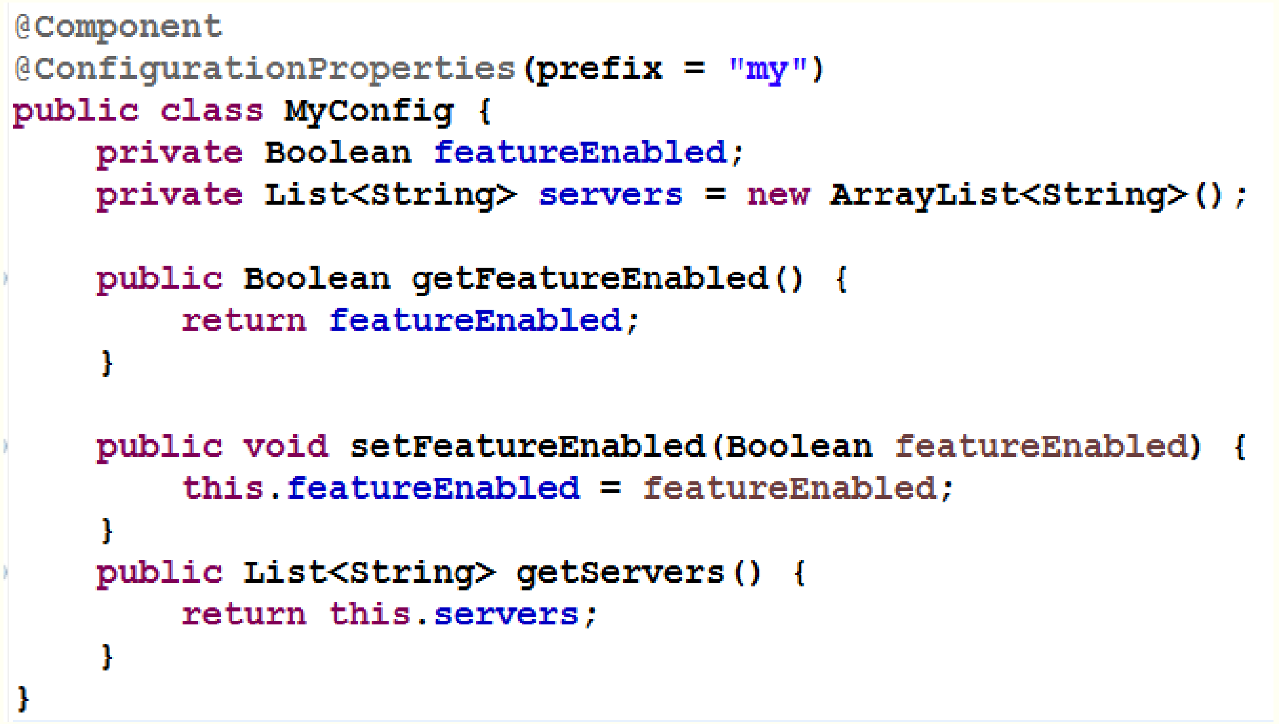


application.properties application.yml

```yaml
my:
 feature-enabled: true
 servers:
   - dev.bar.com
   - foo.bar.com
```

Typesafe Configuration

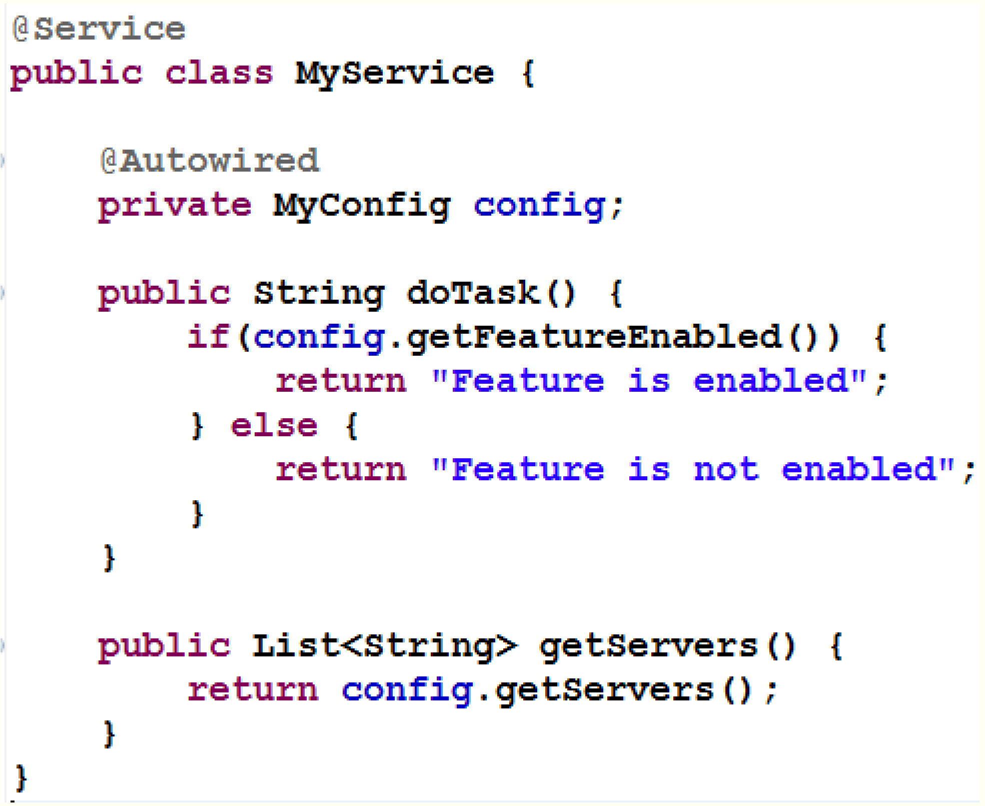
- Application Configuration into type safe POJOs
  - Create an Instance Variable for Your Property
  - Annotate class with @ConfigurationProperties
  - Annotate class with @Component



```
@Component
@ConfigurationProperties(prefix = "my")
public class MyConfig {
    private Boolean featureEnabled;
    private List<String> servers = new ArrayList<String>();

    public Boolean getFeatureEnabled() {
        return featureEnabled;
    }

    public void setFeatureEnabled(Boolean featureEnabled) {
        this.featureEnabled = featureEnabled;
    }
    public List<String> getServers() {
        return this.servers;
    }
}
```

Typesafe Configuration

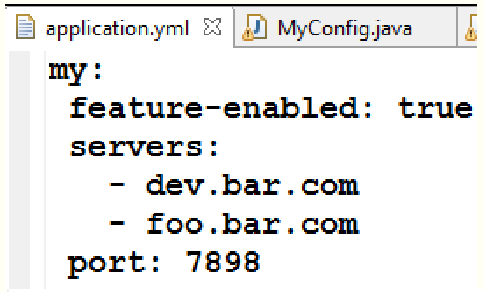
- Autowire It into Any Class



```
@Service
public class MyService {

    @Autowired
    private MyConfig config;

    public String doTask() {
        if(config.getFeatureEnabled()) {
            return "Feature is enabled";
        } else {
            return "Feature is not enabled";
        }
    }

    public List<String> getServers() {
        return config.getServers();
    }
}
```

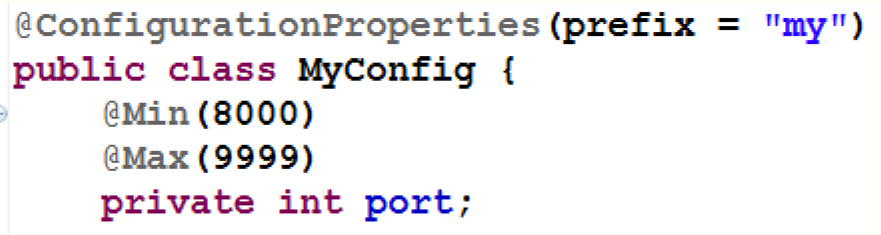
Validating Your Configuration

- Simply annotate your instance variables with JSR-303 Annotations
  - @NotNull
  - @Pattern
  - @Max
  - @Min
  - @Digits
  - And more

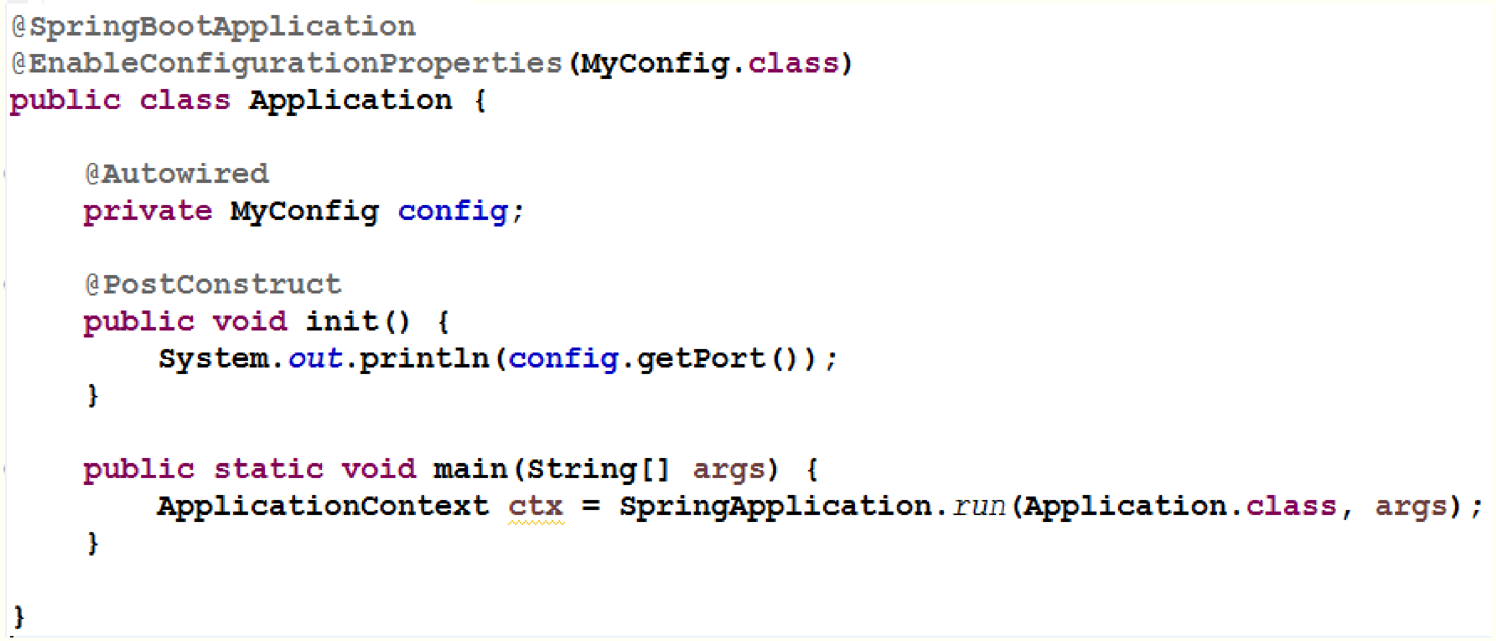


application.yml MyConfig.java

```yaml
my:
 feature-enabled: true
 servers:
   - dev.bar.com
   - foo.bar.com
 port: 7898
```

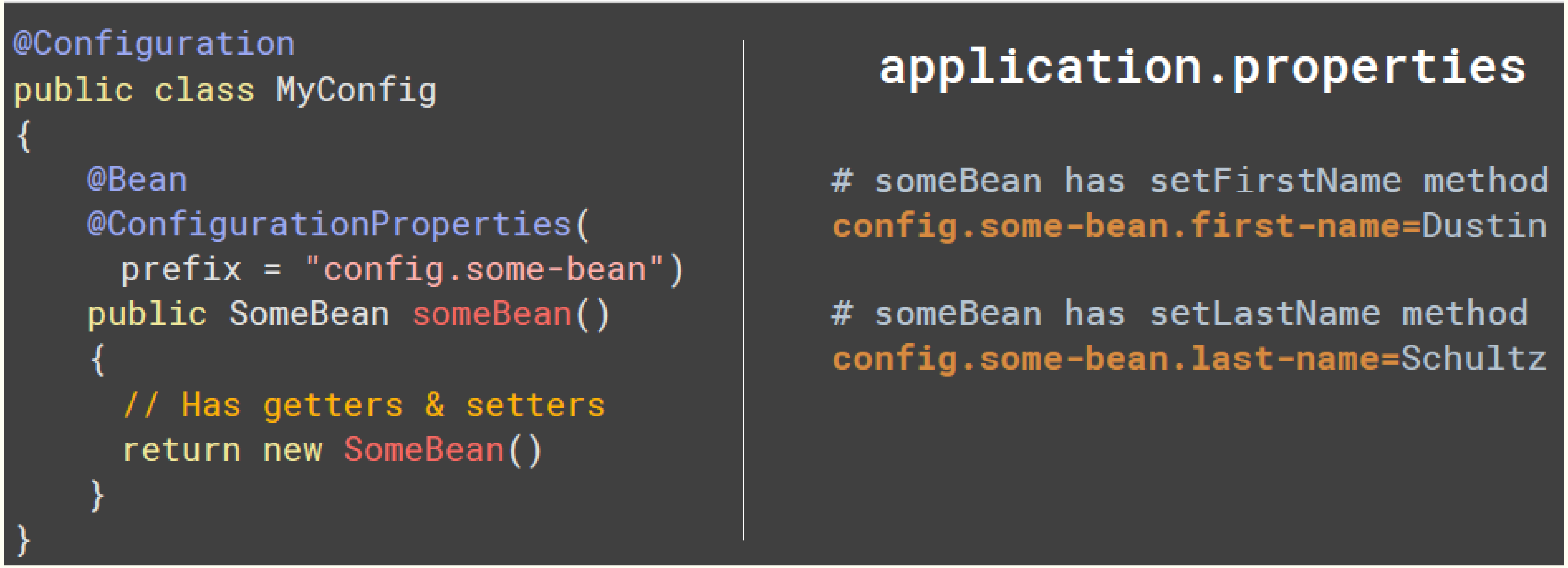


```java
@ConfigurationProperties(prefix = "my")
public class MyConfig {
    @Min(8000)
    @Max(9999)
    private int port;
```



```java
@SpringBootApplication
@EnableConfigurationProperties(MyConfig.class)
public class Application {

    @Autowired
    private MyConfig config;

    @PostConstruct
    public void init() {
        System.out.println(config.getPort());
    }

    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Application.class, args);
    }

}
```

Configuring Third Party Beans



```java
@Configuration
public class MyConfig
{
    @Bean
    @ConfigurationProperties(
      prefix = "config.some-bean")
    public SomeBean someBean()
    {
      // Has getters & setters
      return new SomeBean()
    }
}
```

application.properties

```properties
# someBean has setFirstName method
config.some-bean.first-name=Dustin

# someBean has setLastName method
config.some-bean.last-name=Schultz
```

Relaxed Configuration Names

Camel Case

`featureEnabled`

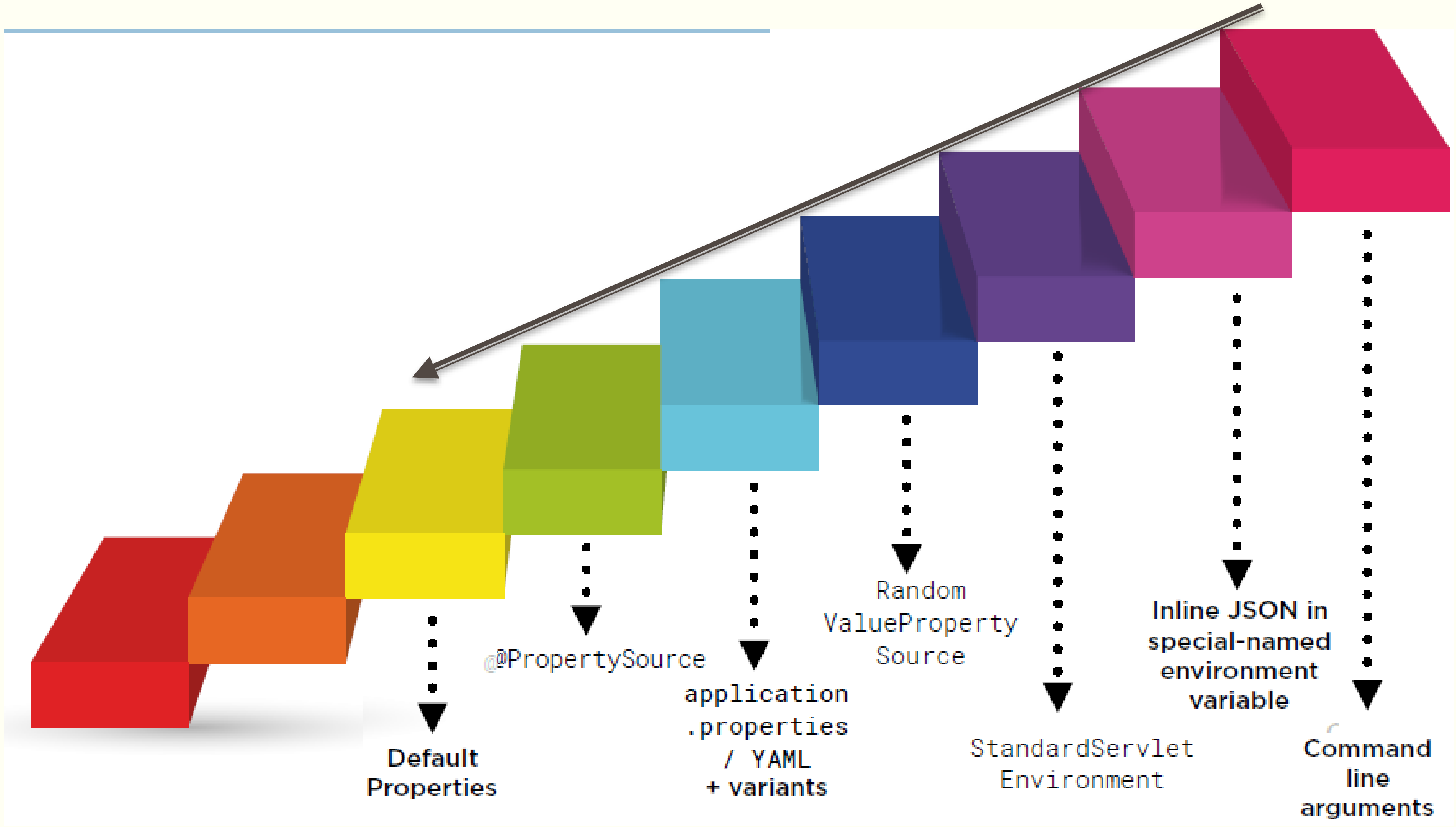
Dash Notation

`feature-enabled`

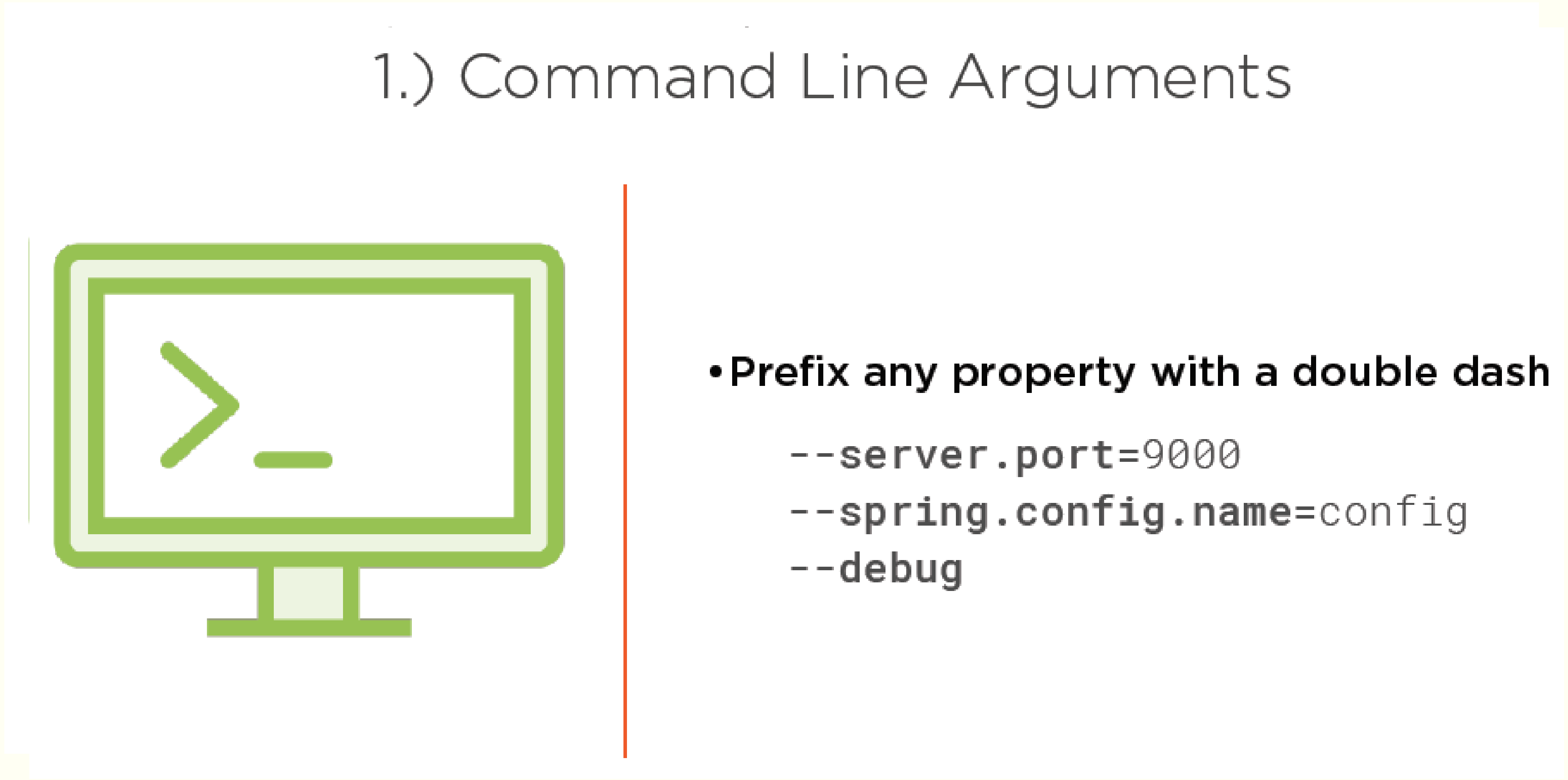
Underscore

`PREFIX_FEATURE_ENABLED`

Resolving Configuration in Spring Boot


Default Properties
@PropertySource
application .properties / YAML + variants
Random ValueProperty Source
StandardServlet Environment
Inline JSON in special-named environment variable
Command line arguments

Resolving Configuration in Spring Boot



1.) Command Line Arguments

- Prefix any property with a double dash

```
--server.port=9000
--spring.config.name=config
--debug
```

Resolving Configuration in Spring Boot

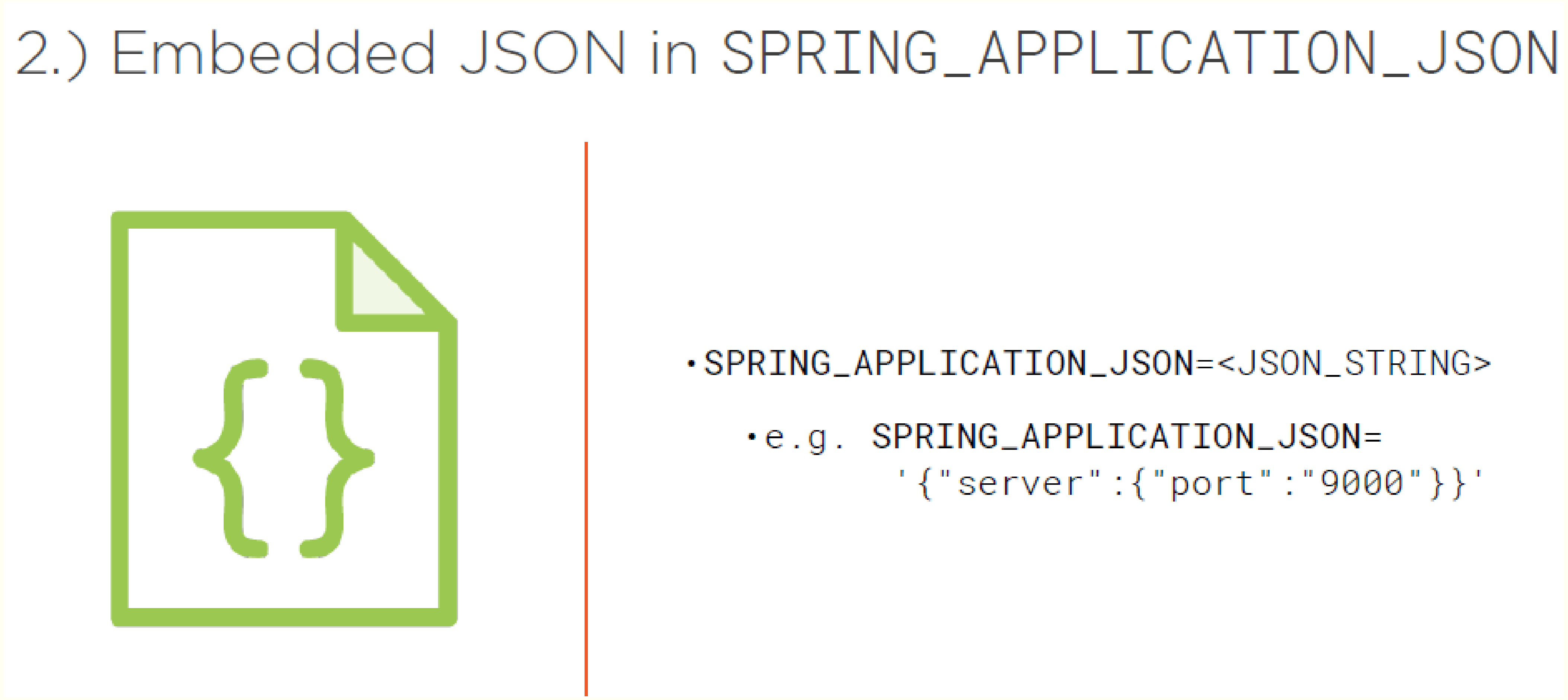
```
@Value("${name}")
private String name;
```

```
$ java -jar yourapp.jar --name=Dave
```

You can also configure many aspects of Spring Boot itself:

```
$ java -jar target/*.jar --server.port=9000
```

Resolving Configuration in Spring Boot



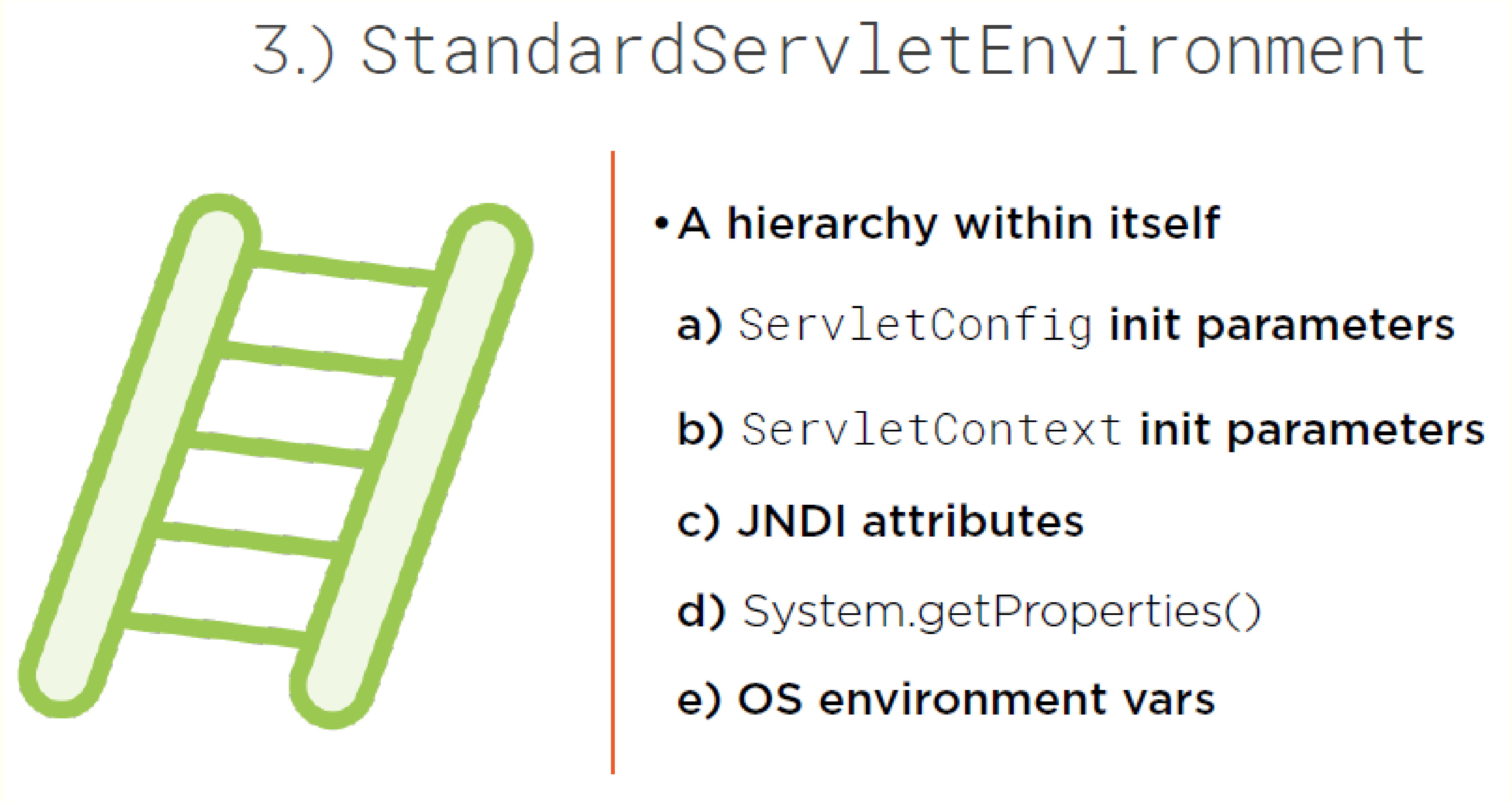
2.) Embedded JSON in SPRING_APPLICATION_JSON

- SPRING_APPLICATION_JSON=<JSON_STRING>
  - e.g. SPRING_APPLICATION_JSON= '{"server":{"port":"9000"}}'

Resolving Configuration in Spring Boot

- The SPRING_APPLICATION_JSON properties can be supplied on the command line with an environment variable:
  - $ SPRING_APPLICATION_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar
  - In this example you will end up with foo.bar=spam in the Spring Environment.
- You can also supply the JSON as spring.application.json in a System variable:
  - $ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar
- or command line argument:
  - $ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'
- or as a JNDI variable
  - java:comp/env/spring.application.json.

Resolving Configuration in Spring Boot



3.) StandardServletEnvironment

- A hierarchy within itself

a) `ServletConfig` **init parameters**

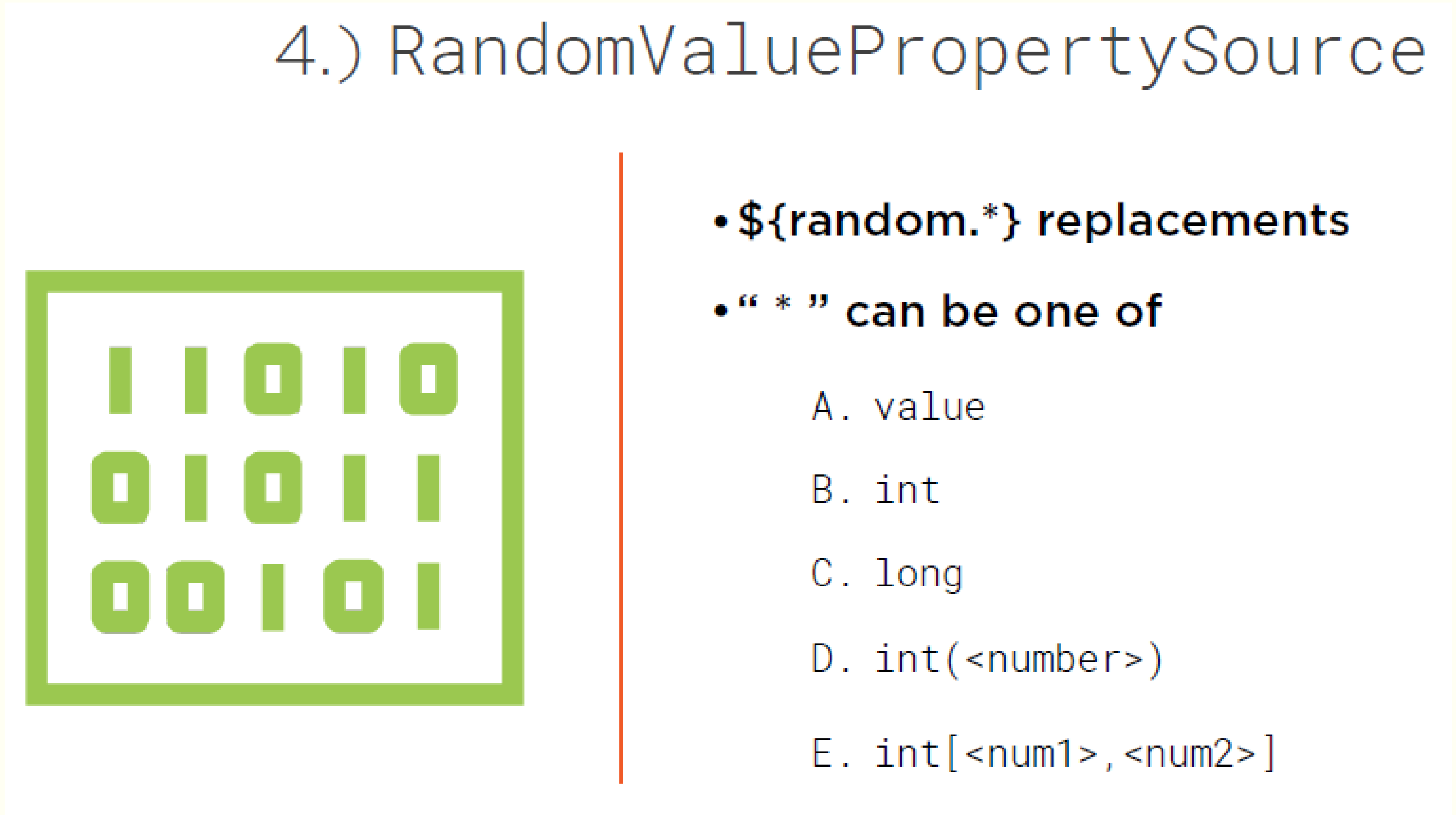
b) `ServletContext` **init parameters**

c) **JNDI attributes**

d) System.getProperties()

e) **OS environment vars**

Resolving Configuration in Spring Boot



4.) RandomValuePropertySource

- ${random.*} replacements
- " * " can be one of

A. value

B. int

C. long

D. int(<number>)

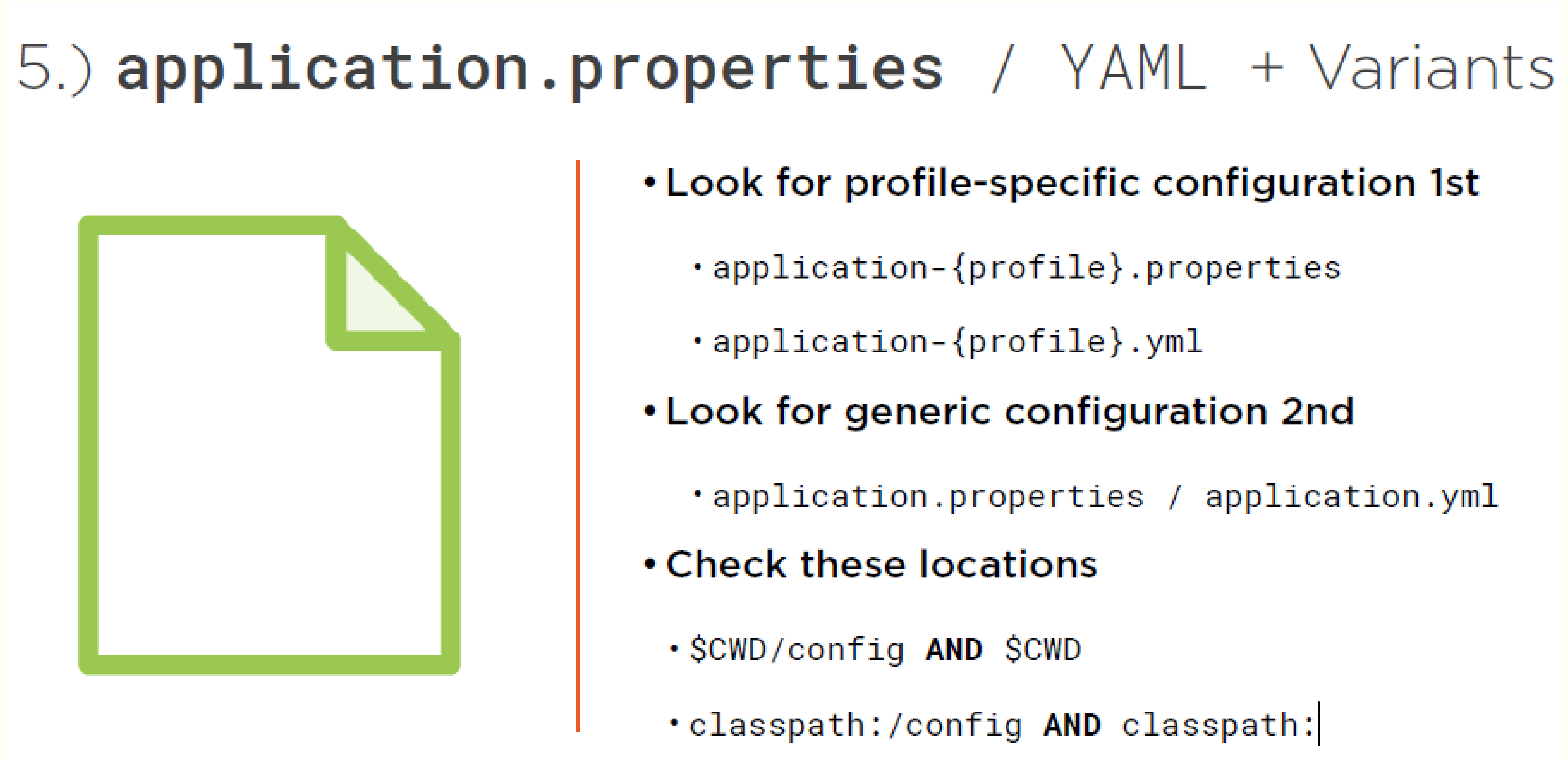
E. int[<num1>,<num2>]

