Tutorial de Introducción a Grails

Adrián Cepillo Macías

7 de marzo de 2012

Índice

1. Instalación de Grails

Siguiendo con el tutorial anterior ahora vamos a instalar Grails de nuevo manualmente. Primero descargar la aplicacion desde aquí. La extraemos en el directorio que prefiramos.

Luego establecer una variable de entorno:

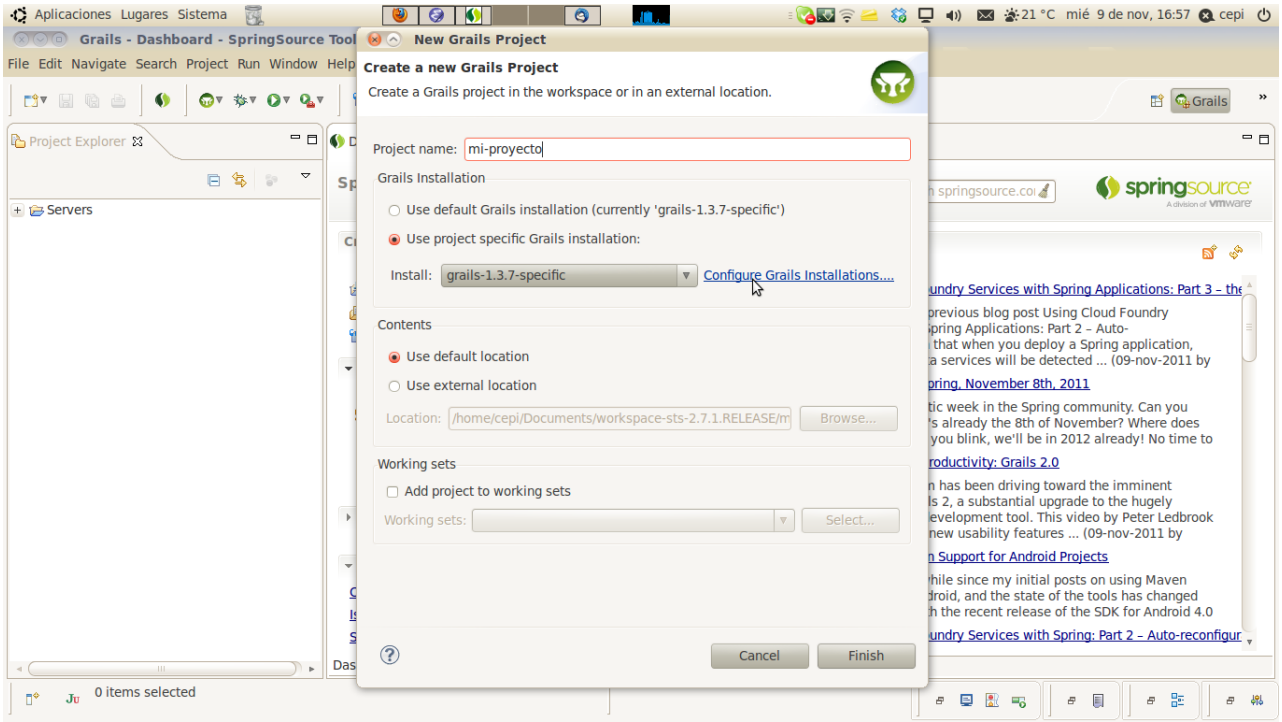
- $ export GRAILS_HOME= /RUTA_EXTRACCION/bin

Podemos hacer esto permanente introduciendo este comando al final del fichero /.bashrc.

Aún así nosotros no usaremos grails por consola, se muestra esta instalación por si se quiere usar grails independientemente de eclipse. Nosotros veremos en este tutorial algunos de los comandos que grails contiene para crear una aplicacion, testearla, ... pero todo desde el prompt que nos ofrece Eclipse.

