

[BT](#)

- [About InfoQ](#)
- [Our Audience](#)
- [Contribute](#)
- [About C4Media](#)

• Exclusive updates on:

- 
- 
- 
- 
- 

Facilitating the spread of knowledge and innovation in professional software development

[gavin](#)

Welcome, gavin !

- [My Reading List](#)
- [Preferences](#)

Personalize Your Main Interests

- ☒ Development
- ☒ Architecture & Design
- ☒ Data Science
- ☒ Culture & Methods
- ☒ DevOps

This affects what content you see on the homepage & your RSS feed. Click preferences to access more fine-grained personalization.

[Sign out](#)

**InfoQ**  
queue

- [En](#)

5/13/2016

- [中文](#)
- [日本](#)
- [Fr](#)
- [Br](#)

1,455,522 Apr unique visitors

- [Development](#)
  - [Java](#)
  - [Clojure](#)
  - [Scala](#)
  - [.Net](#)
  - [Mobile](#)
  - [Android](#)
  - [iOS](#)
  - [IoT](#)
  - [HTML5](#)
  - [JavaScript](#)
  - [Functional Programming](#)
  - [Web API](#)

## Featured in Development

### [An Introduction to Property Based Testing](#)



[Aaron Bedra focuses on describing your system as a series of models that can be used to systematically and automatically generate input data and ensure that your code is behaving as expected. Bedra discusses property based testing and how it can help you build more resilient systems and even reduce your time maintaining your current test suite.](#)

### [All in Development](#)

- [Architecture & Design](#)
  - [Architecture](#)
  - [Enterprise Architecture](#)
  - [Scalability/Performance](#)
  - [Design](#)

Exploring Micro frameworks: Spring Boot

5/13/2016

Exploring Microservices with Spring Boot

- [Case Studies](#)
- [Microservices](#)
- [Patterns](#)
- [Security](#)

## Featured in Architecture & Design

### [The Morning Paper Quarterly Review](#)



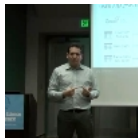
[A summary of five CS papers chosen from the 66 that Adrian Coyler has reviewed for his Morning Paper blog during the first quarter of 2016. Topics include distributed transactions, transaction recovery, and Hyperloglog.](#)

### [All in Architecture & Design](#)

- [Data Science](#)
  - [Big Data](#)
  - [Machine Learning](#)
  - [NoSQL](#)
  - [Database](#)
  - [Data Analytics](#)
  - [Streaming](#)

## Featured in Data Science

### [Hunting Criminals with Hybrid Analytics](#)



[David Talby demos using Python libraries to build a ML model for fraud detection, scaling it up to billions of events using Spark, and what it took to make the system perform and ready for production.](#)

### [All in Data Science](#)

- [Culture & Methods](#)
  - [Agile](#)
  - [Leadership](#)

5/13/2016

Exploring Micro-frameworks Spring Boot

- [Team Collaboration](#)
- [Testing](#)
- [Project Management](#)
- [UX](#)
- [Scrum](#)
- [Lean/Kanban](#)
- [Personal Growth](#)

## Featured in Culture & Methods

### [Q&A with Shawn Callahan on Putting Stories to Work](#)



[The book Putting Stories to Work by Shawn Callahan provides a process with a practical approach to master business storytelling; a leadership skill that helps to achieve results. It contains many stories that can help you to use storytelling for business communication and culture change.](#)

### [All in Culture & Methods](#)

- [DevOps](#)
  - [Infrastructure](#)
  - [Continuous Delivery](#)
  - [Automation](#)
  - [Containers](#)
  - [Cloud](#)

## Featured in DevOps

### [Acceptance Testing for Continuous Delivery](#)



[Dave Farley discusses using acceptance testing to work quickly and effectively, building functional coverage for complex enterprise-scale systems, and managing and maintaining those tests.](#)

### [All in DevOps](#)

5/13/2016  
New York

San Francisco

London

- [Mobile](#)
- [HTML5](#)
- [JavaScript](#)
- [APM](#)
- [DevOps](#)
- [Java](#)
- [API Design](#)
- [Cloud](#)
- [Database](#)

[All topics](#)

You are here: [InfoQ Homepage](#) [Articles](#) Exploring Micro-frameworks: Spring Boot

# Exploring Micro-frameworks: Spring Boot



Posted by [Dan Woods](#) on Mar 18, 2014 | [15 Discuss](#)

- Share
- |
- 
- 
- 
- 
- 
- 
- 
- ["Read later"](#)
- ["My Reading List"](#)

Exploring Micro-frameworks: Spring Boot [Jun 13-17](#)

[Nov 7-11](#)

[Mar 6-10, 2017](#)

5/13/2016

[Spring Boot](#) is a brand new framework from the team at [Pivotal](#), designed to simplify the bootstrapping and development of a new Spring application. The framework takes an opinionated approach to configuration, freeing developers from the need to define boilerplate configuration. In that, Boot aims to be a front-runner in the ever-expanding rapid application development space.

The [Spring IO platform](#) has been criticized over the years for having bulky XML configuration with complex dependency management. During last year's SpringOne 2GX conference, Pivotal CTO, Adrian Colyer [acknowledged those criticisms](#), and made special note that a goal of the platform going forward is to embrace an XML-free development experience. Boot takes that mission statement to the extreme, not only freeing developers from the need for XML, but also, in some scenarios, releasing them from the tedium of writing import statements. In the days following its public beta release, Boot gained some viral popularity by demonstrating the framework's simplicity with a runnable web application that fit in under 140-characters, delivered in a [tweet](#).

Spring Boot is not, however, an alternative to the many projects that comprise the ["Foundation" layer](#) of the Spring IO platform. Indeed, the goal of Spring Boot is not to provide new solutions for the many problem domains already solved, but rather to leverage the platform in fostering a development experience that simplifies the use of those already-available technologies. This makes Boot an ideal choice for developers who are familiar with the Spring ecosystem, while also catering to new adopters by allowing them to embrace Spring technologies in a simplified manner.

Related Vendor Content

[Start your FREE TRIAL of AppDynamics Pro](#)

[Top five PHP performance metrics, tips and tricks](#)

[Reactive: Learn how the Java community and Spring in particular is adopting reactive programming at QConNY](#)

[Top 10 Java Performance Problems](#)

[Cloud-Based Java Development: Choosing the Right PaaS](#)

Related Sponsor

APPDYNAMICS

[AppDynamics](#) is an [application intelligence company](#) solution that simplifies the management of complex, business-critical apps.

In pursuit of such an improved development experience, Spring Boot — and, indeed, the entire Spring ecosystem — has embraced the [Groovy programming language](#). Groovy's powerful MetaObject protocol, pluggable AST transformation process, and embedded dependency resolution engine are what facilitate many of the shortcuts that Boot affords. At the core of its compilation model, Boot utilizes Groovy to build project files, so that it can decorate a class' generated bytecode with common imports and boilerplate methods, such as a class' main method. This allows applications written with Boot to remain concise, while still offering a breadth of functionality.

At its most fundamental level, Spring Boot is little more than a set of libraries that can be leveraged by any project's build system. As a convenience, the framework also offers a command-line interface, which can be used to run and test Boot applications. The framework distribution, including the integrated CLI, can be [manually downloaded and installed](#) from the Spring repository. A more convenient approach is to use the [Groovy enVironment Manager \(GVM\)](#), which will handle the installation and management of Boot versions. Boot and its CLI can be installed by GVM with the command line, `gvm install springboot`. Formulas are available for installing Boot on OS X through the [Homebrew](#) package manager. To do so, first tap the Pivotal repository with `brew tap pivotal/tap`, followed by the `brew install springboot` command.