Resolving Configuration in Spring Boot

RandomValuePropertySource

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

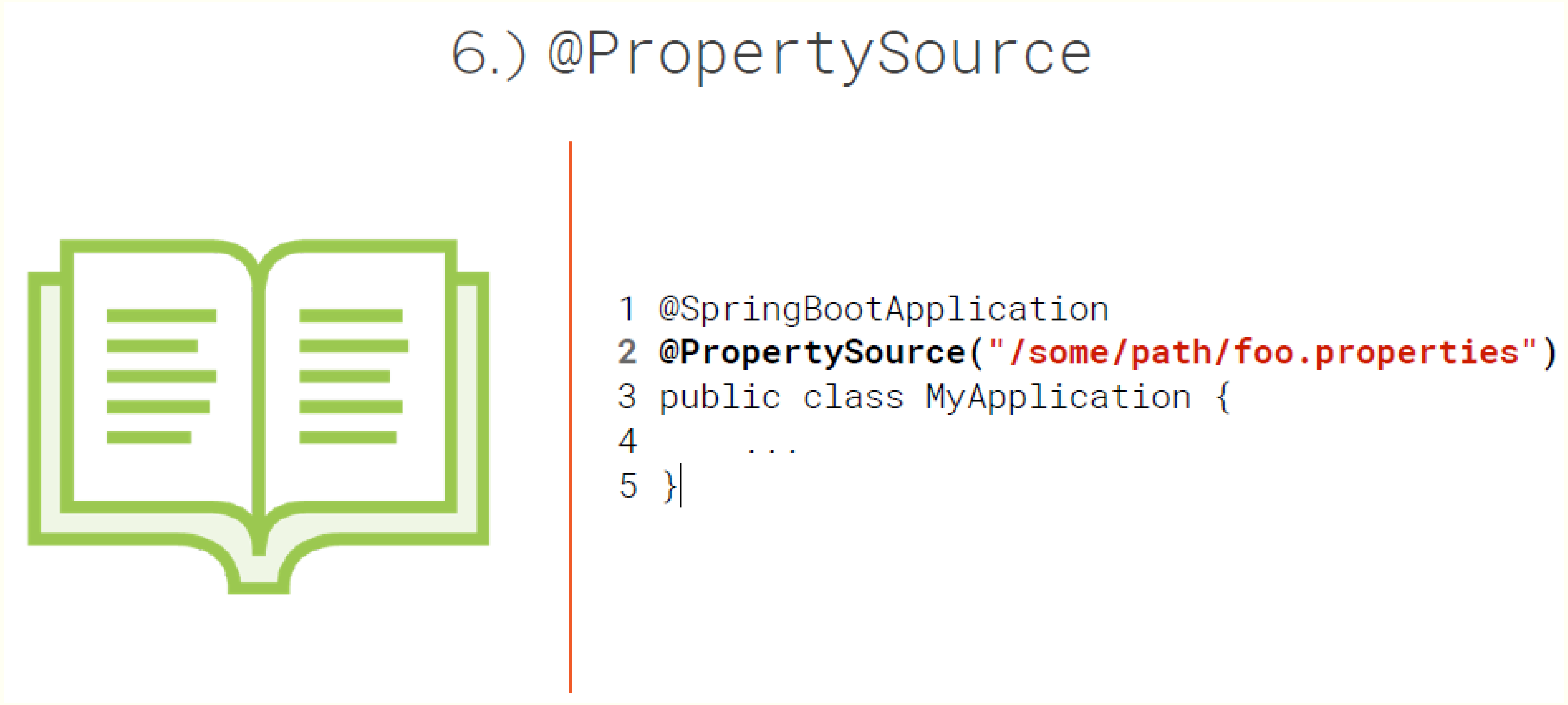
Resolving Configuration in Spring Boot



5.) **application.properties** / YAML + Variants

- **Look for profile-specific configuration 1st**
  - `application-{profile}.properties`
  - `application-{profile}.yml`
- **Look for generic configuration 2nd**
  - `application.properties / application.yml`
- **Check these locations**
  - `$CWD/config` **AND** `$CWD`
  - `classpath:/config` **AND** `classpath:`

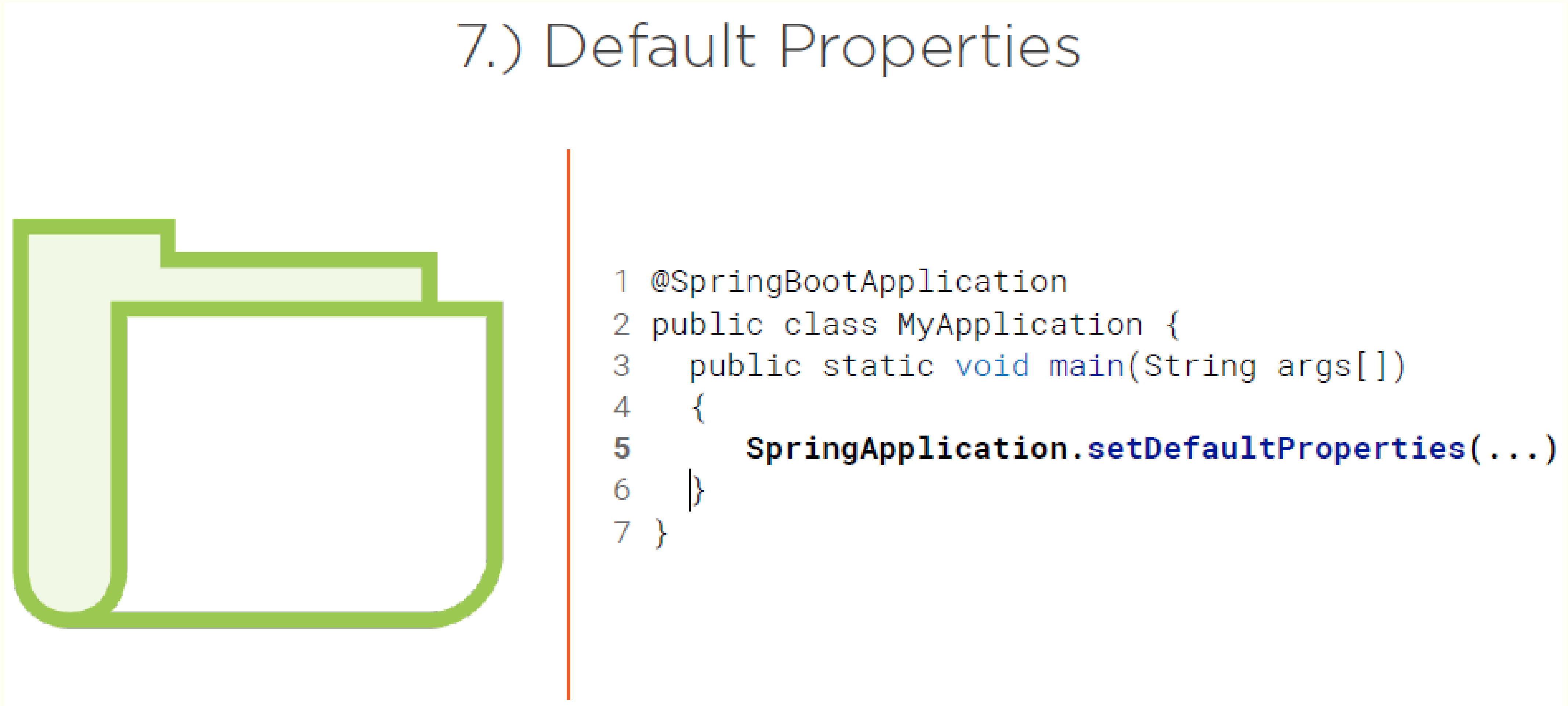
Resolving Configuration in Spring Boot



6.) @PropertySource

```
@SpringBootApplication
@PropertySource("/some/path/foo.properties")
public class MyApplication {
    ...
}
```

Resolving Configuration in Spring Boot

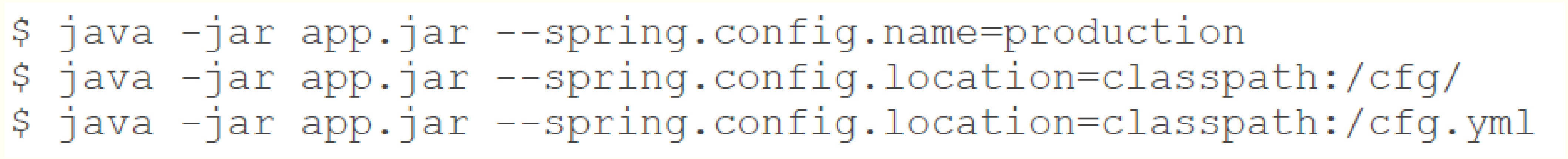


7.) Default Properties

```
@SpringBootApplication
public class MyApplication {
  public static void main(String args[])
  {
      SpringApplication.setDefaultProperties(...)
  }
}
```

Customizing Configuration Location

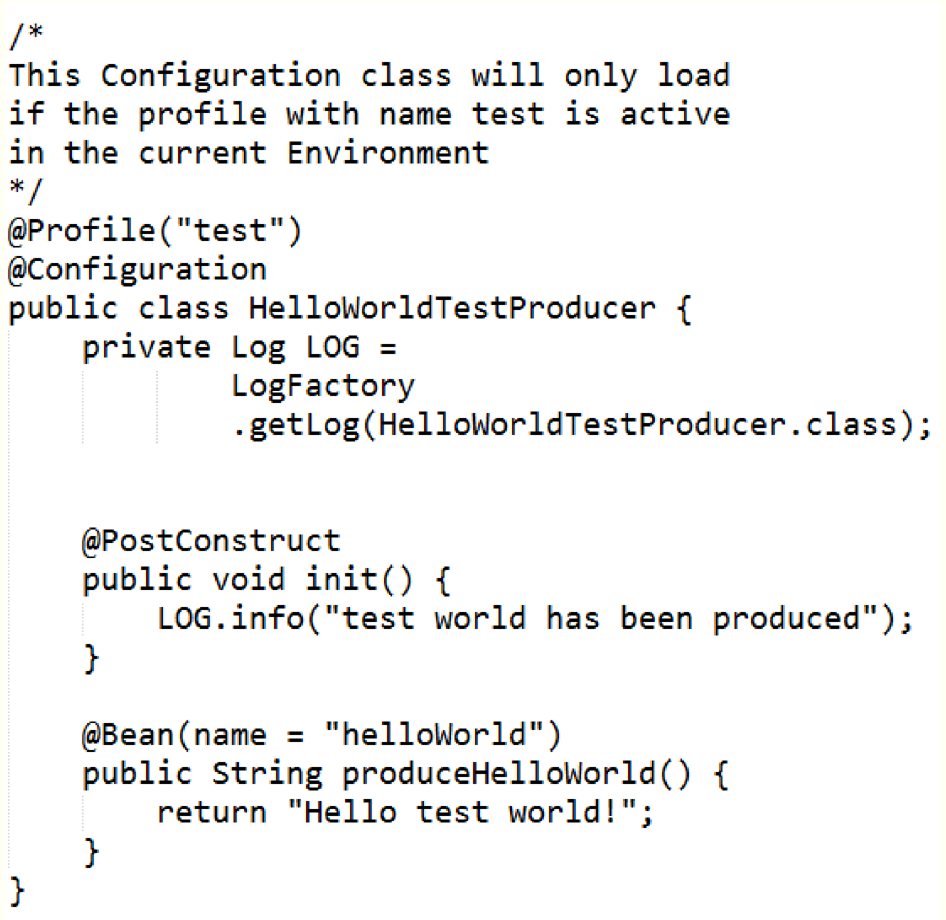
- spring.config.name - default **application**, can be comma-separated list
- spring.config.location - a Resource path
- Ends with / to define a directory
- Otherwise overrides name



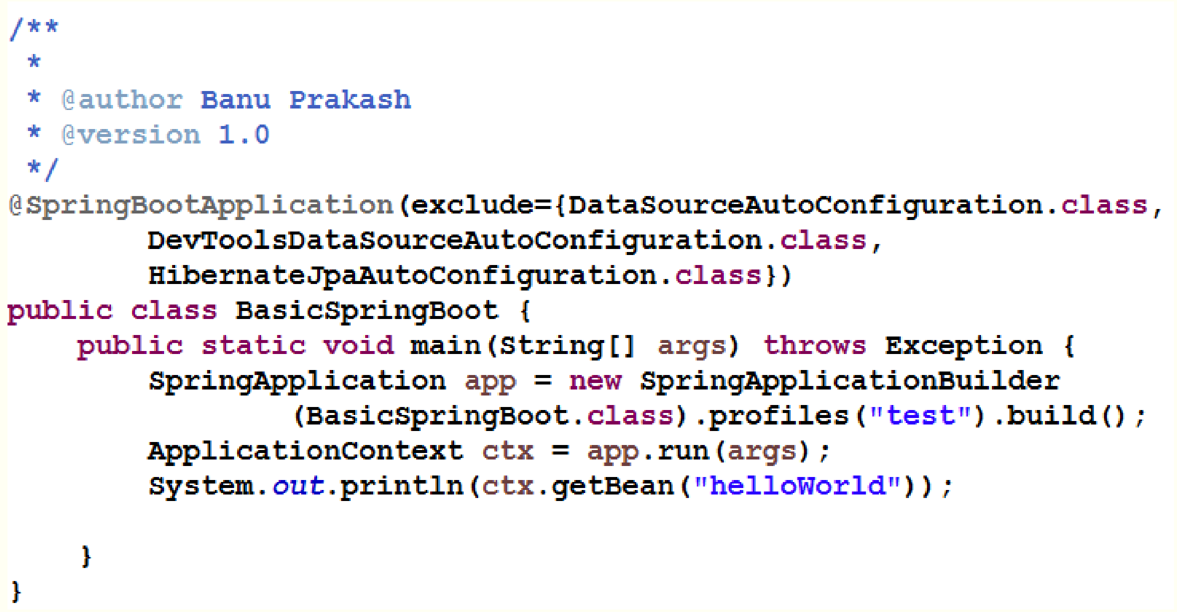
```
$ java -jar app.jar --spring.config.name=production
$ java -jar app.jar --spring.config.location=classpath:/cfg/
$ java -jar app.jar --spring.config.location=classpath:/cfg.yml
```

Spring Profiles

- **Using the @Profile annotation**



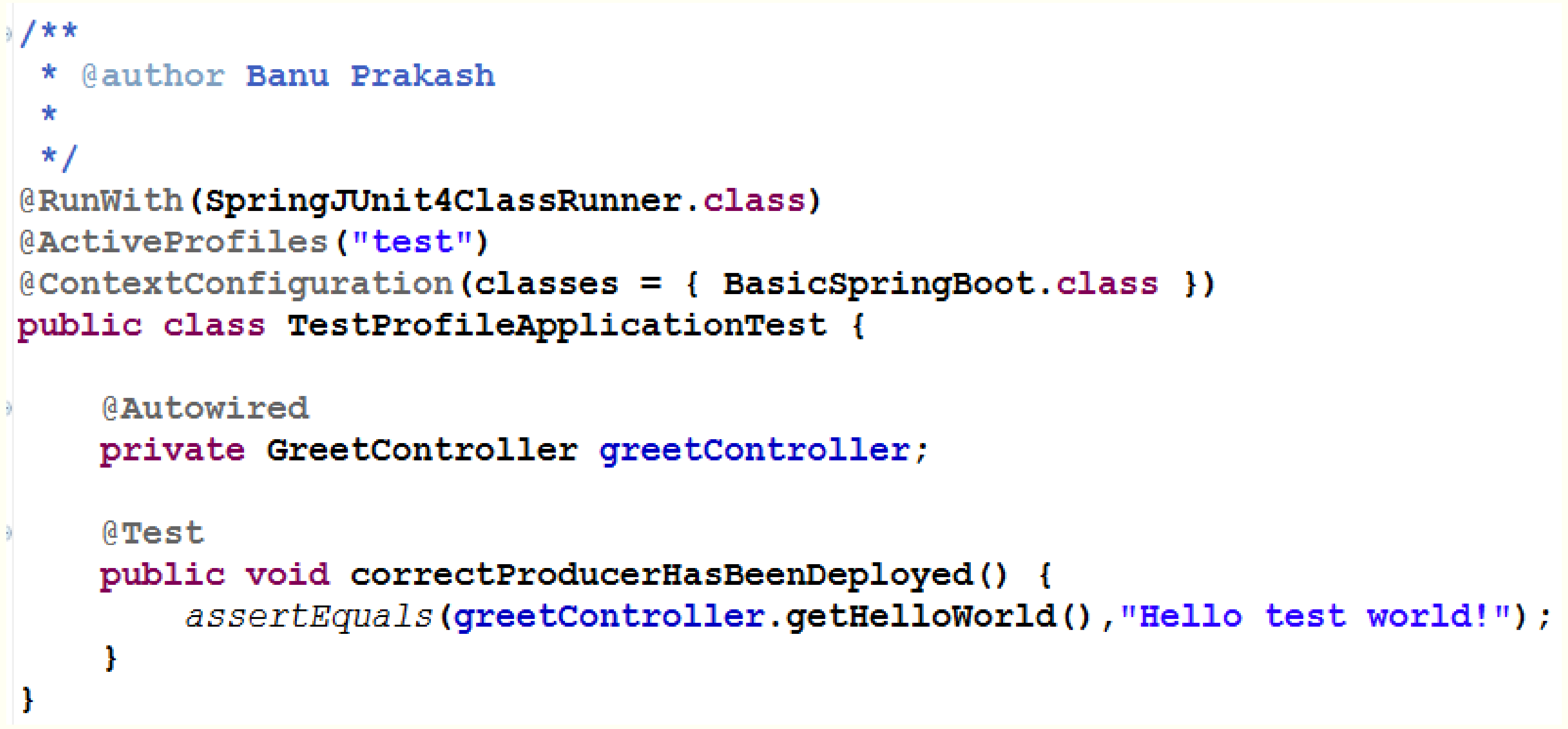
```
/*
This Configuration class will only load
if the profile with name test is active
in the current Environment
*/
@Profile("test")
@Configuration
public class HelloWorldTestProducer {
    private Log LOG =
            LogFactory
            .getLog(HelloWorldTestProducer.class);


    @PostConstruct
    public void init() {
        LOG.info("test world has been produced");
    }

    @Bean(name = "helloWorld")
    public String produceHelloWorld() {
        return "Hello test world!";
    }
}
```



```
/**
 *
 * @author Banu Prakash
 * @version 1.0
 */
@SpringBootApplication(exclude={DataSourceAutoConfiguration.class,
        DevToolsDataSourceAutoConfiguration.class,
        HibernateJpaAutoConfiguration.class})
public class BasicSpringBoot {
    public static void main(String[] args) throws Exception {
        SpringApplication app = new SpringApplicationBuilder
                (BasicSpringBoot.class).profiles("test").build();
        ApplicationContext ctx = app.run(args);
        System.out.println(ctx.getBean("helloWorld"));

    }
}
```

Spring Profiles

- How can we make sure that, when we activate the test-profile, our production profile isn't loaded?
- **@ActiveProfiles** will make sure that the provided profiles will be activated when tests are run



```
/**
 * @author Banu Prakash
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles("test")
@ContextConfiguration(classes = { BasicSpringBoot.class })
public class TestProfileApplicationTest {

    @Autowired
    private GreetController greetController;

    @Test
    public void correctProducerHasBeenDeployed() {
        assertEquals(greetController.getHelloWorld(),"Hello test world!");
    }
}
```

Spring Profiles

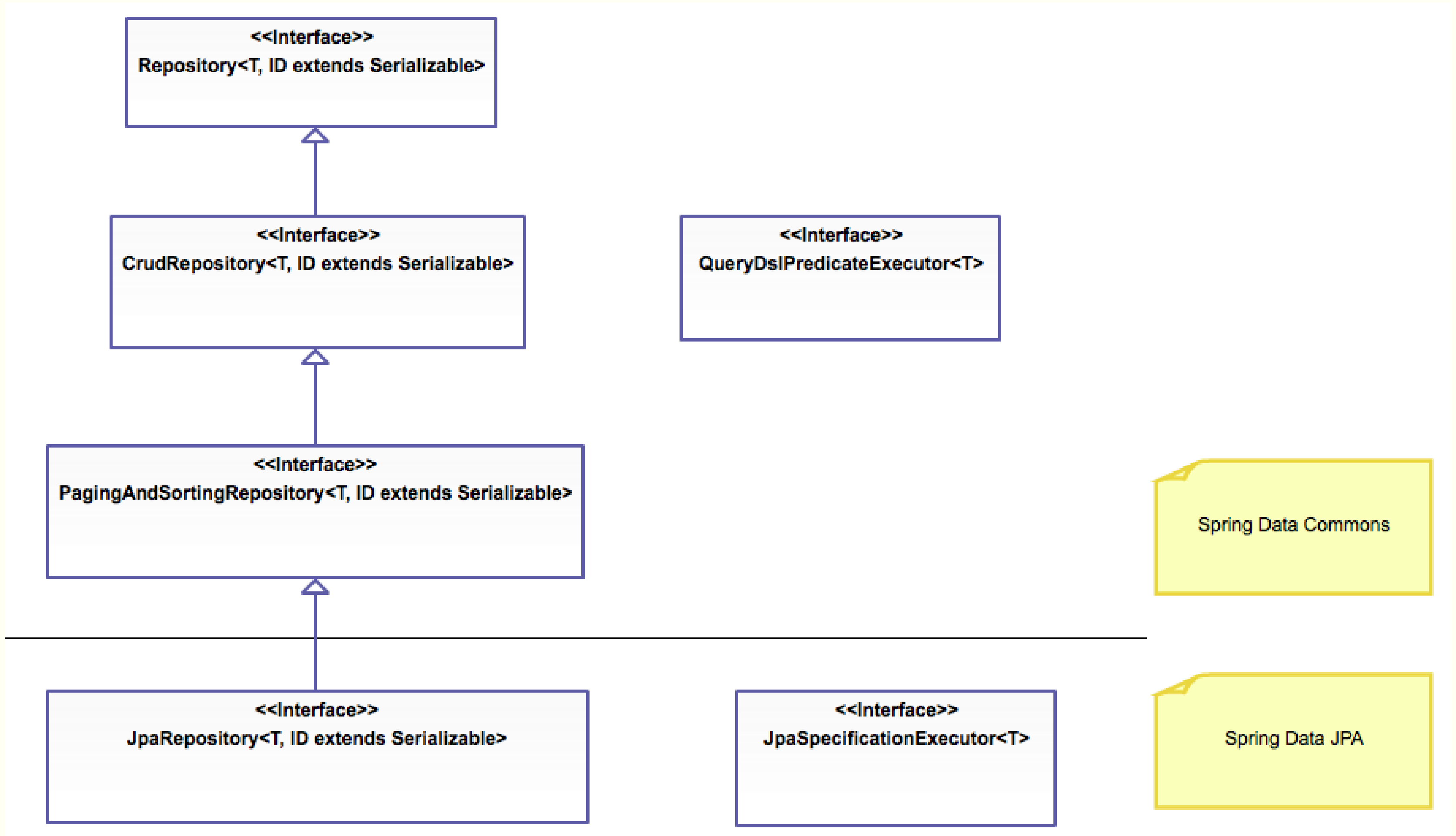
- **Using application.properties**
  - If you're using the application.properties file, you can add additional profiles by adding the following key to the file.
    spring.profiles.active=profile1, profile2
- **Using jvm-arguments**
  - **java -jar myapp.jar --spring.profiles.active=profile1,profile2**

Spring Data JPA - Simplifying persistence - even more!

- **JPA**
  - **THE STANDARD ORM SOLUTION FOR JAVA**
  - Hibernate is used underneath
  - Very complete, impressive set of features

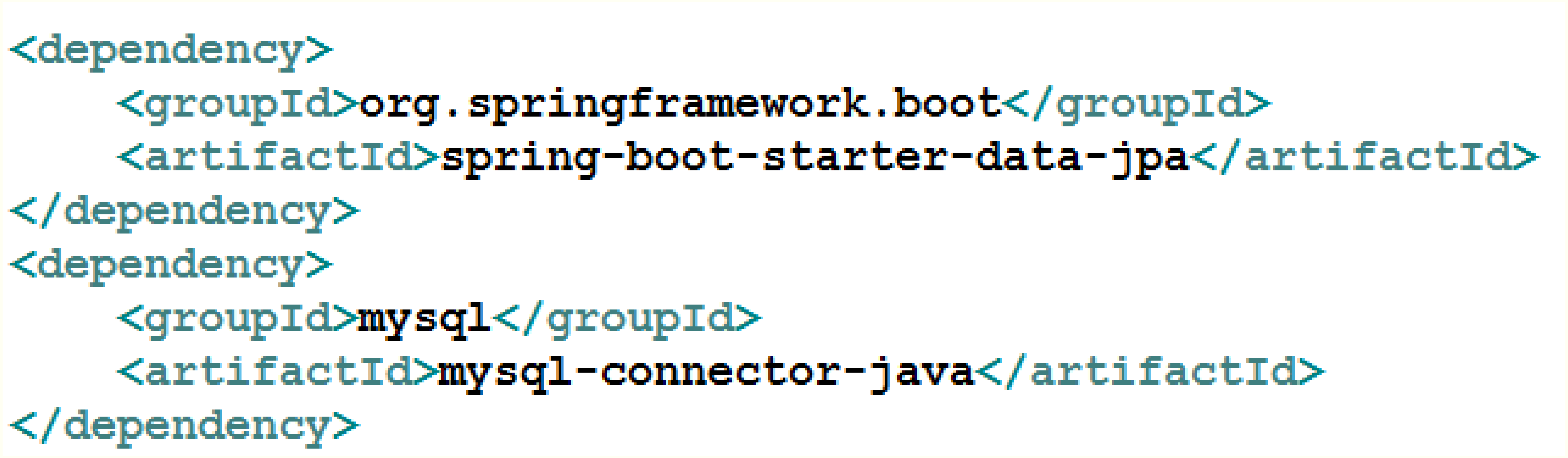
- **SPRING DATA JPA**
  - **ADDS EXTRA SYNTAXIC SUGAR ON TOP OF JPA**
  - Generates your JPA repositories *automatically*
  - Removes a lot of boilerplate code

Spring Data Repositories



<<Interface>>
Repository<T, ID extends Serializable>
<<Interface>>
CrudRepository<T, ID extends Serializable>
<<Interface>>
QueryDslPredicateExecutor<T>
<<Interface>>
PagingAndSortingRepository<T, ID extends Serializable>
Spring Data Commons
<<Interface>>
JpaRepository<T, ID extends Serializable>
<<Interface>>
JpaSpecificationExecutor<T>
Spring Data JPA

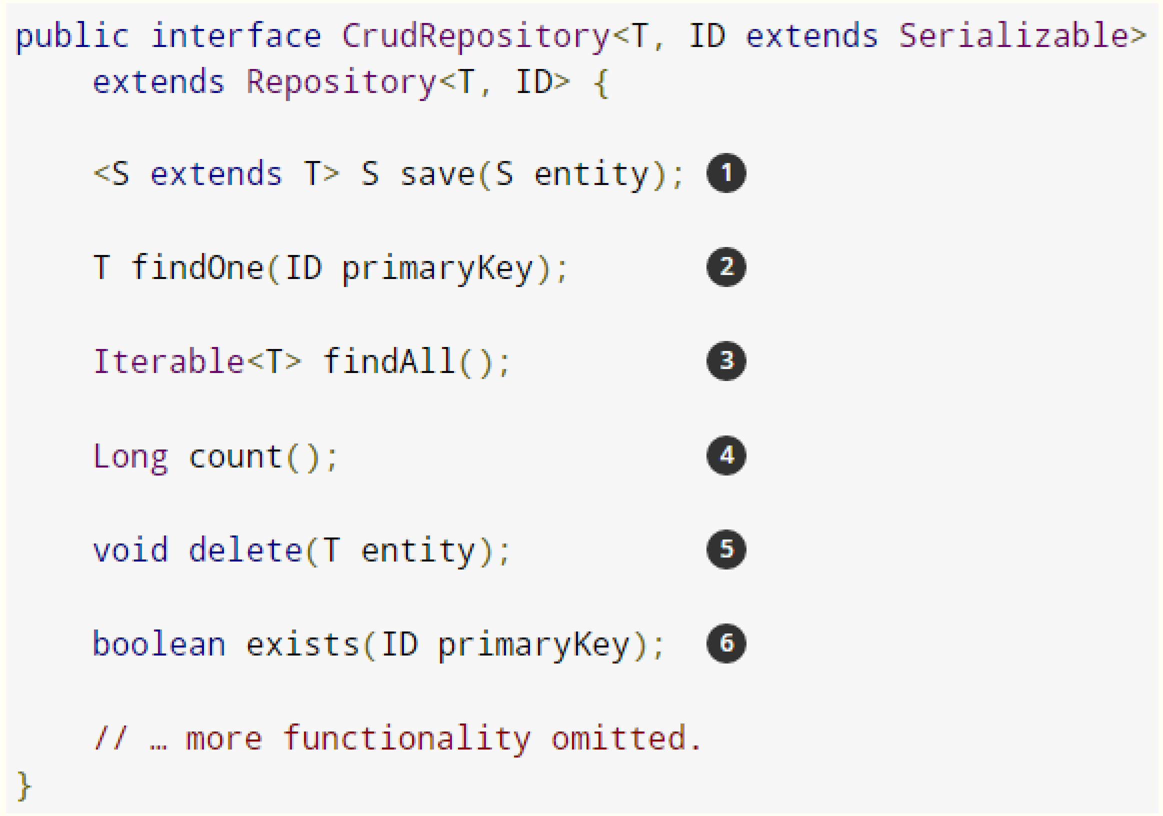
Declaring a dependency to a Spring Data module



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

CrudRepository

- The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed:
    1. Saves the given entity.
    2. Returns the entity identified by the given id.
    3. Returns all entities.
    4. Returns the number of entities.
    5. Deletes the given entity.
    6. Indicates whether an entity with the given id exists.

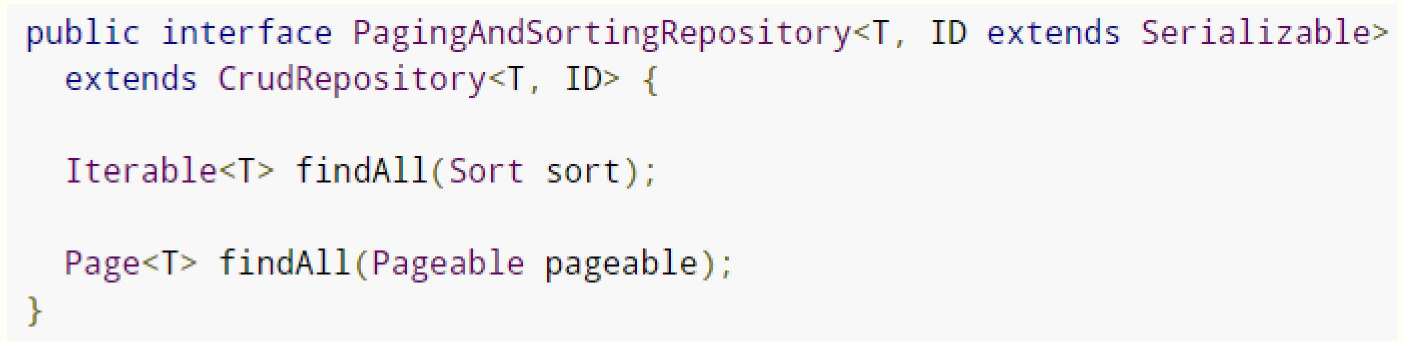


```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);      ❶

    T findOne(ID primaryKey);            ❷

    Iterable<T> findAll();               ❸

    Long count();                        ❹

    void delete(T entity);               ❺

    boolean exists(ID primaryKey);       ❻

    // … more functionality omitted.
}
```

JpaRepository, MongoRepository and PagingAndSortingRepository

- These interfaces extend CrudRepository and expose the capabilities of the underlying persistence technology in addition to the rather generic persistence technology-agnostic interfaces like e.g. CrudRepository.

- PagingAndSortingRepository



```java
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

- Accessing the second page of User by a page size of 20 you could simply do something like this:
  - PagingAndSortingRepository<User, Long> repository = // … get access to a bean
  - Page<User> users = repository.findAll(new PageRequest(1, 20));

Query creation

- The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository.
- The mechanism strips the prefixes find…By, read…By, query…By, count…By, and get…By from the method and starts parsing the rest of it.
- The introducing clause can contain further expressions such as a Distinct to set a distinct flag on the query to be created.
- However, the first By acts as delimiter to indicate the start of the actual criteria.
- At a very basic level you can define conditions on entity properties and concatenate them with And and Or.

Query creation example

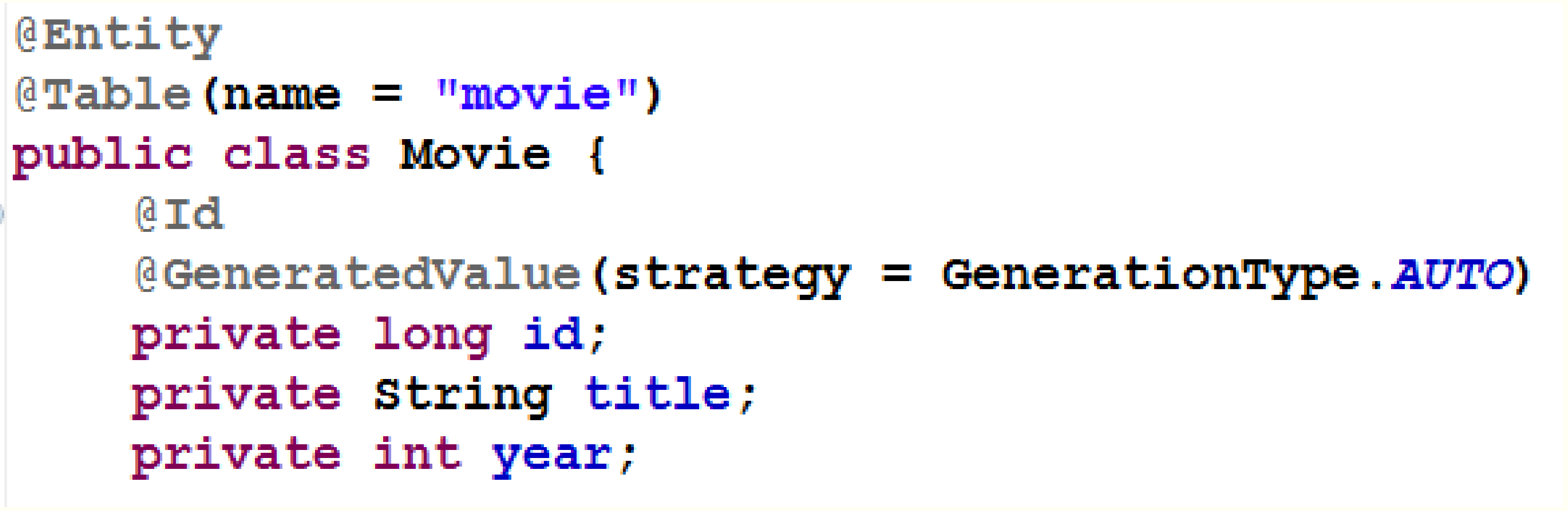
```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property
List<Person> findByLastnameIgnoreCase(String lastname);
// Enabling ignoring case for all suitable properties
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

Spring Data JPA Example

```
create database spring_movies;
use spring_movies;
 create table `movie`(
        `id` int(3) NOT NULL AUTO_INCREMENT,
         `title` VARCHAR(100) NOT NULL,
         `year` int (4),
         primary key (`id`)
    );
```

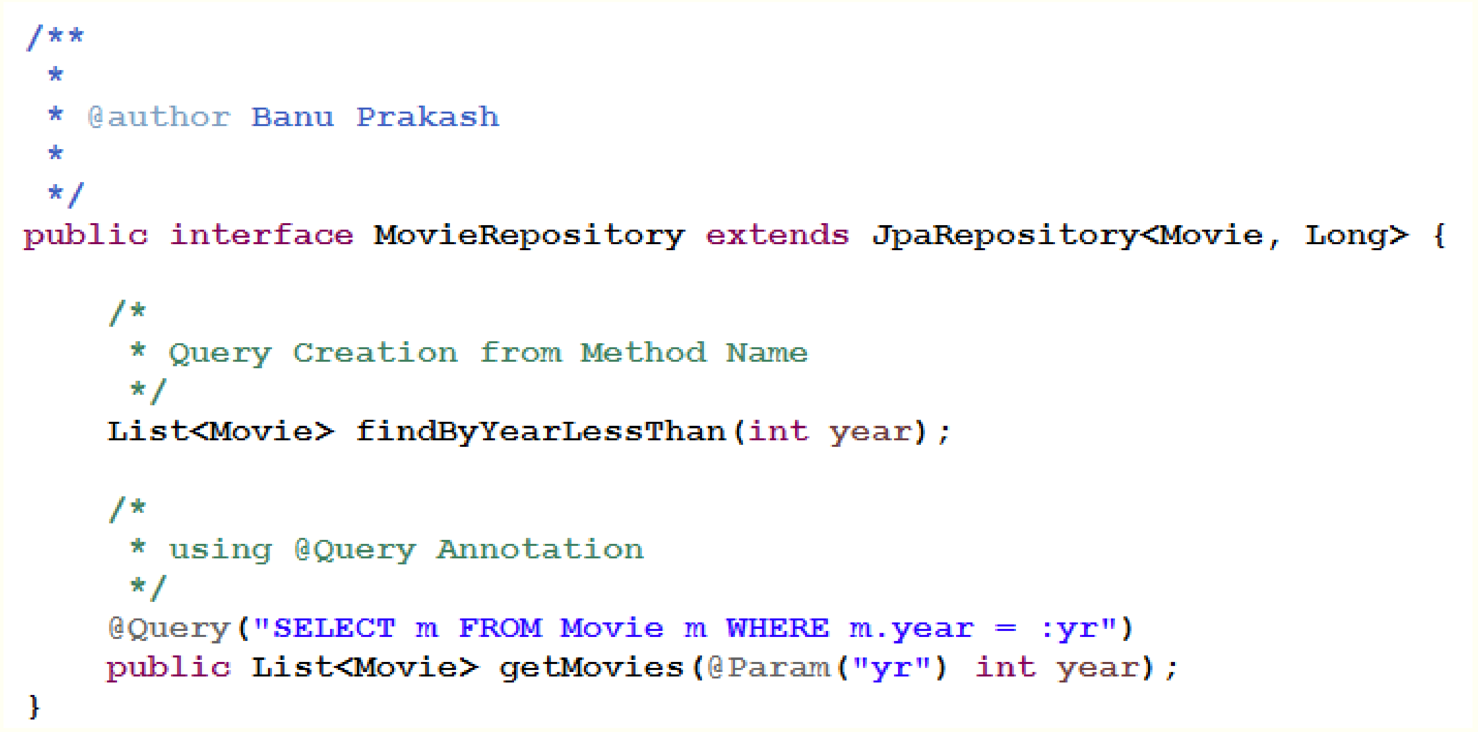
Entity class

- Movie class



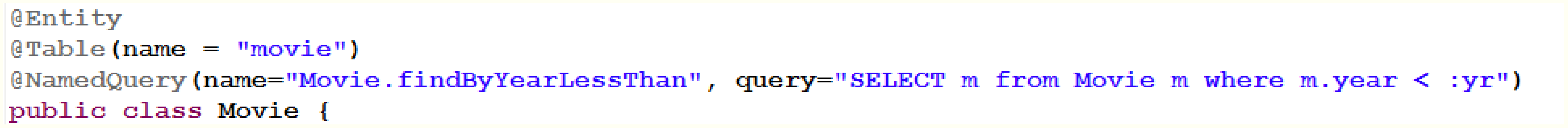
```
@Entity
@Table(name = "movie")
public class Movie {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private String title;
    private int year;
```

Repository



```
/**
 *
 * @author Banu Prakash
 *
 */
public interface MovieRepository extends JpaRepository<Movie, Long> {

    /*
     * Query Creation from Method Name
     */
    List<Movie> findByYearLessThan(int year);

    /*
     * using @Query Annotation
     */
    @Query("SELECT m FROM Movie m WHERE m.year = :yr")
    public List<Movie> getMovies(@Param("yr") int year);
}
```

