Leveraging Groovy Goodness with Spring 4.0

Presented to the Spring Dallas User Group

by Jack Frosch

16 July 2014

Overview

- A Brief Overview of Groovy 2.3
- Adding Groovy to your Spring projects
- Why Groovy Beans Make Better Spring Beans
- Spring Configuration using Groovy
- Leveraging Groovy in SpringBoot
- Spring Data, Meet GORM
- Using Gradle as Your Spring Project Build Tool (time permitting)

A Brief Overview of Groovy 2.3

- In many ways, Groovy > Java
- In Groovy, we can do what Java does
  - And we can do much more!

objistics™

A Brief Overview of Groovy 2.3

- Some Groovy Advantages
  - Groovy code is cleaner code
  - Groovy code has less ceremony than Java
    - i.e. All exceptions are treated as runtime exceptions
  - Groovy code can be executed as a script
  - Groovy code can respond dynamically to runtime state
  - Groovy != Java, but is very close
    - Groovy objects *are* java.lang.Objects
    - Groovy classes compile into JVM bytecodes
    - Thus interoperability between Java & Groovy is very high

A Brief Overview of Groovy 2.3

- Groovy imports more packages by default
  - java.io.*
  - java.lang.*
  - java.math.BigDecimal
  - java.math.BigInteger
  - java.net.*
  - java.util.*
  - groovy.lang.*
  - groovy.util.*

Groovy Beans

```
public class Pizza {
  private String name;
  private BigDecimal price;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(BigDecimal price) {
    this.price = price;
  }
}
```

This is how we do it in Java

Groovy Beans

```
class Pizza {
  String name
  BigDecimal price
}
```

This is how we do it in Groovy

- Classes are automatically public
- Fields are private by default
- Public getters and setters are created at compile-time
- Semicolons are generally not needed

Parentheses

- Method calls generally don't require parentheses
  - println('Hello')
  - println 'Hello'
- No arg methods require parentheses be used on calls
  - runnable.run()
  - Not runnable.run

Groovy Constructors

```
public class Pizza {
  private String name;
  private BigDecimal price;

  public Pizza(String name, BigDecimal price) {
    this.name = name;
    this.price = price;
  }
}

// to use it
Pizza p1 = new Pizza("Chicago Deep Dish",
                     new BigDecimal("15.99"));
// this won't compile
Pizza p2 = new Pizza(new BigDecimal("15.99"),
                     "Chicago Deep Dish");
```

This is how we do it in Java

Groovy Constructors

```
class Pizza {
  String name
  BigDecimal price
}

// to use it
Pizza p1 = new Pizza(name: “Chicago Deep Dish”,
                     price: new BigDecimal(“15.99”));
// this compiles fine
Pizza p2 = new Pizza(price: new BigDecimal(“15.99”),
                     name: “Chicago Deep Dish”);
```

This is how we do it in Groovy

- In Groovy a map entry is represented as a key : value
- We can define an empty map like this: Map map = [:]
- Classes automatically get public c’tor that accepts a map
  - Thus we can specify args in any order

Groovy Collections

```
List<String> names = new ArrayList<String>();
names.add("Fred");
names.add("Wilma");
names.add("Barney");
names.add("Betty");

for(String name: names) {
  System.out.println(name);
}

List<String> ucNames = new ArrayList<String>();
for(String name: names) {
  ucNames.add(name.toUpperCase());
}
```

This is how we do it in Java

objistics™

Groovy Collections

```
List<String> names = []
names << “Fred” << “Wilma” << “Barney” << “Betty”
//or: List names = [“Fred”, “Wilma”, “Barney”, “Betty”]

for(String name: names) {
  println(name)
}

List<String> ucNames = names.collect { name ->
  name.toUpperCase())
}
```

This is how we do it in Groovy

- Empty list is defined with []
  - It’s just a java.util.Arraylist
- << is an overloaded operator corresponding to leftShift method
  - Groovy adds leftShift to ArrayList dynamically
- For loop still works as in Java
- No need to type System.out.println
- Collect method takes a *closure* argument that defines behavior to be invoked for each element and collected into new collection
  - If closure arg name not specified, ‘it’ is the name

objistics™

Groovy Ranges

```
(1..10).each {
  print "${it} "        // outputs 1 2 3 4 5 6 7 8 9 10
}
println()

('a'..'d').each {
  print "${it} "        // outputs 'a', 'b', 'c', 'd'
}
println()
```