Projects that are to be packaged and distributed will need to rely on build systems like [Maven](#) or [Gradle](#). To simplify the dependency graph, Boot's functionality is modularized, and groups of dependencies can be brought in to a project by importing Boot's so-called "starter" modules. To easily manage dependency versions and to make use of default configuration, the framework exposes a parent POM, which can be inherited by projects. An example POM for a Spring Boot project is defined in Listing 1.

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <!-- Inherit defaults from Spring Boot -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.0.0.RC1</version>
    </parent>

    <!-- Add typical dependencies for a web application -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>
</project>
```

```

</dependencies>

<repositories>
  <repository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/libs-snapshot</url>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/libs-snapshot</url>
  </pluginRepository>
</pluginRepositories>

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

```

For a more-simplified build configuration, developers can leverage the Gradle build system's concise Groovy DSL, as depicted in Listing 1.1.

Listing 1.1

```

buildscript {
  repositories {
    maven { url "http://repo.spring.io/libs-snapshot" }
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.0.RC1")
  }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

repositories {

```



```

mavenCentral()
maven { url "http://repo.spring.io/libs-snapshot" }
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-actuator:1.0.0.RC1'
}

```

To help with getting Boot projects up and running quickly, Pivotal provides the so-called ["Spring Initializr" web interface](#), which can be used to download pre-built Maven or Gradle build configurations. Projects can also be quick-started through the use of a [Lazybones](#) template, which will create the necessary project structure and gradle build file for a Boot application after executing the `lazybones create spring-boot-actuator my-app` command.

## Developing a Spring Boot Application

The most popular example of a Spring Boot application is one that was delivered via Twitter shortly following the public announcement of the framework. As demonstrated in its entirety in Listing 1.2, a very simple Groovy file can be crafted into a powerful Spring-backed web application.

Listing 1.2

```

@RestController
class App {
    @RequestMapping("/")
    String home() {
        "hello"
    }
}

```

This application can be run from the Spring Boot CLI, by executing the `spring run App.groovy` command. Boot analyzes the file and — through various identifiers known as "compiler auto-configuration" — determines that it is intended to be a web application. It then, in turn, bootstraps the Spring Application context inside of an embedded Tomcat container on the default port of 8080. Opening a browser and navigating to the provided URL will land you on a page with a simple text response, "hello". This process of providing a default application context and embedded container allows developers to focus on the process of developing application and business logic, and frees them from the tedium of otherwise boiler-plate configuration.

Boot's ability to ascertain the desired functionality of a class is what makes it such a powerful tool for rapid application development. When applications are executed from the Boot CLI, they are built using the internal Groovy compiler, which allows the ability to programmatically inspect and modify a class while its bytecode is being generated. In this way, developers who use the CLI are not only freed from the need to define default configuration, but, to an extent, they also do not need to define certain import statements that can otherwise be recognized and automatically added during the compilation process. Additionally, when applications are run from the CLI, Groovy's built-in dependency manager, ["Grape"](#), is used to resolve classpath dependencies that are needed to bootstrap the compilation and runtime environments, as determined by Boot's compiler auto-configuration mechanisms. This idiom not only makes the framework more user-friendly, but also allows different versions of Spring Boot to be coupled with specific versions of libraries from the Spring IO platform, which in turn means that developers do not need to be concerned with managing a complex dependency graph and versioning structure. Additionally, it opens the door for rapid

prototyping and quick generation of proof-of-concept project code.

For projects that are not built with the CLI, Boot provides a host of "starter" modules, which define a set of dependencies that can be brought into a build system in order to resolve the specific libraries needed from the framework and its parent platform. As an example of this, the `spring-boot-starter-actuator` dependency pulls in a set of base Spring projects to get an application quickly configured and up-and-running. The emphasis of this dependency is on developing web applications, and specifically RESTful web services. When included in conjunction with the `spring-boot-starter-web` dependency, it will provide auto-configuration to bootstrap an embedded Tomcat container, and will map endpoints useful to micro-service applications, like server information, application metrics, and environment details. Additionally, when the `spring-boot-starter-security` module is brought in, the actuator will auto-configure [Spring Security](#) to provide the application with basic authentication and other advanced security features. For any application structure, it will also include an internal auditing framework that can be used for reporting purposes or application-specific needs, like developing an authentication-failure lock-out policy.

To demonstrate quickly getting a Spring web application up-and-running from within a Java Maven project, consider the application code outlined in Listing 1.3.

Listing 1.3

```
package com.infoq.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Application {

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

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

The presence of the `@EnableAutoConfiguration` annotation on the `Application` class informs Boot that it should take an opinionated approach to configuring the application. This defers otherwise boilerplate configuration to the defaults assumed by the framework, which focuses on getting the application up-and-running as quickly as possible. The `Application` class is also runnable, which means that the application, and its embedded container, can be started and actively developed by choosing to run the class as a Java application.

When it is time to build the project for distribution, Boot's Maven and Gradle plugins hook into those build systems' packaging process to produce an executable "fat jar", which embeds all of the project's dependencies and can be executed as a runnable jar. Packaging a Boot application with Maven is as simple as running

the `mvn package` command, and likewise with Gradle, executing the `gradle build` command will output a runnable jar to the build's target location.

## Developing Micro-Services

Given Boot's simplifications to Spring application development, its provided ability to import dependencies in a modular fashion, its inherent emphasis on developing RESTful web services, and its capability to produce a runnable jar, the framework is clearly a formidable utility in the development of deployable micro-services. As demonstrated in prior examples, getting a RESTful web application up-and-running is a fairly trivial task with Boot; but to realize the full potential of Boot, we will demonstrate the intricacies of developing a full-featured RESTful micro-service. Micro-services are an increasingly popular application architecture in enterprise infrastructure, as they allow for rapid development, smaller code-bases, enterprise integration, and modular deployables. There are many frameworks that are targeting this development vertical, and this section will discuss utilizing Boot's simplifications for this purpose.

### Database Access

Micro-services can be built for a variety of purposes, but one guarantee is that most will need the ability to read and write to a database. Spring Boot makes database integration a trivial task with its ability to auto-configure Spring Data for database access. By simply including the `spring-boot-starter-data-jpa` module as part of your project, Boot's auto-configuration engine will detect that your project requires database access, and will create the necessary beans within the Spring application context, so that you can create and use repositories and services. To demonstrate this more specifically, consider the Gradle build file in Listing 1.4, which outlines the build structure of a Groovy-based Boot micro-service web application that uses Spring Data's JPA support for database access.

Listing 1.4

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/libs-snapshot" }
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.0.RC1")
    }
}

apply plugin: 'groovy'
apply plugin: 'spring-boot'

repositories {
    mavenCentral()
    maven { url "http://repo.spring.io/libs-snapshot" }
}

ext {
    springBootVersion = "1.0.0.RC1"
```

```
} 5/13/2016
```

Exploring Micro-frameworks: Spring Boot

```
dependencies {  
    compile 'org.codehaus.groovy:groovy-all:2.2.1'  
    compile "org.springframework.boot:spring-boot-starter-web:$springBootVersion"  
    compile "org.springframework.boot:spring-boot-starter-data-jpa:$springBootVersion"  
    compile "org.springframework.boot:spring-boot-starter-actuator:$springBootVersion"  
}
```

In this configuration, Boot's actuator module provides a dependency on `hsqldb`, and will set up all of the necessary configuration — including schema creation — so that Spring Data can use the relational in-memory database as its `datasource`. This shortcut frees developers from the need to create and manage complicated `datasource` XML configuration in development, and quickly opens the door to developing database-driven micro-services. This same auto-configuration capability exists if the H2 or Derby embedded databases are found on the classpath. An additional convenience offered by Boot is its ability to quickly and easily bootstrap an application's database schema with relevant data. This is incredibly useful in development, where a database may be in-memory or otherwise volatile, and where developers need to be sure that certain data points are available when the application starts. To demonstrate this, consider the example JPA entity shown in Listing 1.5, which represents a "User" data structure that the micro-service will provide.

#### Listing 1.5