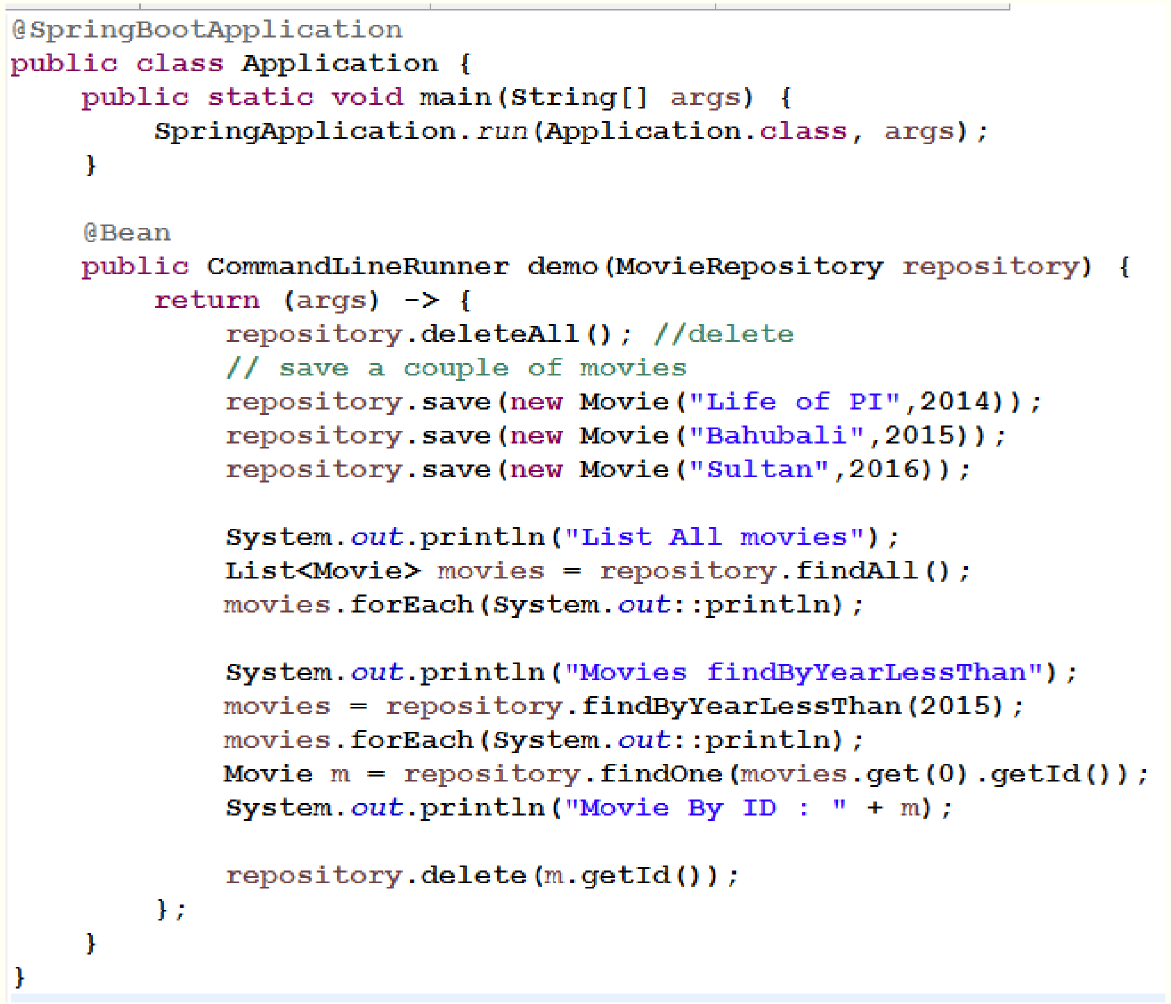
- You can also map the repository method to Named Query



```
@Entity
@Table(name = "movie")
@NamedQuery(name="Movie.findByYearLessThan", query="SELECT m from Movie m where m.year < :yr")
public class Movie {
```

CommandLineRunner

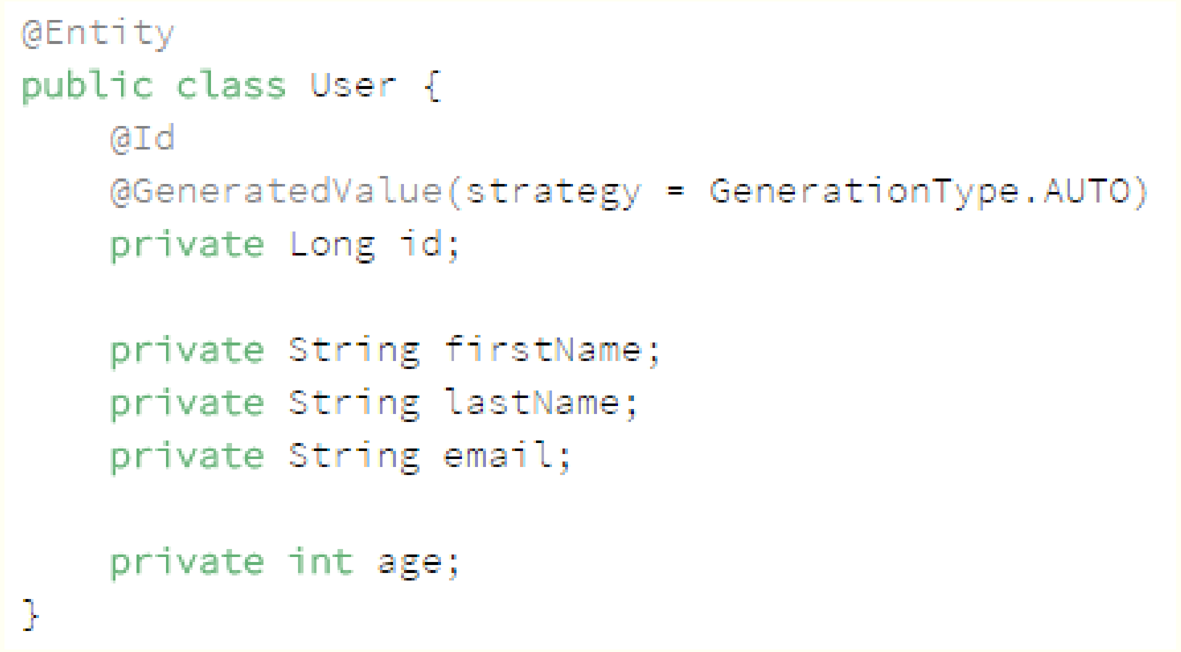
- Test code



```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public CommandLineRunner demo(MovieRepository repository) {
        return (args) -> {
            repository.deleteAll(); //delete
            // save a couple of movies
            repository.save(new Movie("Life of PI",2014));
            repository.save(new Movie("Bahubali",2015));
            repository.save(new Movie("Sultan",2016));

            System.out.println("List All movies");
            List<Movie> movies = repository.findAll();
            movies.forEach(System.out::println);

            System.out.println("Movies findByYearLessThan");
            movies = repository.findByYearLessThan(2015);
            movies.forEach(System.out::println);
            Movie m = repository.findOne(movies.get(0).getId());
            System.out.println("Movie By ID : " + m);

            repository.delete(m.getId());
        };
    }
}
```

Creating Database Queries With the JPA Criteria API

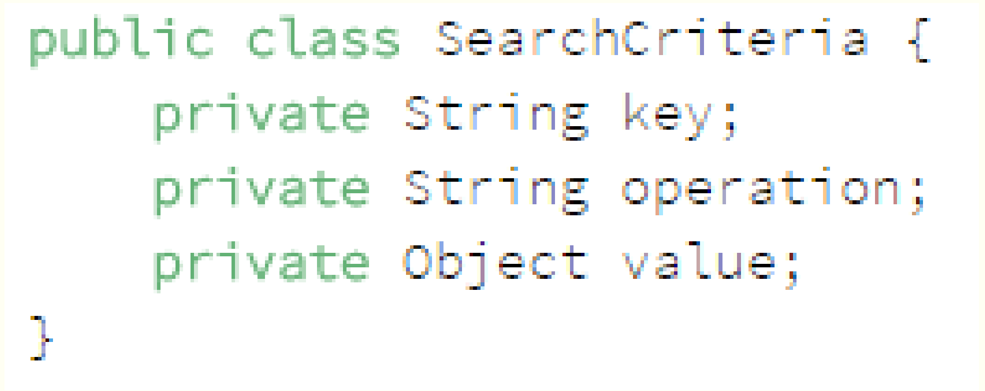
- *User* Entity



```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String firstName;
    private String lastName;
    private String email;

    private int age;
}
```

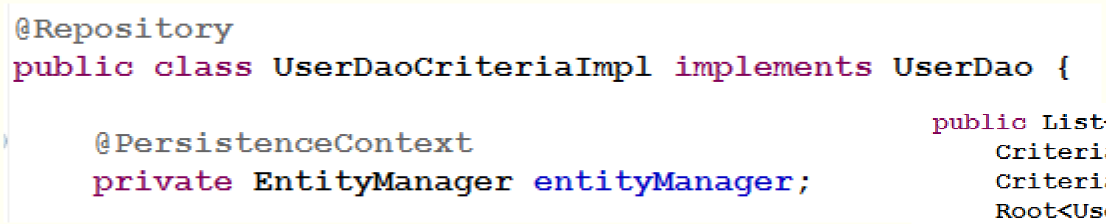
The Constraint class

- The SearchCriteria implementation holds our Query parameters:
  - key: used to hold field name – for example: firstName, age, … etc.
  - operation: used to hold the operation – for example: Equality, less than, … etc.
  - value: used to hold the field value – for example: Ganesh, 24, … etc.

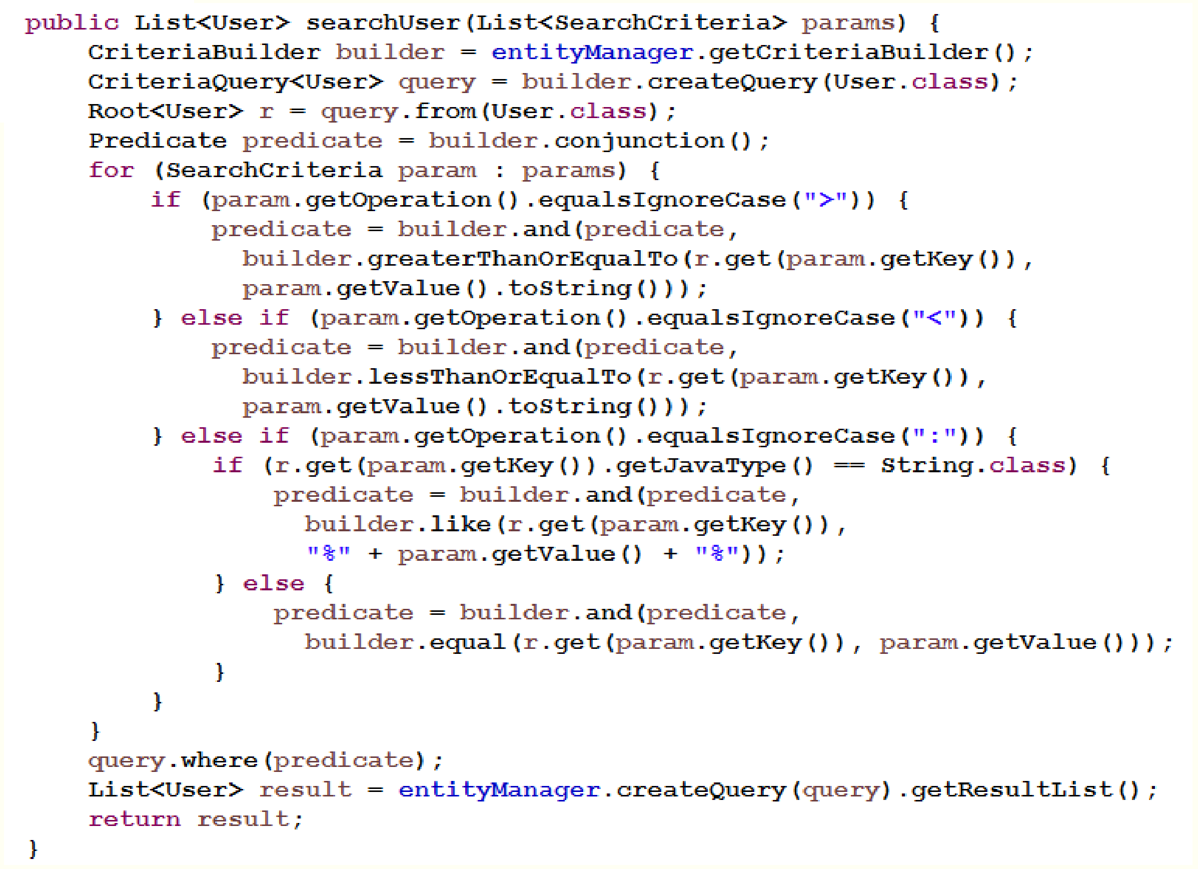


```
public class SearchCriteria {
    private String key;
    private String operation;
    private Object value;
}
```

CriteriaBuilder



```
@Repository
public class UserDaoCriteriaImpl implements UserDao {

    @PersistenceContext
    private EntityManager entityManager;
```



```
public List<User> searchUser(List<SearchCriteria> params) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<User> query = builder.createQuery(User.class);
    Root<User> r = query.from(User.class);
    Predicate predicate = builder.conjunction();
    for (SearchCriteria param : params) {
        if (param.getOperation().equalsIgnoreCase(">")) {
            predicate = builder.and(predicate,
              builder.greaterThanOrEqualTo(r.get(param.getKey()),
              param.getValue().toString()));
        } else if (param.getOperation().equalsIgnoreCase("<")) {
            predicate = builder.and(predicate,
              builder.lessThanOrEqualTo(r.get(param.getKey()),
              param.getValue().toString()));
        } else if (param.getOperation().equalsIgnoreCase(":")) {
            if (r.get(param.getKey()).getJavaType() == String.class) {
                predicate = builder.and(predicate,
                  builder.like(r.get(param.getKey()),
                  "%" + param.getValue() + "%"));
            } else {
                predicate = builder.and(predicate,
                  builder.equal(r.get(param.getKey()), param.getValue()));
            }
        }
    }
    query.where(predicate);
    List<User> result = entityManager.createQuery(query).getResultList();
    return result;
}
```

Test the Search Queries

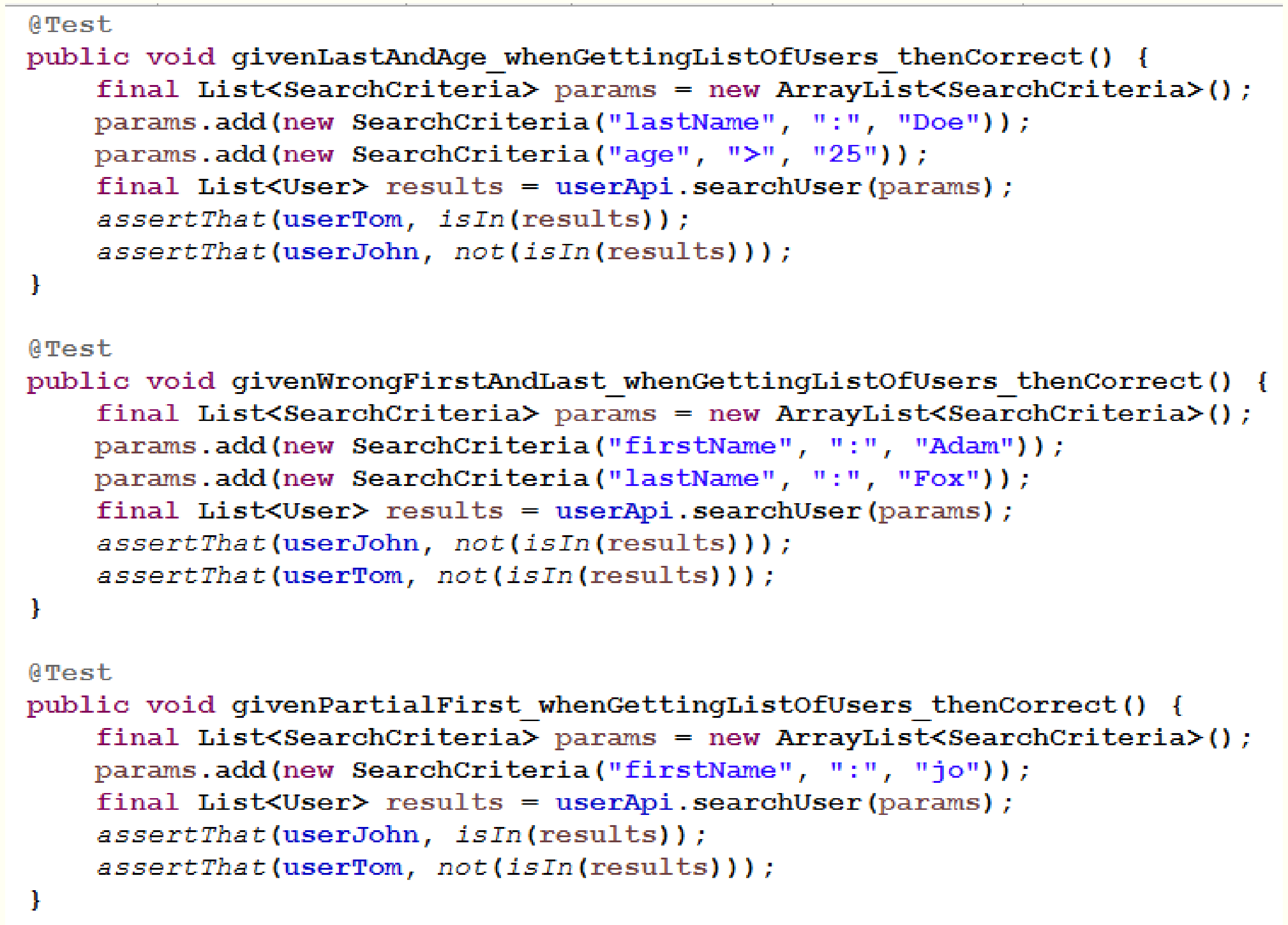
```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { PersistenceConfig.class })
@Transactional
public class JPACriteriaQueryTest {

    @Autowired
    private UserDao userApi;
    private User userJohn;

    private User userTom;

    @Before
    public void init() {
        userJohn = new User();
        userJohn.setFirstName("");
        userJohn.setLastName("Doe");
        userJohn.setEmail("john@doe.com");
        userJohn.setAge(22);
        userApi.save(userJohn);

        userTom = new User();
        userTom.setFirstName("Tom");
        userTom.setLastName("Doe");
        userTom.setEmail("tom@doe.com");
        userTom.setAge(26);
        userApi.save(userTom);
    }
```

```java
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.collection.IsIn.isIn;
import static org.hamcrest.core.IsNot.not;
```

Test the Search Queries

```
@Test
public void givenFirstAndLastName_whenGettingListOfUsers_thenCorrect() {
    final List<SearchCriteria> params = new ArrayList<SearchCriteria>();
    params.add(new SearchCriteria("firstName", ":", "John"));
    params.add(new SearchCriteria("lastName", ":", "Doe"));
    final List<User> results = userApi.searchUser(params);
    assertThat(userJohn, isIn(results));
    assertThat(userTom, not(isIn(results)));
}

@Test
public void givenLast_whenGettingListOfUsers_thenCorrect() {
    final List<SearchCriteria> params = new ArrayList<SearchCriteria>();
    params.add(new SearchCriteria("lastName", ":", "Doe"));
    final List<User> results = userApi.searchUser(params);
    assertThat(userJohn, isIn(results));
    assertThat(userTom, isIn(results));
}

@Test
public void givenLastAndAge_whenGettingListOfUsers_thenCorrect() {
    final List<SearchCriteria> params = new ArrayList<SearchCriteria>();
    params.add(new SearchCriteria("lastName", ":", "Doe"));
    params.add(new SearchCriteria("age", ">", "25"));
    final List<User> results = userApi.searchUser(params);
    assertThat(userTom, isIn(results));
    assertThat(userJohn, not(isIn(results)));
}
```

Test the Search Queries



```
@Test
public void givenLastAndAge_whenGettingListOfUsers_thenCorrect() {
    final List<SearchCriteria> params = new ArrayList<SearchCriteria>();
    params.add(new SearchCriteria("lastName", ":", "Doe"));
    params.add(new SearchCriteria("age", ">", "25"));
    final List<User> results = userApi.searchUser(params);
    assertThat(userTom, isIn(results));
    assertThat(userJohn, not(isIn(results)));
}

@Test
public void givenWrongFirstAndLast_whenGettingListOfUsers_thenCorrect() {
    final List<SearchCriteria> params = new ArrayList<SearchCriteria>();
    params.add(new SearchCriteria("firstName", ":", "Adam"));
    params.add(new SearchCriteria("lastName", ":", "Fox"));
    final List<User> results = userApi.searchUser(params);
    assertThat(userJohn, not(isIn(results)));
    assertThat(userTom, not(isIn(results)));
}

@Test
public void givenPartialFirst_whenGettingListOfUsers_thenCorrect() {
    final List<SearchCriteria> params = new ArrayList<SearchCriteria>();
    params.add(new SearchCriteria("firstName", ":", "jo"));
    final List<User> results = userApi.searchUser(params);
    assertThat(userJohn, isIn(results));
    assertThat(userTom, not(isIn(results)));
}
```

Domain Driven Design

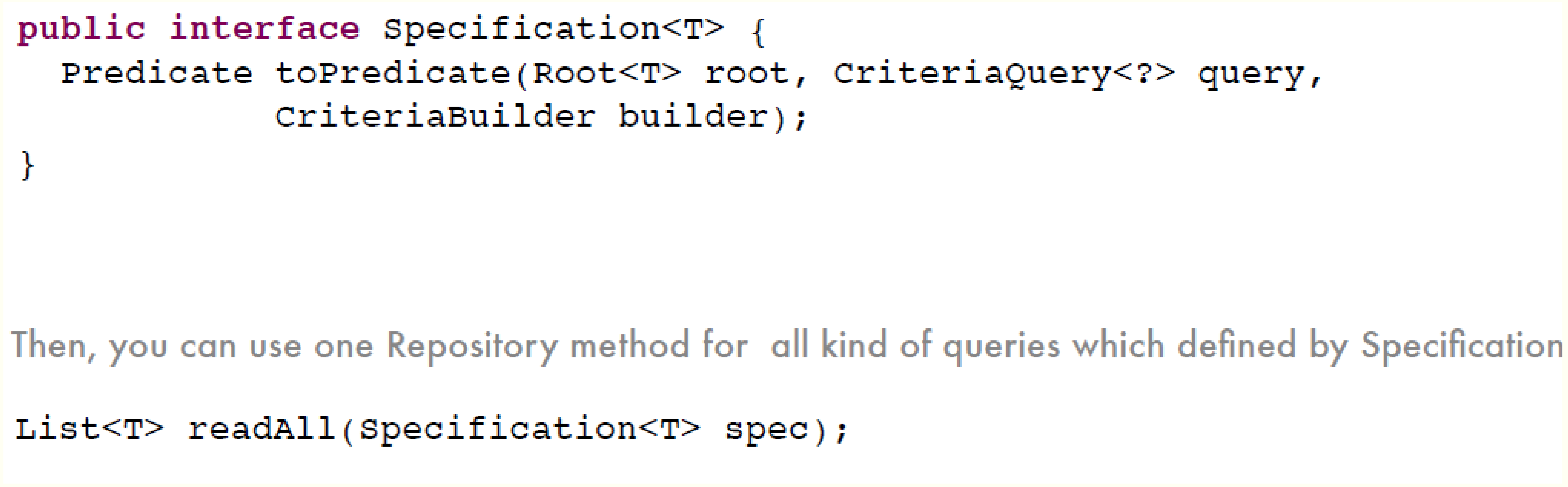
- DDD main approaches
  - Entity
  - Value Object
  - Aggregate
  - Repository

The Specification

- **Problem:** Business rules often do not fit the responsibility of any of the obvious Entities or Value Objects, and their variety and combinations can overwhelm the basic meaning of the domain object.
- But moving the rules out of the domain layer is even worse, since the domain code no longer expresses the model.

- **Solution:** Create explicit predicate-like Value Objects for specialized purposes.
- A Specification is a predicate that determines if an object does or does not satisfy some criteria.

Spring Data JPA Specification API

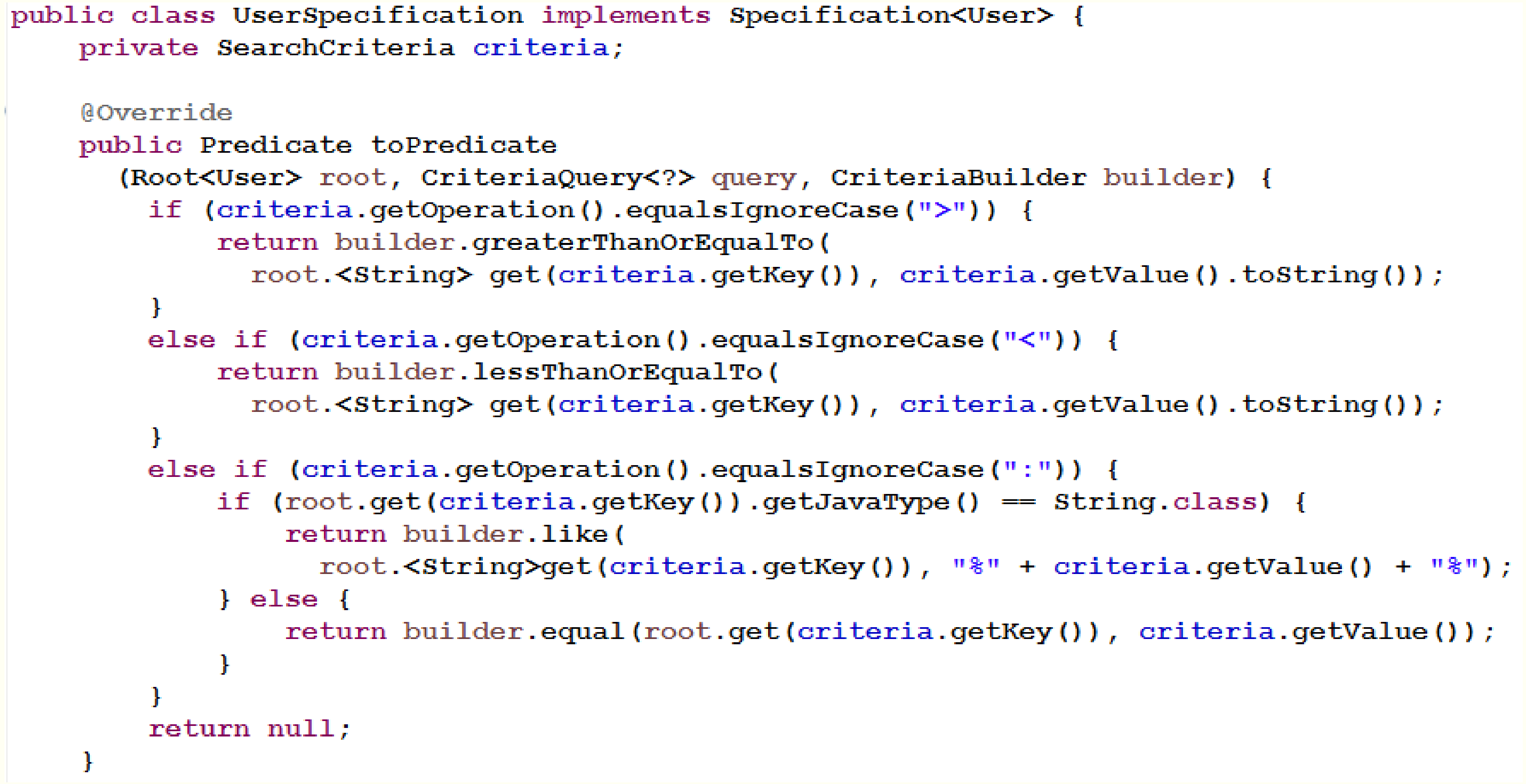


```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder);
}
```

Then, you can use one Repository method for all kind of queries which defined by Specification

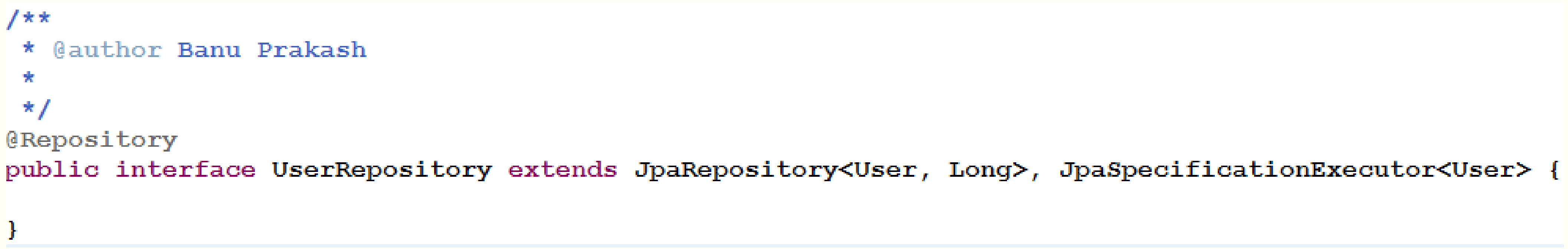
```
List<T> readAll(Specification<T> spec);
```

Spring Data JPA Specification API



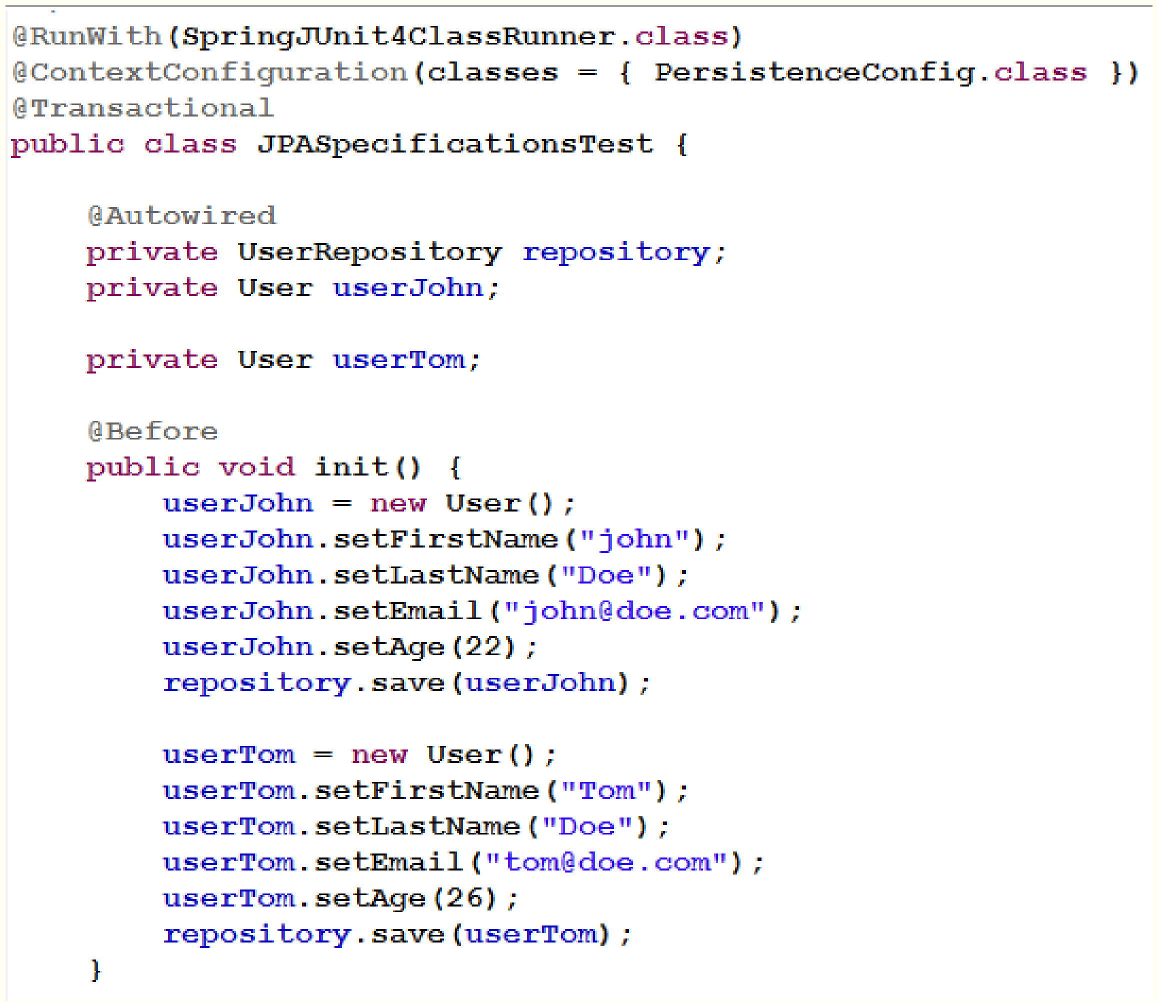
```
public class UserSpecification implements Specification<User> {
    private SearchCriteria criteria;

    @Override
    public Predicate toPredicate
      (Root<User> root, CriteriaQuery<?> query, CriteriaBuilder builder) {
        if (criteria.getOperation().equalsIgnoreCase(">")) {
            return builder.greaterThanOrEqualTo(
              root.<String> get(criteria.getKey()), criteria.getValue().toString());
        }
        else if (criteria.getOperation().equalsIgnoreCase("<")) {
            return builder.lessThanOrEqualTo(
              root.<String> get(criteria.getKey()), criteria.getValue().toString());
        }
        else if (criteria.getOperation().equalsIgnoreCase(":")) {
            if (root.get(criteria.getKey()).getJavaType() == String.class) {
                return builder.like(
                  root.<String>get(criteria.getKey()), "%" + criteria.getValue() + "%");
            } else {
                return builder.equal(root.get(criteria.getKey()), criteria.getValue());
            }
        }
        return null;
    }
```

Repository using Specification



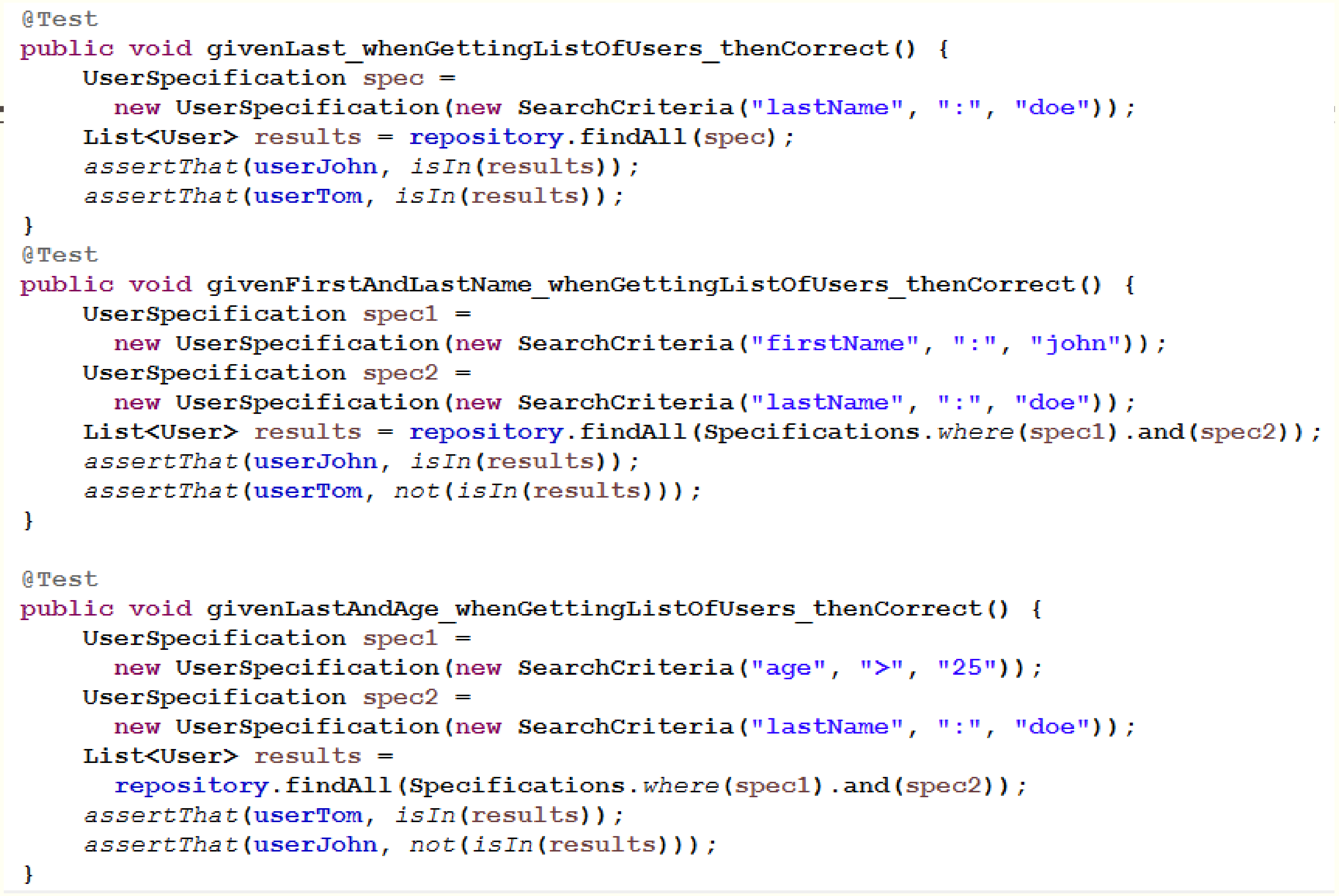
```java
/**
 * @author Banu Prakash
 *
 */
@Repository
public interface UserRepository extends JpaRepository<User, Long>, JpaSpecificationExecutor<User> {

}
```

Test the Search Queries



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { PersistenceConfig.class })
@Transactional
public class JPASpecificationsTest {

    @Autowired
    private UserRepository repository;
    private User userJohn;

    private User userTom;

    @Before
    public void init() {
        userJohn = new User();
        userJohn.setFirstName("john");
        userJohn.setLastName("Doe");
        userJohn.setEmail("john@doe.com");
        userJohn.setAge(22);
        repository.save(userJohn);

        userTom = new User();
        userTom.setFirstName("Tom");
        userTom.setLastName("Doe");
        userTom.setEmail("tom@doe.com");
        userTom.setAge(26);
        repository.save(userTom);
    }
```

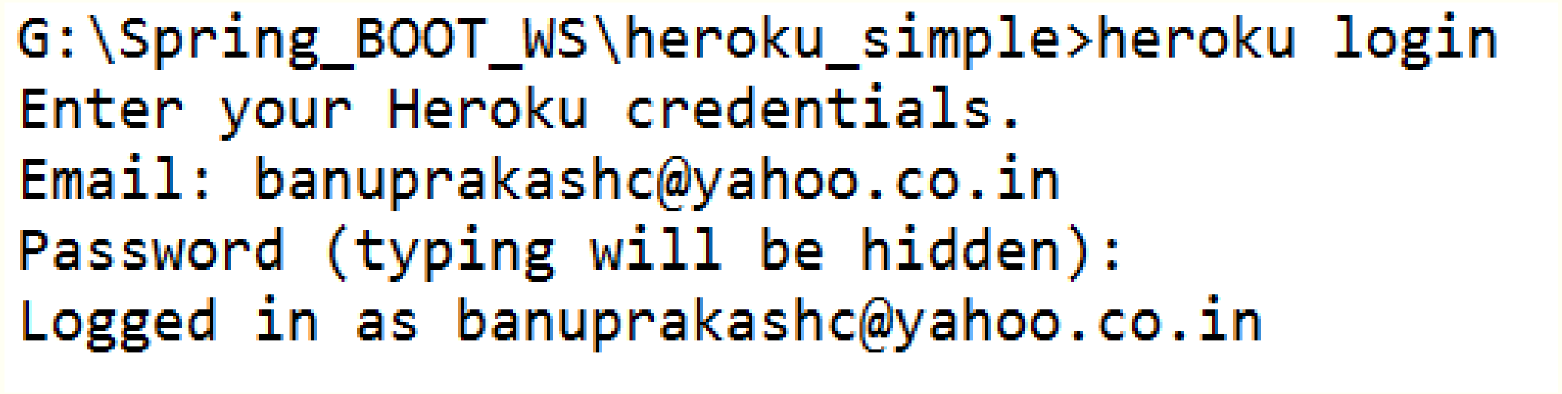
Test the Search Queries



```java
@Test
public void givenLast_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec =
      new UserSpecification(new SearchCriteria("lastName", ":", "doe"));
    List<User> results = repository.findAll(spec);
    assertThat(userJohn, isIn(results));
    assertThat(userTom, isIn(results));
}
@Test
public void givenFirstAndLastName_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec1 =
      new UserSpecification(new SearchCriteria("firstName", ":", "john"));
    UserSpecification spec2 =
      new UserSpecification(new SearchCriteria("lastName", ":", "doe"));
    List<User> results = repository.findAll(Specifications.where(spec1).and(spec2));
    assertThat(userJohn, isIn(results));
    assertThat(userTom, not(isIn(results)));
}

@Test
public void givenLastAndAge_whenGettingListOfUsers_thenCorrect() {
    UserSpecification spec1 =
      new UserSpecification(new SearchCriteria("age", ">", "25"));
    UserSpecification spec2 =
      new UserSpecification(new SearchCriteria("lastName", ":", "doe"));
    List<User> results =
      repository.findAll(Specifications.where(spec1).and(spec2));
    assertThat(userTom, isIn(results));
    assertThat(userJohn, not(isIn(results)));
}
```

Cloud Services

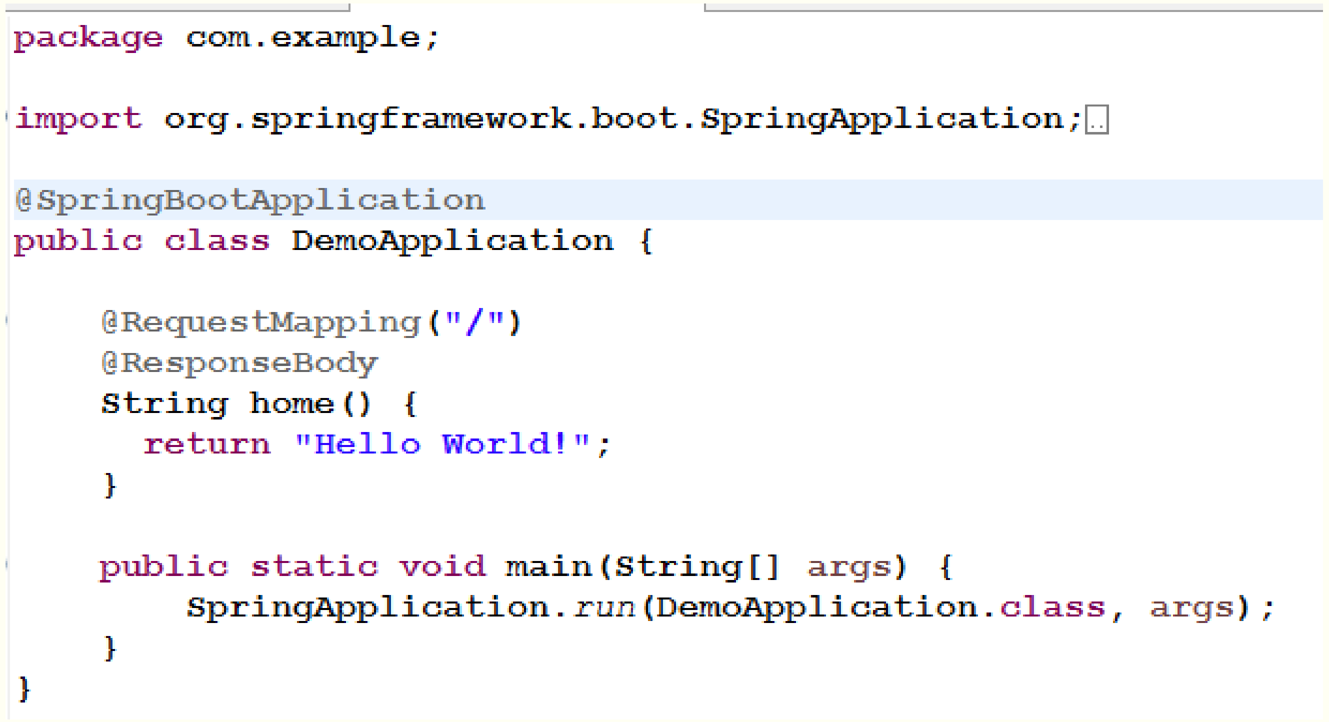
- **Installing the Spring Boot CLI [https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-installing-spring-boot.html#getting-started-manual-cli-installation]**
- GIT CLI: https://git-scm.com/downloads
- To begin, create a free Heroku account.
- Then download and install the Heroku Toolbelt.