- Groovy gives us Range types
  - Nothing like it in Java
- Range literal declared like this: start..end
  - Start and end can be int or char types
- Parentheses optional in method calls (except for no arg methods)
- Groovy Strings (GStrings) expands macros defined in ${} expressions
  - Groovy String literals can be declared with single quote, but won't get macro expansion

Groovy multi-line strings

```
'''
<html>
    <head>
       <title>A title</title>
    </head>
    <body>
      <p>Some content</p>
    </body>
</html>
'''
```

- Multiline strings are defined with triple quotes
- Formatting is preserved
- Single quotes good for hardcoded text

Groovy Strings

- Groovy has three flavors of Strings
  - A normal (Java) String literal is formed with single quotes

```
def msg = 'Hello'
```

  - A Groovy String (GString) is formed with double quotes
    - This type of String can include expressions evaluated at runtime
    - If no expressions are in the String, it falls back to a Java String

```
def msg = "Hello"              // a java.lang.String
def msg = "Hello $name"        // a groovy.lang.Gstring
def msg = "Hello ${name.toUpperCase()}"
```

Groovy multi-line strings

```
"""
<html>
    <head>
       <title>${title}</title>
    </head>
    <body>
      <p>${content}</p>
    </body>
</html>
"""
```

- Multiline strings are defined with triple quotes
- Formatting is preserved
- Use double quotes for expression evaluation

Groovy Misc operators

```
// Elvis
void lookup(String code) {
  String properCode = code ?: “”
  …
}

// Safe-navigation
void lookup(Map<String, Integer> map, String code) {
  Integer value = map[code?.toUpperCase()]
}

// Spread
def dates = [new Date(), new Date() + 1]
println dates*.date // outputs [today date,tomorrow date]

// Spaceship
println dates[0] <=> dates[1] // outputs -1
println dates[1] <=> dates[0] // outputs 1
```

Groovy Duck Typing

```
// Groovy is dynamically typed. This means the type is
// evaluated at runtime, not compile time
// If it walks like a duck and talks like a duck, Groovy
// treats it like a duck

// y acts like a String by responding to a call to
// toUpperCase, so Groovy treats it like a String

Object y = ‘Good Bye’
println y.toUpperCase() // Java won’t allow this

// def means we’re explicitly deferring type
def x = 100
x = ‘Hello’
```

Groovy 101 – Duck Typing

```
// Duck typing and Groovy behaviors can fool us.
// The following works only because any object can be
// transformed into a String

String z = new Date()
println z.toLowerCase()

// And the following compiles fine, but will it work?
Date d = ‘05—07-2014’

// No! Because Groovy can’t convert from String to Date
// at runtime, we get a GroovyCastException
```

```
org.codehaus.groovy.runtime.typehandling.GroovyCastExcept
ion: Cannot cast object '05—07-2014' with class
'java.lang.String' to class 'java.util.Date'
```

objistics™

Groovy 101 – Monkey Patching

- Since Groovy is a dynamic language, we can add properties and behaviors to classes and instances at runtime
- We can do this even for final classes and Java API classes
  - Don't worry, we're not really changing the class!
- Dynamically added things are available to Groovy code, but not to Java

```
Integer.metaClass.squared = { delegate * delegate }
println 16.squared()            // 256
println 16.squared().squared()  // 65536
```

This ‘n’ That

- To reference a class, you need only specify the class name
  - Class clazz = MyClass // not MyClass.class
- Any method that starts with ‘get’ or ‘is’ can be accessed like a property
  - Class class = myObj.class // same as myObj.getClass()

A Brief Overview of Groovy 2.3

- Groovy Disadvantages
  - Being dynamic, we don't get as much help from the compiler
    - We can mitigate this with @TypeChecked if we know we're not relying on dynamic features in a method
  - Dynamic dispatching imposes a performance penalty
    - We can mitigate this with @StaticCompile if we know we're not relying on dynamic features in a method
  - Groovy is **not** 100% syntactically identical to Java
    - A few of the differences can be real gotchas

objistics™

A Brief Overview of Groovy 2.3

- **Static vs Dynamic**
  - Java is statically typed
  - Groovy is dynamically typed