```
@Entity  
class User {  
    @Id  
    @GeneratedValue  
    Long id  
  
    String username  
    String firstName  
    String lastName  
    Date createdAt  
    Date lastAccessed  
  
    Boolean isActive = Boolean.TRUE  
}
```

To bootstrap some common data that represents `User` objects, we can simply create a file named `schema.sql` or `data.sql`, and include it on our classpath. This file will be executed after the schema has been created, so, given the entity depicted in Listing 1.5, we can bootstrap a user account with a SQL statement, as shown in Listing 1.6.

#### Listing 1.6

```
insert into user(username, first_name, last_name, created_date) values ('daneloper', 'Dan', 'Woods', now())
```

Upon startup, the provided SQL code will be executed, and we can be sure that we have a test account to work with. Now that the micro-service has a data point

to begin with, Listing 1.7 demonstrates how we can follow Spring Data's development pattern and create a `Repository` interface that will act as the Data Access Object for the `User` entity.

#### Listing 1.7

```
public interface UserRepository extends CrudRepository<User, Long> {  
}
```

The `CrudRepository` provides some common interface methods for creating, retrieving, updating, and deleting objects and sets of objects. Any specific capabilities that our application may need beyond this can be defined by following Spring Data's [repository development conventions](#). Once the `UserRepository` interface is created, Boot's `spring-data-jpa` layer will detect it within the project, and will bring it into the Spring application context, making it an autowire candidate for controllers and services. This automatic configuration occurs only when a Boot application requests that an opinionated approach be taken, which is identified by the presence of the `@EnableAutoConfiguration` annotation. The micro-service can now define a RESTful endpoint for consumers to retrieve a list of users or an individual user through the controller implementation shown in Listing 1.8.

#### Listing 1.8

```
@RestController  
@EnableAutoConfiguration  
@RequestMapping("/user")  
class UserController {  
  
    @Autowired  
    UserRepository repository  
  
    @RequestMapping(method=[RequestMethod.GET])  
    def get(Long id) {  
        id ? repository.findOne(id) : repository.findAll()  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run UserController, args  
    }  
}
```

At startup, the application will output logging that shows Hibernate creating the database structure as defined by the `User` entity, and, as well, will show Boot importing the data from the `schema.sql` file as the final part of the application's initialization.

It's important to note the use of the `@RequestMapping` annotation when developing a micro-service application. This is not an annotation that is Boot specific. However, because Boot installs its own endpoints for the purposes of monitoring the application's performance, health, and configuration, we want to ensure that our application code doesn't conflict with the resolution of those built-in detail providers. Given that, when there is a requirement to resolve a property (in this case, the `id` of the user) from the request path, then we need to carefully consider how that dynamic property resolution will affect the rest of the micro-service's behavior. In this case, simply mapping the controller to the `/user` endpoint takes it out of the root context, and allows Boot's endpoints to be accessible as well.

All data provided by our micro-service might not best fit into a relational structure, and for that, Spring Boot exposes modules that give developers the ability to work with Spring Data's MongoDB and Redis projects, while still taking an opinionated approach to their configuration. Spring Data's higher-level framework for defining Data Access Objects makes it easy to quickly interchange between JPA and non-JPA data sources. Consider the example in Listing 1.9, which demonstrates a redefined `UserRepository` interface designed to work with MongoDB instead of JPA.

Listing 1.9

```
public interface UserRepository extends MongoRepository<User, Long> {  
}
```

The `MongoRepository` interface also extends `CrudRepository`, so the micro-service's controller code from Listing 1.8 needn't change. To facilitate MongoDB integration as demonstrated, the project must only include the `spring-boot-starter-data-mongodb` module on the application's classpath. The dependency block from the Gradle build file in Listing 1.4 will only need to change slightly, as demonstrated in Listing 1.10.

Listing 1.10

```
dependencies {  
    compile 'org.codehaus.groovy:groovy-all:2.2.1'  
    compile "org.springframework.boot:spring-boot-starter-web:$springBootVersion"  
    compile "org.springframework.boot:spring-boot-starter-data-mongodb:$springBootVersion"  
    compile "org.springframework.boot:spring-boot-starter-actuator:$springBootVersion"  
}
```

Now that the MongoDB dependency is on the classpath, Boot will auto-configure Spring Data to connect to the database on localhost, and by default to the `test` database. From there, the `User` collection will automatically be created (as is standard with MongoDB), and the micro-service is now backed by MongoDB. Bootstrapping data for non-JPA datastores is less trivial than otherwise, but this is mostly rooted in the fact that it wouldn't make sense to run a SQL file against MongoDB's document store or Redis' key-value store. Since Spring Data will be using persistent instances of these datastores, it also means that data that is created in development will survive a restart. To begin working with persistent data, we need to modify the micro-service controller so that `User` instances can be created by consumers. We can also start to evolve the micro-service's `UserController` to conform to a common RESTful API structure by having the controller handle different HTTP methods in different ways. Listing 1.11 demonstrates adding the ability to create new `User` instances through the controller.

Listing 1.11

```
@RestController  
@RequestMapping("/user")  
@EnableAutoConfiguration  
class UserController {  
  
    @Autowired  
    UserRepository repository  
  
    @RequestMapping(method=[RequestMethod.GET])
```

```

def get(Long id) {
    id ? repository.findOne(id) : repository.findAll()
}

@RequestMapping(method=[RequestMethod.POST])
def create(@RequestBody User user) {
    repository.save user
    user
}

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

```

When a consumer of the micro-service performs an HTTP POST to the application's endpoint, Spring will coerce the request body into a `User` instance. The code will then use the `UserRepository` to store the object in the MongoDB collection. An example of using curl to create a new `User` instance is shown in Listing 1.12.

Listing 1.12

```
curl -v -H "Content-Type: application/json" -d '{"username": "danveloper", "firstName": "Dan", "lastName": "Woo"
```

With Boot providing an opinion about how the Mongo datasource should be configured, the new `User` instance will, by default, be persisted to the `user` collection on the test database within the Mongo instance running on localhost. If we open a web browser and make an HTTP GET request to the micro-service, we will see the user that we created returned in the list.

## Configuration

Spring Boot gets out of your way pretty quickly when you need to override its configuration defaults. By default, application configuration can be defined using a Java properties file at the root of the application's classpath named `application.properties`. A preferred approach, however, is to use [YAML](#) configuration, which gives structure and depth to nested configuration. Given the presence of the `snakeyaml` dependency on the application's runtime classpath, your project can then define configuration directives in an `application.yml` file. To demonstrate this, consider the example YAML configuration in Listing 1.13, which outlines the various settings that are available for an application's embedded HTTP server (Tomcat by default, Jetty by option).

Listing 1.13

```

# Server settings (ServerProperties)
server:
  port: 8080
  address: 127.0.0.1
  sessionTimeout: 30

```

```
5/13/2016  
contextPath: /
```

Enabling Micro-frameworks Spring Boot

```
# Tomcat specifics  
tomcat:  
  accessLogEnabled: false  
  protocolHeader: x-forwarded-proto  
  remoteIpHeader: x-forwarded-for  
  basedir:  
  backgroundProcessorDelay: 30 # secs
```

The ability to override Boot's auto-configuration is what will allow you to take your application from prototype to production, and Boot makes this easy to do within the same `application.yml` file. Auto-configuration directives are designed to be as short as possible, so when building your micro-service with actuator, an endpoint for configuration properties, `/configprops` is installed, and can be referred to when determining what directives to override. When we're ready for our micro-service to use a persistent datasource, like [MySQL](#), then we can simply add the MySQL Java Driver to the runtime classpath, and add the necessary configuration directive to the `application.yml` file, as demonstrated in Listing 1.14.

Listing 1.14