```
G:\Spring_BOOT_WS\heroku_simple>heroku login
Enter your Heroku credentials.
Email: banuprakashc@yahoo.co.in
Password (typing will be hidden):
Logged in as banuprakashc@yahoo.co.in
```

Heroku Cloud services

- G:\Spring_BOOT_WS\>spring init --dependencies=web heroku-banuapp
- Import Maven project into eclipse



```java
package com.example;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class DemoApplication {

    @RequestMapping("/")
    @ResponseBody
    String home() {
      return "Hello World!";
    }

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

Preparing a Spring Boot app for Heroku

- Before you can deploy the app to Heroku, you'll need to create a Git repository for the application and add all of the code to it by running these commands:
  - G:\Spring_BOOT_WS\heroku-banuapp>git init
  - G:\Spring_BOOT_WS\heroku-banuapp>git add .
  - G:\Spring_BOOT_WS\heroku-banuapp>git commit -m "Inital Commit“
- In order to deploy to Heroku, you'll first need to provision a new Heroku app. Run this command:
  - **G:\Spring_BOOT_WS\heroku-banuapp>heroku create**
    - Creating app... done, enigmatic-hamlet-26764
    - https://enigmatic-hamlet-26764.herokuapp.com/ | https://git.heroku.com/enigmatic-hamlet-26764.git
  - **G:\Spring_BOOT_WS\heroku-banuapp>heroku apps:rename heroku-banuapp**
    - Renaming enigmatic-hamlet-26764 to heroku-banuapp... done
    - https://heroku-banuapp.herokuapp.com/ | https://git.heroku.com/heroku-banuapp.git
    - Git remote heroku updated
    - ! Don't forget to update git remotes for all other local checkouts of the app.

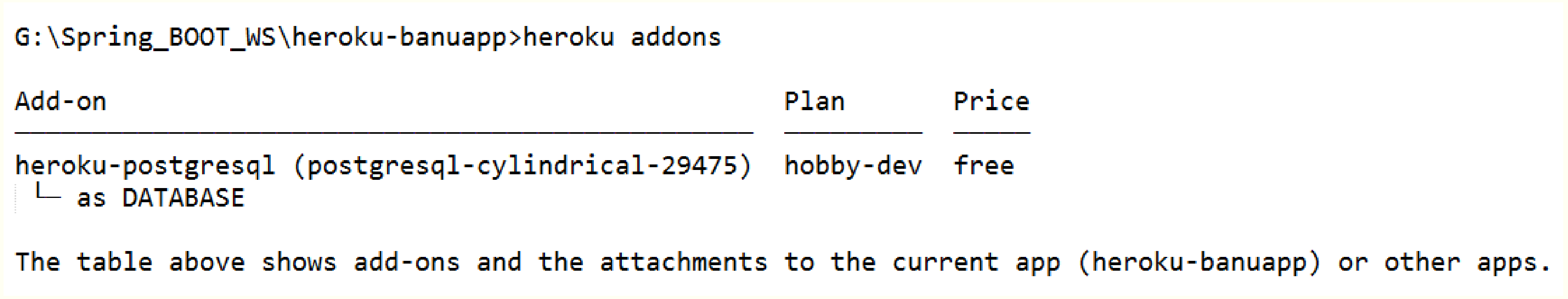
Deploy your code

- **G:\Spring_BOOT_WS\heroku-banuapp>git push heroku master**
- Heroku automatically detects the application as a Maven/Java app due to the presence of a pom.xml file. It installed Java 8 by default, but you can easily configure this with a system.properties file

- All that said, the application is now deployed.
- You can visit the app’s URL by running this command:
  - **G:\Spring_BOOT_WS\heroku-banuapp> heroku open**
- You can view the logs for the application by running this command:
  - **G:\Spring_BOOT_WS\heroku-banuapp> heroku logs --tail**

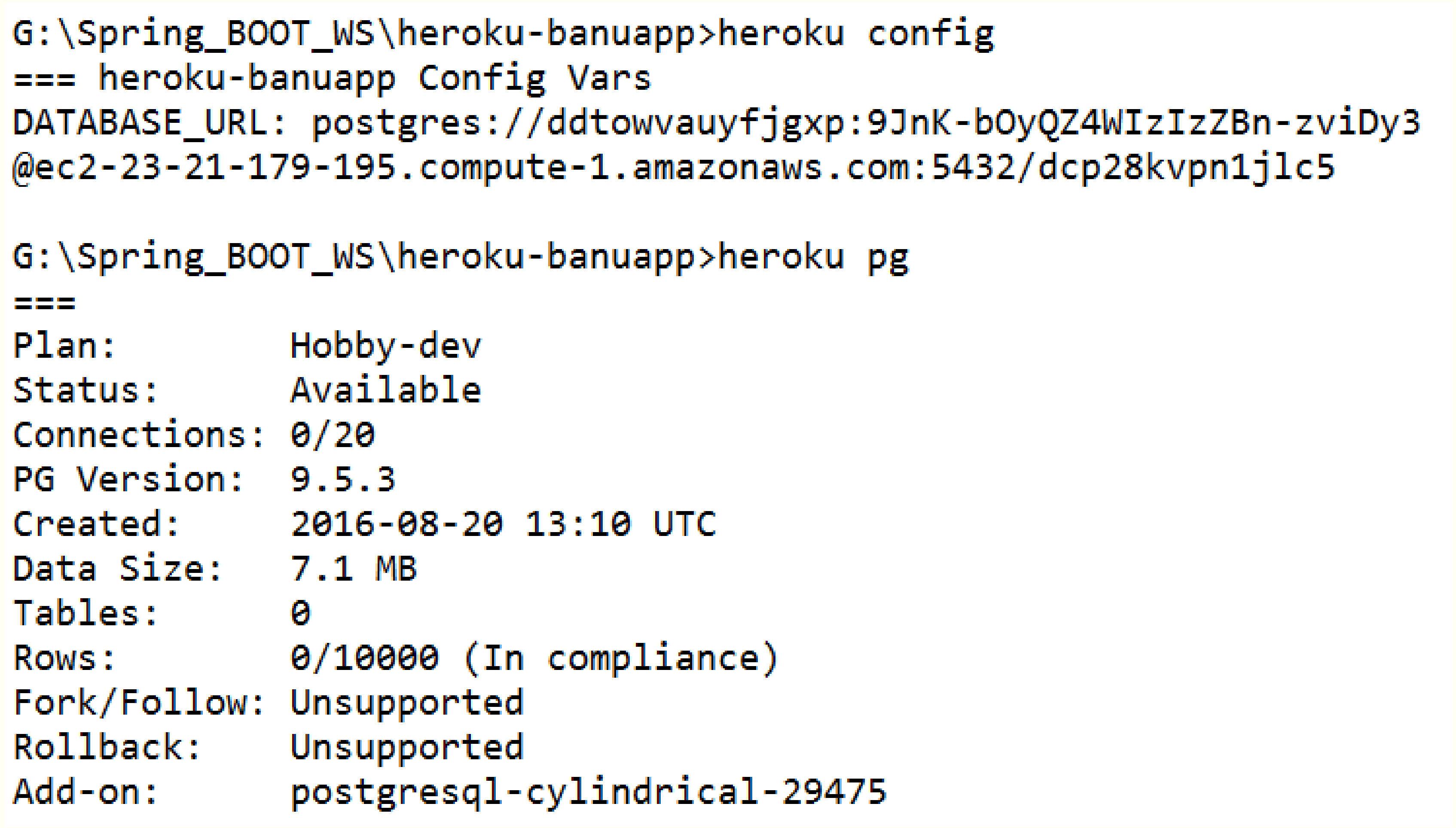
Connection to database

- Fortunately, a database has already been provision for your app. You can find more about it by running the addons command in the CLI:



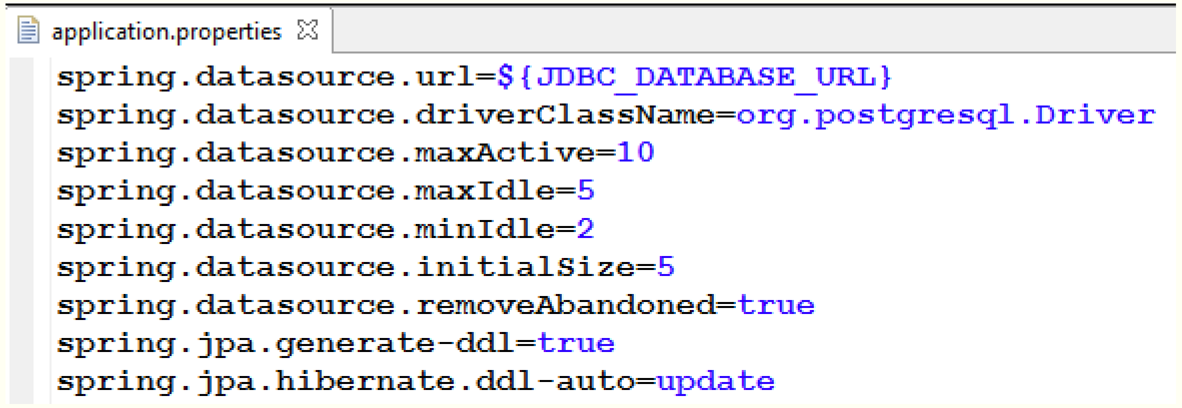
```
G:\Spring_BOOT_WS\heroku-banuapp>heroku addons

Add-on                                            Plan       Price
────────────────────────────────────────────────  ─────────  ─────
heroku-postgresql (postgresql-cylindrical-29475)  hobby-dev  free
 └─ as DATABASE

The table above shows add-ons and the attachments to the current app (heroku-banuapp) or other apps.
```

Connection to database



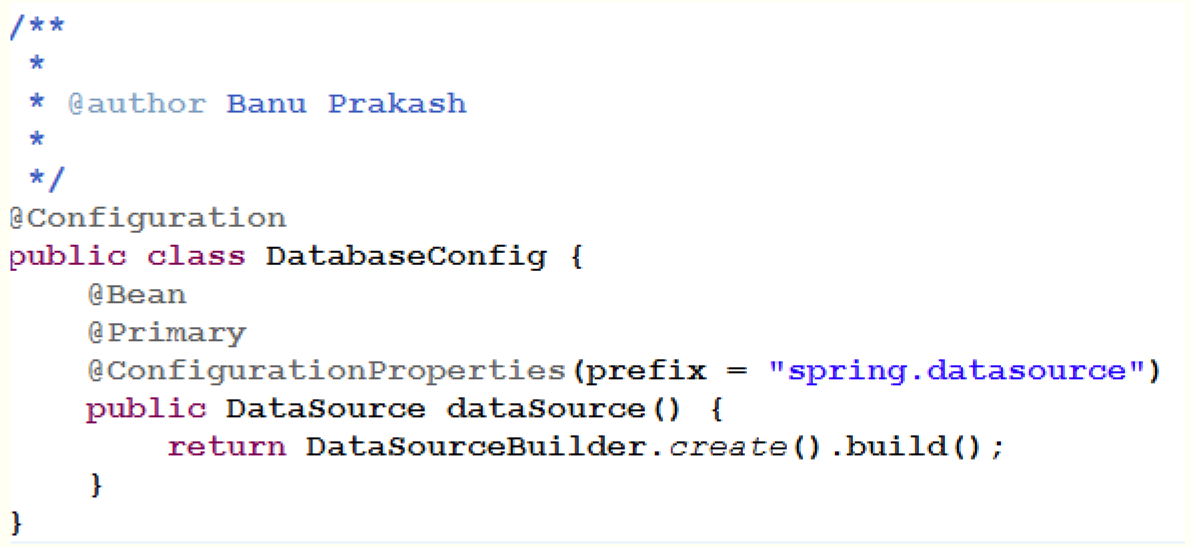
```
G:\Spring_BOOT_WS\heroku-banuapp>heroku config
=== heroku-banuapp Config Vars
DATABASE_URL: postgres://ddtowvauyfjgxp:9JnK-bOyQZ4WIzIzZBn-zviDy3
@ec2-23-21-179-195.compute-1.amazonaws.com:5432/dcp28kvpn1jlc5

G:\Spring_BOOT_WS\heroku-banuapp>heroku pg
===
Plan:        Hobby-dev
Status:      Available
Connections: 0/20
PG Version:  9.5.3
Created:     2016-08-20 13:10 UTC
Data Size:   7.1 MB
Tables:      0
Rows:        0/10000 (In compliance)
Fork/Follow: Unsupported
Rollback:    Unsupported
Add-on:      postgresql-cylindrical-29475
```

Connecting to database



application.properties

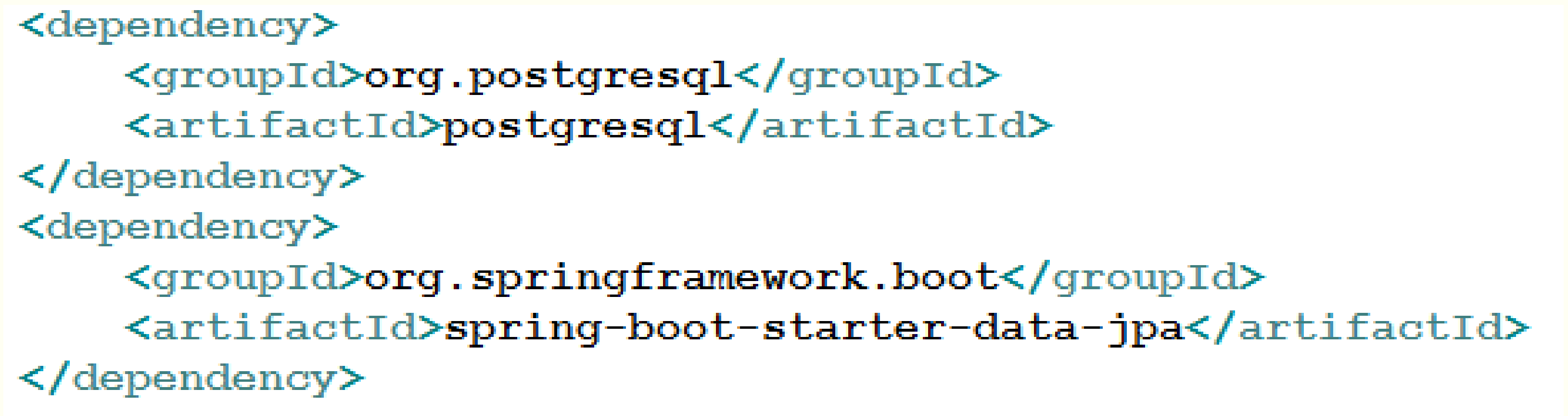
```
spring.datasource.url=${JDBC_DATABASE_URL}
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.maxActive=10
spring.datasource.maxIdle=5
spring.datasource.minIdle=2
spring.datasource.initialSize=5
spring.datasource.removeAbandoned=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
```



```
/**
 *
 * @author Banu Prakash
 *
 */
@Configuration
public class DatabaseConfig {
    @Bean
    @Primary
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource dataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

Connecting to database

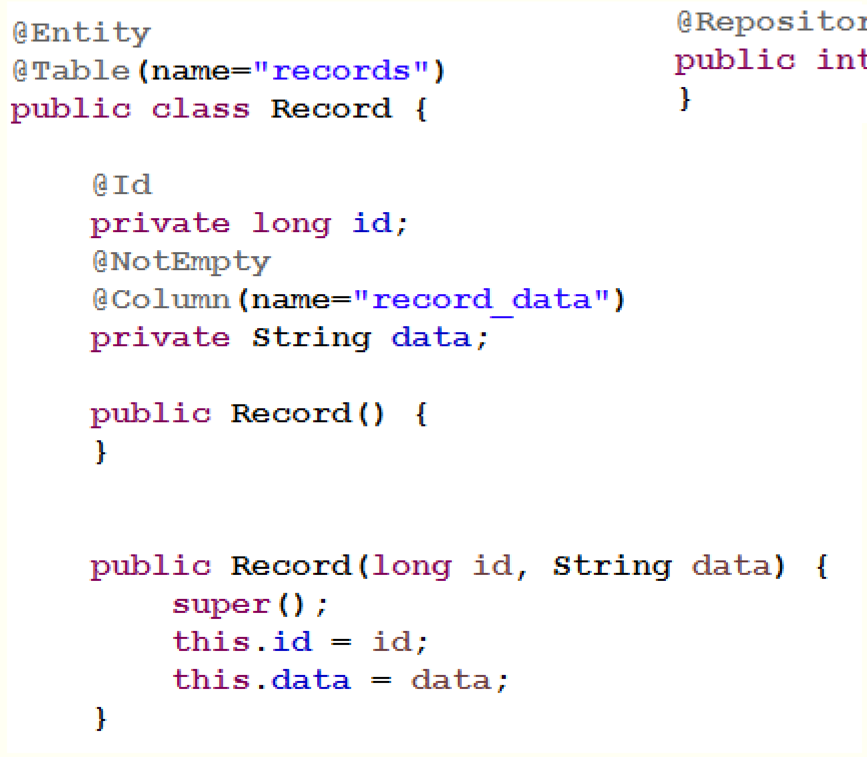
- Add the following dependencies for Postgresql and spring-data-jpa



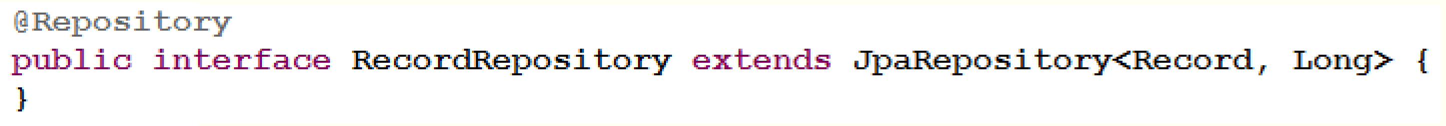
```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Connecting to database

- Entity class and Repository

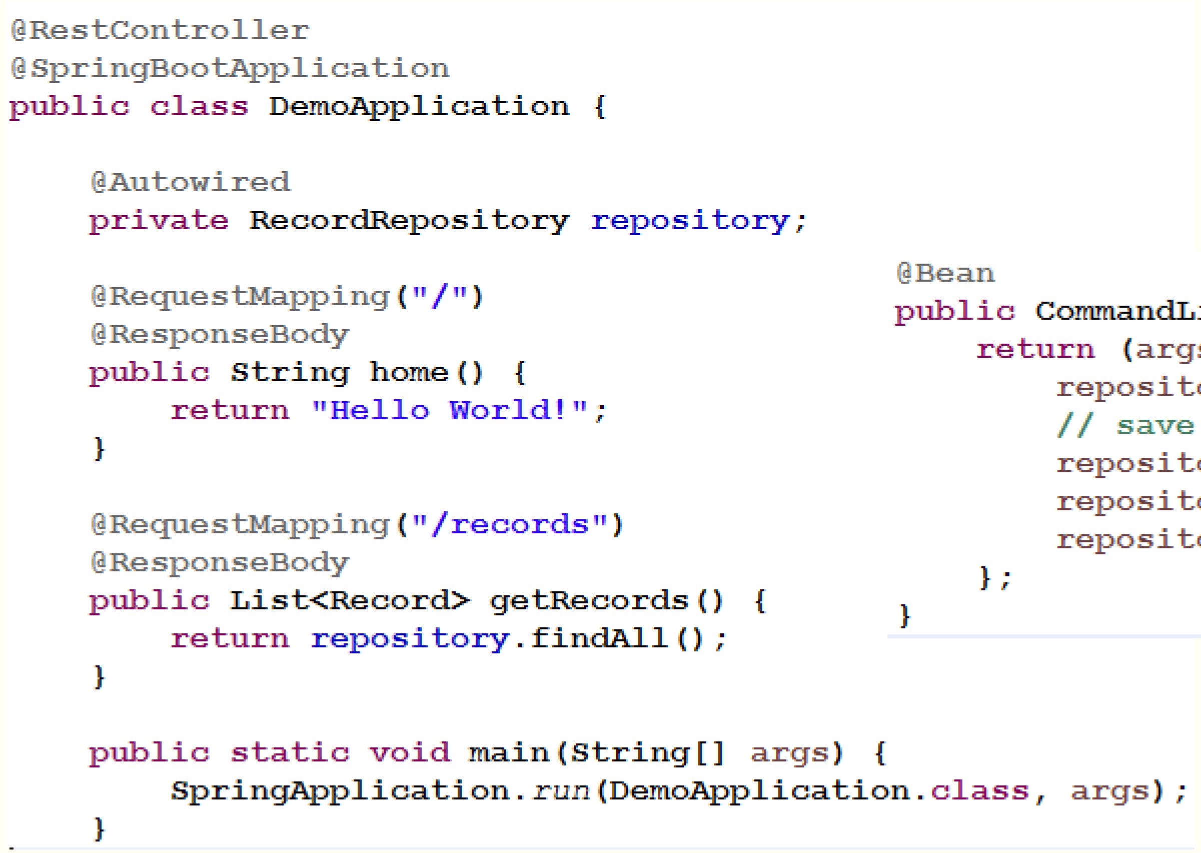


```
@Entity
@Table(name="records")
public class Record {

    @Id
    private long id;
    @NotEmpty
    @Column(name="record_data")
    private String data;

    public Record() {
    }


    public Record(long id, String data) {
        super();
        this.id = id;
        this.data = data;
    }
```

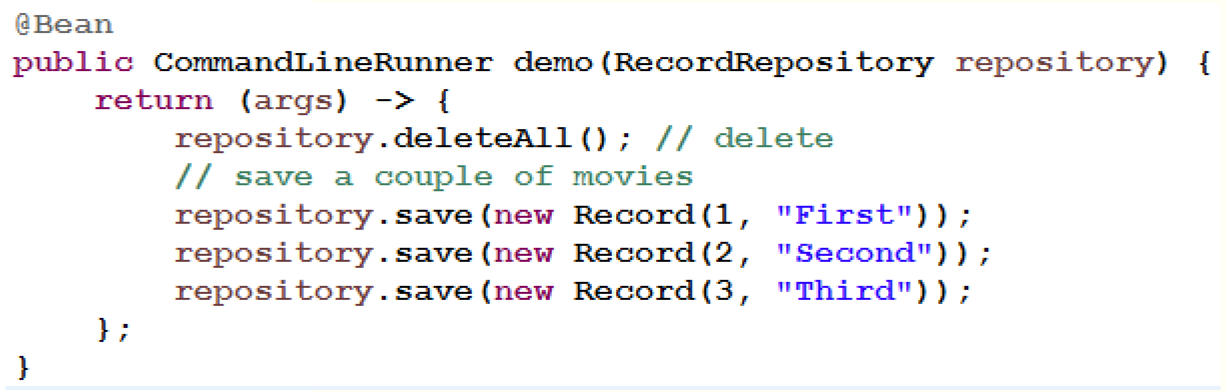


```
@Repository
public interface RecordRepository extends JpaRepository<Record, Long> {
}
```

Connecting to database [Launch the application]



```
@RestController
@SpringBootApplication
public class DemoApplication {

    @Autowired
    private RecordRepository repository;

    @RequestMapping("/")
    @ResponseBody
    public String home() {
        return "Hello World!";
    }

    @RequestMapping("/records")
    @ResponseBody
    public List<Record> getRecords() {
        return repository.findAll();
    }

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



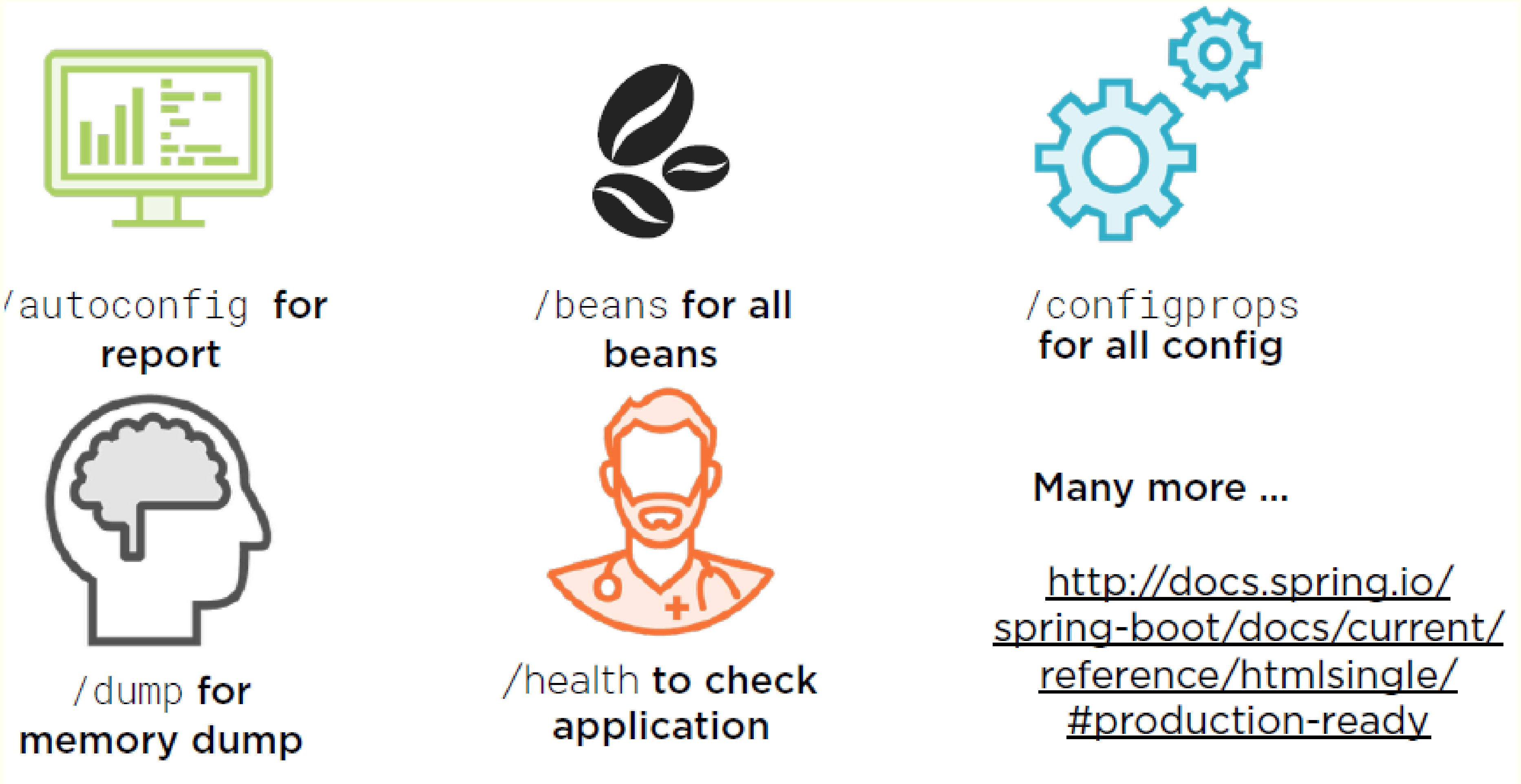
```
@Bean
public CommandLineRunner demo(RecordRepository repository) {
    return (args) -> {
        repository.deleteAll(); // delete
        // save a couple of movies
        repository.save(new Record(1, "First"));
        repository.save(new Record(2, "Second"));
        repository.save(new Record(3, "Third"));
    };
}
```

Monitoring Your Spring Boot Apps in the Cloud

- Spring Boot Actuator

- Production ready **monitoring** and **management features** out of the box
  - Health, autoconfig report, beans, etc
- HTTP or JMX
  - Feed into Nagios / Zabbix / New Relic
- Easy to add your own

Built-in Production Ready Endpoints



/autoconfig for report

/beans for all beans

/configprops for all config

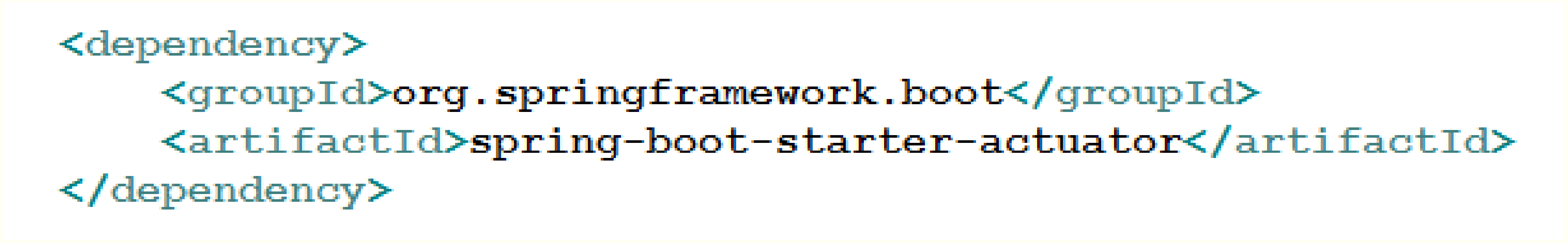
/dump for memory dump

/health to check application

Many more ...

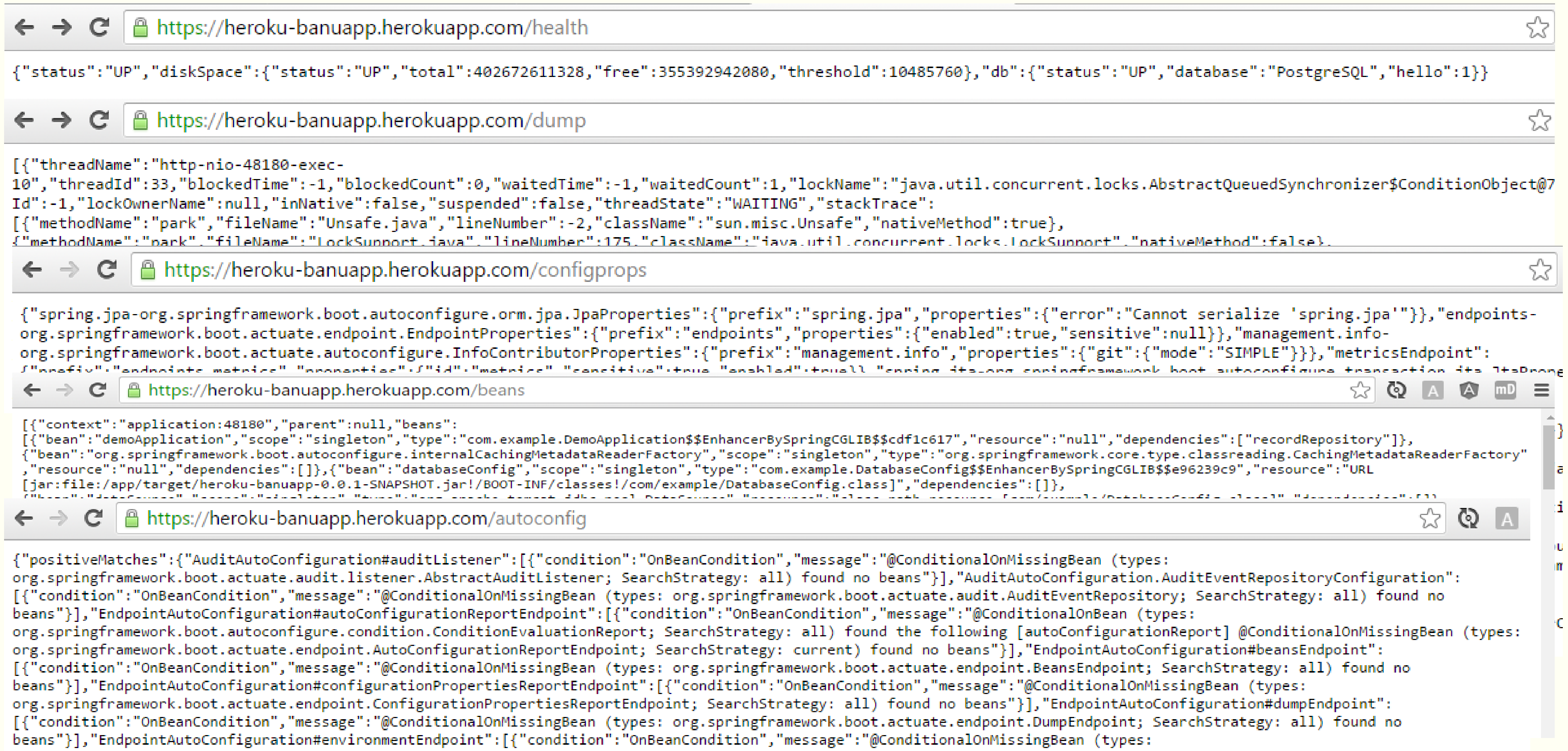
http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready

Adding Spring Boot Actuator to Your Project