- **int x = "Hello"**
  - Java compiler flags this is as compile error
  - Groovy compiler doesn't see this as an error
  - Ditto for passing the wrong type as method args
    - However, at runtime, you'll get a **GroovyCastException**
  - We often omit the type and just declare vars with *def;* e.g. **def x = 5**

- **"Hello".doSomethingStrange()**
  - Java compiler flags this is as compile error
  - Groovy compiler doesn't see this as an error
    - However, at runtime, you'll get a **MissingMethodException**
    - We can exploit that to do some cool dynamic things

objistics™

A Brief Overview of Groovy 2.3

- **foo == bar**
  - In Java foo == bar means "does foo reference the same object as bar?"
    - We hardly ever want to do that!
  - In Groovy, it means the same as **foo.equals(bar)**
    - This is actually how Java *should* have done it
    - If you want to compare if the two object reference the same object, **foo.is(bar)**

```
def x = "Hello"
def y = new String("Hello")
println x == y      // true
println x.is(y)     // false
```

objistics™

A Brief Overview of Groovy 2.3

- **‘a’ == “a”**
  - In Java, single quotes are used for literal chars
  - In Groovy, single quotes are used for literal Strings
  - We can coerce the type to a char with the ***as*** operator

```
void doSomething(char c) {
    // use it…
}

doSomething(‘a’ as char);
```

A Brief Overview of Groovy 2.3

- The word ***in*** is a Groovy keyword
  - If your code has **InputStream in = ...** you have to change it
- All literals in Groovy are really Object types
- Floating point types are BigDecimal by default

```
def calculateAreas(List radii) {
   def pi = 3.14159
   List areas = []
   for(radius in radii) { areas << pi * radius * radius }
   areas
}
```

  - To avoid the performance penalties of BigDecimal, coerce the type
    - def pi = 3.14159d // we can coerce to Double
    - def pi = 3.14159f // or Float

objistics™

A Brief Overview of Groovy 2.3

- Java array declarations won't work with Groovy
  - Java way
    - **int[] a = {1,2,3};**
  - Groovy way
    - **int[] a = [1,2,3]**

- Visibility differences
  - Methods and classes are public by default
    - No public modifier needed
    - This means by default, there's no package-local visibility
  - Fields are private by default
    - No private modifier needed
  - Groovy permits accessing and mutating private data
    - But don't do it!

That's a very, very brief overview of Groovy.

Now let's see how we can bring Groovy goodness to Spring 4…

Adding Groovy to your Spring projects

- The current release of Groovy is v2.3.4
  - Groovy 2.0 introduced more modular Groovy distributions making the core groovy JAR 50% smaller than getting everything in one JAR
- However, the easiest way to get all you need is still with the groovy-all JAR
  - There are two flavors
    - Groovy for Java <7
    - Groovy “indy” for Java 7+ with invokeDynamic support
- There are a few ways to add Groovy to your Spring projects

Adding Groovy to your Spring projects

- Developers should download the Groovy distribution to get tools, source code, documentation, etc.
  - http://groovy.codehaus.org/Download
  - The distribution has a groovy-all JAR, but…
- If you're managing dependencies manually…
  Please seek counseling immediately!
- Until your appointment, you can find the JARs in the embedded folder in your Groovy install directory
  - groovy-all-2.3.4.jar
  - groovy-all-2.3.4-indy.jar

objistics™

Adding Groovy to your Spring projects

- Adding Groovy via Maven

```
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>2.3.4</version>
    <!-- remove the classifier if using Java 6 -->
    <classifier>indy</classifier>
</dependency>
```

- Besides just the Groovy JAR, you'll probably want to use the GMavenPlus plugin to handle compiling Groovy code in src/main/groovy and more
  - http://groovy.github.io/GMavenPlus/index.html

objistics™

Adding Groovy to your Spring projects

- Adding Groovy dependency via Gradle

```
dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.4:indy'
}
```

- Besides just the Groovy JAR, you'll need to use the Groovy plugin to handle compiling Groovy code in src/main/groovy and more
  - http://www.gradle.org/docs/current/userguide/groovy_plugin.html