```
spring:  
  datasource:  
    driverClassName: com.mysql.jdbc.Driver  
    url: jdbc:mysql://localhost:3306/proddb  
    username: root  
    password
```

In scenarios where you need a more flexible configuration, Boot allows you to override much of its default configuration through the use of Java System properties. As an example, if your application needs a different database user when it is deployed to production, the username configuration directive can be passed to the application using standard Java System property switches to the command-line execution, `-Dspring.datasource.username=user`. A more practical scenario for this is in a cloud deployment, like Cloud Foundry or Heroku, where those platforms require the application to start on a specific HTTP port, which is made available through an environment variable in the host operating system. Boot's ability to derive configuration from System properties allows your application to inherit the HTTP port through the command line execution using, `-Dserver.port=$PORT`. This is an incredibly useful feature of the framework when developing micro-services, because it allows a micro-service application to run on a variety of environment configurations.

## Externalized Configuration

One very important thing that a micro-service must be able to do is support *externalized* configuration. This configuration can hold anything from placeholder messages to database configuration, and the architecture behind this is an area that must be considered during the initial planning and prototyping of an application. Various strategies for importing configuration exist today within the Spring IO platform, however supporting the various ways in which an application may want to consume configuration often results in verbose programmatic coupling.

A niche feature of Boot is its ability to automatically manage externalized configuration, and coerce it into an object structure that is usable throughout the



application context. By creating a Plain Old Java/Groovy Object, and decorating it with the `@ConfigurationProperties` annotation, that object will consume its defined name subset of configuration from Boot's configuration structure. To describe this more plainly, consider the POGO in Listing 1.15, which brings in configuration directives from under the `application.` key.

Listing 1.15

```
@ConfigurationProperties(name = "application")
class ApplicationProperties {
    String name
    String version
}
```

When the `ApplicationProperties` object is created within the Spring context, Boot will recognize that it is a configuration object, and will populate its properties in alignment with configuration directives from the `application.properties` or `application.yml` file on the runtime classpath. With that given, if we add an application block to the micro-service's `application.yml` file, as shown in Listing 1.16, we will be able to access those directives in a programmatic fashion from the rest of our application.

Listing 1.16

```
application:
  name: sb-ms-custdepl
  version: 0.1-CUSTOMER
```

These configuration directives can be used for a variety of purposes, and the only requirement to gain access to them is that their represented POJO/POGO be a participant in the Spring application context. Boot lets us easily manage the configuration bean's integration to the application context by allowing us to treat a controller as a Spring Java configuration object, as demonstrated in Listing 1.17.

Listing 1.17

```
@RestController
@Configuration
@RequestMapping("/appinfo")
@EnableAutoConfiguration
class AppInfoController {

    @Autowired
    ApplicationProperties applicationProperties

    @RequestMapping(method=[RequestMethod.GET])
    def get() {
        [
            name: applicationProperties.name,
            version: applicationProperties.version
        ]
    }
}
```

```

    }
}

@Bean
ApplicationProperties applicationProperties() {
    new ApplicationProperties()
}

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

```

The code in Listing 1.17 is a contrived example, though given a more-complex scenario, the principles remain the same for using Boot to set up access to application-specific configuration. Configuration classes can also support nested object graphs to give depth and meaning to the data as it's coming from the configuration. For example, if we wanted to also have configuration directives for our application's metrics keys under the `application.root`, we can add a nested object to the `ApplicationProperties` POGO to represent those values, as shown in Listing 1.18.

Listing 1.18

```

@ConfigurationProperties(name = "application")
class ApplicationProperties {
    String name
    String version

    final Metrics metrics = new Metrics()

    static class Metrics {
        String dbExecutionTimeKey
    }
}

```

Now our `application.yml` file can be crafted as demonstrated in Listing 1.19 to include the `metrics` configuration under the `application.` block.

Listing 1.19

```

application:
  name: sb-ms-custdepl
  version: 0.1-CUSTOMER
  metrics:
    dbExecutionTimeKey: user.get.db.time

```

When we need access to the `application.metrics.dbExecutionTimeKey` value, we can access it programmatically through the `ApplicationProperties` object.

These configuration directives within the `application.properties` or `application.yml` file do not necessarily need to be coerced to an object graph for them to be usable throughout the application. Indeed, Boot also provides the Spring application context with a `PropertySourcesPlaceholderConfiguration`, so that directives that are derived from either the `application.properties` file, the `application.yml` file, or from Java System property overrides can be utilized as Spring property placeholders. This mechanism of Spring allows you to define a placeholder value for a property using a specific syntax, and Spring will fill it in if it finds a placeholder configuration that provides it. As an example of this, we can use the `@Value` annotation to directly access the `application.metrics.dbExecutionTimeKey` within our controller, as shown in Listing 1.20.

Listing 1.20

```
@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {

    @Autowired
    UserRepository repository

    @Autowired
    GaugeService gaugeService

    @Value('${application.metrics.dbExecutionTimeKey}')
    String dbExecutionKey

    @RequestMapping(method=[RequestMethod.GET])
    def get(Long id) {
        def start = new Date().time
        def result = id ? repository.findOne(id) : repository.findAll()
        gaugeService.submit dbExecutionKey, new Date().time - start
        result
    }

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

There will be more discussion on the intricacies of metrics reporting later, but for now the important thing to understand is how the `@Value` annotation can be used with a Spring property placeholder to have Boot auto-populate the value for our micro-service's specific configuration needs.

## Security

In micro-service development, the need for a comprehensive security context will invariably arise. To service this need, Boot brings in the powerful,

comprehensive Spring Security and provides auto-configuration to quickly and easily bootstrap a security layer. Just the presence alone of the `spring-boot-starter-security` module on the application's classpath will let Boot employ some of its security features like cross-site scripting protection and adding the headers that prevent click-jacking. Additionally, adding a simple configuration directive, as shown in Listing 1.21, will secure your application with basic authentication.

Listing 1.21

```
security:
  basic:
    enabled: true
```

Boot will provide you with a default user account of `user`, a default role of `USER`, and will output a randomly generated password to the console when the application starts up. Like most other things Boot, it is easy to specify a different username and password for the built-in user account with explicitly defined configuration directives ("`secured`" and "`foo`" respectively), as demonstrated in Listing 1.22.

Listing 1.22

```
security:
  basic:
    enabled: true
  user:
    name: secured
    password: foo
```

Boot's built-in faculties for quickly bootstrapping basic authentication within your micro-service prove very useful for simple, internal applications, or for development prototyping. As your requirements evolve, your application will undoubtedly need some granular level of security, such as the ability to secure endpoints to specific roles. From this perspective, we may want to secure read-only data (ie. **GET** requests) for consumers who are represented with the `USER` role, while read-write data (ie. **POST** requests) should be secured with the `ADMIN` role. To facilitate this, we'll disable Boot's basic authentication auto-configuration inside of the project's `application.yml` file, and define our own for both user and admin accounts for their respective roles. This is another example of an area where Boot gets out of your way quickly when you need to move beyond its by-default functionality. To demonstrate this more practically, consider the code outlined in Listing 1.23. This example can be expounded upon to make use of Spring Security's full potential and leverage more-intricate authentication strategies, such as JDBC-backed, OpenID, or Single-Sign On.