```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Built-in Production Ready Endpoints



https://heroku-banuapp.herokuapp.com/health

```
{"status":"UP","diskSpace":{"status":"UP","total":402672611328,"free":355392942080,"threshold":10485760},"db":{"status":"UP","database":"PostgreSQL","hello":1}}
```

https://heroku-banuapp.herokuapp.com/dump

```
[{"threadName":"http-nio-48180-exec-
10","threadId":33,"blockedTime":-1,"blockedCount":0,"waitedTime":-1,"waitedCount":1,"lockName":"java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@7
Id":-1,"lockOwnerName":null,"inNative":false,"suspended":false,"threadState":"WAITING","stackTrace":
[{"methodName":"park","fileName":"Unsafe.java","lineNumber":-2,"className":"sun.misc.Unsafe","nativeMethod":true},
{"methodName":"park","fileName":"LockSupport.java","lineNumber":175,"className":"java.util.concurrent.locks.LockSupport","nativeMethod":false},
```

https://heroku-banuapp.herokuapp.com/configprops

```
{"spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties":{"prefix":"spring.jpa","properties":{"error":"Cannot serialize 'spring.jpa'"}},"endpoints-
org.springframework.boot.actuate.endpoint.EndpointProperties":{"prefix":"endpoints","properties":{"enabled":true,"sensitive":null}},"management.info-
org.springframework.boot.actuate.autoconfigure.InfoContributorProperties":{"prefix":"management.info","properties":{"git":{"mode":"SIMPLE"}}},"metricsEndpoint":
{"prefix":"endpoints.metrics","properties":{"id":"metrics","sensitive":true,"enabled":true}},"spring.jta-org.springframework.boot.autoconfigure.transaction.jta.JtaPrope
```

https://heroku-banuapp.herokuapp.com/beans

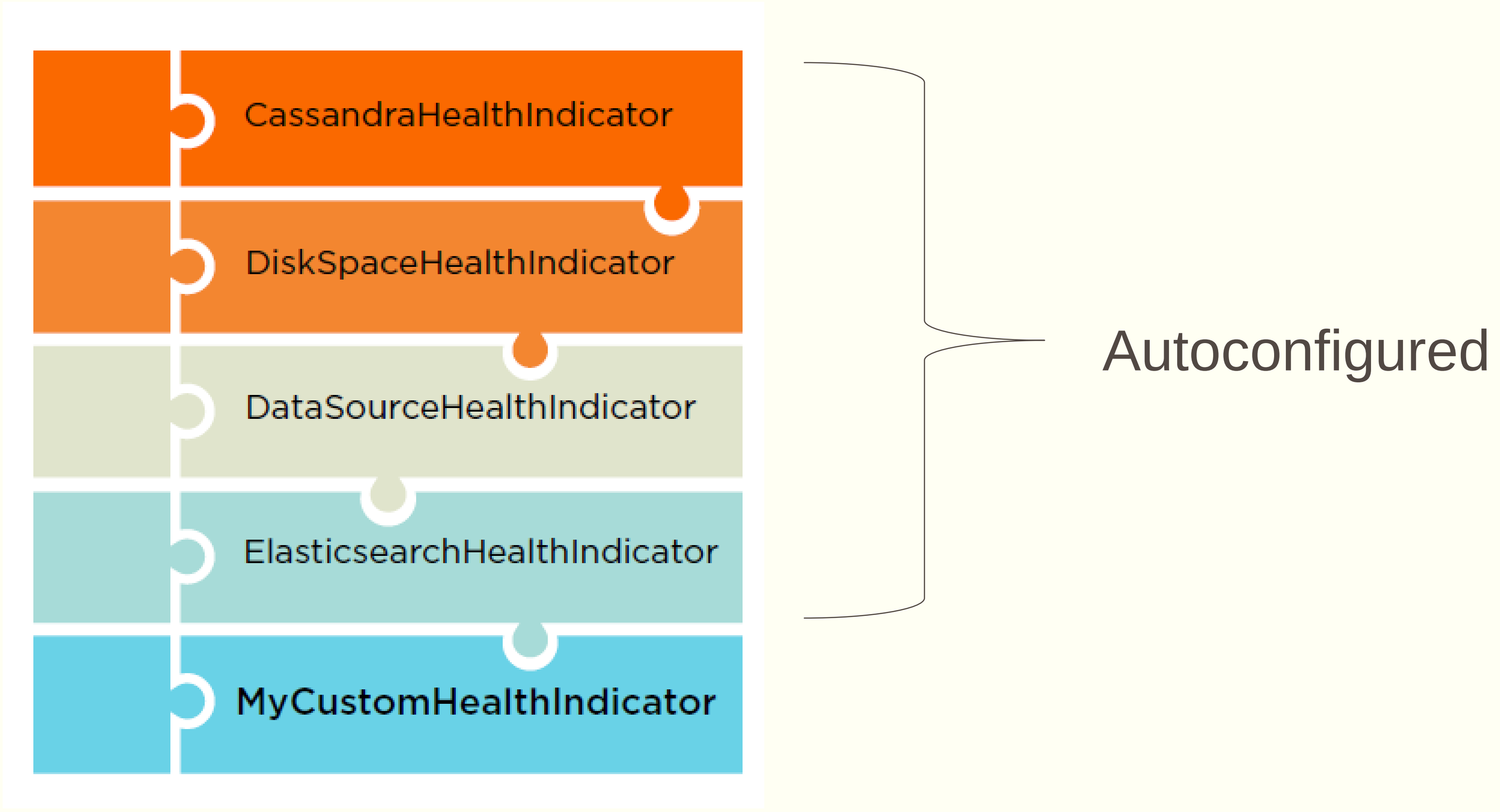
```
[{"context":"application:48180","parent":null,"beans":
[{"bean":"demoApplication","scope":"singleton","type":"com.example.DemoApplication$$EnhancerBySpringCGLIB$$cdf1c617","resource":"null","dependencies":["recordRepository"]},
{"bean":"org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory","scope":"singleton","type":"org.springframework.core.type.classreading.CachingMetadataReaderFactory"
,"resource":"null","dependencies":[]},{"bean":"databaseConfig","scope":"singleton","type":"com.example.DatabaseConfig$$EnhancerBySpringCGLIB$$e96239c9","resource":"URL
[jar:file:/app/target/heroku-banuapp-0.0.1-SNAPSHOT.jar!/BOOT-INF/classes!/com/example/DatabaseConfig.class]","dependencies":[]},
```

https://heroku-banuapp.herokuapp.com/autoconfig

```
{"positiveMatches":{"AuditAutoConfiguration#auditListener":[{"condition":"OnBeanCondition","message":"@ConditionalOnMissingBean (types:
org.springframework.boot.actuate.audit.listener.AbstractAuditListener; SearchStrategy: all) found no beans"}],"AuditAutoConfiguration.AuditEventRepositoryConfiguration":
[{"condition":"OnBeanCondition","message":"@ConditionalOnMissingBean (types: org.springframework.boot.actuate.audit.AuditEventRepository; SearchStrategy: all) found no
beans"}],"EndpointAutoConfiguration#autoConfigurationReportEndpoint":[{"condition":"OnBeanCondition","message":"@ConditionalOnBean (types:
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReport; SearchStrategy: all) found the following [autoConfigurationReport] @ConditionalOnMissingBean (types:
org.springframework.boot.actuate.endpoint.AutoConfigurationReportEndpoint; SearchStrategy: current) found no beans"}],"EndpointAutoConfiguration#beansEndpoint":
[{"condition":"OnBeanCondition","message":"@ConditionalOnMissingBean (types: org.springframework.boot.actuate.endpoint.BeansEndpoint; SearchStrategy: all) found no
beans"}],"EndpointAutoConfiguration#configurationPropertiesReportEndpoint":[{"condition":"OnBeanCondition","message":"@ConditionalOnMissingBean (types:
org.springframework.boot.actuate.endpoint.ConfigurationPropertiesReportEndpoint; SearchStrategy: all) found no beans"}],"EndpointAutoConfiguration#dumpEndpoint":
[{"condition":"OnBeanCondition","message":"@ConditionalOnMissingBean (types: org.springframework.boot.actuate.endpoint.DumpEndpoint; SearchStrategy: all) found no
beans"}],"EndpointAutoConfiguration#environmentEndpoint":[{"condition":"OnBeanCondition","message":"@ConditionalOnMissingBean (types:
```

HealthIndicator’s

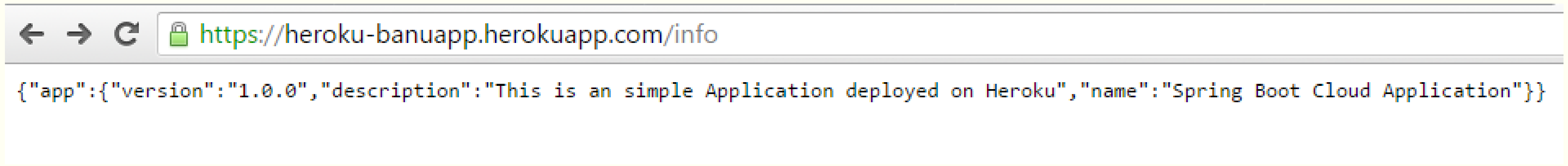
- We can add custom health indicators along with existing Autoconfigured HealthIndicator’s



/info Endpoint

- application.properites

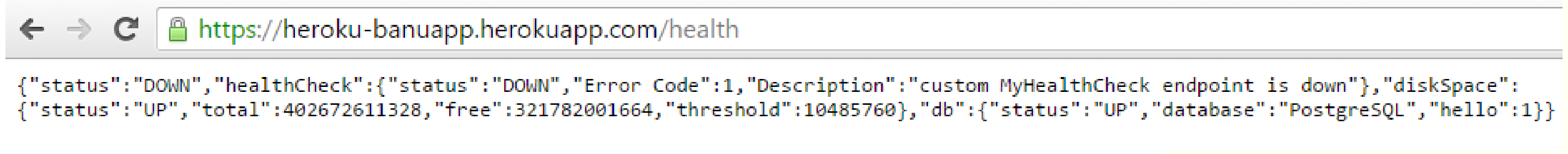
```
info.app.name=Spring Boot Cloud Application
info.app.description=This is an simple Application deployed on Heroku
info.app.version=1.0.0
```


https://heroku-banuapp.herokuapp.com/info

```
{"app":{"version":"1.0.0","description":"This is an simple Application deployed on Heroku","name":"Spring Boot Cloud Application"}}
```

Custom HealthIndicator

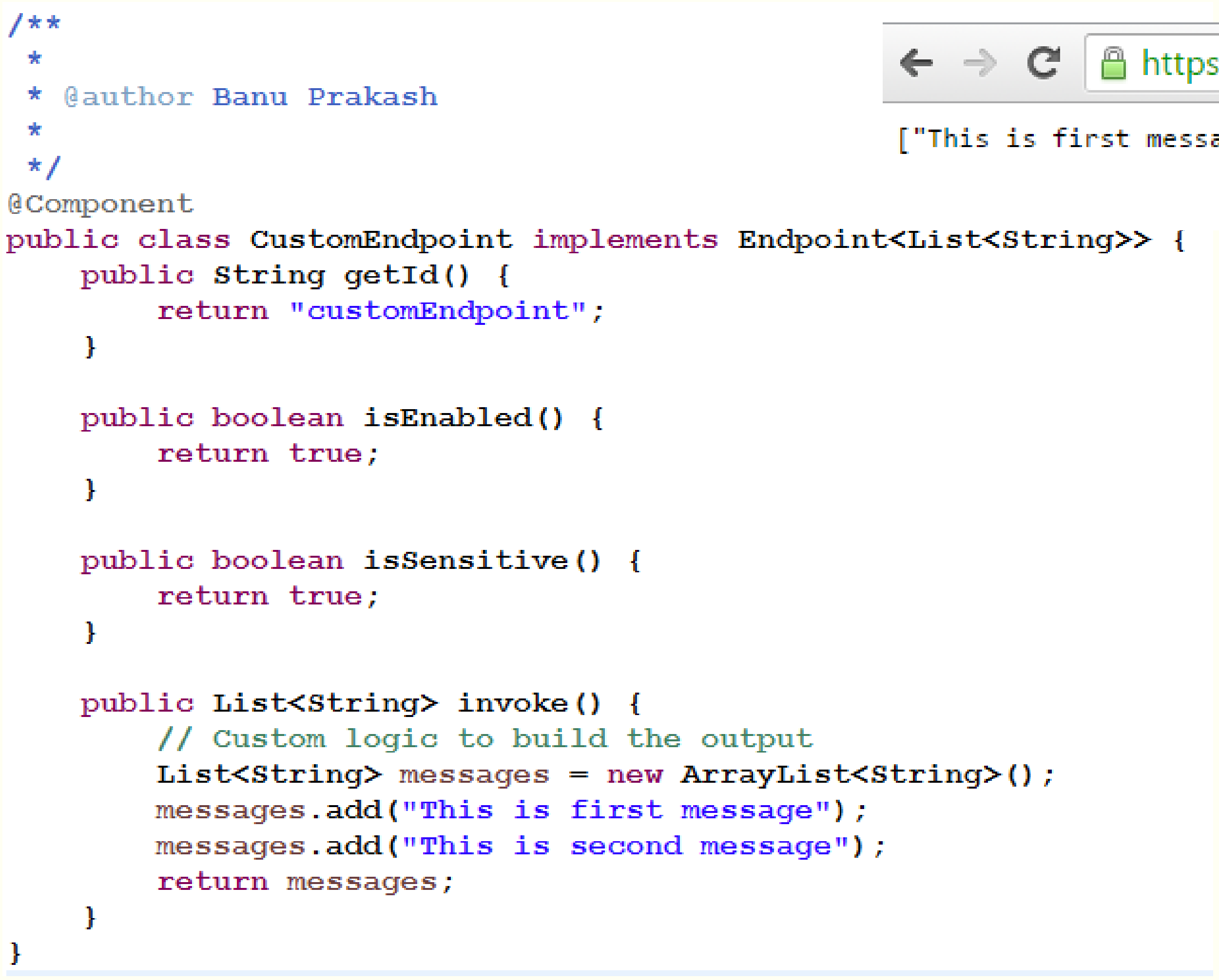
```java
/**
 * Custom Health Check.
 *
 * @author Banu Prakash
 *
 */
@Component
public class HealthCheck implements HealthIndicator {
    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down()
                    .withDetail("Error Code", errorCode)
                    .withDetail("Description", "custom MyHealthCheck endpoint is down")
                    .build();
        }
        return Health.up().build();
    }

    public int check() {
        // Your logic to check health
        return 1;
    }
}
```



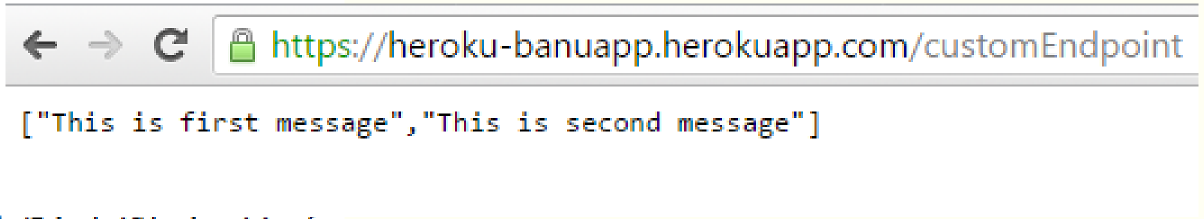
https://heroku-banuapp.herokuapp.com/health

```
{"status":"DOWN","healthCheck":{"status":"DOWN","Error Code":1,"Description":"custom MyHealthCheck endpoint is down"},"diskSpace":
{"status":"UP","total":402672611328,"free":321782001664,"threshold":10485760},"db":{"status":"UP","database":"PostgreSQL","hello":1}}
```

Custom Endpoints



```
/**
 *
 * @author Banu Prakash
 *
 */
@Component
public class CustomEndpoint implements Endpoint<List<String>> {
    public String getId() {
        return "customEndpoint";
    }

    public boolean isEnabled() {
        return true;
    }

    public boolean isSensitive() {
        return true;
    }

    public List<String> invoke() {
        // Custom logic to build the output
        List<String> messages = new ArrayList<String>();
        messages.add("This is first message");
        messages.add("This is second message");
        return messages;
    }
}
```



https://heroku-banuapp.herokuapp.com/customEndpoint

```
["This is first message","This is second message"]
```

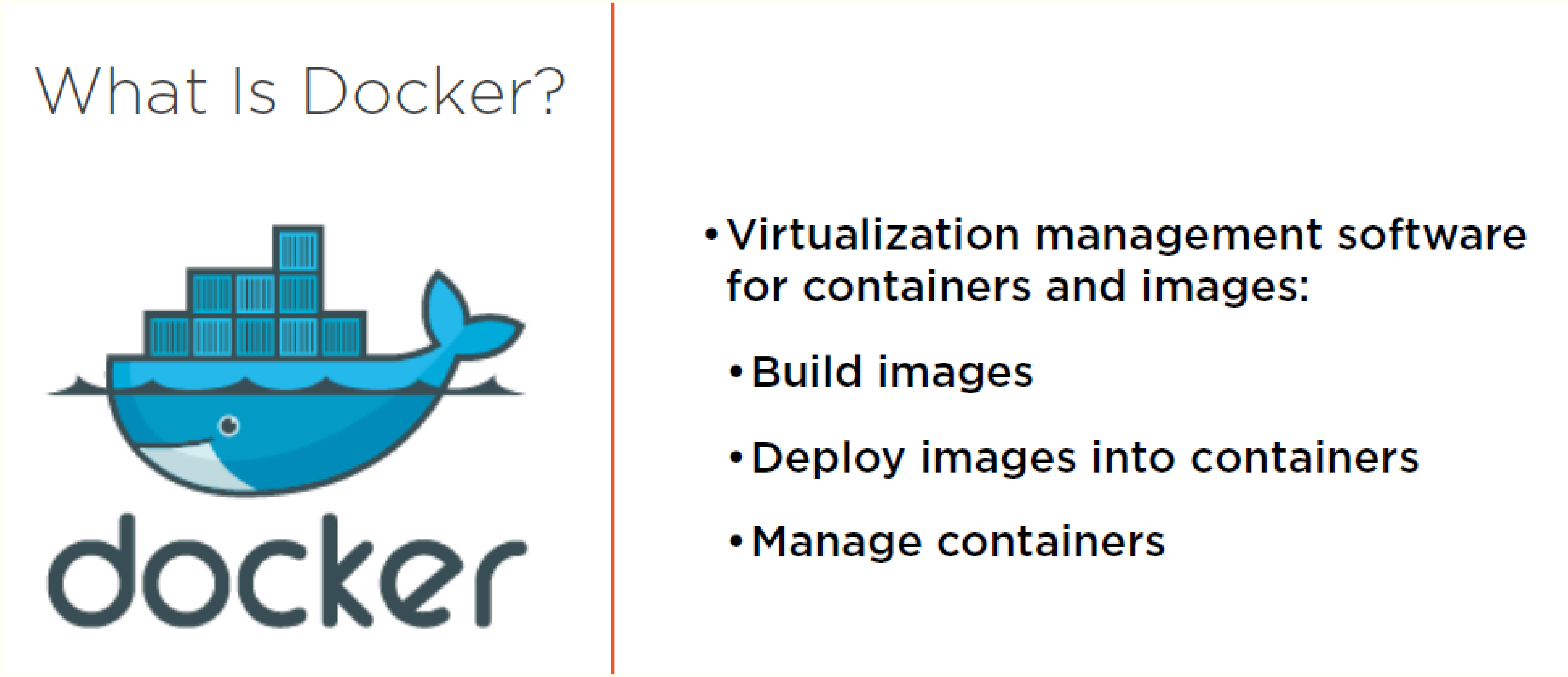
Dynos

- Dynos
    - A dyno is a lightweight Linux container that runs a single user-specified command. A dyno can run any command available in its default environment or in your app's slug (a compressed and pre-packaged copy of your application and its dependencies).
- Dyno configurations
    - Web Dynos: Web dynos are dynos of the “web” process type that is defined in your Procfile. Only web dynos receive HTTP traffic from the routers.
    - Worker Dynos: Worker dynos can be of any process type declared in your Procfile, other than “web”. Worker dynos are typically used for background jobs, queueing systems, and timed jobs..
- The Dyno Manager
    - The dyno manager keeps dynos running automatically; so operating your app is generally hands-off and maintenance free
- Scalability
    - To scale horizontally, add more dynos. For example, adding more web dynos allows you to handle more concurrent HTTP requests, and therefore higher volumes of traffic.
- Refer: [https://devcenter.heroku.com/articles/dynos#cli-commands-for-dyno-management]

Procfile

- A Procfile is a mechanism for declaring what commands are run by your application’s dynos on the Heroku platform.
- A Procfile is a file named Procfile. and not anything else
- Declaring process types : <process type>: <command>
- The syntax is defined as:
  - <process type> – an alphanumeric string, is a name for your command, such as web, worker, urgentworker, clock, etc.
  - <command> – a command line to launch the process
- Procfile for Spring Boot:
  - web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
- Deploying to Heroku
  - A Procfile is not necessary to deploy apps written in most languages supported by Heroku. The platform automatically detects the language, and creates a default web process type to boot the application server.

Docker



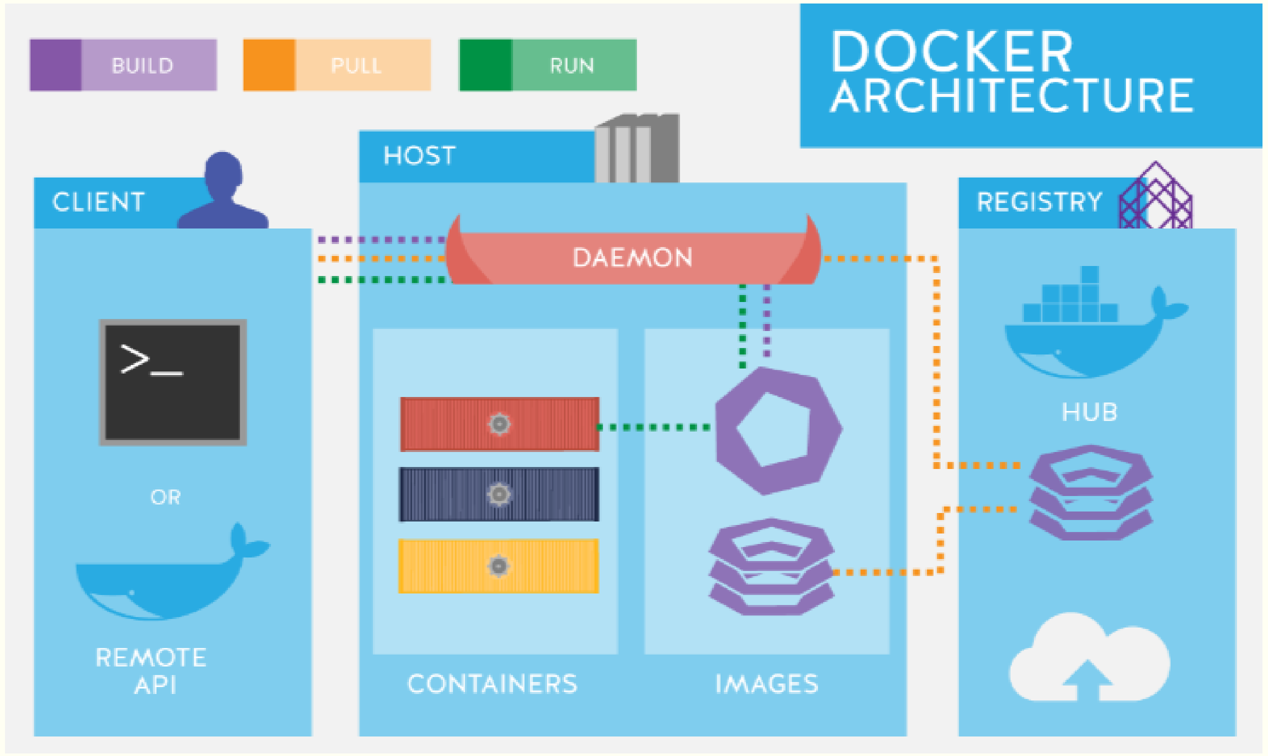
What Is Docker?

docker

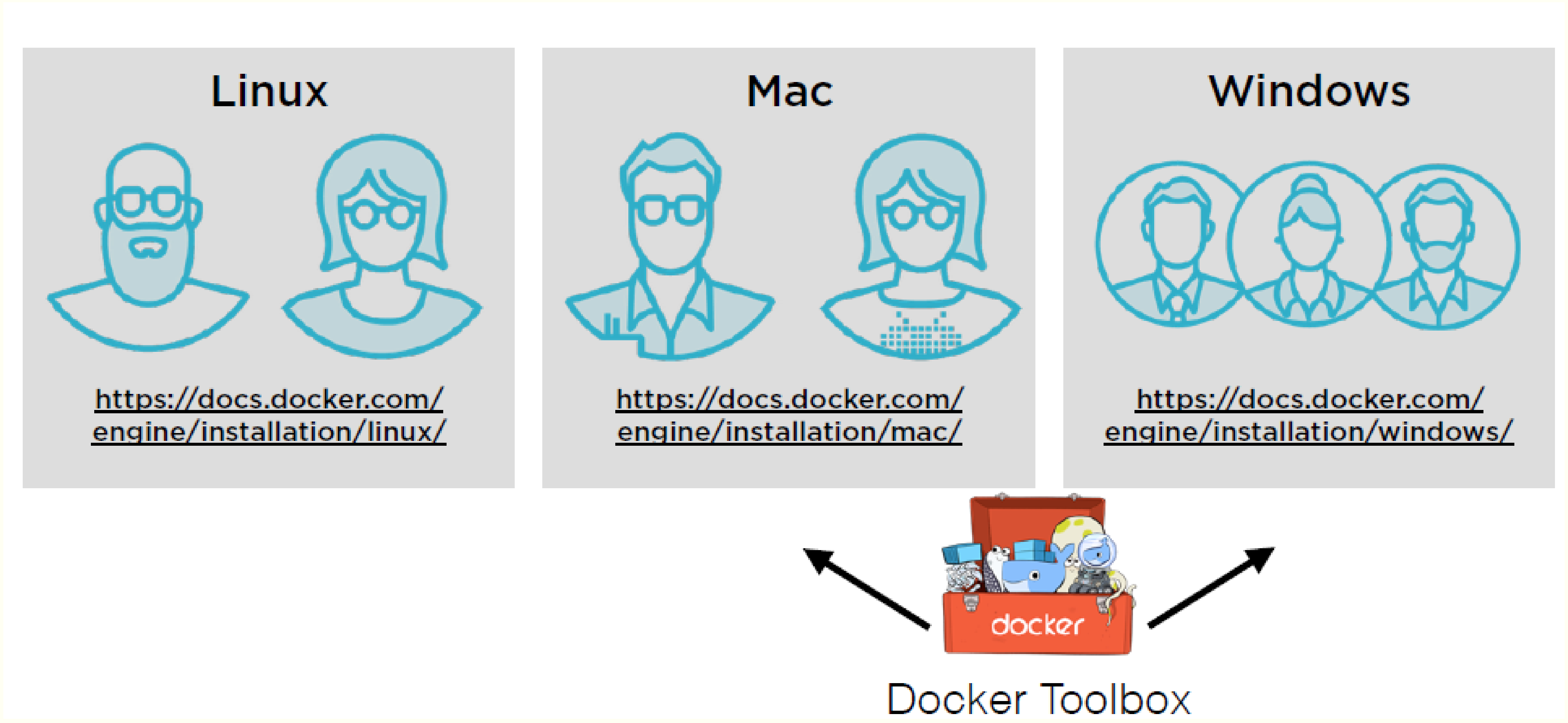
- Virtualization management software for containers and images:
  - Build images
  - Deploy images into containers
  - Manage containers

Docker

- "A container is a stripped-to-basics version of a Linux operating system
- The software you run in a container is called an *image*
- *Why Docker?*
  - *Easy*
  - *Lightweight*
  - *Cloud Agnostic*
  - *Scales*



Installing Docker

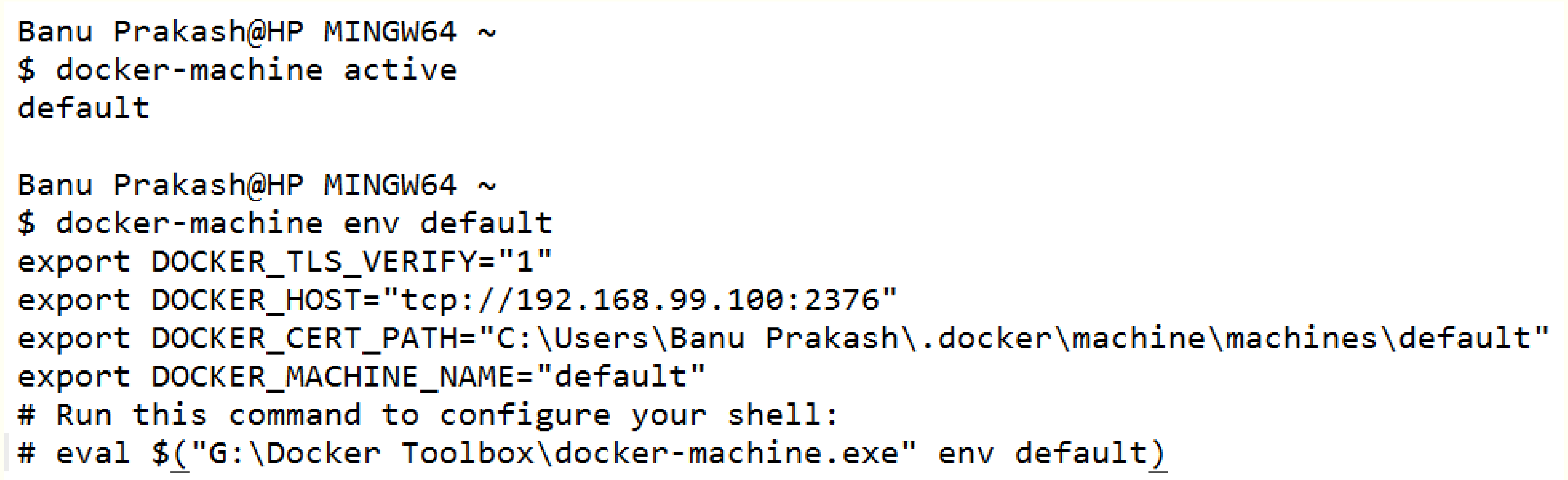

Linux
https://docs.docker.com/engine/installation/linux/
Mac
https://docs.docker.com/engine/installation/mac/
Windows
https://docs.docker.com/engine/installation/windows/
docker
Docker Toolbox

INTRODUCTION AND TOOLS

- Docker Toolbox: containing VirtualBox (for creating the VM that will run your containers), Docker Machine (runs within a VM to run Docker Containers), Docker Kitematic (a GUI for managing containers running in your Docker Machine), and Docker Compose (tool to orchestrate multiple container templates)
- Git: command line git is fine
- Java 8 SDK: Java 8 had me at PermGen improvements; the collection streaming and lambda support are great
- A build tool of choice: Let’s use Maven.
- IDE of choice: We’ll work with the Eclipse/Spring Tool Suite (STS).
- A REST tool: this is very handy for any web service project. I’m a big fan of the Chrome extension Postman. If you’re good at cURL, that works too
- uMongo or other Mongo GUI: a document database fits the model of self-containment pretty well — objects are retrieved automatically, and reference objects are referred to by ID in a microservice architecture, which maps to a document store pretty well. Plus, MongoDB has an “official” Docker image which works very well

Environment variables

- If everything is installed correctly, a command prompt will contain key environment variables:



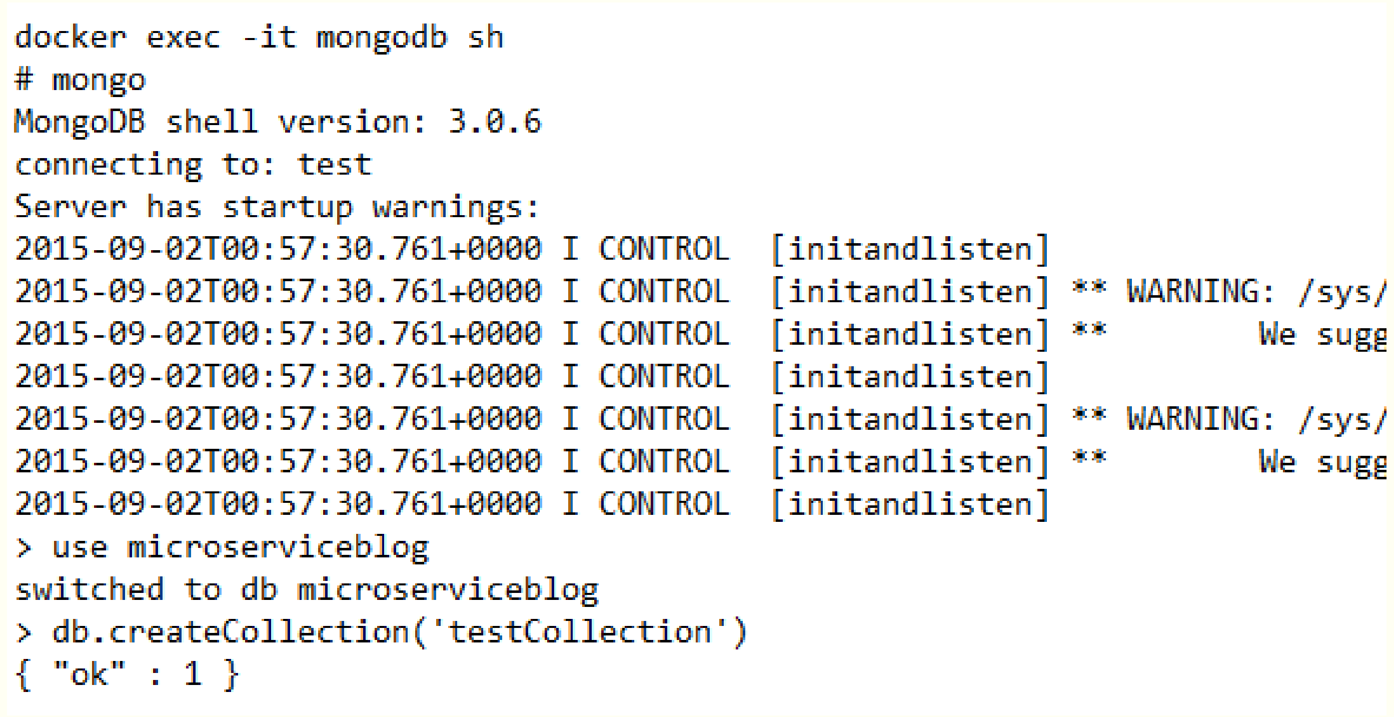
```
Banu Prakash@HP MINGW64 ~
$ docker-machine active
default

Banu Prakash@HP MINGW64 ~
$ docker-machine env default
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="C:\Users\Banu Prakash\.docker\machine\machines\default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval $("G:\Docker Toolbox\docker-machine.exe" env default)
```

Firing up a Mongo container

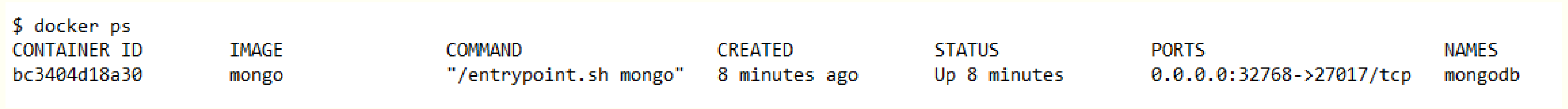
- **docker run -P -d --name mongodb mongo**

- -P tells Docker to expose any container-declared port
- -d says to run the container as a daemon (e.g. in the background)
- - -name mongodb says what name to assign to the container instance (names must be unique across all running container instances. If you don’t supply one, you will get a random semi-friendly name like: modest_poitras)
- the mongo at the end indicates which image definition to use

Firing up a Mongo container

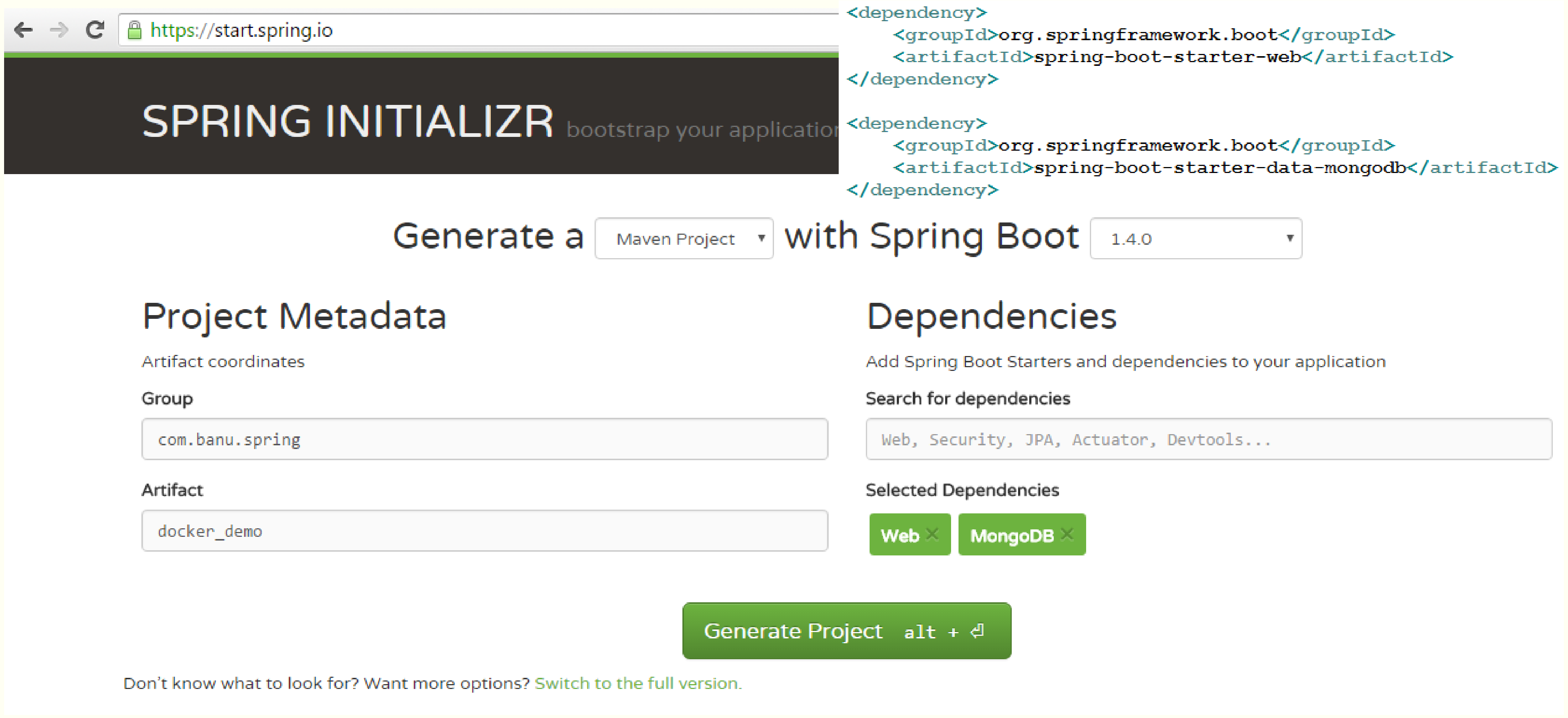


```
docker exec -it mongodb sh
# mongo
MongoDB shell version: 3.0.6
connecting to: test
Server has startup warnings:
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten]
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten] ** WARNING: /sys/
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten] **        We sugg
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten]
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten] ** WARNING: /sys/
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten] **        We sugg
2015-09-02T00:57:30.761+0000 I CONTROL  [initandlisten]
> use microserviceblog
switched to db microserviceblog
> db.createCollection('testCollection')
{ "ok" : 1 }
```