Why Groovy Beans Make Better Spring Beans

- There are all kinds of Spring beans, like
  - Configuration
  - Spring MVC Controllers
  - Services
  - Repositories (i.e. DAOs)
  - Value objects (i.e. POJO DTOs)
  - Property Place Holders
  - Infrastructure
    - Datasource
    - SessionFactory / EntityManagerFactory
- Just about any object could be a Spring bean

objistics™

Why Groovy Beans Make Better Spring Beans

- So which kinds can be coded as Groovy beans?
  - All of them!
- But *maybe* there are some beans that shouldn't be done in Groovy
  - First, do NOT rewrite every complex bean class in Groovy just to be Groovy. Do rewrite them in Groovy when you touch them
  - If a Spring bean is doing very heavy computations where every ounce of performance is needed, *maybe* Groovy isn't a good choice even if @TypeChecked or @CompileStatic are used
- For all others, Groovy should be considered for its concise code

objistics™

Why Groovy Beans Make Better Spring Beans

- Would you rather code this (or even wade through this code generated by your IDE)?

```
// Address.java
public class Address {
    private String street1;
    private String street2;
    private String street3;
    private String city;
    private String state;
    private String zip;

    ...
```

Why Groovy Beans Make Better Spring Beans

```
    ...
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Address address = (Address) o;
        if (city != null ? !city.equals(address.city) : address.city
!= null) return false;
        if (state != null ? !state.equals(address.state) :
address.state != null) return false;
        if (street1 != null ? !street1.equals(address.street1) :
address.street1 != null) return false;
        if (street2 != null ? !street2.equals(address.street2) :
address.street2 != null) return false;
        if (street3 != null ? !street3.equals(address.street3) :
address.street3 != null) return false;
        if (zip != null ? !zip.equals(address.zip) : address.zip !=
null) return false;
        return true;    } ...
```

Why Groovy Beans Make Better Spring Beans

```
...
public String getStreet1() {
    return street1;
}

public void setStreet1(String street1) {
    this.street1 = street1;
}

public String getStreet2() {
    return street2;
}

public void setStreet2(String street2) {
    this.street2 = street2;
}
...
```

objistics™

Why Groovy Beans Make Better Spring Beans

```
...
public String getStreet3() {
    return street3;
}

public void setStreet3(String street3) {
    this.street3 = street3;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}
...
```

objistics™

Why Groovy Beans Make Better Spring Beans

```
...
public String getState() {
    return state;
}

public void setState(String state) {
    this.state = state;
}

public String getZip() {
    return zip;
}

public void setZip(String zip) {
    this.zip = zip;
}
...
```

objistics™

Why Groovy Beans Make Better Spring Beans

```
    ...
    @Override
    public int hashCode() {
        int result = street1 != null ? street1.hashCode() : 0;
        result = 31 * result + (street2 != null ? street2.hashCode()
: 0);
        result = 31 * result + (street3 != null ? street3.hashCode()
: 0);
        result = 31 * result + (city != null ? city.hashCode() : 0);
        result = 31 * result + (state != null ? state.hashCode() :
0);
        result = 31 * result + (zip != null ? zip.hashCode() : 0);
        return result;
    }
    ...
```

Why Groovy Beans Make Better Spring Beans

```
    ...
    @Override
    public String toString() {
        return "Address{" +
                "street1='" + street1 + '\'' +
                ", street2='" + street2 + '\'' +
                ", street3='" + street3 + '\'' +
                ", city='" + city + '\'' +
                ", state='" + state + '\'' +
                ", zip='" + zip + '\'' +
                '}';
    }
}
```

objistics™

Why Groovy Beans Make Better Spring Beans

- Or would you rather code/maintain this?

```
// Address.groovy

@EqualsAndHashCode
@ToString(includeNames=true)
class Address {
    String street1
    String street2
    String street3
    String city
    String state
    String zip
}
```

- That's all there is. Really!
- At runtime, these two will have the same behavior

objistics™

Why Groovy Beans Make Better Spring Beans

- The beans I'll show will all be Groovy
- You decide whether the small learning curve to write simple, elegant, fluent Groovy code is worth it for you

objistics™

Spring Configuration using Groovy

- We can configure Spring beans three ways
  - XML
  - Annotations
  - Code-based Configuration
- These can be mixed and matched, but let's look at them separately

Spring Configuration using Groovy

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="custService" class="my.company.CustServiceImpl">
           <property name="someProperty" value="2014"/>
           <property name="customerDao" ref="customerDao"/>
       </bean>

       <bean id="customerDao" class="my.company.CustomerDaoImpl">
           <property name="sessionFactory" ref="sessionFactory"/>
       </bean>
</beans>
```

Found elsewhere

Spring Configuration using Groovy

- With annotations, we tell Spring what base package (and subpackages) to scan for annotated types

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="my.company"/>

</beans>
```