Listing 1.23

```
@RestController
@RequestMapping("/user")
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
@EnableAutoConfiguration
class UserController extends WebSecurityConfigurerAdapter {

    @Autowired
```

```

@Repository repository

@RequestMapping(method = [GET])
@Secured(['ROLE_USER'])
def get(Long id) {
    id ? repository.findOne(id) : repository.findAll()
}

@RequestMapping(method = [POST])
@Secured(['ROLE_ADMIN'])
def create(@RequestBody User user) {
    repository.save user
    user
}

@Override
void configure(AuthenticationManagerBuilder auth) {
    auth
        .inMemoryAuthentication()
        .withUser "user" password "password" roles "USER" and() withUser "admin" password "password" roles "USER", "ADMIN"
}

@Override
void configure(HttpSecurity http) throws Exception {
    BasicAuthenticationEntryPoint entryPoint = new BasicAuthenticationEntryPoint()
    entryPoint.realmName = "Spring Boot"
    http.exceptionHandling().authenticationEntryPoint(entryPoint)
    http.requestMatchers().antMatchers("/**").anyRequest()
        .and().httpBasic().and().anonymous().disable().csrf().disable()
}

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

```

Given the example in Listing 1.23, the application now configures authentication to explicitly provide access to user accounts of user and admin, both with the password, password, and with respective roles of USER and ADMIN. The micro-service's **GET** and **POST** endpoints are also secured for USER and ADMIN roles respectively, meaning that read-only data can now be accessed by regular users, while performing read-write operations require the admin user credentials.

Basic authentication is a great choice for micro-services because it follows a very practical and widely usable authentication protocol. In other words, many API consumers, including mobile applications, can very easily make use of this to gain access to your micro-service. When your authentication needs out-grow basic authentication (ie. OpenID or OAuth), your micro-service can leverage the full capabilities of Spring Security for your requirement.

## Messaging Integration

Messaging is a very powerful utility in the toolkit of any application, and micro-services are no exception when it comes to this. Developing these applications with a message-driven architecture can support reusability and scalability. Spring Boot allows developers to write micro-services with messaging as a core tenant of the architecture, through its use of the Spring IO platform's implementation of the Enterprise Integration Patterns, Spring Integration. Spring Integration provides the structure for developing a message-driven architecture, as well as providing modules for integrating with a distributed enterprise platform. This capability allows micro-services to utilize business objects from an abstract messaging source, whether that source be within the application or provided from another service within the organization.

While Boot does not provide any explicit Spring context auto-configuration, it does offer a starter module for Spring Integration, which is responsible for bringing in a host of dependencies from the Spring Integration project. These dependencies include Spring Integration's Core library, its HTTP module (for HTTP-oriented enterprise integration), its IP module (for Socket-based integration operations), its File module (for filesystem integration), and its Stream module (for working with Stream, like stdin and stdout). This starter module gives developers a robust toolkit of messaging functionality for adapting an existing infrastructure to a micro-service API.

In addition to the starter module, Boot also offers compiler auto-configuration for applications that are built through the CLI. This provides some shortcuts for developers who are rapidly prototyping micro-services, and are demonstrating viability. Applications that leverage an enterprise platform can be quickly developed, and their value rapidly determined before they are moved to a formal project and build system. Getting a message-driven micro-service up-and-running with Spring Boot and Spring Integration is as easy as the code sample depicted in Listing 1.24.

Listing 1.24

```
@RestController
@EnableIntegrationPatterns
class App {

    @Bean
    def userLookupChannel() {
        new DirectChannel()
    }

    @Bean
    def userTemplate() {
        new MessagingTemplate(userLookupChannel())
    }

    @RequestMapping(method=[RequestMethod.GET])
    def get(@RequestParam(required=false) Long id) {
        userTemplate().convertSendAndReceive( id ? id : "" )
    }
}
```

```
class User {
    Long id
}

@MessageEndpoint
class UserLookupObject {

    @ServiceActivator(inputChannel="userLookupChannel")
    def get(Long id) {
        id ? new User(id:id) : new User()
    }
}
```

Taking a message-driven approach to micro-service development offers a high amount of code reusability and decoupling from the underlying service provider implementation. In a less contrived scenario, the code in Listing 1.18 may be responsible for the composition of data from database calls and external service integration within an enterprise organization. Spring Integration has built-in constructs for payload routing and handler chaining, which makes it an appealing solution to the composition of disparate data, to which we may find our micro-service being a provider.

## Providing Metrics

Perhaps the most important feature of a micro-service is its ability to provide metrics to a reporting agent. Unlike thick web applications, micro-services are lightweight and aren't designed with the intent of providing reporting screens or robust interfaces to analyze the service's activity. These types of operations are best left to applications that are strictly responsible for the aggregation and analysis of data for the purposes of stability, performance, and business intelligence monitoring. With that as a given, a micro-service will provide endpoints for those reporting tools to easily consume data about its activity. From there, it is the responsibility of the reporting tool to compose that data into a view or report that makes sense to whomever cares about that data.

While some of the metrics about a micro-service, such as stability and performance, can be generalized across all applications, metrics related to business operations must be managed specifically by the application. For this purpose, Spring Boot's actuator module exposes a mechanism for developers to programmatically expose details about the micro-service's state through the `/metrics` endpoint. Boot breaks down metrics to the categories of "counters" and "gauges"; a counter is any metric that is represented as a `Number`, where a gauge is a metric that measures some calculation with double precision. To make working with metrics easy for micro-service developers, Boot exposes a `CounterService` and a `GaugeService` as autowirable candidates to the application context. Consider the example in Listing 1.25, which demonstrates exposing a hit count through the `CounterService`.

Listing 1.25

```
@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {

    @Autowired
    UserRepository repository
```

```

@Autowired
CounterService counterService

@RequestMapping(method = [GET])
def get() {
    get(null)
}

@RequestMapping(value="/{id}", method = [GET])
def get(@PathVariable Long id) {
    counterService.increment id ? "queries.by.id.$id" : "queries.without.id"
    id ? repository.findOne(id) : repository.findAll()
}
}

```

After hitting the /user endpoint with and without a provided ID, the /metrics endpoint will report new keys under the counter . parent. For example, if we simply query the /user endpoint with no ID, then the counter.queries.without.id metric will be registered and available. Likewise, when we do provide an ID, we will see the counter.queries.by.id.<id> key shown to denote how many queries have been made for a provided ID. These metrics may provide some insight into the most commonly accessed User objects, and may denote some actions that need to be taken, such as caching or database indexing. In the same manner of incrementing a metric count, the CounterService also allows for a metric to be decremented to zero. This may be useful for tracking open connections or other histogrammic measurements.

Gauges are a slightly different type of metric, in that they provide heuristics for calculated or otherwise upon-request determined values. As the GaugeService JavaDocs note, the "gauge" measurement can be anything from method execution time to the temperature of a meeting room. Those types of measurements are uniquely suited for the use of the GaugeService when exposing details for a reporting tool. Gauge metrics will be prefixed in the /metrics endpoint with gauge.. They are registered in a slightly different manner than counters, which is demonstrated in Listing 1.26.

Listing 1.26

```

@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {

    @Autowired
    UserRepository repository

    @Autowired
    CounterService counterService

    @RequestMapping(method = [GET])
    def get() {

```



```

    def get(null)
    }

    @RequestMapping(value="/{id}", method = [GET])
    def get(@PathVariable Long id) {
        def start = new Date().time
        def result = id ? repository.findOne(id) : repository.findAll()
        def time = new Date().time - start
        gaugeService.submit("user.get.db.time", time.doubleValue())
        result
    }
}

```

By default, metrics will be stored in a volatile, in-memory database, but providing an implementation of a `MetricsRepository` to the application context will allow for a more-persistent storage behavior. Boot ships with a `RedisMetricsRepository`, which can be autowired to store metrics in a Redis keystore, though custom implementations for any datastore can be crafted to persist the metrics.