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                      NAMES
bc3404d18a30        mongo               "/entrypoint.sh mongo"   8 minutes ago       Up 8 minutes        0.0.0.0:32768->27017/tcp   mongodb
```

Spring Boot Project


https://start.spring.io

SPRING INITIALIZR bootstrap your applicatio

Generate a Maven Project with Spring Boot 1.4.0

Project Metadata

Artifact coordinates

Group

com.banu.spring

Artifact

docker_demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

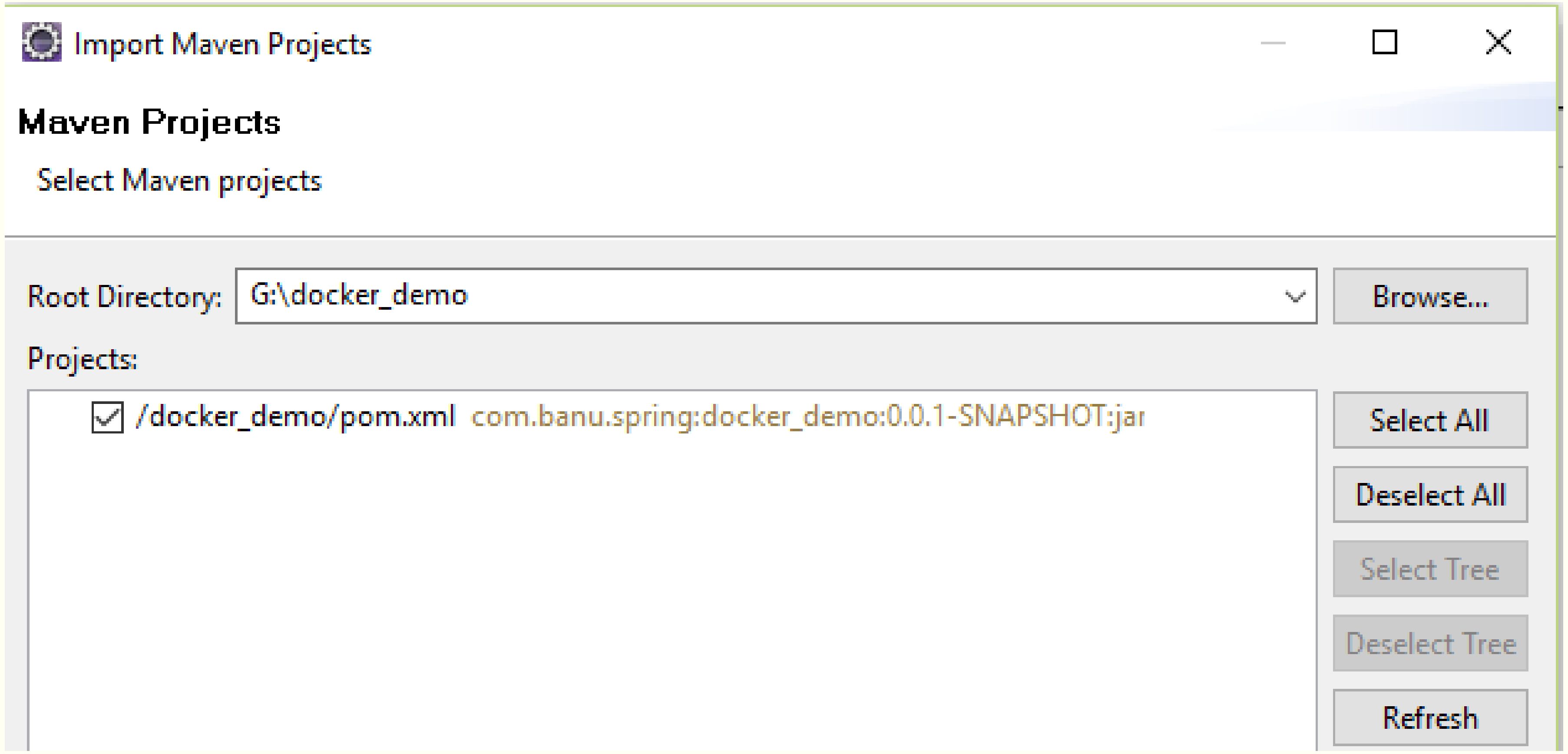
Web × MongoDB ×

Generate Project alt + ⏎

Don't know what to look for? Want more options? Switch to the full version.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

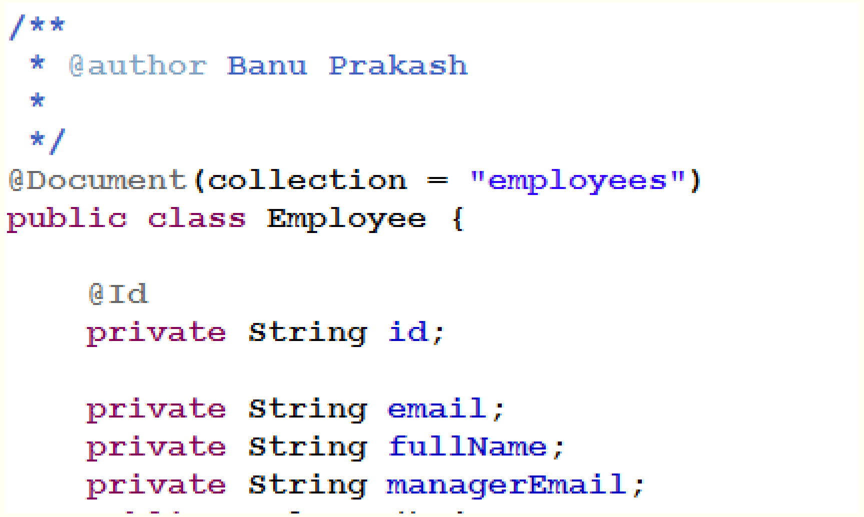
Import Maven Project generated from https://start.spring.io/



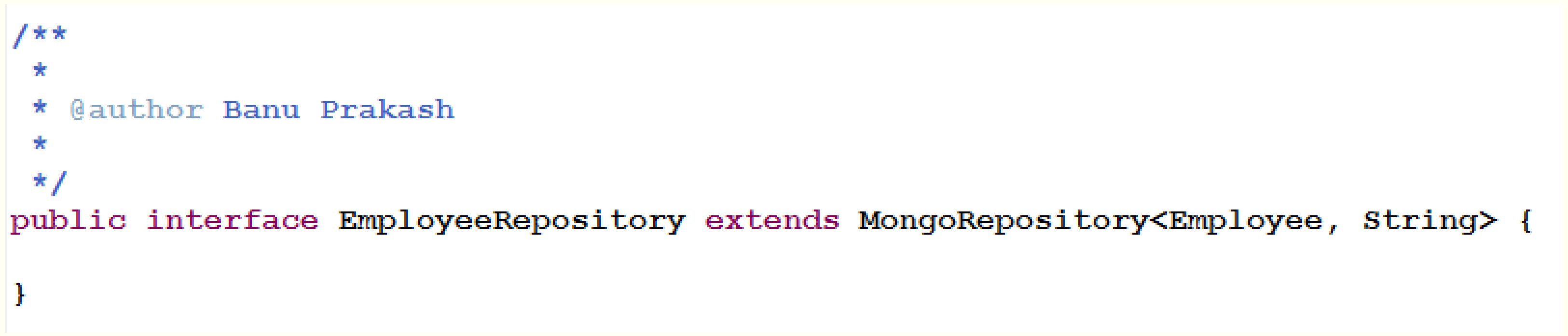
Root Directory: G:\docker_demo

[✓] /docker_demo/pom.xml com.banu.spring:docker_demo:0.0.1-SNAPSHOT:jar

Document and Repository

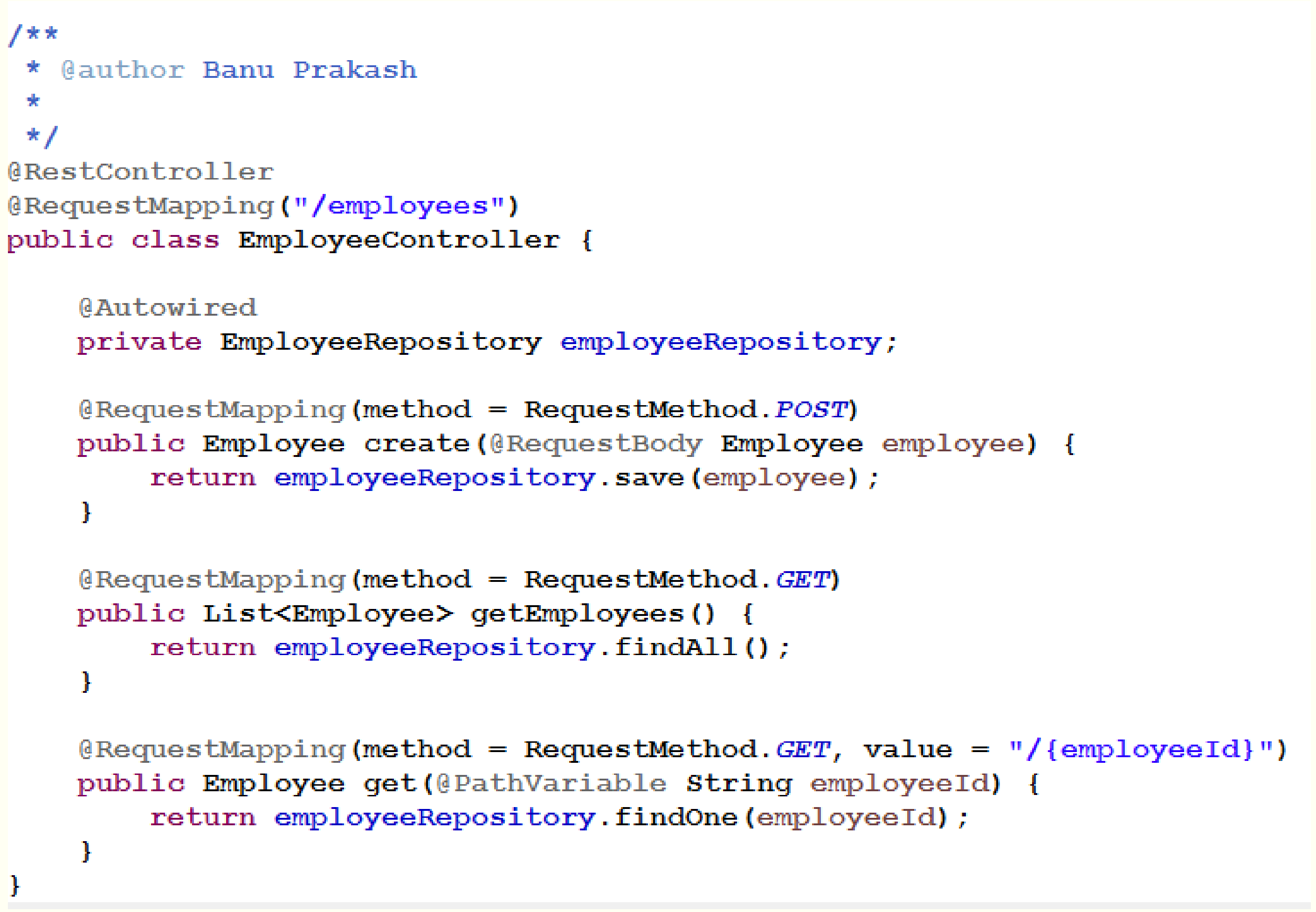


```
/**
 * @author Banu Prakash
 *
 */
@Document(collection = "employees")
public class Employee {

    @Id
    private String id;

    private String email;
    private String fullName;
    private String managerEmail;
```



```
/**
 *
 * @author Banu Prakash
 *
 */
public interface EmployeeRepository extends MongoRepository<Employee, String> {

}
```

RestController



```
/**
 * @author Banu Prakash
 *
 */
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    private EmployeeRepository employeeRepository;

    @RequestMapping(method = RequestMethod.POST)
    public Employee create(@RequestBody Employee employee) {
        return employeeRepository.save(employee);
    }

    @RequestMapping(method = RequestMethod.GET)
    public List<Employee> getEmployees() {
        return employeeRepository.findAll();
    }

    @RequestMapping(method = RequestMethod.GET, value = "/{employeeId}")
    public Employee get(@PathVariable String employeeId) {
        return employeeRepository.findOne(employeeId);
    }
}
```

Execute the application

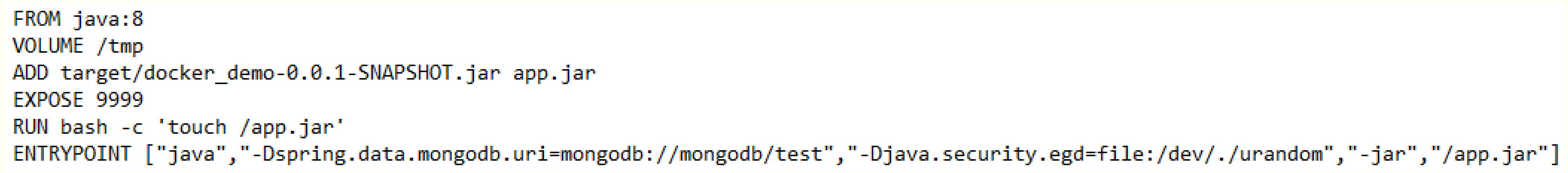
- java -Dspring.data.mongodb.uri=mongodb://192.168.99.100:32768/test target/..jar

- Perform CRUD using POSTMAN/any other REST client
- http://localhost:8080/employees

- docker exec -it mongodb sh
- # mongo
- > use test
- switched to db test
- > show collections
- > db.employees.find()

TURNING A BOOT INTO A CONTAINER

- Dockerfile

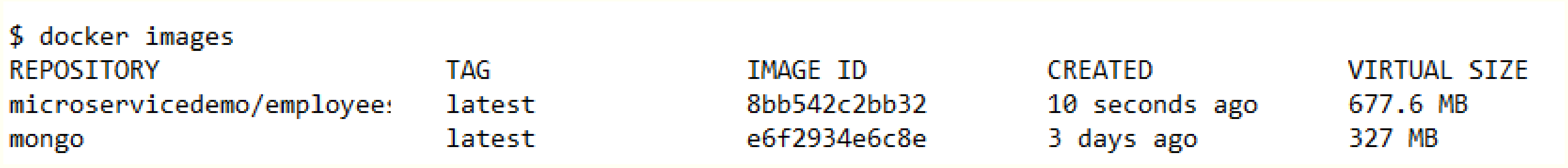


```
FROM java:8
VOLUME /tmp
ADD target/docker_demo-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 9999
RUN bash -c 'touch /app.jar'
ENTRYPOINT ["java","-Dspring.data.mongodb.uri=mongodb://mongodb/test","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

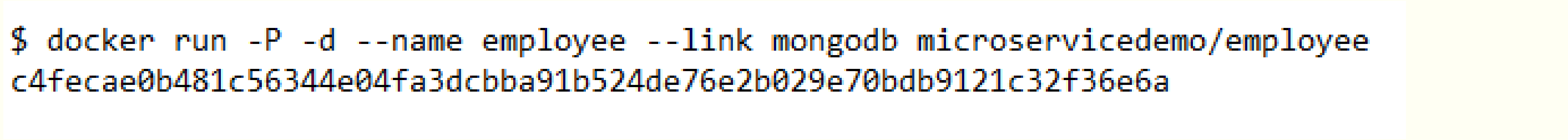
- We start with a “standard” image that already includes Java 8 installed (called “Java” and tagged “8”)
- We then define that a volume named /tmp should exist. We added a VOLUME pointing to "/tmp" because that is where a Spring Boot application creates working directories for Tomcat by default
- We then add a file from the local filesystem, naming it “app.jar.”
- We run a command on the system to “touch” the file. This ensures a file modification date on the app.jar file
- The ENTRYPOINT command is the “what to run to ‘start'” command- we run Java, setting our Spring Mongo property and a quick additional property to speed up the Tomcat startup time, and then point it at our jar

TURNING A BOOT INTO A CONTAINER

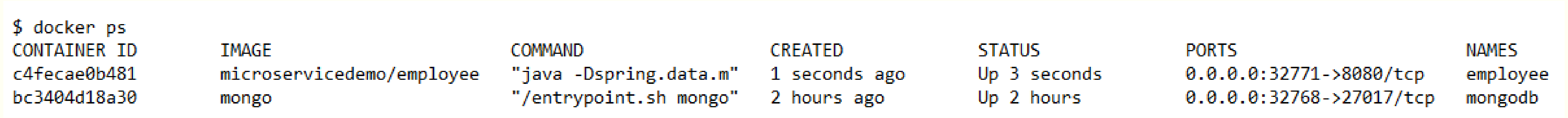
$ docker build -t microservicedemo/employee .



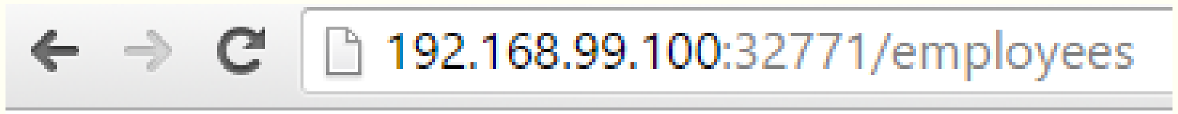
```
$ docker images
REPOSITORY                   TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
microservicedemo/employee    latest              8bb542c2bb32        10 seconds ago      677.6 MB
mongo                        latest              e6f2934e6c8e        3 days ago          327 MB
```



```
$ docker run -P -d --name employee --link mongodb microservicedemo/employee
c4fecae0b481c56344e04fa3dcbba91b524de76e2b029e70bdb9121c32f36e6a
```



```
$ docker ps
CONTAINER ID        IMAGE                       COMMAND                  CREATED             STATUS              PORTS                        NAMES
c4fecae0b481        microservicedemo/employee   "java -Dspring.data.m"   1 seconds ago       Up 3 seconds        0.0.0.0:32771->8080/tcp      employee
bc3404d18a30        mongo                       "/entrypoint.sh mongo"   2 hours ago         Up 2 hours          0.0.0.0:32768->27017/tcp     mongodb
```



192.168.99.100:32771/employees

TURNING A BOOT INTO A CONTAINER

- clear out our old running container with a new version:
- **$ docker build -t microservicedemo/employee .**
- $ docker stop employee
- $ docker rm employee
- $ docker run -P -d --name employee --link mongodb microservicedemo/employee

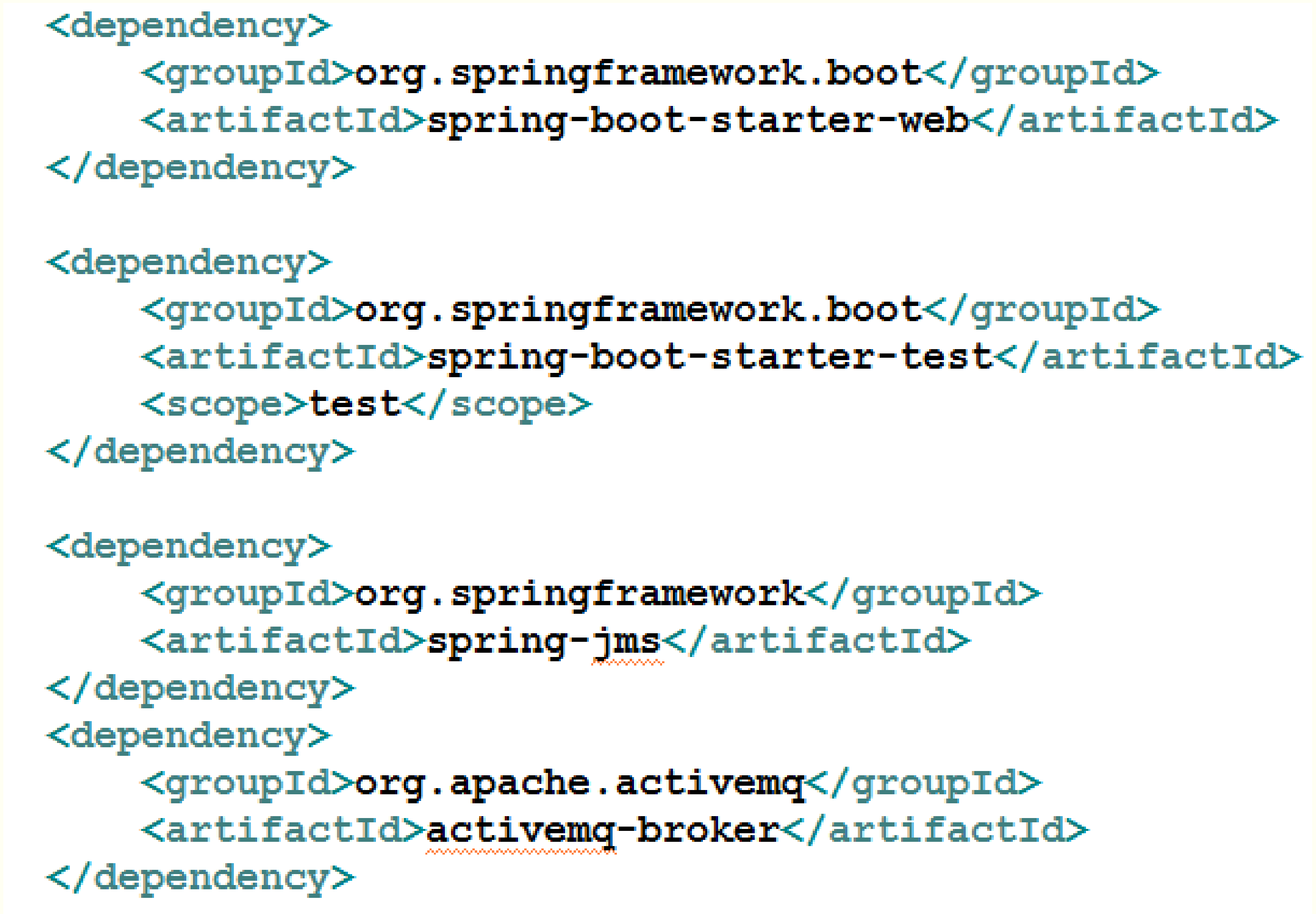
Spring JMS and Active MQ

- Spring JMS(Java Message Service) is a powerful mechanism to integrate in distributed system.
  ActiveMq is a Java Open Source, it is simple JMS solution for concurrent, consumers and producers architecture in integrated development.

ActiveMQ on Docker

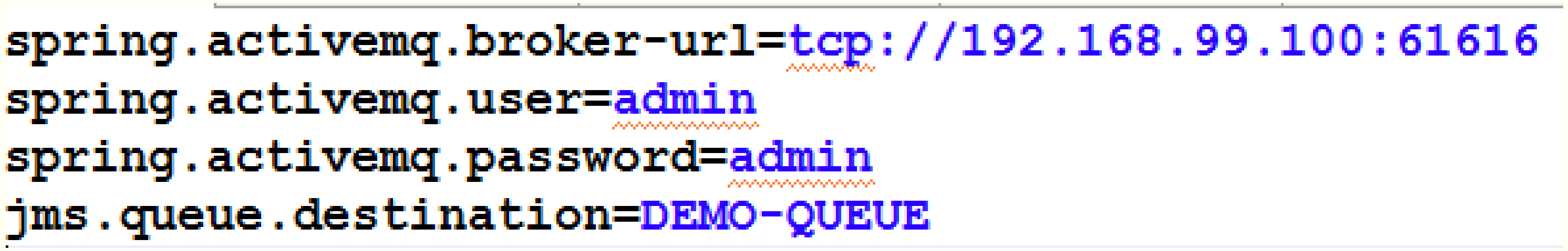
- docker pull webcenter/activemq:latest
- Or
- git clone https://github.com/disaster37/activemq.git cd activemq docker build --tag="$USER/activemq" .
- docker run -p 61616:61616 -p 8161:8161 -t webcenter/activemq

Dependencies



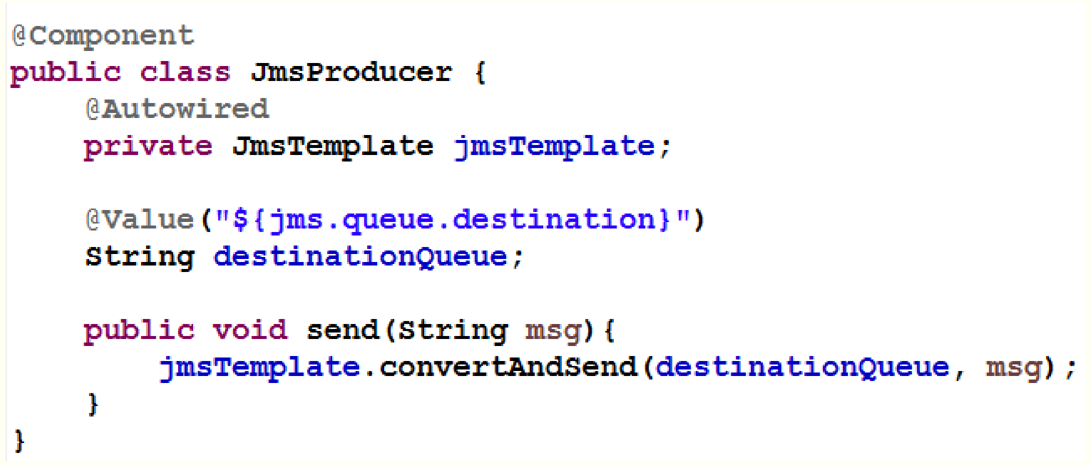
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jms</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-broker</artifactId>
</dependency>
```

application.properties

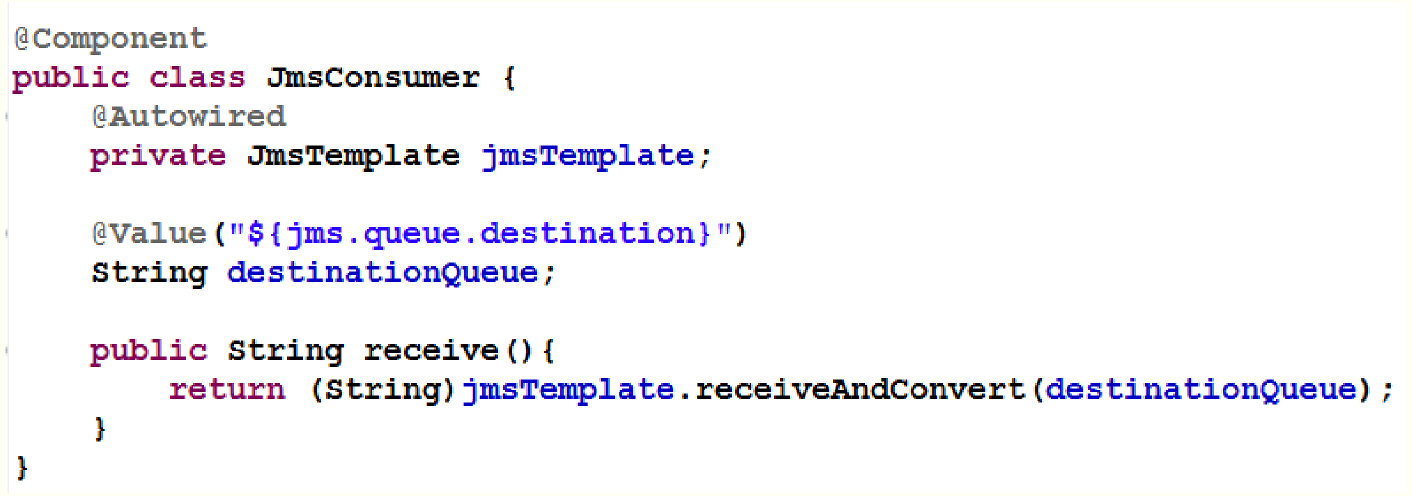


```
spring.activemq.broker-url=tcp://192.168.99.100:61616
spring.activemq.user=admin
spring.activemq.password=admin
jms.queue.destination=DEMO-QUEUE
```

Producer and Consumer

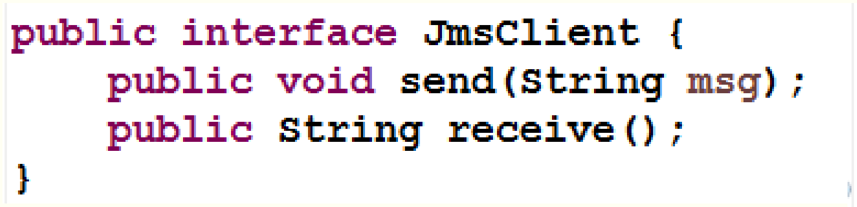


```
@Component
public class JmsProducer {
    @Autowired
    private JmsTemplate jmsTemplate;

    @Value("${jms.queue.destination}")
    String destinationQueue;

    public void send(String msg){
        jmsTemplate.convertAndSend(destinationQueue, msg);
    }
}
```

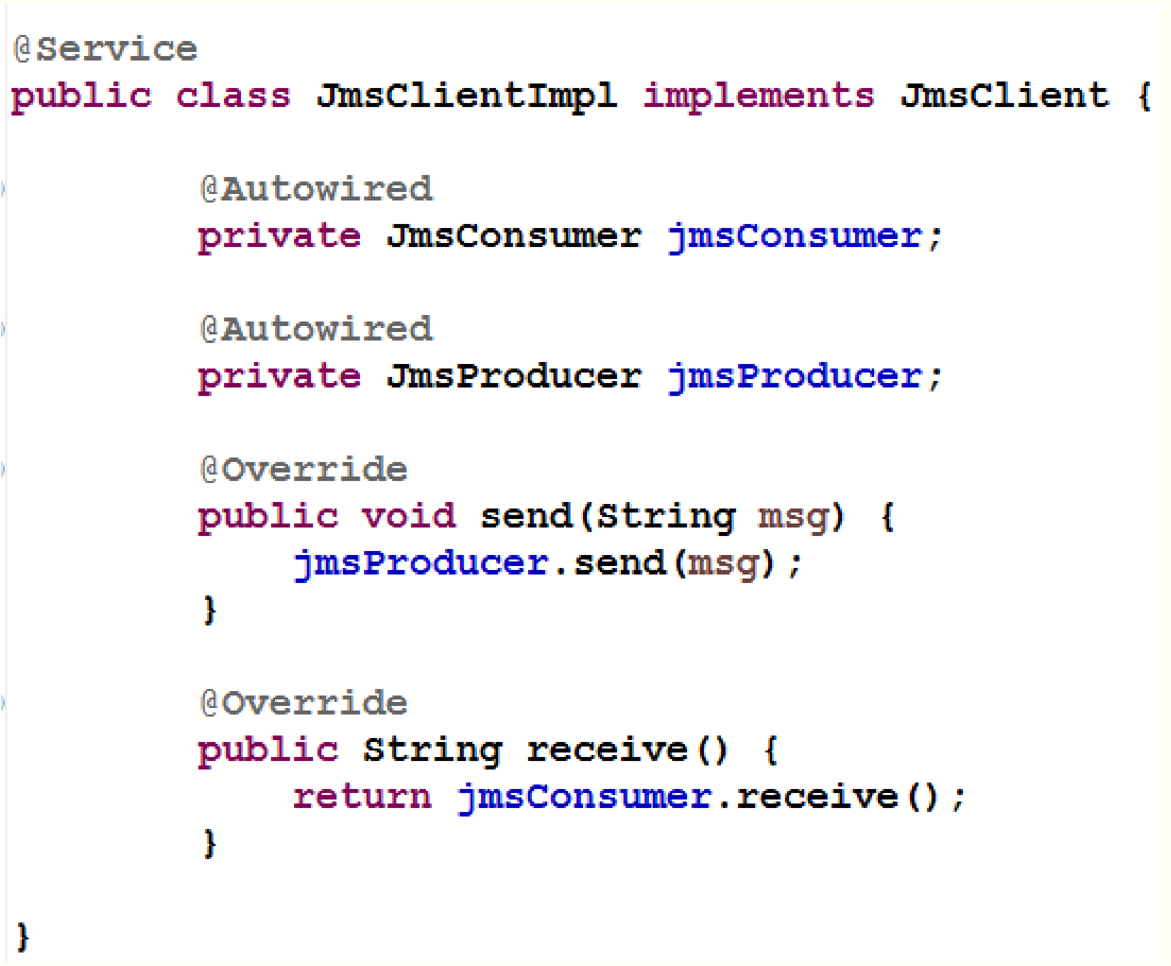


```
@Component
public class JmsConsumer {
    @Autowired
    private JmsTemplate jmsTemplate;

    @Value("${jms.queue.destination}")
    String destinationQueue;

    public String receive(){
        return (String)jmsTemplate.receiveAndConvert(destinationQueue);
    }
}
```

JMSClient

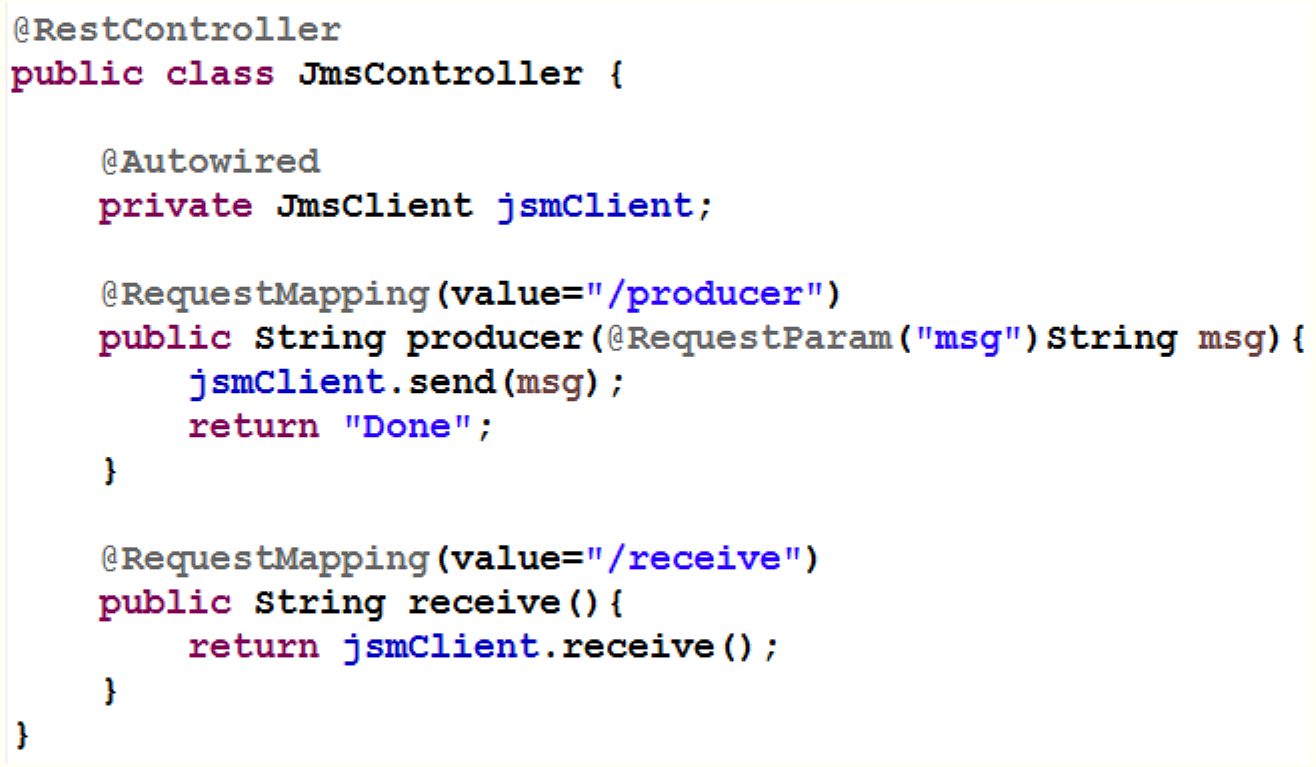


```
public interface JmsClient {
    public void send(String msg);
    public String receive();
}
```

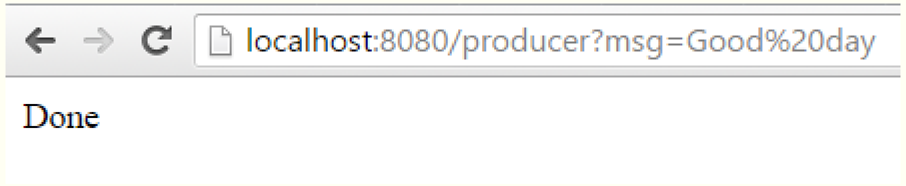


```
@Service
public class JmsClientImpl implements JmsClient {

        @Autowired
        private JmsConsumer jmsConsumer;

        @Autowired
        private JmsProducer jmsProducer;

        @Override
        public void send(String msg) {
            jmsProducer.send(msg);
        }

        @Override
        public String receive() {
            return jmsConsumer.receive();
        }

}
```

Controller

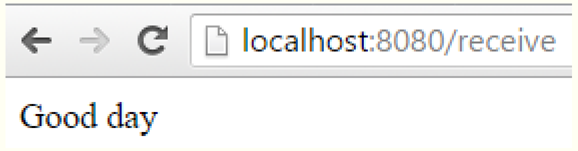


```
@RestController
public class JmsController {

    @Autowired
    private JmsClient jsmClient;

    @RequestMapping(value="/producer")
    public String producer(@RequestParam("msg")String msg){
        jsmClient.send(msg);
        return "Done";
    }

    @RequestMapping(value="/receive")
    public String receive(){
        return jsmClient.receive();
    }
}
```



localhost:8080/producer?msg=Good%20day

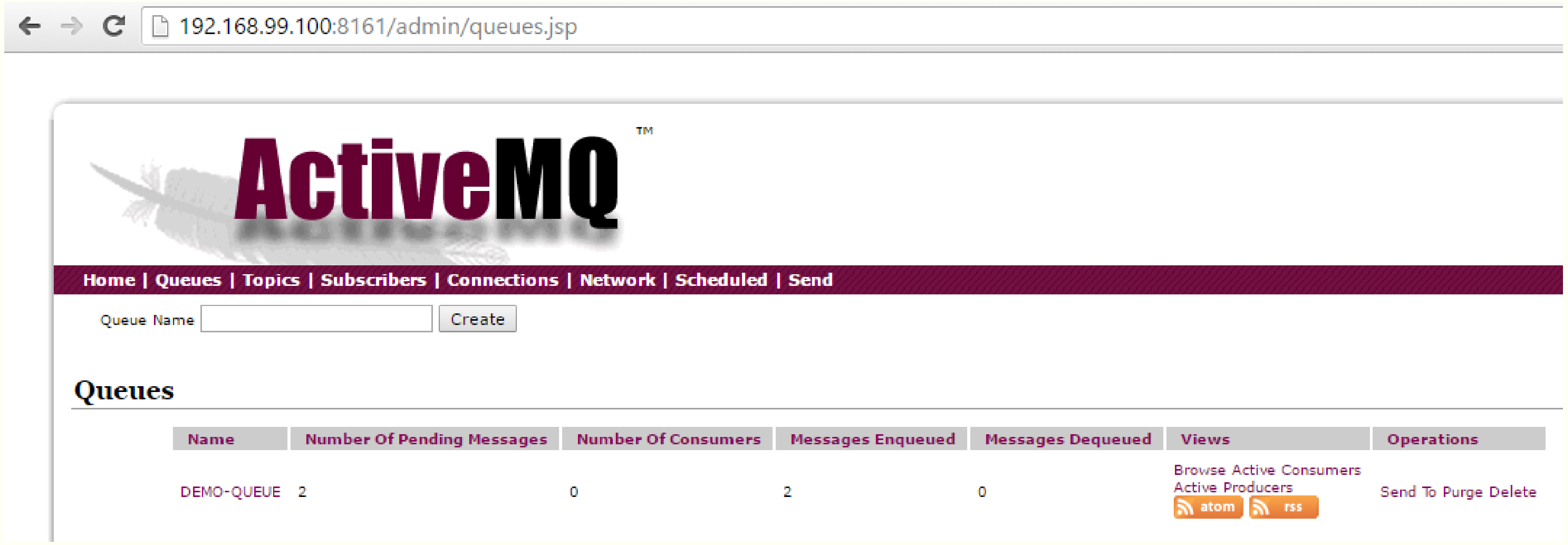
Done



localhost:8080/receive

Good day

ActiveMQ Admin console


192.168.99.100:8161/admin/queues.jsp

ActiveMQ™
Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send
Queue Name
Create

Queues

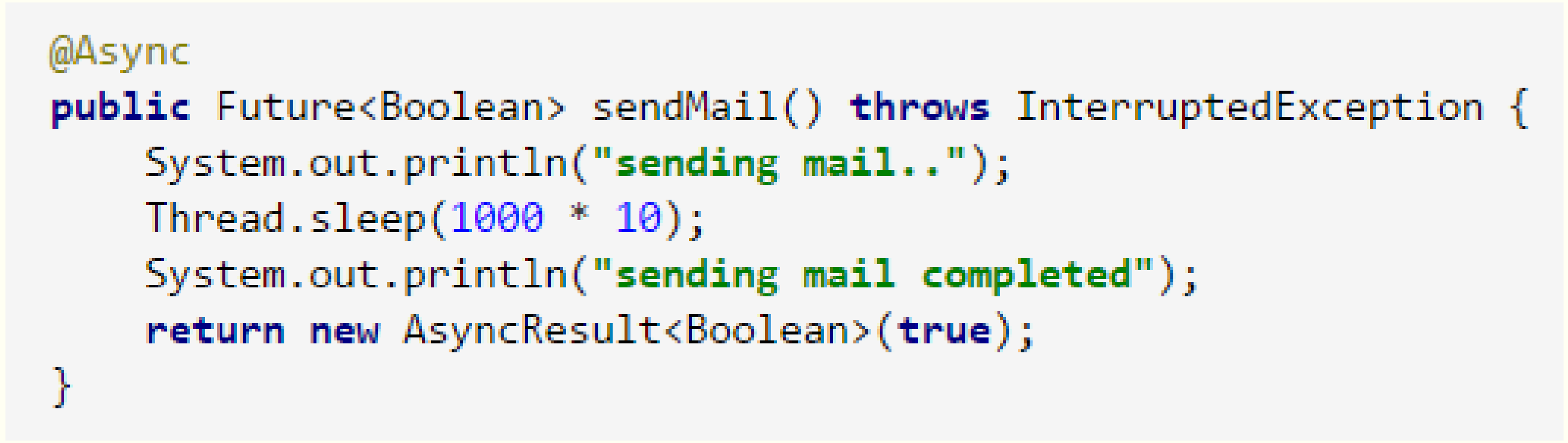
Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
DEMO-QUEUE	2	0	2	0	Browse Active Consumers Active Producers atom rss	Send To Purge Delete

Asynchronous Service With Spring @Async And Java Future

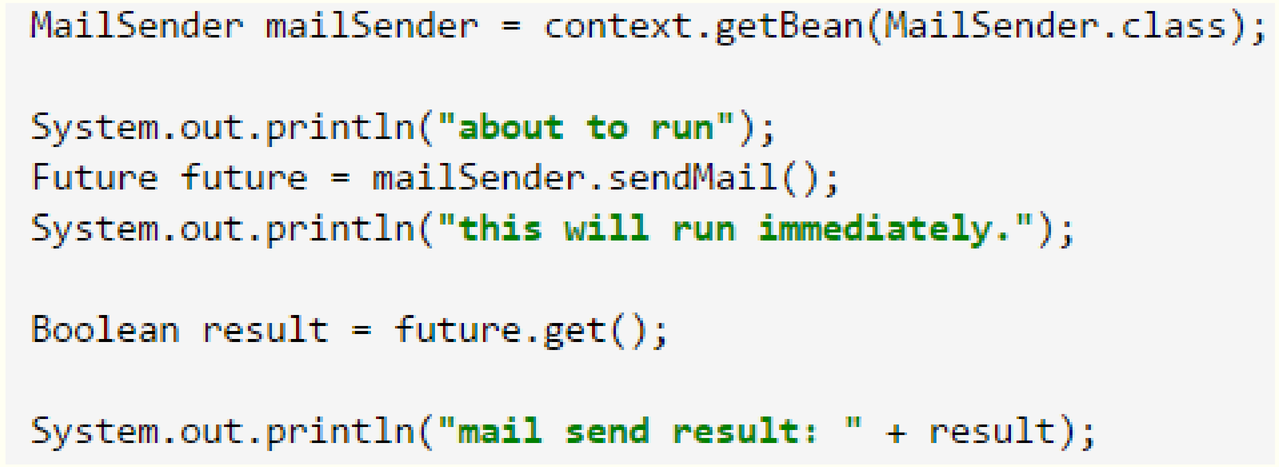
@Async

- Annotation that marks a method as a candidate for asynchronous execution. Can also be used at the type level, in which case all of the type's methods are considered as asynchronous.
- In terms of target method signatures, any parameter types are supported.
- However, the return type is constrained to either void or Future.
- The Future handle returned from the proxy will be an actual asynchronous Future that can be used to track the result of the asynchronous method execution.
- However, since the target method needs to implement the same signature, it will have to return a temporary.

Example