Spring Configuration using Groovy

```
package my.company
import my.company.CustomerDao;

@Service // a specialized stereotype of @Component
public class CustServiceImpl implements CustService {
  private int someProperty;
  private CustomerDao customerDao;

  @Autowired
  public void setSomeProperty(int value) { someProperty = value; }

  @Autowired
  public void setCustomerDao(CustomerDao dao) { customerDao = dao; }

  @Override
  public void execute() { ... }
}
```

Spring Configuration using Groovy

- With code-based Configuration, we use code to tell Spring how to wire beans
  - One big advantage of this approach is we get programmatic control over the wiring
- Classes that do the configuration are tagged with the @Configuration annotation
  - These are @Components
- Classes that are beans being configured in the Configuration are tagged with @Bean

Spring Configuration using Groovy

```
/* We can get away from XML any for component:scan by passing
   the config file(s) to our context creator:
   AnnotationConfigApplicationContext ctx = new
      AnnotationConfigApplicationContext(AppConfig.class); */

@Configuration
public class AppConfig {

    @Bean
    public CustService custService() {
       CustServiceImpl cs = new CustServiceImpl();
       cs.setCustomerDao(customerDao());
       cs.setSomeProperty(2014);
       return cs;
    }
    @Bean
    public CustomerDao customerDao() {
      return new CustomerDaoImpl();
    }  }
```

Spring Configuration using Groovy

```
@Configuration
public class AppConfig {

    @Bean
    public CustService custService() {
       // C'tor injection makes the code cleaner
       return new CustServiceImpl(customerDao(), 2014);
    }

    @Bean
    public CustomerDao customerDao() {
      return new CustomerDaoImpl();
    }
}
```

Spring Configuration using Groovy

- SpringSource is definitely favoring moving to code-based configuration
  - We leave XML behind as a bad memory
  - We get finer grained control over configuration
  - We can use programming logic to drive configuration
- Using Groovy makes this even easier using the Groovy Bean Builder
  - This provides us a simple *DSL* for configuring beans

Spring Configuration using Groovy

- Configuration with Groovy

```
import my.company.*

// this is processed as a groovy script
beans {
  customerDao(CustServiceImpl) // must be defined first

  // for setter injection
  custService(CustServiceImpl) {
    someProperty = 2014
    customerDao = ref(customerDao) // ref not needed
  }
}
```

Spring Configuration using Groovy

- Configuration with Groovy

```
import my.company.*

beans {

  customerDao(CustServiceImpl) // must be defined first

  // for c'tor injection
  custService(CustServiceImpl, customerDao, 2014)
}
```

Implementation

Dao being injected

Other injected prop

Spring Configuration using Groovy

- We can even declare anonymous inner beans

```
import my.company.*

beans {

  custService(CustServiceImpl) {
    someProperty = 2014

    // inject anonymous inner bean
    customerDao = bean CustomerDaoImpl
  }

}
```

Spring Configuration using Groovy

- Setting bean metadata

```
import my.company.*

beans {
  customerDao(CustServiceImpl) // must be defined first

  // for setter injection
  custService(CustServiceImpl) { bean ->
    someProperty = 2014
    customerDao = customerDao

    bean.scope = 'prototype'
  }
}
```

Leveraging Groovy in SpringBoot

- Spring Boot

  … takes an opinionated view of building production-ready Spring applications.

  Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible.

Leveraging Groovy in SpringBoot

- How can Groovy be used in Spring Boot?
  - CLI
    - spring run MyApp.groovy
  - Spring beans written in Groovy
    - Write less, drink more!
  - Bean Configuration
  - Groovy Template Engine
  - Use standalone GORM

Leveraging Groovy in SpringBoot

- Let's make a simple REST service using Groovy

```
package my.company.boot

import ...

@RestController
@EnableAutoConfiguration
class RestExample {
    @RequestMapping("/sayHello/{audience}")
    String home(@PathVariable String audience) {
        "Hello $audience!"
    }

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

Leveraging Groovy in SpringBoot

- Now let's run it **`> mvn spring-boot:run`**

```
C:\Windows\system32\cmd.exe - Example_Rest.cmd
[INFO] --- spring-boot-maven-plugin:1.1.3.RELEASE:run (default-cli) @ spring.groovy.goodness ---
[INFO] Attaching agents: []