Boot also provides support for the [Coda Hale Metrics library](#), and will coerce metrics that start with certain names to their respective Metrics types. For example, if a metric is provided that starts with `histogram.`, then Boot will provide that value as a `Histogram` object type. This automatic coercion also works for `meter.` and `timer.` keys, while regular metrics are sent as `Gauge` types.

Once micro-service metrics are being registered with Boot, they can then be retrieved by a reporting tool through the `/metrics` endpoint. Named metrics can be provided to the `/metrics` endpoint by providing the metric key name as part of the query string. For example, to access *only* the gauge metric, `"user.get.db.time"`, a reporting tool can make a query to `/metrics/gauge.user.get.db.time`.

## Packaging Boot Applications

As discussed earlier, Boot ships with plugins for both Maven and Gradle, which provide a hook into the build systems' packaging phase to produce the so-called "fat jar" with all of the project's dependencies included. When the fat jar is executed, the application code will run inside of the same embedded container in which the project was developed. This shortcut gives developers the peace-of-mind that their deployable package has the same dependency structure and runtime environment from which they developed. This alleviates operations teams from worrying about deployment scenarios where a mis-configured runtime container may have one specification of dependencies, while the project was developed under another.

To perform the packaging under Maven, simply execute the `mvn package` command. The Spring Boot plugin will make a backup of the originally created project jar and rename it with an appended `".original"` to the filename. From here, the runnable jar will be available following the Maven artifact naming conventions, and can be deployed in the manner most suitable to the project. Building Boot projects with Gradle is equally as trivial, and requires only the execution of the standard `gradle build` command. Similar to Maven, the Boot plugin installs a lifecycle event with Gradle that follows the original packaging task, and will produce the fat jar to the `build/libs` directory. An examination of the produced fat jar will reveal all of the dependent jars in the `lib/` directory of the archive.

Once packaged, the fat jar can be executed from the command line as a regular runnable jar file, using the command `$JAVA_HOME/bin/java -jar`

path/to/myproject.jar. When started, the Boot application logging will be shown in the console.

For applications that need the ability to deploy to a traditional servlet container, Boot provides a path for programmatically initializing a web configuration. To facilitate this, Boot offers an opinionated `WebApplicationInitializer`, which registers the application with the servlet container through the Servlet 3.0 API to programmatically register servlets with the container's `ServletContext`. By providing a subclass of `SpringBootServletInitializer`, Boot applications can register their configuration with the embedded Spring context that is created during the container's initialization. To demonstrate this functionality, consider the example code demonstrated in Listing 1.27.

Listing 1.27

```
@RestController
@EnableAutoConfiguration
class Application extends SpringBootServletInitializer {

    @RequestMapping(method = RequestMethod.GET)
    String get() {
        "home"
    }

    static void main(String[] args) {
        SpringApplication.run this, args
    }

    @Override
    SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        application.sources Application
    }
}
```

The `Application` class' overridden `configure` method is what is used to register the application code with the embedded Spring context. In a less-contrived scenario, this method may be used to register a Spring Java configuration class, which would define the beans for all of the controllers and services in the application.

When packaging the application for deployment to a servlet container, the project must be built as a war file. To accommodate this in a Maven project, the Boot plugin needs to be removed, and the packaging needs to be defined explicitly with a type of "war", as shown in Listing 1.28.

Listing 1.28

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```

<groupId>com.example</groupId>
<artifactId>myproject</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>war</packaging>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.0.0.RC1</version>
</parent>

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

<repositories>
  <repository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/libs-snapshot</url>
  </repository>
</repositories>
</project>

```

Executing a `mvn install` command for this project will result in a `myproject-1.0.0-SNAPSHOT.war` file being built to the `target` directory. Projects that are built with Gradle can make use of the Gradle War Plugin, which exposes a `war` task for building the war file. Similar to Maven configurations, Boot Gradle projects will also need to remove their inclusion of the Boot plugin. A sample Gradle build script for producing a war file can be depicted in Listing 1.29.

Listing 1.29

```

apply plugin: 'java'
apply plugin: 'war'

repositories {
  mavenCentral()
  maven { url "http://repo.spring.io/snapshot" }
  maven { url "http://repo.spring.io/milestone" }
}

```