```
@Async
public Future<Boolean> sendMail() throws InterruptedException {
    System.out.println("sending mail..");
    Thread.sleep(1000 * 10);
    System.out.println("sending mail completed");
    return new AsyncResult<Boolean>(true);
}
```



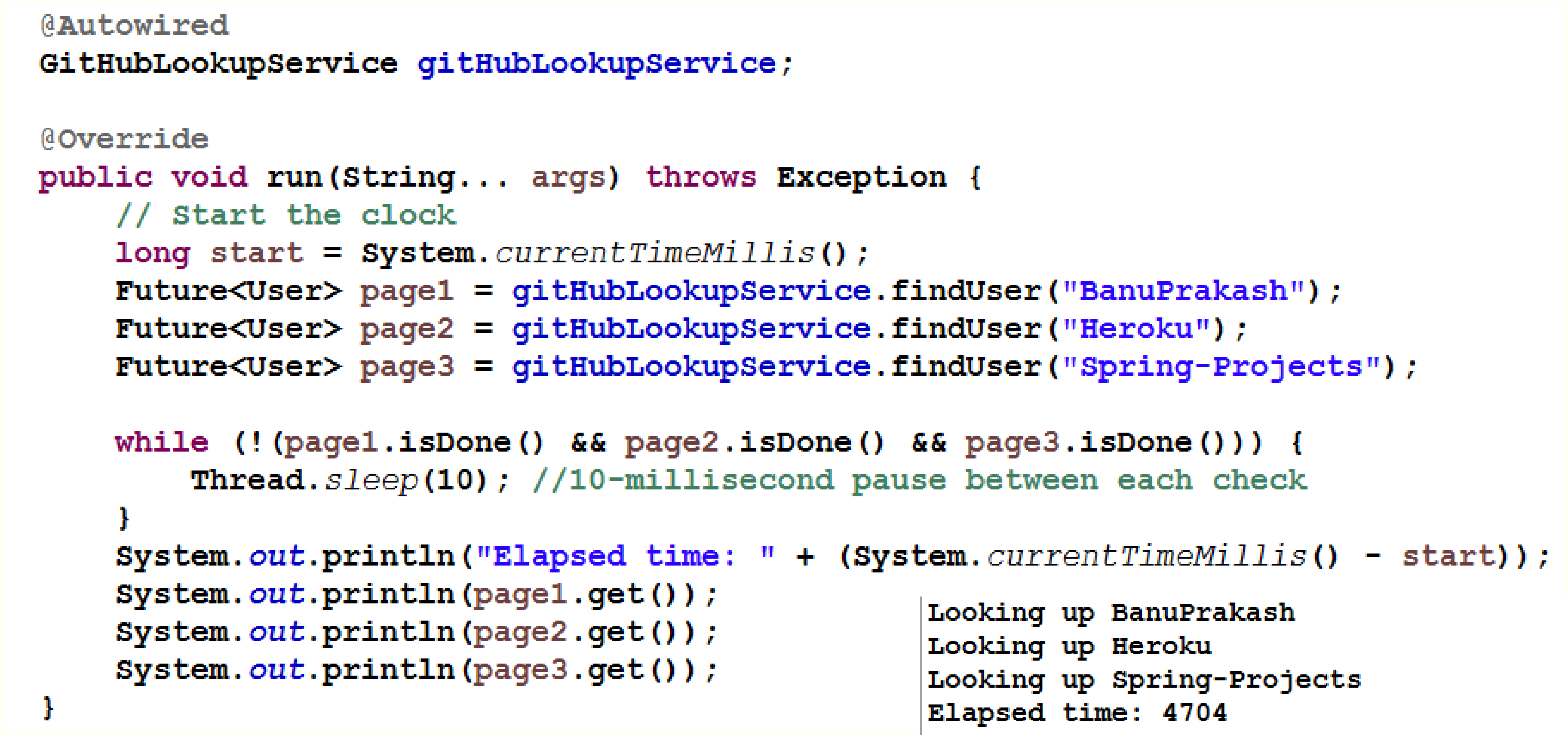
```
MailSender mailSender = context.getBean(MailSender.class);

System.out.println("about to run");
Future future = mailSender.sendMail();
System.out.println("this will run immediately.");

Boolean result = future.get();

System.out.println("mail send result: " + result);
```

GitHub lookup service

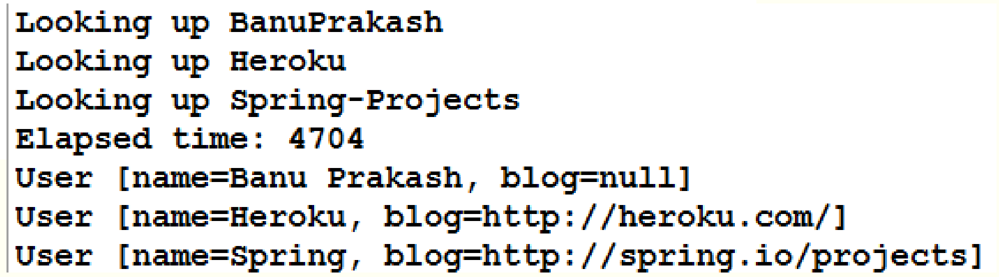
```
/**
 *
 * @author Banu Prakash
 *
 */
@Service
public class GitHubLookupService {
    RestTemplate restTemplate = new RestTemplate();
    @Async
    public Future<User> findUser(String user) throws InterruptedException {
        System.out.println("Looking up " + user);
        User results =
                restTemplate.getForObject("https://api.github.com/users/" + user, User.class);
        Thread.sleep(1000L);
        return new AsyncResult<User>(results);
    }
}
```

```
public class User {
    private String name;
    private String blog;
```

Application



```
@Autowired
GitHubLookupService gitHubLookupService;

@Override
public void run(String... args) throws Exception {
    // Start the clock
    long start = System.currentTimeMillis();
    Future<User> page1 = gitHubLookupService.findUser("BanuPrakash");
    Future<User> page2 = gitHubLookupService.findUser("Heroku");
    Future<User> page3 = gitHubLookupService.findUser("Spring-Projects");

    while (!(page1.isDone() && page2.isDone() && page3.isDone())) {
        Thread.sleep(10); //10-millisecond pause between each check
    }
    System.out.println("Elapsed time: " + (System.currentTimeMillis() - start));
    System.out.println(page1.get());
    System.out.println(page2.get());
    System.out.println(page3.get());
}
```



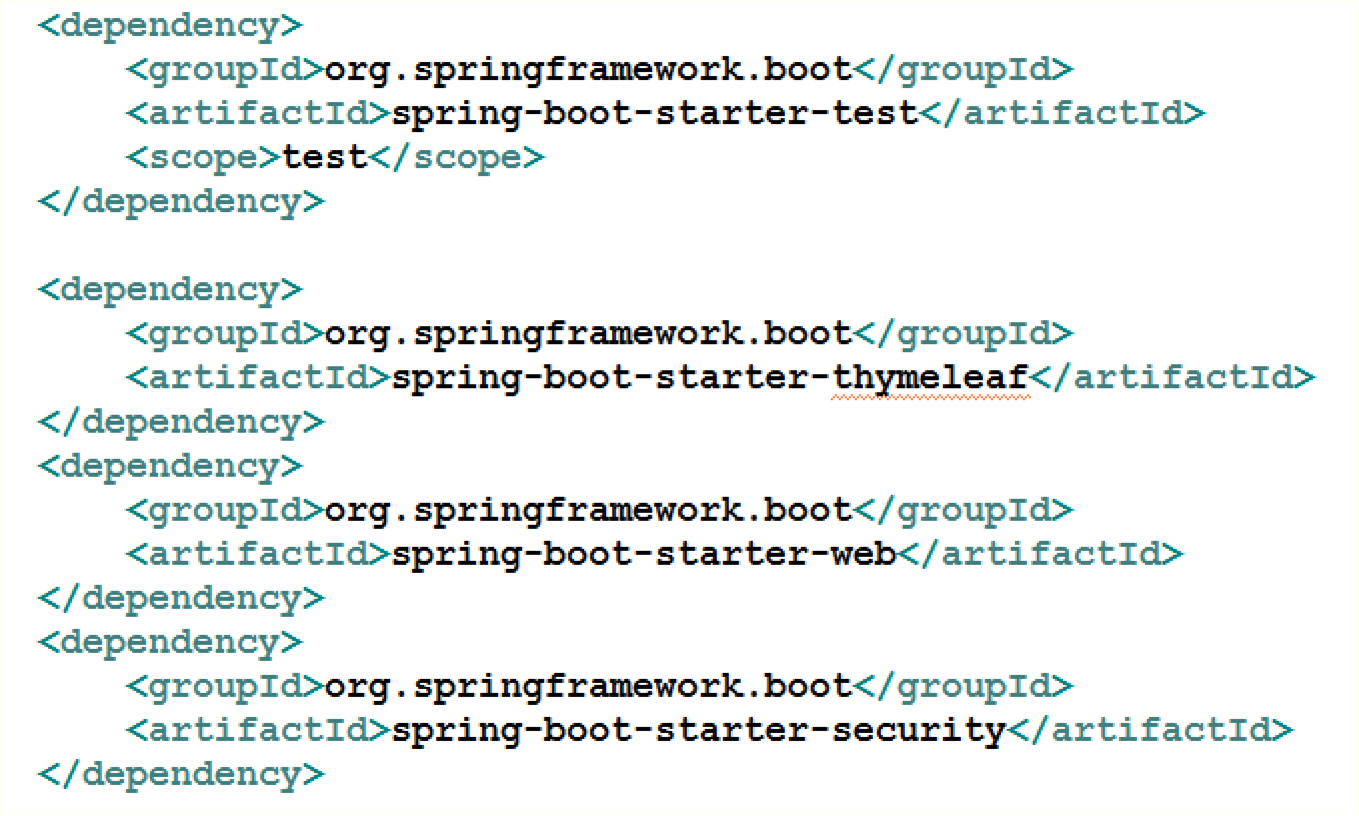
```
Looking up BanuPrakash
Looking up Heroku
Looking up Spring-Projects
Elapsed time: 4704
User [name=Banu Prakash, blog=null]
User [name=Heroku, blog=http://heroku.com/]
User [name=Spring, blog=http://spring.io/projects]
```

SPRING SECUIRTY AND SOCIAL

banuprakashc@yahoo.co.in

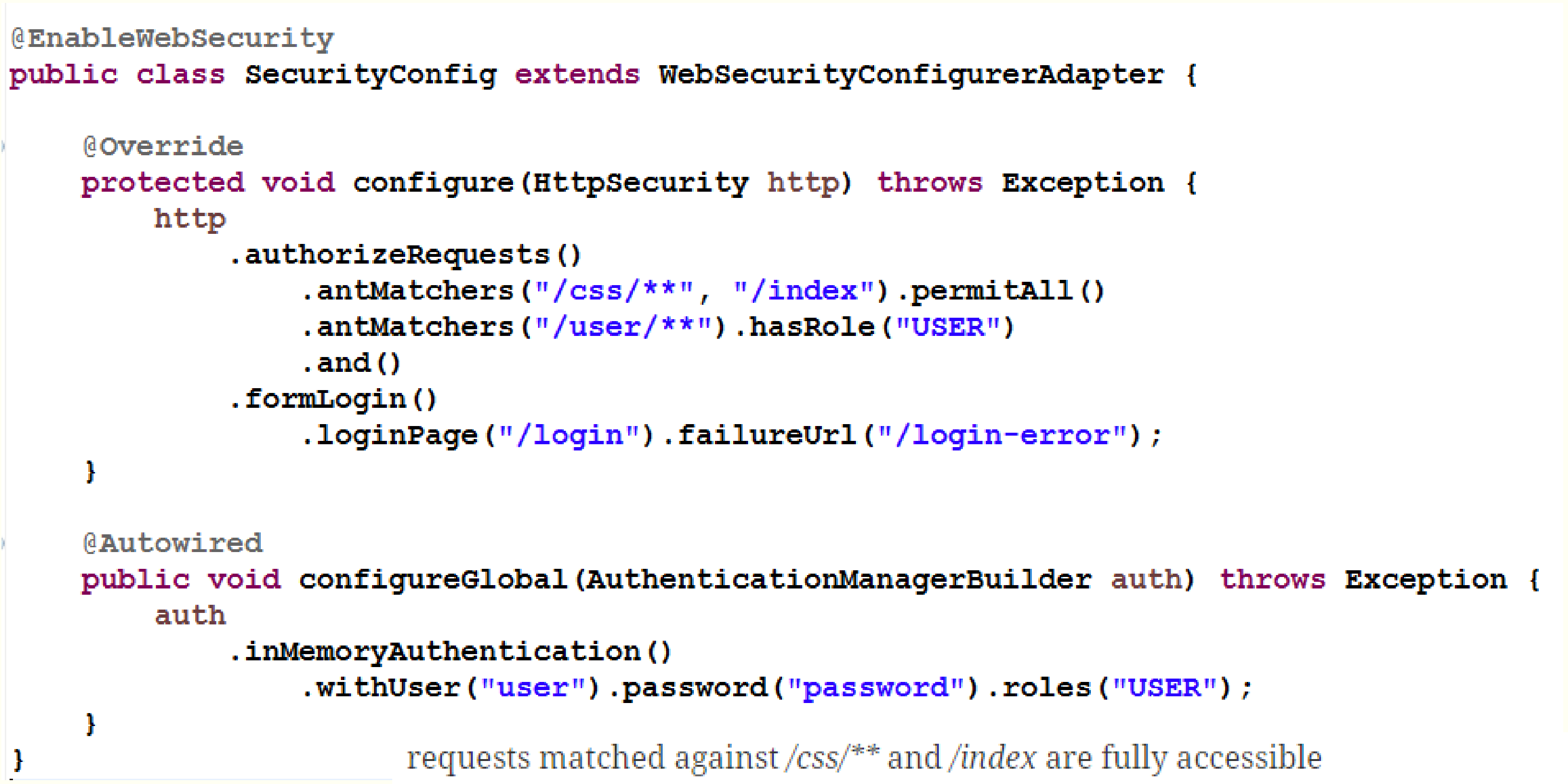
Spring Security With Boot

- Adding dependencies

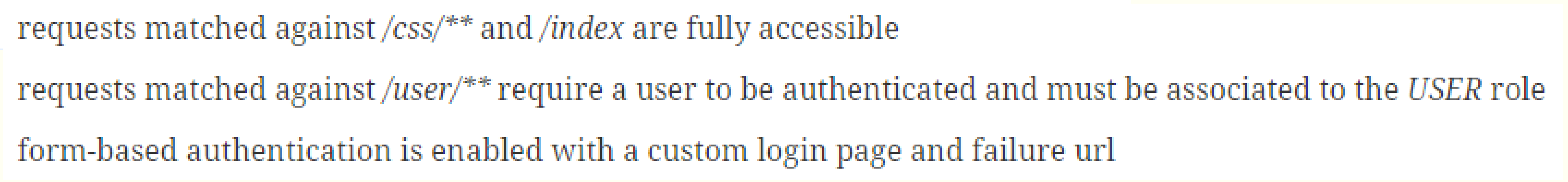


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Spring Security configuration



```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/css/**", "/index").permitAll()
                .antMatchers("/user/**").hasRole("USER")
                .and()
            .formLogin()
                .loginPage("/login").failureUrl("/login-error");
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
                .withUser("user").password("password").roles("USER");
    }
}
```

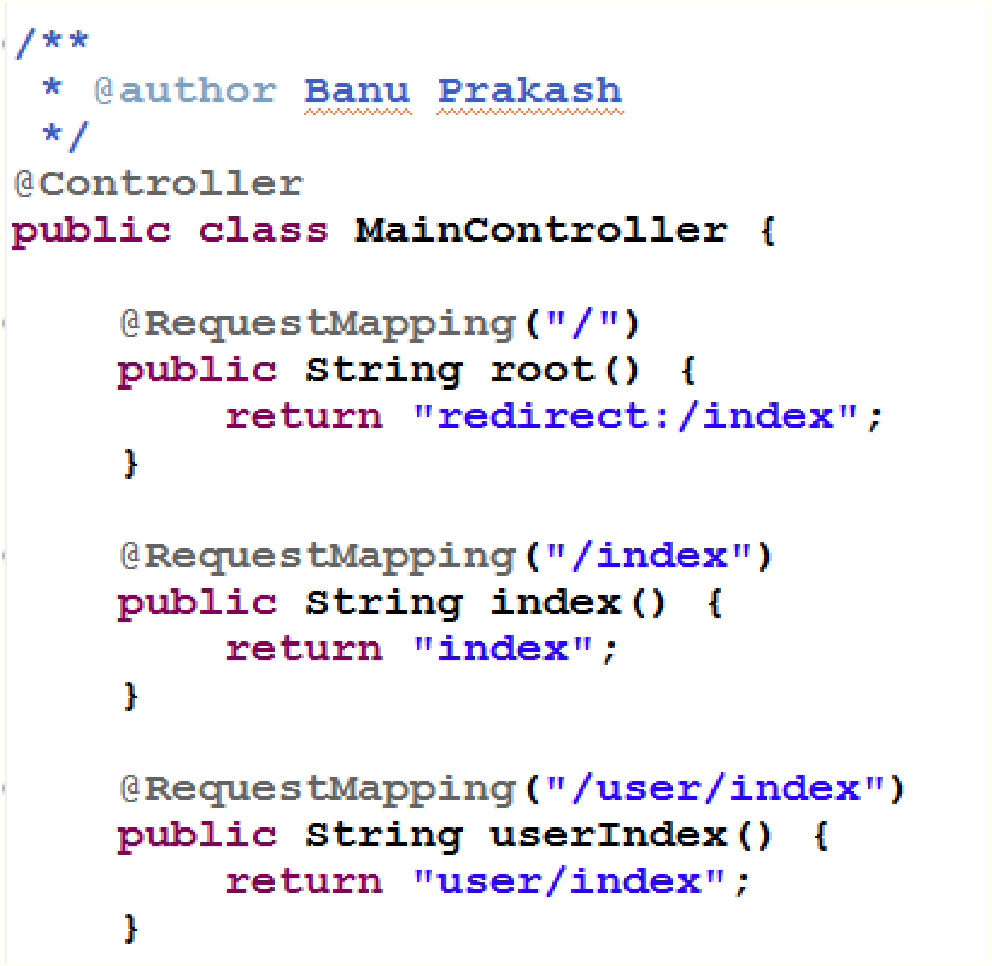


requests matched against */css/*** and */index* are fully accessible

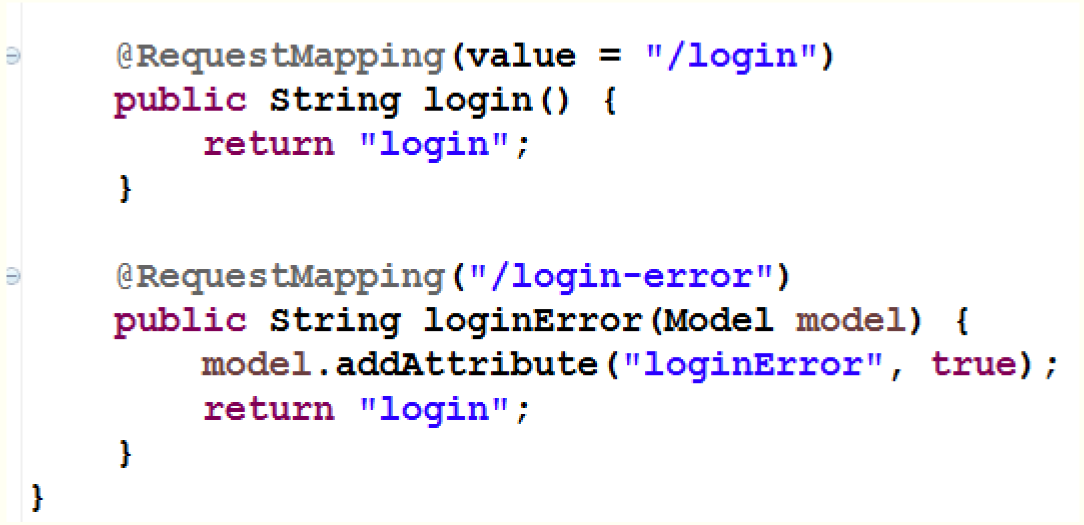
requests matched against */user/*** require a user to be authenticated and must be associated to the *USER* role

form-based authentication is enabled with a custom login page and failure url

Controller

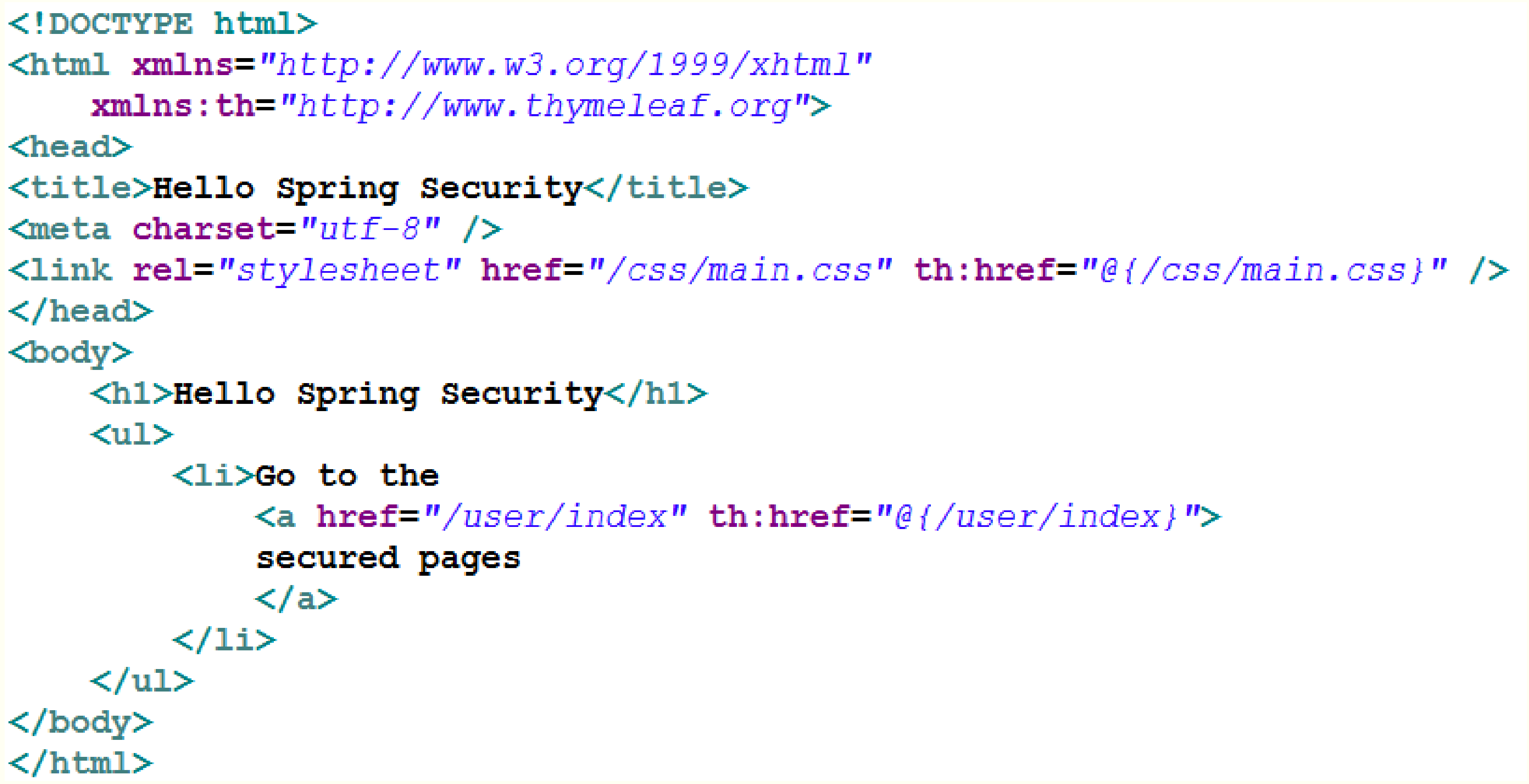


```java
/**
 * @author Banu Prakash
 */
@Controller
public class MainController {

    @RequestMapping("/")
    public String root() {
        return "redirect:/index";
    }

    @RequestMapping("/index")
    public String index() {
        return "index";
    }

    @RequestMapping("/user/index")
    public String userIndex() {
        return "user/index";
    }
```



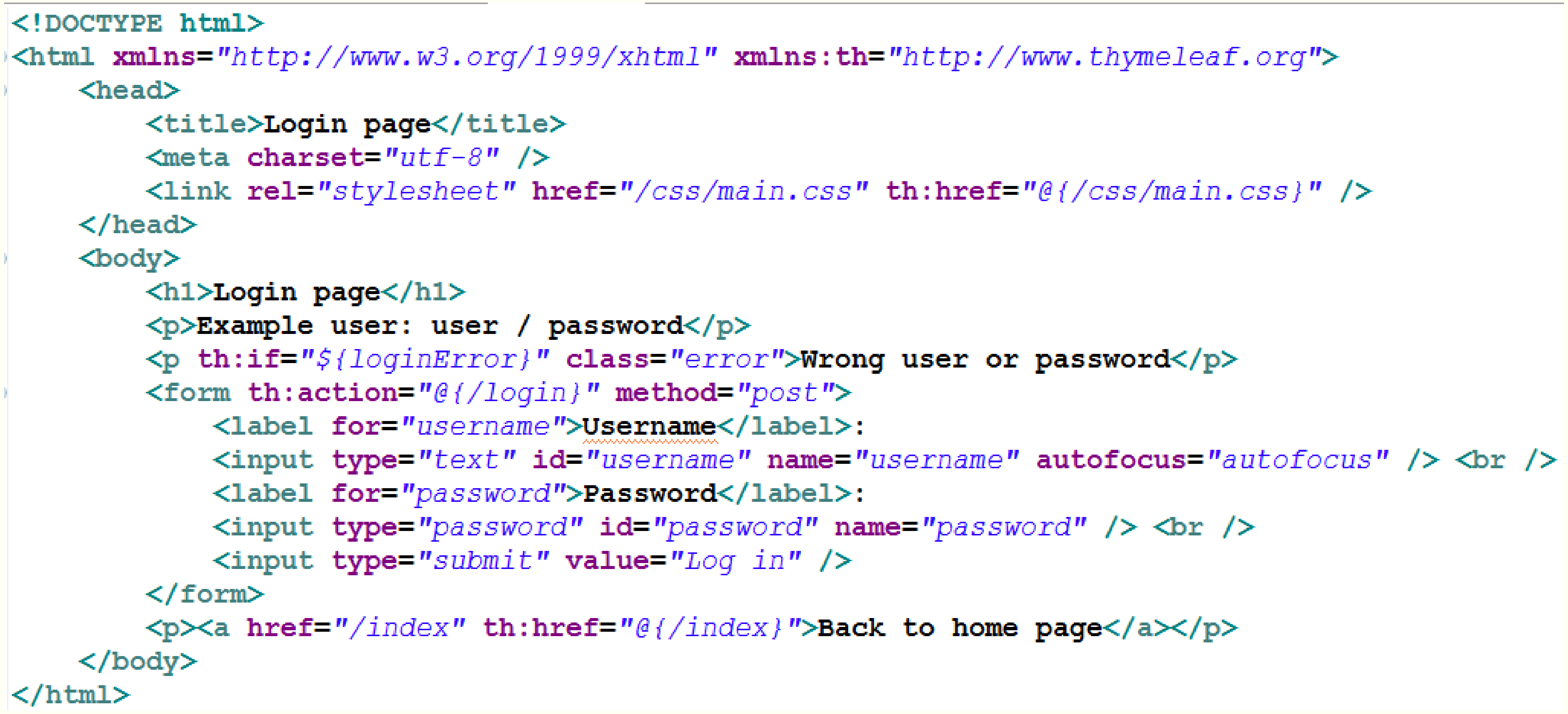
```java
    @RequestMapping(value = "/login")
    public String login() {
        return "login";
    }

    @RequestMapping("/login-error")
    public String loginError(Model model) {
        model.addAttribute("loginError", true);
        return "login";
    }
}
```

The index.html page



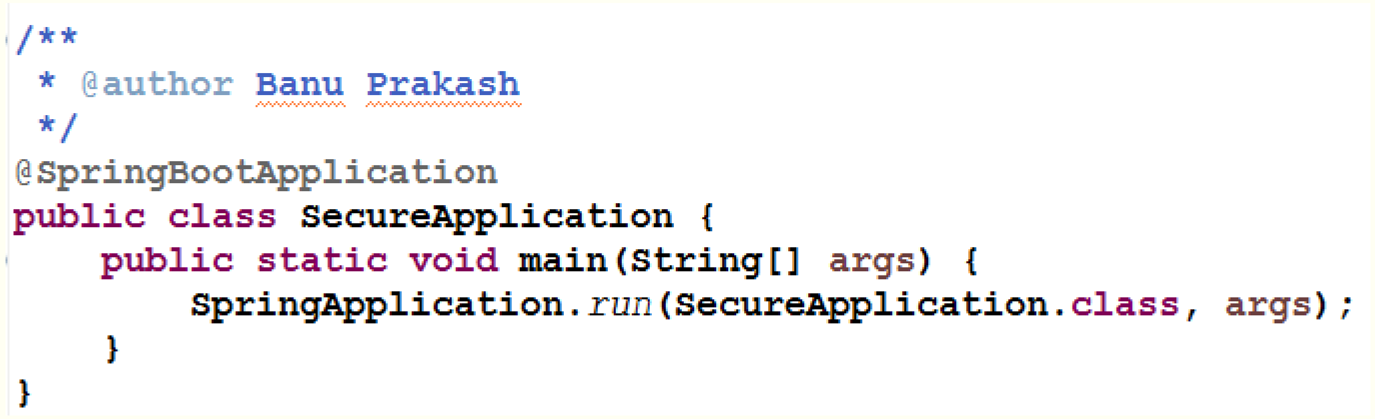
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:th="http://www.thymeleaf.org">
<head>
<title>Hello Spring Security</title>
<meta charset="utf-8" />
<link rel="stylesheet" href="/css/main.css" th:href="@{/css/main.css}" />
</head>
<body>
    <h1>Hello Spring Security</h1>
    <ul>
        <li>Go to the
            <a href="/user/index" th:href="@{/user/index}">
            secured pages
            </a>
        </li>
    </ul>
</body>
</html>
```

The login.html page

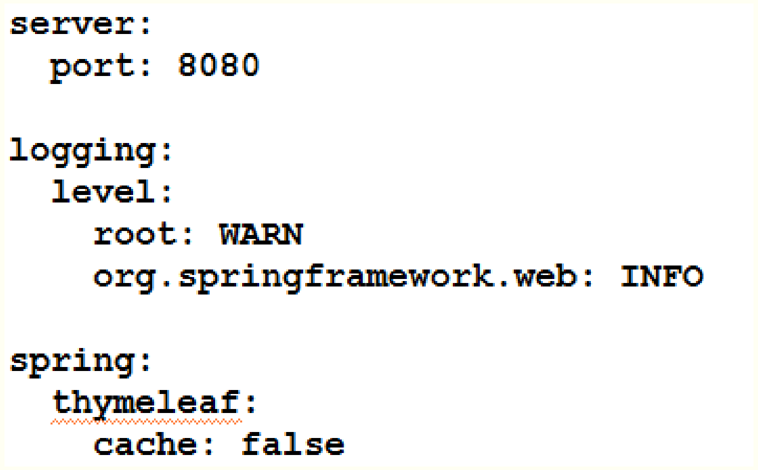


```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Login page</title>
        <meta charset="utf-8" />
        <link rel="stylesheet" href="/css/main.css" th:href="@{/css/main.css}" />
    </head>
    <body>
        <h1>Login page</h1>
        <p>Example user: user / password</p>
        <p th:if="${loginError}" class="error">Wrong user or password</p>
        <form th:action="@{/login}" method="post">
            <label for="username">Username</label>:
            <input type="text" id="username" name="username" autofocus="autofocus" /> <br />
            <label for="password">Password</label>:
            <input type="password" id="password" name="password" /> <br />
            <input type="submit" value="Log in" />
        </form>
        <p><a href="/index" th:href="@{/index}">Back to home page</a></p>
    </body>
</html>
```

Main App and application.yml



```java
/**
 * @author Banu Prakash
 */
@SpringBootApplication
public class SecureApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecureApplication.class, args);
    }
}
```



```yaml
server:
  port: 8080

logging:
  level:
    root: WARN
    org.springframework.web: INFO

spring:
  thymeleaf:
    cache: false
```

Why Social Login?

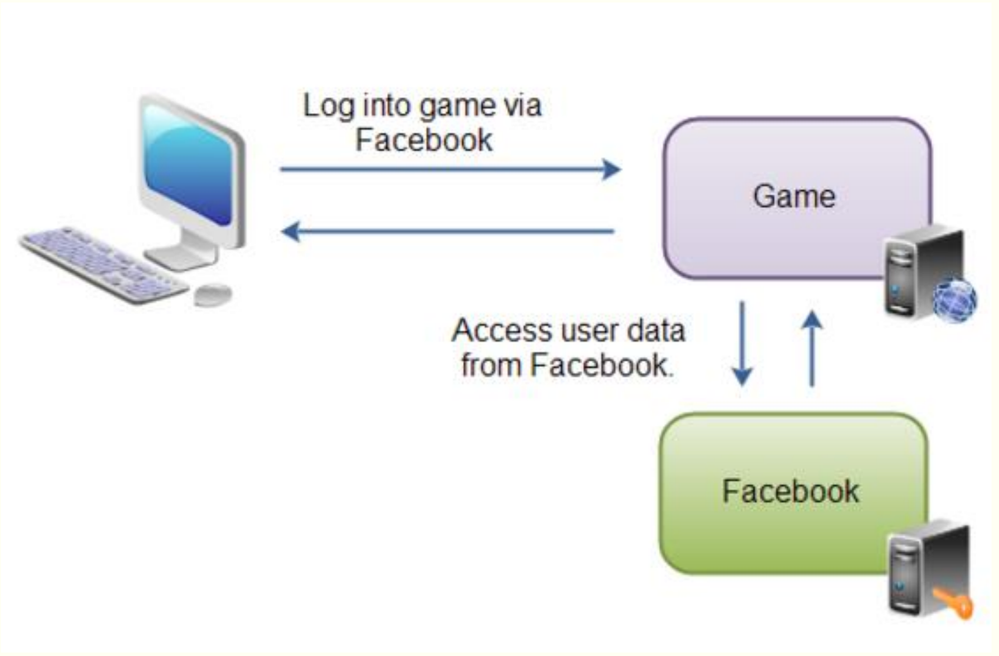
- Handling user registration and authentication on a website is hard, both for the users required to remember yet another pair of username/password and for developers implementing a secure handling of the user credentials.
- With requirements of more sophisticated login-methods such as two-factor authentication and single sign-on it gets even worse.
- A few years ago websites started to use social login, i.e. delegating the sign in process to social network services such as Facebook, Twitter and Google+ allowing users to sign in to the website using their social network accounts.
- Initially social login was rather complex to setup and a number of commercial offerings from social network integration providers evolved to simplify the setup.
- Today a free and open source based alternative exists, the Spring Social project.

Social login benefits:

- The benefits of social login are two-folded.
  - Website users can benefit from single sign-on using their social networking accounts to identify themselves for the website.
  - Website developers can use social login to automatically create local account for the user in the website based on the login information from the selected social network. As a website developer you can delegate the authentication process to the social network services

OAuth 2.0

- OAuth 2.0 is an open authorization protocol which enables applications to access each others data.
- Example:
  - The user accesses the game web application.
  - The game web application asks the user to login to the game via Facebook.
  - The user logs into Facebook, and is sent back to the game.
  - The game can now access the users data in Facebook, and call functions in Facebook on behalf of the user (e.g. posting status updates).