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.1.3.RELEASE)

2014-07-15 23:01:54.315  INFO 7064 --- [           main] my.company.boot.RestExample              : Starting RestExample on TitanVM with PID 7064 (start
2014-07-15 23:01:54.370  INFO 7064 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded
2014-07-15 23:01:54.848  INFO 7064 --- [           main] o.s.b.f.s.DefaultListableBeanFactory     : Overriding bean definition for bean 'beanNameViewRes
ry=false; factoryBeanName=org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration; factoryMethodName=beanN
MvcAutoConfiguration$WhitelabelErrorViewConfiguration.class]] with [Root bean: class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3; dep
ConfigurationAdapter; factoryMethodName=beanNameViewResolver; initMethodName=null; destroyMethodName=(inferred); defined in class path resource [org/spr
2014-07-15 23:01:55.641  INFO 7064 --- [           main] .t.TomcatEmbeddedServletContainerFactory : Server initialized with port: 8080
2014-07-15 23:01:55.854  INFO 7064 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat
2014-07-15 23:01:55.855  INFO 7064 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/7.0.54
2014-07-15 23:01:55.946  INFO 7064 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2014-07-15 23:01:55.946  INFO 7064 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed
2014-07-15 23:01:56.387  INFO 7064 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        : Mapping servlet: 'dispatcherServlet' to [/]
2014-07-15 23:01:56.390  INFO 7064 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2014-07-15 23:01:56.675  INFO 7064 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of ty
2014-07-15 23:01:56.745  INFO 7064 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/sayHello/{audience}],methods=[],params=[]
2014-07-15 23:01:56.745  INFO 7064 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/eval/{expr}],methods=[],params=[],headers=
2014-07-15 23:01:56.750  INFO 7064 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],methods=[],params=[],headers=[],co
ramework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
2014-07-15 23:01:56.750  INFO 7064 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],methods=[],params=[],headers=[],co
ErrorController.errorHtml(javax.servlet.http.HttpServletRequest)
2014-07-15 23:01:56.779  INFO 7064 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class or
2014-07-15 23:01:56.780  INFO 7064 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [
2014-07-15 23:01:57.286  INFO 7064 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2014-07-15 23:01:57.328  INFO 7064 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080/http
2014-07-15 23:01:57.331  INFO 7064 --- [           main] my.company.boot.RestExample              : Started RestExample in 3.399 seconds (JVM running fo
```

Leveraging Groovy in SpringBoot

- Whoa!
- Did you notice these lines in the output?

```
: Server initialized with port: 8080
: Starting service Tomcat
: Starting Servlet Engine: Apache Tomcat/7.0.54
: Initializing Spring embedded WebApplicationContext
: Root WebApplicationContext: initialization completed in 1579 ms
```

- Spring Boot embedded Tomcat in our JAR allowing us to run a micro web service without needing to install a server, make a WAR file, deploy, etc.

Leveraging Groovy in SpringBoot

- What else can we do with Groovy in Spring Boot?
- Let's make a full, template-driven web page using Groovy Markup Templates

… And no need to install Tomcat!

objistics™

Leveraging Groovy in SpringBoot

- The Groovy markup template engine provides an innovative templating system based on the builder syntax. It offers various key features:
  - Hierarchical (builder) syntax to generate XML-like contents (in particular, HTML5)
  - Template includes
  - Compilation of templates to byte code for fast rendering
  - Internationalization
  - Layout mechanism for sharing structural patterns
  - Optional type checking

Leveraging Groovy in SpringBoot

```
// The Controller

@Controller
@EnableAutoConfiguration
class SimpleController {

  @RequestMapping("/")
  def home() {
    new ModelAndView("views/home", [bootVersion:
              Banner.package.implementationVersion,
              groovyVersion: GroovySystem.version])
  }
  static void main(String[] args) {
        SpringApplication.run SimpleController, args
    }
}
```

Leveraging Groovy in SpringBoot