```
ext {  
    springBootVersion = '1.0.0.BUILD-SNAPSHOT'  
}  
  
dependencies {  
    compile "org.springframework.boot:spring-boot-starter-web:${springBootVersion}"  
    compile "org.springframework.boot:spring-boot-starter-actuator:${springBootVersion}"  
}
```

Running the Gradle war task against a Boot project with this build script will output the war artifact to the build/libs directory.

In either a Maven or Gradle configuration, once the war file is produced, it can then be deployed to any Servlet 3.0-compliant application container. Some compliant containers include Tomcat 7+, Jetty 8, Glassfish 3.x, JBoss AS 6.x/7.x, and Websphere 8.0.

## Further Reading

The Spring Boot team has produced a comprehensive collection of [guides](#) and samples to demonstrate the framework's capabilities. Blog posts, reference material, and API documentation can all be found on the [Spring.IO website](#). Example projects can be found on the [project's GitHub page](#), and additional low-level detail can be found in the [Spring Boot reference manual](#). The SpringSourceDev YouTube channel has a [webinar on Spring Boot](#), which outlines the project's goals and capabilities. During last year's Groovy & Grails Exchange in London, David Dawson gave a [presentation on developing micro-services](#) with Spring Boot.

## About the Author



**Daniel Woods** is a Senior Software Engineer at Netflix, where he develops continuous delivery and cloud deployment tools. He specializes in JVM stack technologies and is active in the Groovy, Grails, and Spring communities. Daniel can be reached via email at [danielwoods@gmail.com](mailto:danielwoods@gmail.com) or through Twitter [@daneloper](#).

- [Personas](#)
- [Development](#)
- [Topics](#)
- [Pivotal](#)
- [Java](#)
- [Spring](#)

Related Editorial

[Spring Framework 5 - Preview & Roadmap](#)

5/12/2016

Eliminating Micro frameworks from Spring Boot

[#NoXML: Eliminating XML in Your Spring Projects](#)

[Introducing CallTracing\(tm\) Based on RabbitMQ, Spring and Zipkin](#)

[Isomorphic Templating with Spring Boot, Nashorn and React](#)

[Building a Next-generation Cloud e-Commerce Platform with Spring](#)

## Hello stranger!

You need to [Register an InfoQ account](#) or [Login](#) or login to post comments. But there's so much more behind being registered.

## Get the most out of the InfoQ experience.

### Tell us what you think

Please enter a subject	Message
------------------------	---------

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

Community comments [Watch Thread](#)

[Executable WARs are worth considering too](#) by Dave Syer Posted Mar 18, 2014 11:15

[Re: Executable WARs are worth considering too](#) by Jirka Pinkas Posted Mar 19, 2014 02:09

[Excellent article](#) by Rakesh Smith Posted Mar 19, 2014 11:12

[Excellent article](#) by Marcus Vinicius Soliva Sousa Posted Apr 07, 2014 02:36

[XML Free... huh?](#) by Neil Bartlett Posted Jun 20, 2014 07:41

[Re: XML Free... huh?](#) by Neil Jones Posted Feb 23, 2015 06:26

[best article](#) by Bayadr Taha Posted Jul 17, 2014 05:52

[Great article](#) by Oleg KOROLENKO Posted Sep 07, 2014 09:31

[Project on GitHub](#) by Thomas Madison Posted Oct 02, 2014 11:29

[Need help in Spring boot actuator audit](#) by Anant Navagale Posted Oct 15, 2014 02:45

[About use in the production environments](#) by liu chunrong Posted Nov 08, 2014 04:26

[API I/O abstraction](#) by Owen Rubel Posted Nov 17, 2014 09:57

[Ways to change the port in the external Tomcat in a Spring Boot Application.](#) by Sunil Wadkar Posted May 20, 2015 12:21

[Extremely sorry](#) by Sunil Wadkar Posted May 20, 2015 12:29

5/13/2016  
[Re: Extremely sorry by Charles Humble Posted May 20, 2015 06:02](#)

Executable WARs are worth considering too

**Executable WARs are worth considering too** Mar 18, 2014 11:15 by "Dave Syer"

Great article, and lots of great ideas to try out.

One small correction: it's not mandatory to remove the boot plugins to create a WAR, in fact it might be better not to (given the main method still in the application class example). If you don't remove the plugin then you just end up with a WAR that is deployable \*and\* executable. Personally I would always do it that way now (the tomcat jars will not bloat a typical WAR by much). The gs-convert-jar-to-war guide ([spring.io/guides/gs/convert-jar-to-war/](http://spring.io/guides/gs/convert-jar-to-war/)) shows how to do that explicitly. There is a web-static sample in Boot that shows it too.

- [Reply](#)
- [Back to top](#)

**Excellent article** Mar 19, 2014 11:12 by "Rakesh Smith"

I need to write some web services that are mocks of real ones for automated testing. Using Spring Boot seems perfect for the task!

- [Reply](#)
- [Back to top](#)

**Re: Executable WARs are worth considering too** Mar 19, 2014 02:09 by "Jirka Pinkas"

in that case add this to dependencies:

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

so that in your .war.original file won't be embedded Tomcat.

- [Reply](#)
- [Back to top](#)

**Excellent article** Apr 07, 2014 02:36 by "Marcus Vinicius Soliva Sousa"

Hi Daniel,

Congratulations...very nice this article.

5/13/2016

- [Reply](#)
- [Back to top](#)

Embracing Micro frameworks: Spring Boot

**XML Free... huh?** Jun 20, 2014 07:41 by "Neil Bartlett"

This cracks me up. From the first paragraph: "embrace an XML-free development experience ... freeing developers from the need for XML"

First code listing: big old heap of XML.

- [Reply](#)
- [Back to top](#)

**best article** Jul 17, 2014 05:52 by "Bayadr Taha"

this is one of the best articles about the usage of spring-boot in micro-service architecture implementation.  
many thanks

- [Reply](#)
- [Back to top](#)

**Great article** Sep 07, 2014 09:31 by "Oleg KOROLENKO"

Was hesitating to give spring-boot a go on a new project but your article convinced toi try it. Thanks.

- [Reply](#)
- [Back to top](#)

**Project on GitHub** Oct 02, 2014 11:29 by "Thomas Madison"

Can you add this excellent project to GitHub? It would help a lot. Thanks.

- [Reply](#)
- [Back to top](#)

**Need help in Spring boot actuator audit** Oct 15, 2014 02:45 by "Anant Navagale"

Hi, I am using custom spring security with JDBC and AD authentication. I want the audit trails of successful and failure login attempts through boot actuator audit endpoint. But it does not return anything.

I tried registering AuthenticationEventpublisher with spring security providermanager. Still no success. Can you help in in this ?

- [Reply](#)

- 5/13/2016
- [Back to top](#)

**About use in the production environments** Nov 08, 2014 04:26 by "liu chunrong"

I want to use the spring-boot in production environments to build an application store API service. But meet the others colleagues's oppositions, as they think that a new technology would bring some larger risks, and no lots of actual cases to find. Can you give me some real use cases and suggestions? Thanks.

- [Reply](#)
- [Back to top](#)

**API I/O abstraction** Nov 17, 2014 09:57 by "Owen Rubel"

You can do I/O abstraction and avoid the hassle of all the annotations by just using a handlerInterceptor that way forwards, batches and api chaining all use the same request/response and do not spawn new processes/requests.

See api-boot ([github.com/orubel/api-boot](https://github.com/orubel/api-boot))

- [Reply](#)
- [Back to top](#)

**Re: XML Free... huh?** Feb 23, 2015 06:26 by "Neil Jones"

XML Free... huh?

Jun 20, 2014 12:41 by Neil Bartlett

Followed up by an equivalent, xml-free gradle config, showing that you indeed are free from the \*need\* for xml. However, the xml is for Maven, and is nothing really to do with spring boot. The point is that there are no spring xml config files.

- [Reply](#)
- [Back to top](#)

**Ways to change the port in the external Tomcat in a Spring Boot Application.** May 20, 2015 12:21 by "Sunil Wadkar"

I have a scenario in which I need to deploy the war (Restful webservice with actuator) in external tomcat. I followed the instructions on one of the getting started links in STS..extended the main class to SpringBootServletInitializer and did the other necessary changes. The war gets deployed in external server and I could see the SPRING printed in the tomcat console.