OAuth2

- **Resource server**: The server hosting user-owned resources that are protected by OAuth. This is typically an API provider that holds and protects data such as photos, videos, calendars, or contacts.

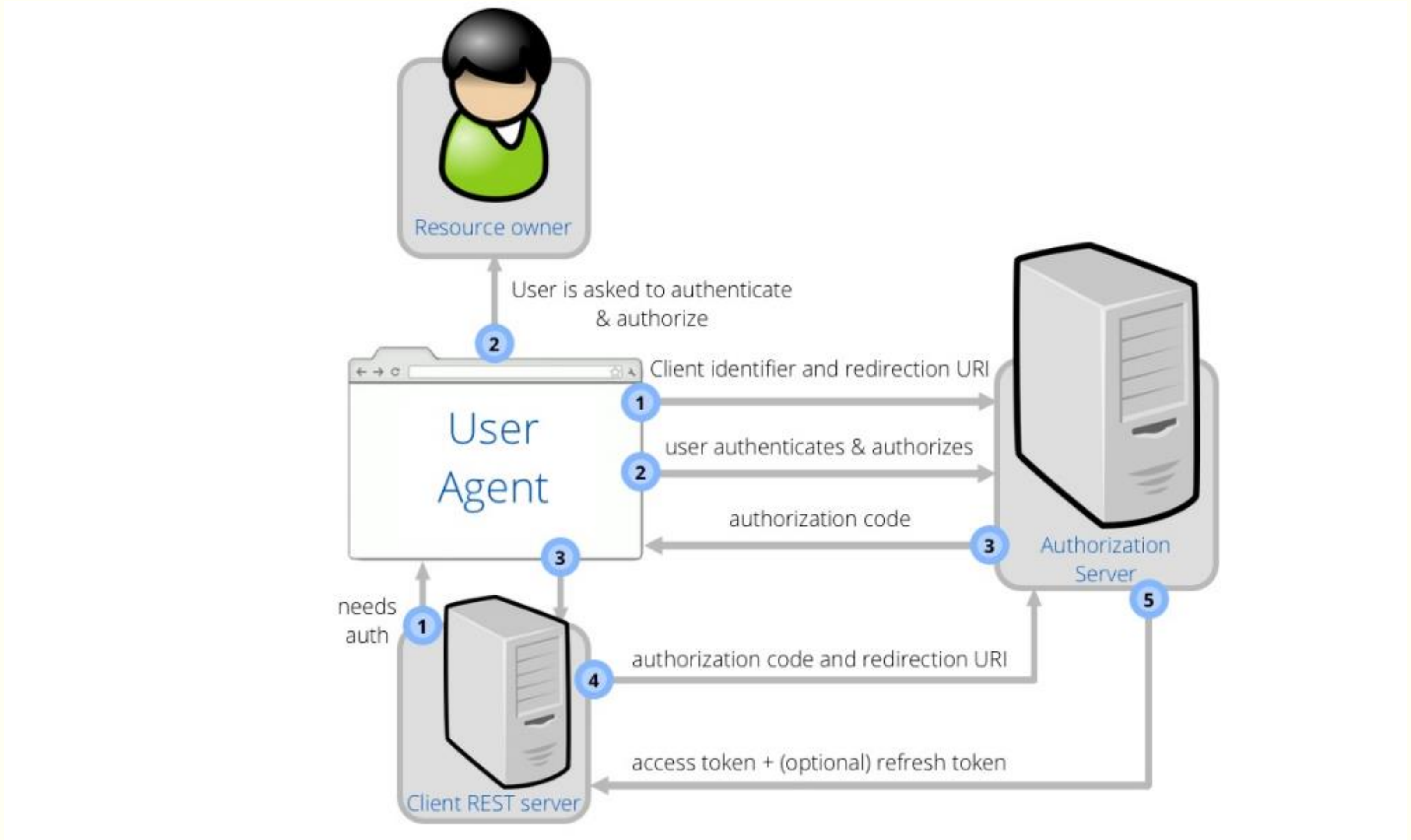
- **Resource owner**: Typically the user of an application, the resource owner has the ability to grant access to their own data hosted on the resource server.

- **Client (or client application)**: An application making API requests to perform actions on protected resources on behalf of the resource owner and with its authorization.

- **Authorization server:** The authorization server gets consent from the resource owner and issues access tokens to clients for accessing protected resources hosted by a resource server. Smaller API providers may use the same application and URL space for both the authorization server and resource server.

- **Access token**: An access token is a key that allows a client, resource owner (via an App or web client) or a resource server, to access the information protected by OAuth. In OAuth 2.0, these access tokens are called "bearer tokens", and can be used alone, with no signature or cryptography, to access the information. Access tokens are usually passed to the servers in the header, in the form of "Authorization: Bearer <token-string>" (and this is the recommended way of sending the access token), although depending on the concrete OAuth implementation made by the service, the access token can also be passed as POST parameters or even as part of the GET URI.

OAuth2

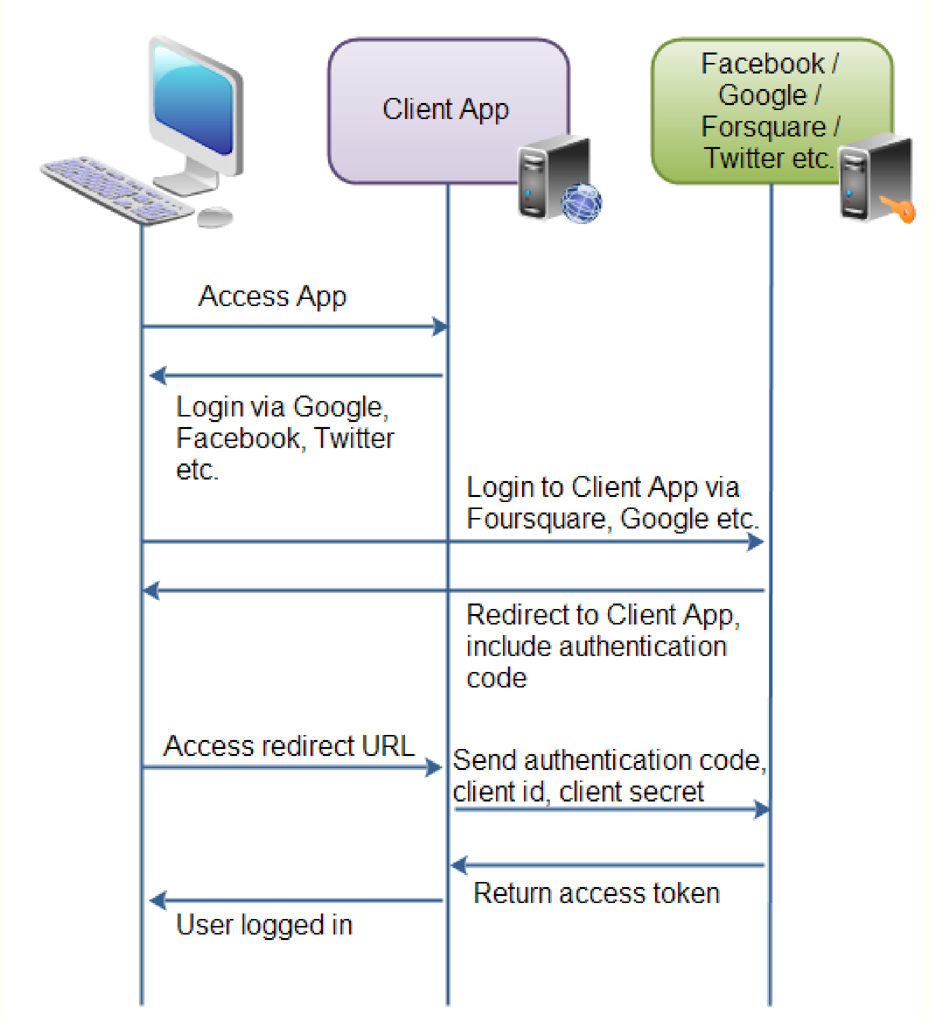

Resource owner
User is asked to authenticate & authorize
2
User Agent
Client identifier and redirection URI
1
user authenticates & authorizes
2
authorization code
3
Authorization Server
3
5
needs auth
1
4
authorization code and redirection URI
access token + (optional) refresh token
Client REST server

OAuth 2.0 Use Cases

- OAuth 2.0 can be used either to create an application that can read user data from another application (e.g. the game in the diagram above), or an application that enables other applications to access its user data (e.g. Facebook in the example above).
- OAuth 2.0 is a replacement for OAuth 1.0, which was more complicated.
- OAuth 1.0 involved certificates etc.
- OAuth 2.0 is more simple. It requires no certificates at all, just SSL / TLS.

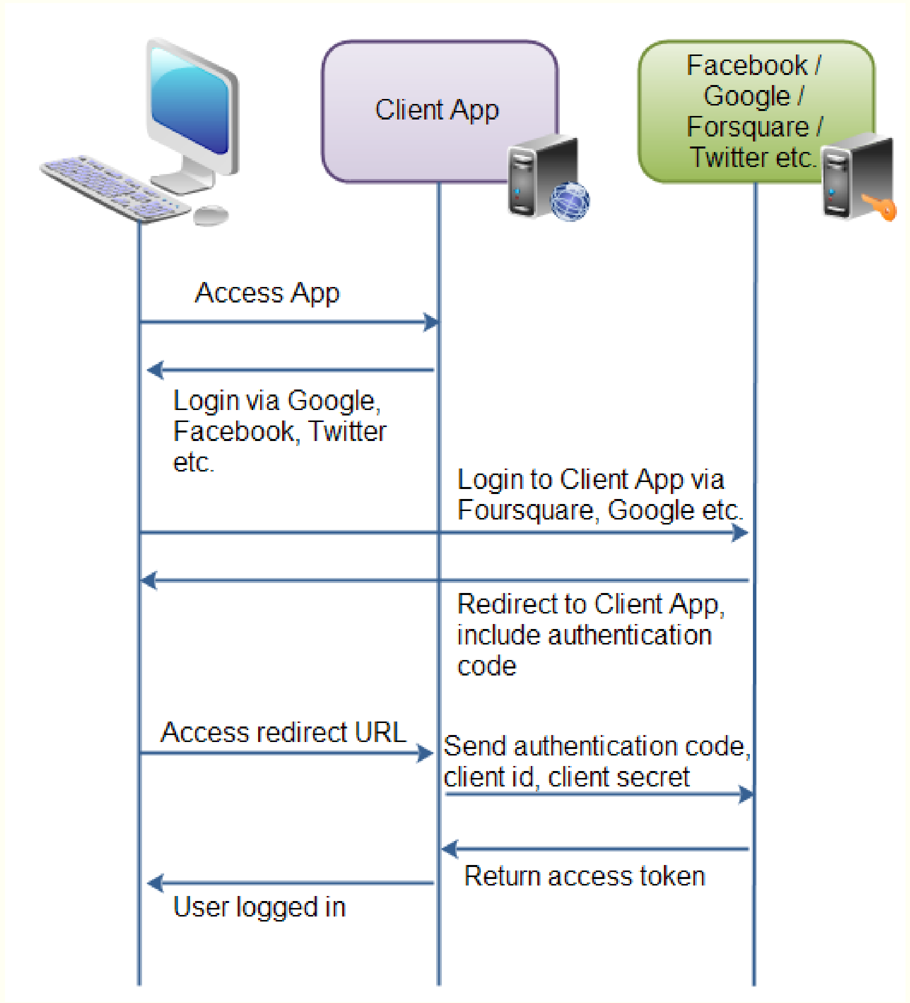
OAuth 2.0

- First the user accesses the client web application. In this web app is button saying "Login via Facebook" (or some other system like Google or Twitter).
- Second, when the user clicks the login button, the user is redirected to the authenticating application (e.g. Facebook). The user then logs into the authenticating application, and is asked if she wants to grant access to her data in the authenticating application, to the client application. The user accepts.


Client App
Facebook / Google / Forsquare / Twitter etc.
Access App
Login via Google, Facebook, Twitter etc.
Login to Client App via Foursquare, Google etc.
Redirect to Client App, include authentication code
Access redirect URL
Send authentication code, client id, client secret
Return access token
User logged in

OAuth 2.0

- Third, the authenticating application redirects the user to a redirect URI, which the client app has provided to the authenticating app. To the URI is appended an authentication code. This code represents the authentication.
- Fourth, the user accesses the page located at the redirect URI in the client application. In the background the client application contacts the authenticating application and sends client id, client password and the authentication code received in the redirect request parameters. The authenticating application sends back an access token.
- Once the client application has obtained an access token, this access token can be sent to the Facebook, Google, Twitter etc. to access resources in these systems, related to the user who logged in.

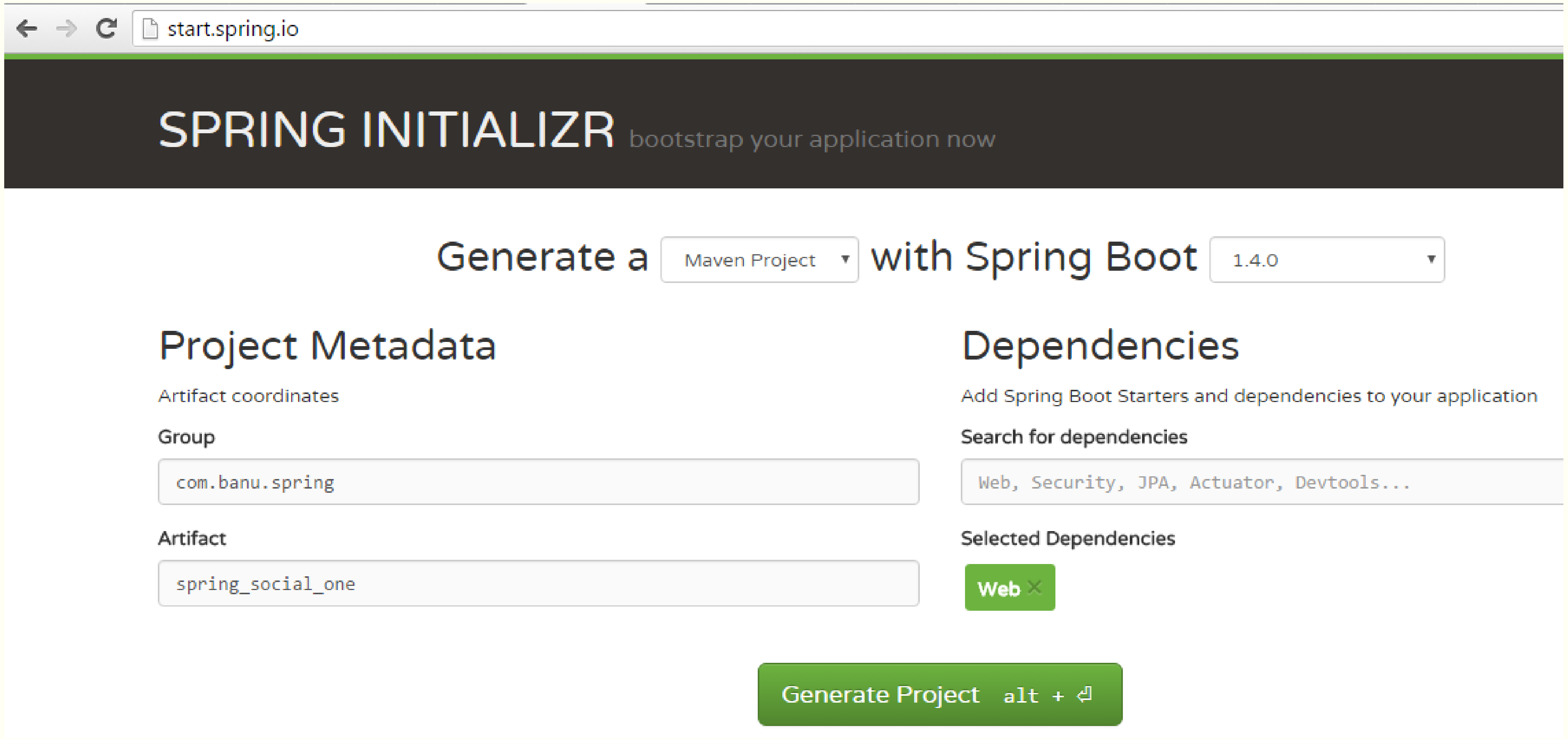

Client App
Facebook / Google / Forsquare / Twitter etc.
Access App
Login via Google, Facebook, Twitter etc.
Login to Client App via Foursquare, Google etc.
Redirect to Client App, include authentication code
Access redirect URL
Send authentication code, client id, client secret
Return access token
User logged in

Spring Social project

- The Spring Social project provides:
  - A standard way to get access to the social network specific API's with Java bindings to popular service provider APIs such as Facebook, Twitter, LinkedIn and GitHub.
  - An extensible service provider framework that greatly simplifies the process of connecting local user accounts to social network provider accounts.

  - Integration with Spring Security and Spring MVC
    - A connect controller that handles the authorization flow between your Java/Spring web application, a service provider, and your users.
    - A sign-in controller that enables users to authenticate with your application by signing in through a service provider.

Spring Boot and OAuth


start.spring.io

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.4.0

Project Metadata

Artifact coordinates

Group

com.banu.spring

Artifact

spring_social_one

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

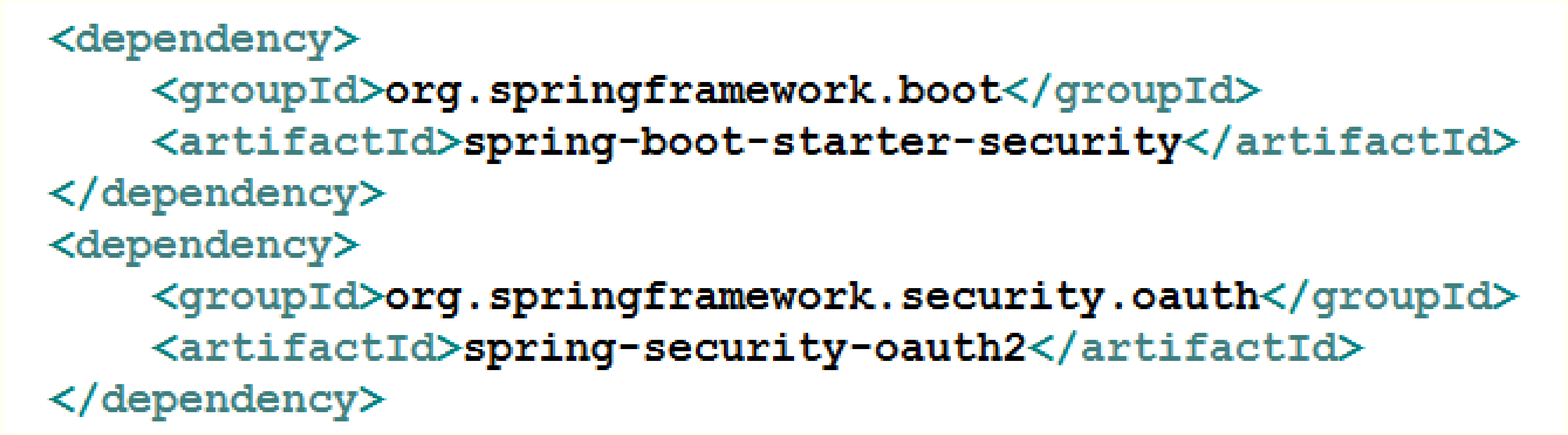
Selected Dependencies

Web ×

Generate Project alt + ⏎

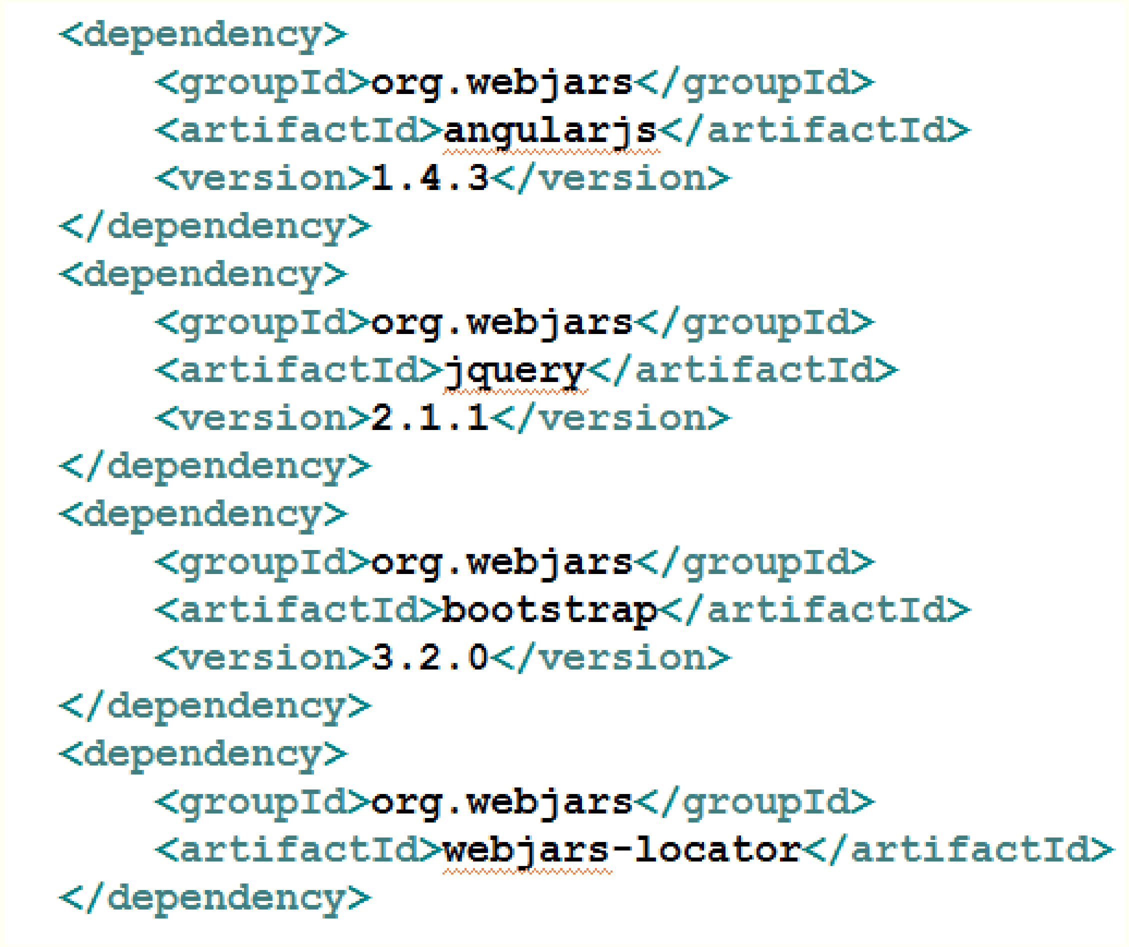
Securing the Application

- HTTP Basic Security:
  - To make the application secure we just need to add Spring Security as a dependency.
- Spring OAuth2 security:
  - We want to do a "social" login (delegate to Facebook), add the Spring Security OAuth2 dependency as well:



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

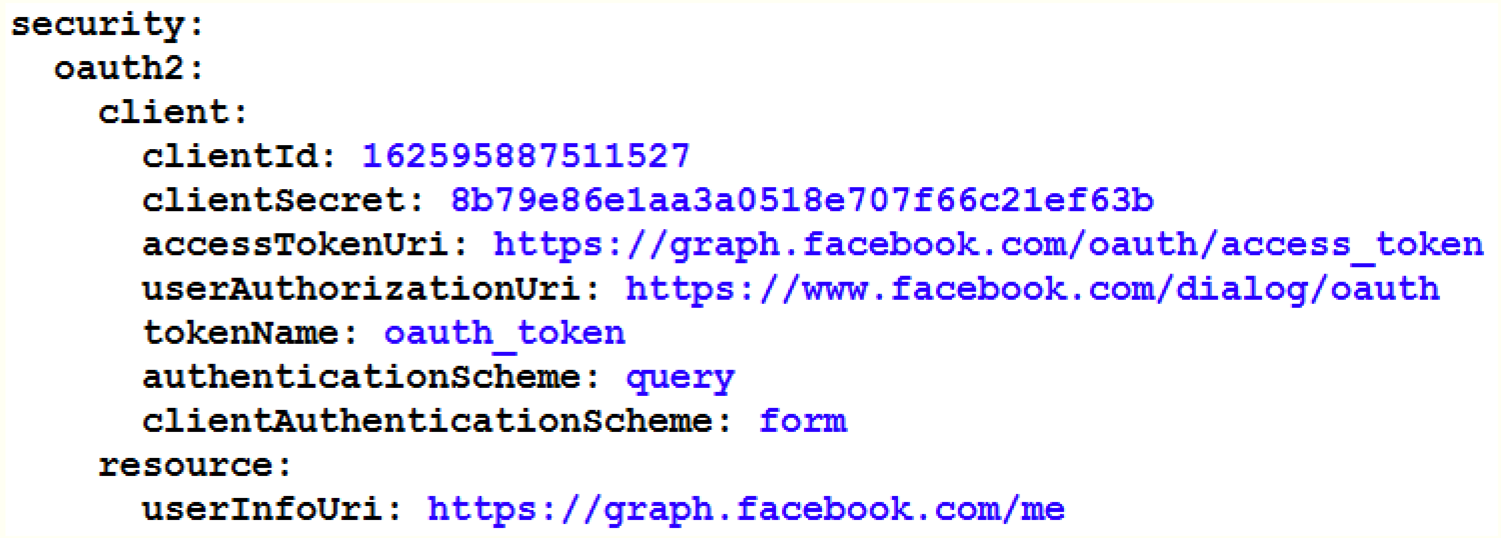
Add the following dependencies for views



```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>angularjs</artifactId>
    <version>1.4.3</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.2.0</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
</dependency>
```

application.yml

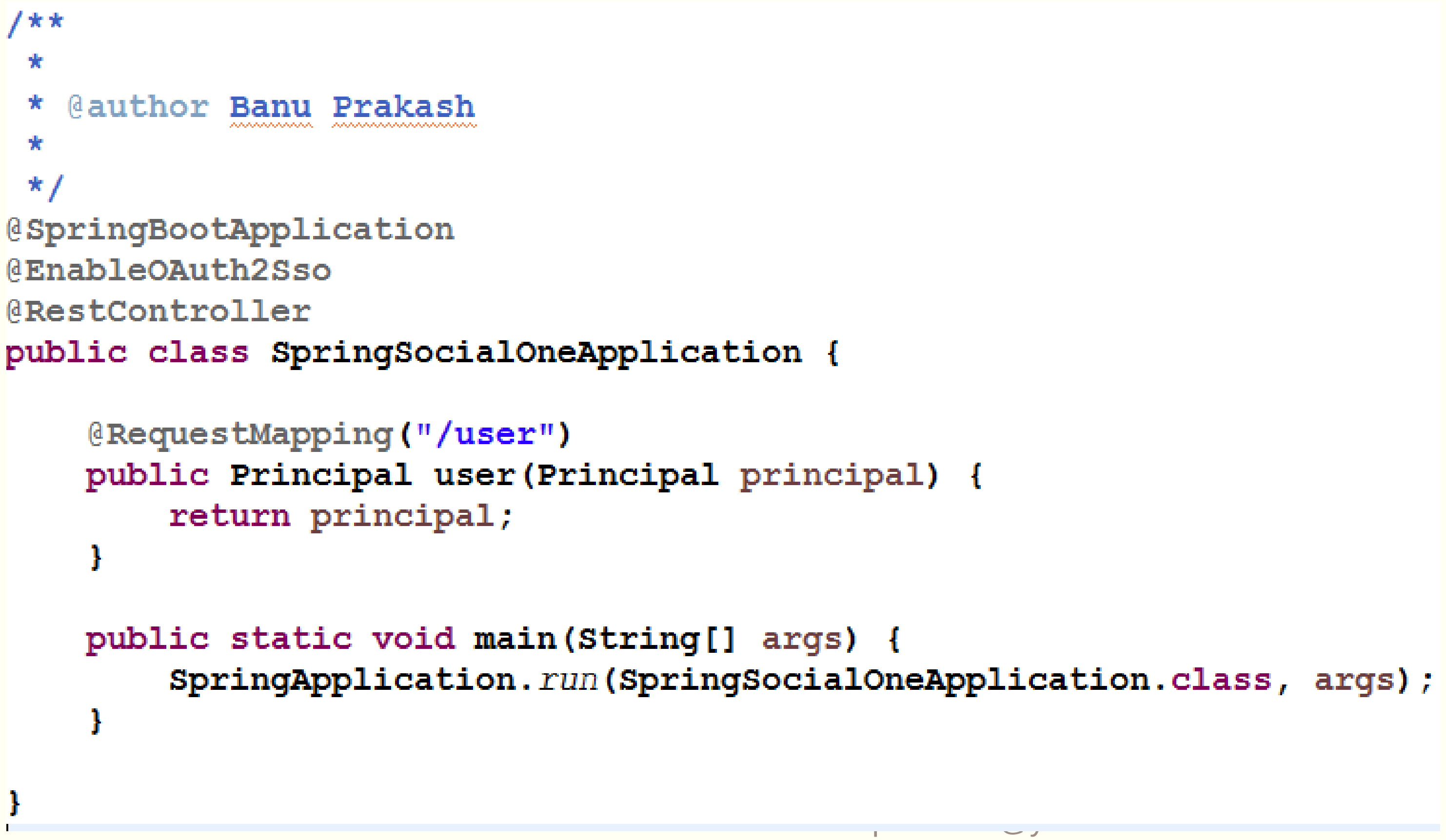
- Configured for facebook



```yaml
security:
  oauth2:
    client:
      clientId: 162595887511527
      clientSecret: 8b79e86e1aa3a0518e707f66c21ef63b
      accessTokenUri: https://graph.facebook.com/oauth/access_token
      userAuthorizationUri: https://www.facebook.com/dialog/oauth
      tokenName: oauth_token
      authenticationScheme: query
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://graph.facebook.com/me
```

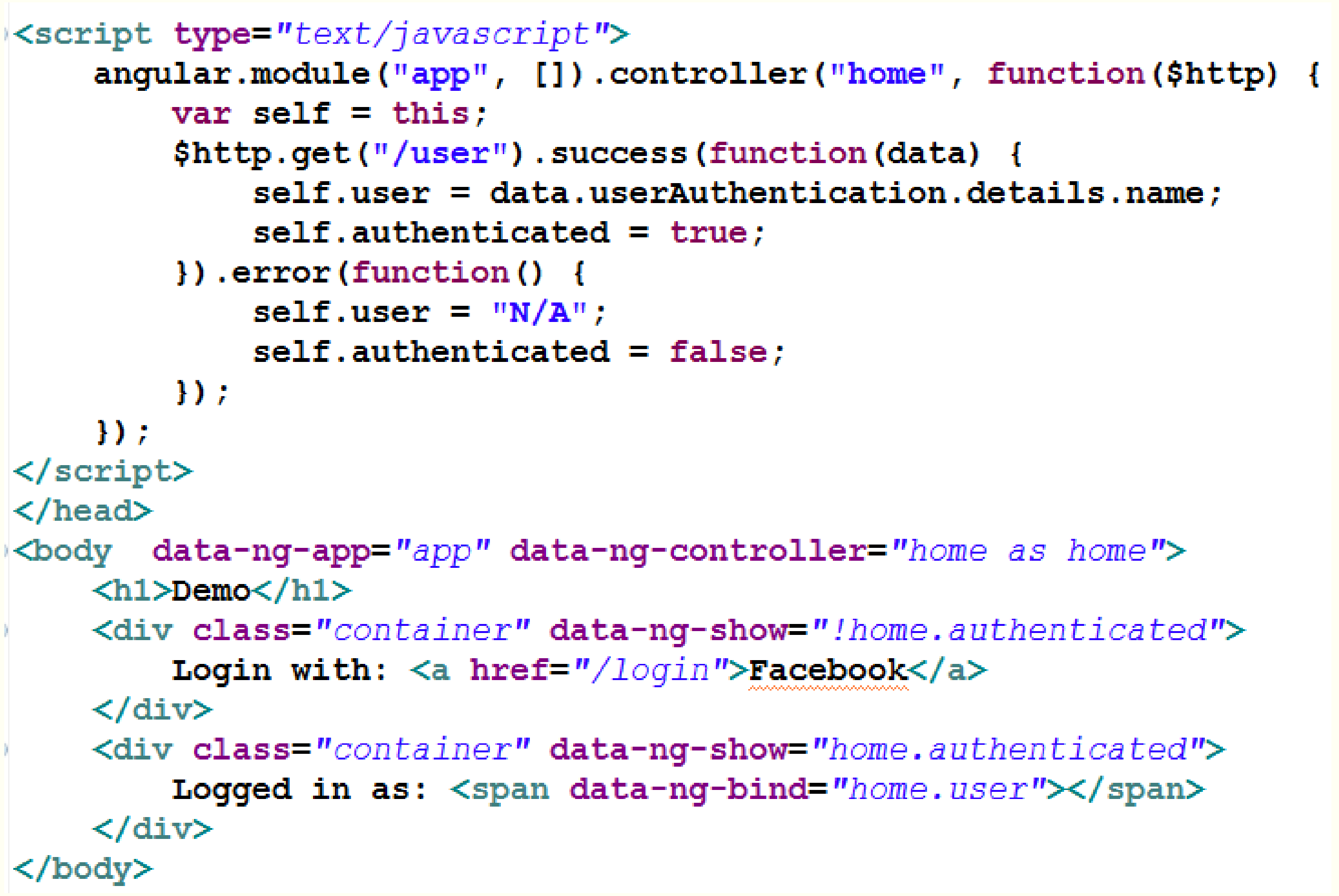
EnableOAuth2

- Add @EnableOAuth2Sso
- Add "/user" that describes the currently authenticated user

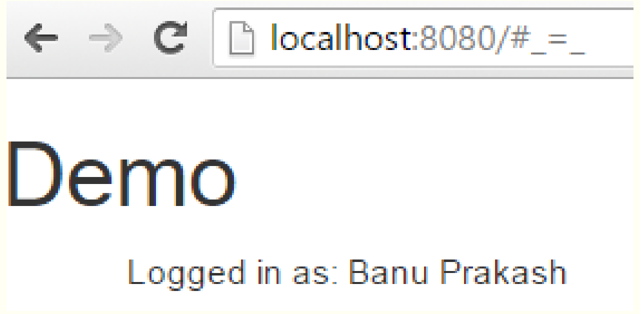


```
/**
 *
 * @author Banu Prakash
 *
 */
@SpringBootApplication
@EnableOAuth2Sso
@RestController
public class SpringSocialOneApplication {

    @RequestMapping("/user")
    public Principal user(Principal principal) {
        return principal;
    }

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

}
```

Client Side content using AngularJS



```
<script type="text/javascript">
    angular.module("app", []).controller("home", function($http) {
        var self = this;
        $http.get("/user").success(function(data) {
            self.user = data.userAuthentication.details.name;
            self.authenticated = true;
        }).error(function() {
            self.user = "N/A";
            self.authenticated = false;
        });
    });
</script>
</head>
<body  data-ng-app="app" data-ng-controller="home as home">
    <h1>Demo</h1>
    <div class="container" data-ng-show="!home.authenticated">
        Login with: <a href="/login">Facebook</a>
    </div>
    <div class="container" data-ng-show="home.authenticated">
        Logged in as: <span data-ng-bind="home.user"></span>
    </div>
</body>
```

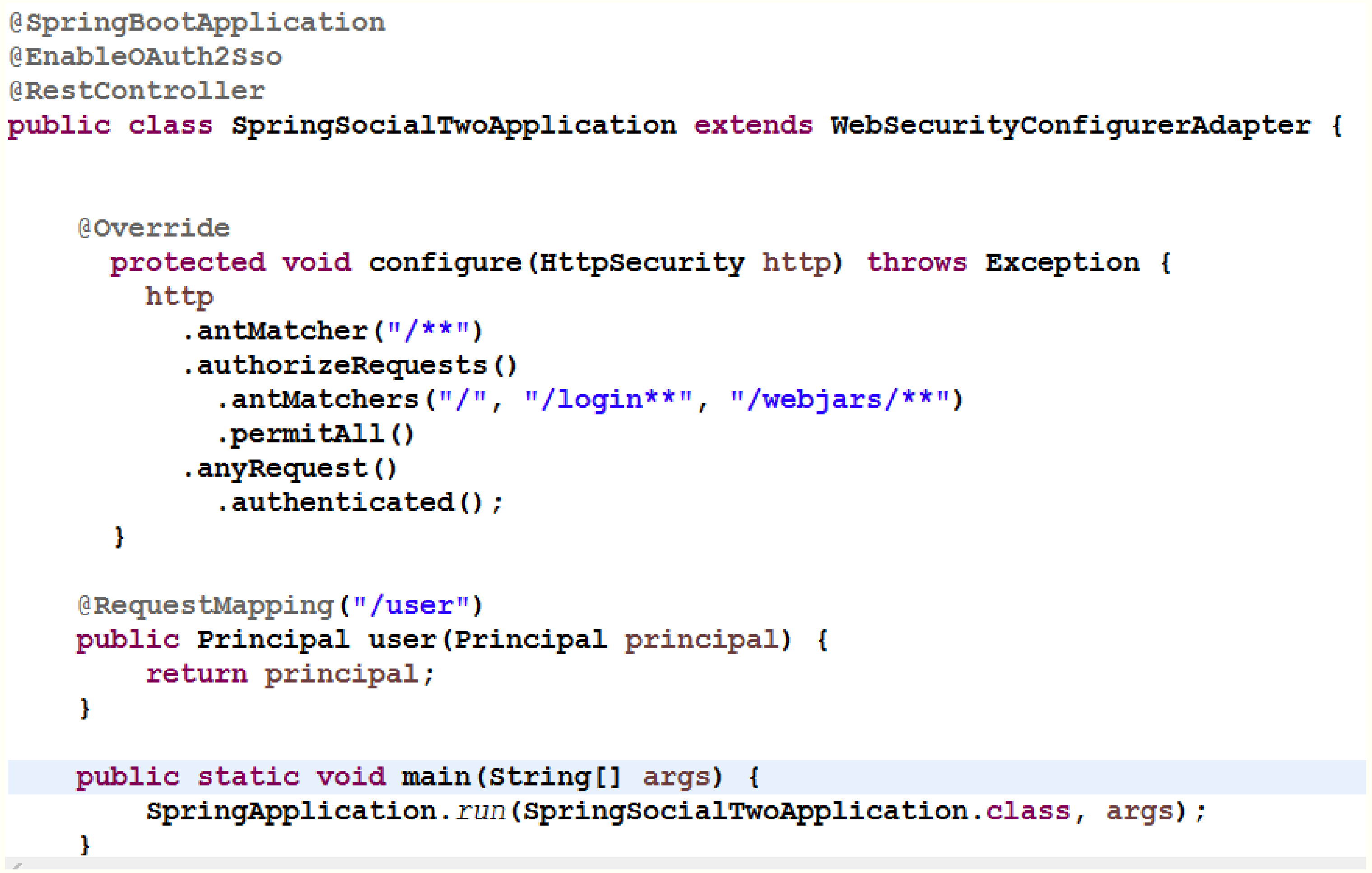


localhost:8080/#_=_

Demo

Logged in as: Banu Prakash

Make link visible



```
@SpringBootApplication
@EnableOAuth2Sso
@RestController
public class SpringSocialTwoApplication extends WebSecurityConfigurerAdapter {


    @Override
      protected void configure(HttpSecurity http) throws Exception {
        http
          .antMatcher("/**")
          .authorizeRequests()
            .antMatchers("/", "/login**", "/webjars/**")
            .permitAll()
          .anyRequest()
            .authenticated();
      }

    @RequestMapping("/user")
    public Principal user(Principal principal) {
        return principal;
    }

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