```
// A Layout Template (/layouts/main.tpl)

html {
  head {
    title(pageTitle)
    link(rel:'stylesheet', href:'/css/bootstrap.min.css')
  }
  body {
    div(class:'container') {
      h1(title)
      div { mainBody() }
    }
  }
}
```

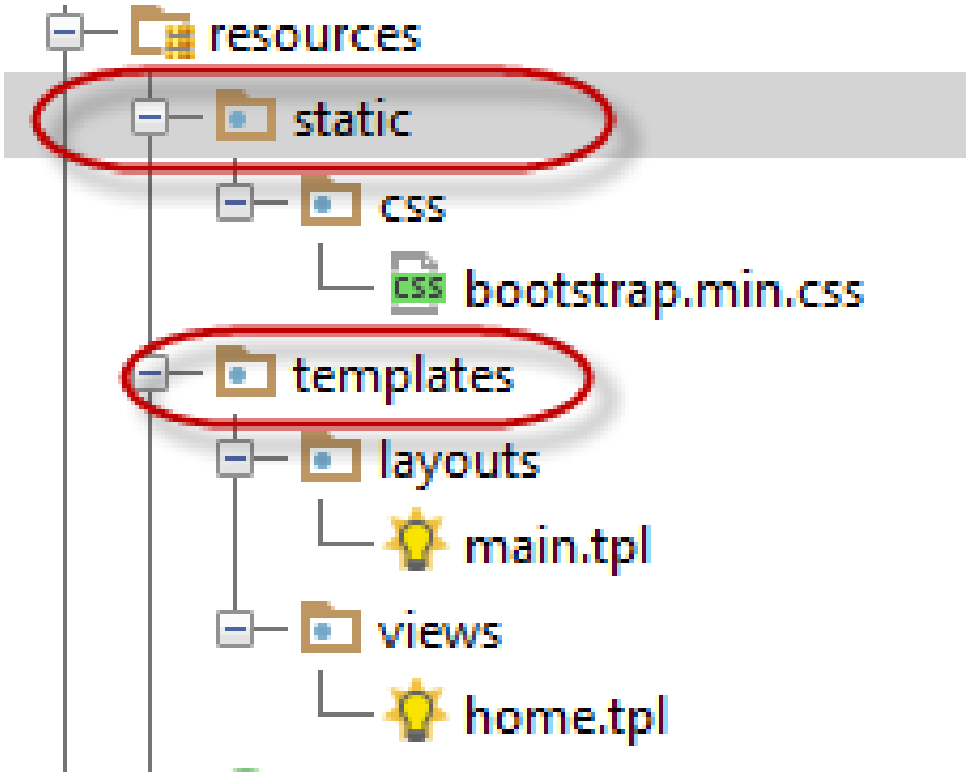
objistics™

Leveraging Groovy in SpringBoot

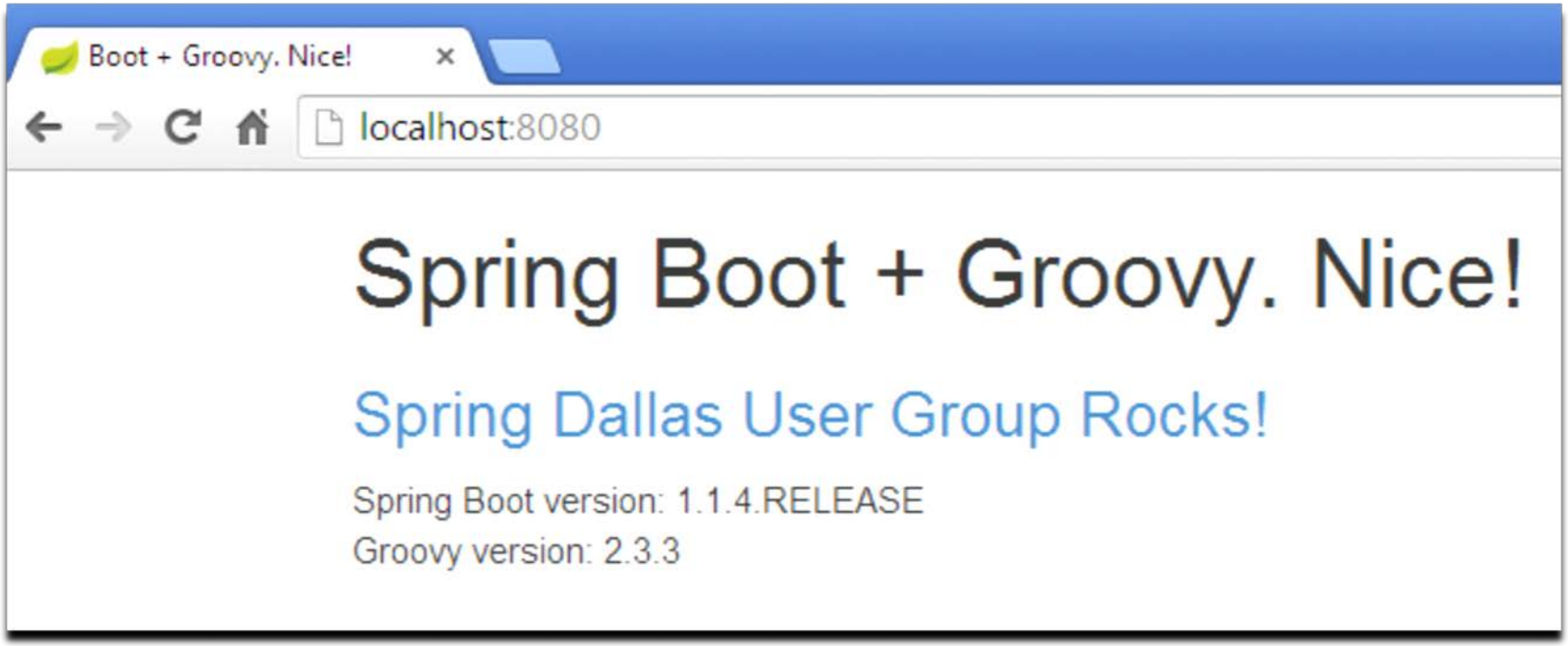
```
// The Home page (/views/home.tpl)