My problem is the services are accessible from the port of external tomcat e.g. [localhost:8080/gs-actuator-service-0.1.0/hello-...](http://localhost:8080/gs-actuator-service-0.1.0/hello-...) where 8080 is the port of external tomcat and gs-actuator-service-0.1.0 is the war name generated by maven install command in STS. Somehow its not reading



5/13/2016

the application.properties and not using the port 9001 defined, which rather works very well with embedded server.

Explaining Microservices with Spring Boot

Sorry for a long post but my assumption was it should also work in external tomcat server and I could be able to deploy multiple restful web services within the same external server (tomcat here) but accessible from different ports configured in application.properties and that's what was my assumption for term/concept called 'microservices'. Please correct my understanding and throw some light on my issue.?

- [Reply](#)
- [Back to top](#)

**Extremely sorry** May 20, 2015 12:29 by "Sunil Wadkar"

I had posted same post multiple times. Since I thought it was not saving as I do not got any success message. Apologies for inconvenience

- [Reply](#)
- [Back to top](#)

**Re: Extremely sorry** May 20, 2015 06:02 by "Charles Humble"

No problem. Your post had got flagged as potential spam and went to the moderation queue.

Have tidied up.

Charles Humble

*Head of editorial*

- [Reply](#)
- [Back to top](#)

[Close](#)

by

on

- View
- [Reply](#)
- [Back to top](#)

[Close](#)

Subject  Your Reply

[Quote original message](#)

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

5/13/2016

☐ Email me replies to any of my messages in this thread

[Close](#)

Subject  Your Reply

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

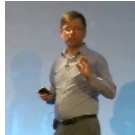
[Close](#)

RELATED CONTENT

- [Developing Cloud-native Applications with Eclipse and the Spring Tool Suite](#) Apr 11, 2016



- [Spring Framework 5 - Preview & Roadmap](#) Apr 01, 2016



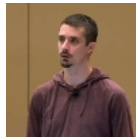
- [#NoXML: Eliminating XML in Your Spring Projects](#) Mar 12, 2016



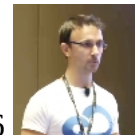
- [Introducing CallTracing\(tm\) Based on RabbitMQ, Spring and Zipkin](#) Feb 29, 2016



- [Isomorphic Templating with Spring Boot, Nashorn and React](#) Feb 29, 2016



- [Building a Next-generation Cloud e-Commerce Platform with Spring](#) Feb 22, 2016



Exploring Micro frameworks: Spring Boot

5/13/2016

Exploring Micro frameworks: Spring Boot

- [Developing Cloud-native Applications with the Spring Tool Suite](#) Feb 05, 2016



- [Get the Most out of Testing with Spring 4.2](#) Jan 09, 2016



- [The State of Securing RESTful APIs with Spring](#) Dec 21, 2015



- [12 Factor or Cloud Native Apps for Spring Developers](#) Nov 18, 2015



- [HTTP/2 for the Web Developer](#) Jan 29, 2016





5/13/2016

Exploring Micro-frameworks: Spring Boot

5/13/2016

Exploring Micro-frameworks: Spring Boot





5/13/2016

Exploring Micro-frameworks: Spring Boot



5/13/2016

Exploring Micro-frameworks: Spring Boot





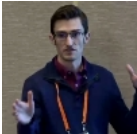
5/13/2016

Exploring Micro-frameworks: Spring Boot

5/13/2016

Exploring Micro-frameworks: Spring Boot

## RELATED CONTENT

- [Documenting RESTful APIs](#) Jan 03, 2016 
- [Oracle to Close Java.net and Kenai.com Forges](#) May 11, 2016
- [Ehcache 3.0 Released with Revamped API and Off-Heap Storage](#) May 02, 2016
- [Vulnerability in Java Reflection Library Fixed after 30 Months](#) Apr 28, 2016
- [Interview with Gil Tene on Hardware Transactional Memory](#) Apr 08, 2016 
- [Neo4j 3.0 Released with Binary Communication Protocol and Standardised Drivers](#) Apr 27, 2016
- [Locating Common Micro Service Performance Anti-Patterns](#) May 03, 2016 
- [Project Jigsaw in JDK 9: Modularity Comes To Java](#) May 02, 2016 
- [RxJava and SWT: Out with Events, in with FRP](#) Apr 29, 2016 
- [Introducing Release Early IDEs 2016.1](#) Apr 13, 2016

• [JetBrains Releases IntelliJ IDEA 2016.1](#) Apr 13, 2016



• [Examining Low Pause Garbage Collection in Java](#) Apr 27, 2016

Exploring Micro-frameworks: Spring Boot

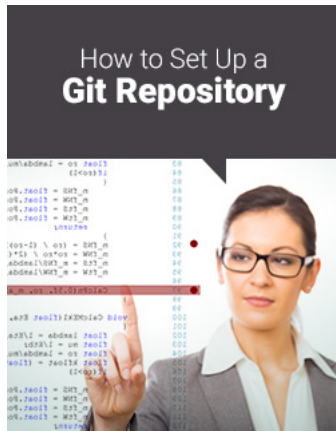
5/13/2016

Exploring Micro-frameworks: Spring Boot

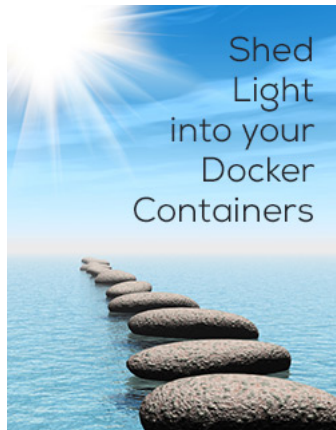
5/13/2016

Exploring Micro-frameworks: Spring Boot

## SPONSORED CONTENT

[How to Set Up a Git Repository](#)

This tutorial provides a succinct overview of the most important Git commands. First, the Setting Up a Repository section explains all of the tools you need to start a new version-controlled project. Then, the remaining sections introduce your everyday Git commands.

[Shed Light into your Docker Containers](#)

Learn how New Relic helps Docker users gain visibility into the performance of their containers within the context of their overall environment.



5/13/2016

Exploring Micro-frameworks: Spring Boot





5/13/2016

Exploring Micro-frameworks: Spring Boot



## RELATED CONTENT

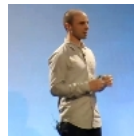
- [Angular 2 and TypeScript - A High Level Overview](#) Apr 26, 2016



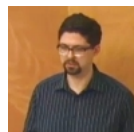
- [Java 9 - The \(G1\) GC Awakens!](#) Apr 25, 2016



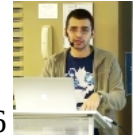
- [Hot Code is Faster Code - Addressing JVM Warm-up](#) Apr 24, 2016



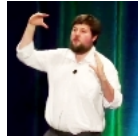
- [Understanding HotSpot JVM Performance with JITWatch](#) Apr 16, 2016



- [CIDER: Building a Clojure Interactive Development Environment that Rocks in Emacs](#) Apr 14, 2016



- [Generics and Java's Evolution](#) Apr 11, 2016



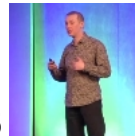
- [Node4J: Running Node.js in a JavaWorld](#) Apr 02, 2016



- [You, Me and Jigsaw](#) Apr 02, 2016



- [The Quest for Low-latency with Concurrent Java](#) Apr 01, 2016



- [High Load Trading Transaction Processing with Reveno CQRS/Event Sourcing Framework](#) Mar 29, 2016



- [Java vs. C Performance](#) Mar 28, 2016



## InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ



Click to view  
how it looks like

Your email here

Subscribe

5/13/2016  
Development

Exploring Micro-frameworks: Spring Boot

[Exploring Azure with F# Azure Storage Type Provider](#)

[An Introduction to Property Based Testing](#)

[Understanding Core Clojure Functions](#)

Architecture & Design

[Integrate 2016 - Day 1 Recap](#)

[Oracle to Close Java.net and Kenai.com Forges](#)

[Azure Stream Analytics Publishing to Power BI Reaches General Availability](#)

Culture & Methods

[Oracle to Close Java.net and Kenai.com Forges](#)

[QCon SF 2016 Registrations Open, Program Committee Announced, & Last year's Top 10 Lists](#)

[Q&A with Shawn Callahan on Putting Stories to Work](#)

Data Science

[Hunting Criminals with Hybrid Analytics](#)

[QCon SF 2016 Registrations Open, Program Committee Announced, & Last year's Top 10 Lists](#)

[Lessons Learned from Eight Years of Using NoSQL](#)

DevOps

[Oracle to Close Java.net and Kenai.com Forges](#)

[Docker Security Scanning](#)

[QCon SF 2016 Registrations Open, Program Committee Announced, & Last year's Top 10 Lists](#)

- [Home](#)
- [All topics](#)



- [QCon Conferences](#)
- [About us](#)
- [About You](#)
- [Contribute](#)
- [Purpose Index](#)
- [Sign out](#)

- **QCons Worldwide**
- [New York](#)  
[Jun 13-17, 2016](#)
- [Rio de Janeiro](#)  
[Oct 5-7, 2016](#)
- [Shanghai](#)  
[Oct 20-22, 2016](#)
- [San Francisco](#)  
[Nov 7-11, 2016](#)
- [Tokyo 2016](#)
- [London](#)  
[Mar 6-10, 2017](#)

## InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ

Click to view  
an example





- [Your personalized RSS](#)
- [For daily content and announcements](#)
- [For major community updates](#)
- [For weekly community updates](#)

Personalize Your Main Interests

Exploring Micro-frameworks: Spring Boot

- ✓ Development
- ✓ Architecture & Design
- ✓ Data Science
- ✓ Culture & Methods
- ✓ DevOps

This affects what content you see on the homepage & your RSS feed. Click preferences to access more fine-grained personalization.

General Feedback   Bugs   Advertising   Editorial   Marketing  
[feedback@infoq.com](mailto:feedback@infoq.com) [bugs@infoq.com](mailto:bugs@infoq.com) [sales@infoq.com](mailto:sales@infoq.com) [editors@infoq.com](mailto:editors@infoq.com) [marketing@infoq.com](mailto:marketing@infoq.com)

InfoQ.com and all content copyright © 2006-2016 C4Media  
Inc. InfoQ.com hosted at [Contegix](#), the best ISP we've ever  
worked with.

[Privacy policy](#)

[BT](#)