2. Creando nuestra aplicación

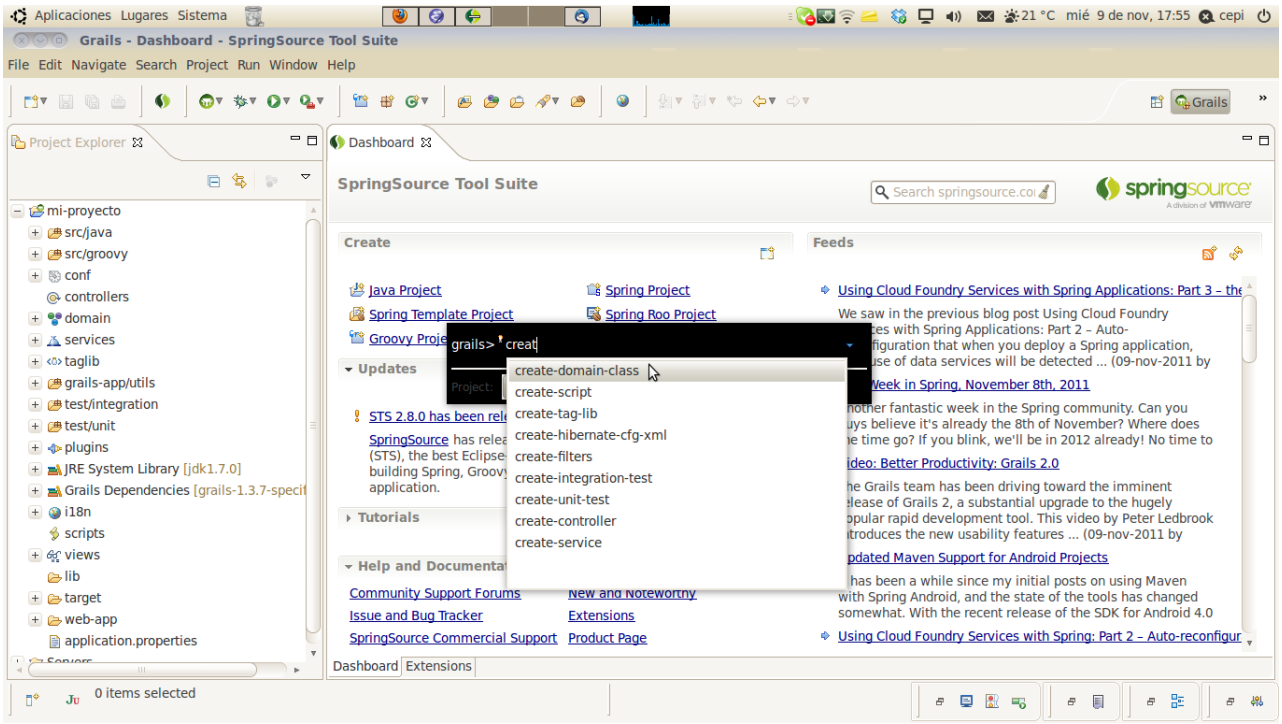
Vamos a crear de nuevo la aplicación *Book*. Creamos un nuevo proyecto, esta vez elegiremos una instalación específica. Para ello debemos clickar en *Configure Grails Installations* y añadiremos la versión que acabamos de descargarnos, indicando el directorio en el que se encuentra.



Nota: El comando de *Grails* para crear un proyecto es, **$ grails create-app mi-proyecto**. *Eclipse* no permite crear un proyecto, introduciendo este comando en el prompt.

Para conocer algunos de los comandos de *Grails* vamos a usar el prompt que nos presenta *Eclipse STS*. Podemos encontrarlo en Navigate->Open Grails Commands Prompt. O clickar en el simbolo de *Grails* en la barra de herramientas o pulsar Ctrl+Alt+Shift+G (Cmd+Alt+Shift+G en Mac).

Con Ctrl+Space (Cmd+Space) nos mostrará el asistente.



Ahora crearemos la clase de dominio introduciendo en el prompt:

- grails> create-domain-class org.example.Book

Modificamos el código:

```
1  package org.example
2
3  class Book {
4          String title
5          String author
6
7          static constraints = {
8                  title(blank: false)
9                  author(blank: false)
10         }
11 }
```

Y lo mismo para el controlador:

- grails> create-controller org.example.Book

```
1  package org.example
2
3  class BookController {
4
```

```
5           def scaffold = Book
6   }
```

Corremos la aplicación ejecutando:

- grails> run-app

3. Importando datos de prueba

Es probable que queramos probar la aplicación con ciertos datos y sería muy pesado tener que introducirlos a mano una y otra vez. Para ello podemos modificar el fichero de configuración *BootStrap.groovy*, dentro de *conf* en la jerarquía del proyecto.

```
1   import org.example.Book
2
3   class BootStrap {
4
5    def init = { servletContext ->
6     // Check whether the test data already exists.
7      if (!Book.count()) {
8        new Book(author: "John R. R. Tolkien", title: "The Simarillion").
             save(failOnError: true)
9        new Book(author: "John R. R. Tolkien", title: "The Hobbit").save(
             failOnError: true)
10       new Book(author: "John R. R. Tolkien", title: "The Lord of the
             Rings").save(failOnError: true)
11       new Book(author: "George R. R. Martin", title: "A Song of Ice and
             Fire: 1 A Game of Thrones").save(failOnError: true)
12       new Book(author: "George R. R. Martin", title: "A Song of Ice and
             Fire: 2 A Clash of Kings").save(failOnError: true)
13       new Book(author: "George R. R. Martin", title: "A Song of Ice and
             Fire: 3 A Storm of Swords").save(failOnError: true)
14       new Book(author: "George R. R. Martin", title: "A Song of Ice and
             Fire: 4 A Feast for Crows").save(failOnError: true)
15     }
16    }
17
18    def destroy = {
19    }
20  }
```

Como podemos comprobar hemos redefinido *init* de esta forma cada vez que se inicie la aplicación se importaran estos datos a el modelo señalado.

Debemos notar la llamada a *Book.count()*. Este método devuelve el número de libros que existen ya en la base de datos. Con el *if(){...}* decimos que en caso que no existan otros datos de testeo, se introduzcan estos datos.

Por otra parte, llamamos al método *Book.save()*, el cuál guarda el objeto en la base de datos. Y establecemos la opción *failOnError: true* indicando que se lance una excepción en caso que el método falle.

4. Configuración del Data Source

Hasta ahora hemos estado trabajando con la base de datos HSQLDB en memoria, para desarrollo y testeo. Vamos a configurar una base de datos alternativa. Para ello modificamos el fichero *DataSource.groovy*.

```
dataSource {
    pooled = true
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class = 'net.sf.ehcache.hibernate.EhCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop
               ','update'
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
                        driverClassName = "com.mysql.jdbc.Driver"
            url = "jdbc:mysql://localhost/mi-proyecto"
                        username="root"
                        password="root"
        }
    }
}
```

En este caso estamos usando Mysql. Grails ya tiene el driver para JDBC pero es necesario descomentar varias líneas del fichero *BuildConfig.groovy*.

```
grails.project.class.dir = "target/classes"
grails.project.test.class.dir = "target/test-classes"
grails.project.test.reports.dir = "target/test-reports"
//grails.project.war.file = "target/${appName}-${appVersion}.war"
grails.project.dependency.resolution = {
    // inherit Grails' default dependencies
    inherits("global") {
        // uncomment to disable ehcache
        // excludes 'ehcache'
    }
    log "warn" // log level of Ivy resolver, either 'error', 'warn', '
       info', 'debug' or 'verbose'
    repositories {
        grailsPlugins()
        grailsHome()
        grailsCentral()

        // uncomment the below to enable remote dependency resolution
        // from public Maven repositories
        mavenLocal()
        mavenCentral()
        mavenRepo "http://snapshots.repository.codehaus.org"
        mavenRepo "http://repository.codehaus.org"
        mavenRepo "http://download.java.net/maven/2/"
        mavenRepo "http://repository.jboss.com/maven2/"
    }
    dependencies {
        // specify dependencies here under either 'build', 'compile', '
           runtime', 'test' or 'provided' scopes eg.

        runtime 'mysql:mysql-connector-java:5.1.13'
    }
}
```

5. Ampliando nuestra aplicación

5.1. Creando la clase Author

Vamos a añadir algunos elementos nuevos a la aplicación empezando por crear otra clase llamada *Author* que relacionar con *Book*. Se muestra primero el código de la *domain class*:

```
package org.example

class Author {

        String name
        String surname
```

```
7
8       static constraints = {
9                   name(blank: false)
10                  surname(blank: false)
11      }
12  }
```

Y por otra parte el controlador:

```
1   package org.example
2
3   class AuthorController {
4
5       def scaffold = Author
6   }
```

5.2. Relacionando con GORM

```
1   package org.example
2
3   class Author {
4           static hasMany = [book:Book]
5
6           String name
7           String surname
8
9           static constraints = {
10                  name(blank:false)
11                  surname(blank:false)
12          }
13
14  }
```

```
1   package org.example
2
3   class Book {
4           String title
5           Author author
6
7   }
```

5.3. Modificando un controlador generado

5.4. Cambiemos de estilo