layout 'layouts/main.tpl',
pageTitle: 'Boot + Groovy. Nice!',
title: 'Spring Boot + Groovy. Nice!',
mainBody: contents {
    h3 {
        a(class:'brand',
          href:'http://www.springdallasug.org/',
          target:'_blank') {
            yield 'Spring Dallas User Group Rocks!'
          }
    }
    div("Spring Boot version: $bootVersion")
    div("Groovy version: $groovyVersion")
}
```

objistics™

Leveraging Groovy in SpringBoot



Leveraging Groovy in SpringBoot



objistics™

Leveraging Groovy in SpringBoot

- Wow! What *else* can we do with Groovy in Spring Boot?
- Boot has a Command Line Interface (CLI)
  - You can experiment with Boot in the CLI using Groovy scripts
- In short, Spring Boot, and Spring in general, is embracing Groovy

Spring Data, Meet GORM

- The latest versions of Groovy and Grails supports GORM (Grails ORM) standalone with Groovy

- What's so special about GORM?
  - It provides a constraints API for easy validation
  - It dynamically and automatically decorates domain entities with CRUD and other persistence methods
  - It allows creation of dynamic finders that create queries for you based on method call
  - Easy to write, fluent where queries
  - Object-oriented criteria queries
  - Works with SQL and NoSQL datastores
  - And much, much more

- Let's use GORM in another Boot example to get a taste …

objistics™

Using Gradle as Your Spring Project Build Tool

- What's a *Gradle*?
  - Gradle is build automation *evolved* (… from Maven, Ant, make, etc.)
  - Gradle can automate
    - Building
    - Testing
    - Publishing
    - Deployment
    - And more!
  - Perfect for software packages, generated static websites, generated documentation, whatever!
- Gradle combines the power and flexibility of Ant with the dependency management and conventions of Maven
- It's powered by a Groovy DSL
- Gradle is quickly becoming the build system of choice

Resources

- Links
  - http://spring.io
  - http://groovy.codehaus.org
  - http://groovy.codehaus.org/Download
  - http://groovy.codehaus.org/Things+you+can+do+but+better+leave+undone
  - http://groovy.codehaus.org/Differences+from+Java
  - http://docs.codehaus.org/display/GroovyJSR/GEP+10+-+Static+compilation
  - http://groovy.codehaus.org/Groovy+2.0+release+notes
  - http://grails.org/doc/latest/guide/spring.html
  - http://beta.groovy-lang.org/docs/groovy-2.3.4/html/documentation/markup-template-engine.html
  - http://www.gradle.org
  - http://www.slideshare.net/vasya10/spring-boot-3g

- Jack Frosch
  - http://www.linkedin.com/in/jackfrosch
  - jackfrosch@gmail.com

objistics™

Questions?

objistics™