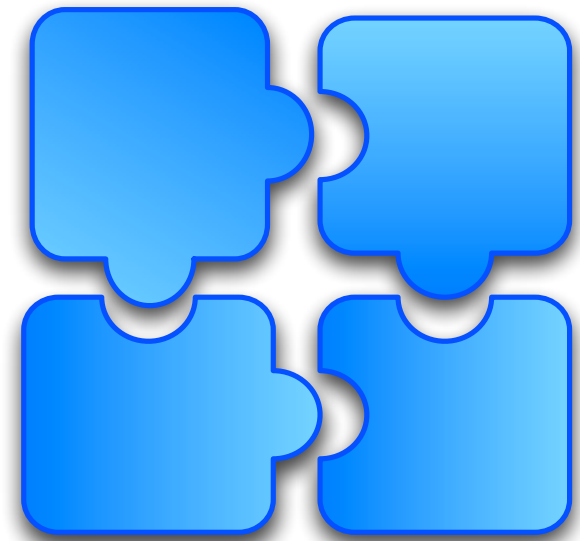


# The Grails Plugin System

Modular Application Development With Grails

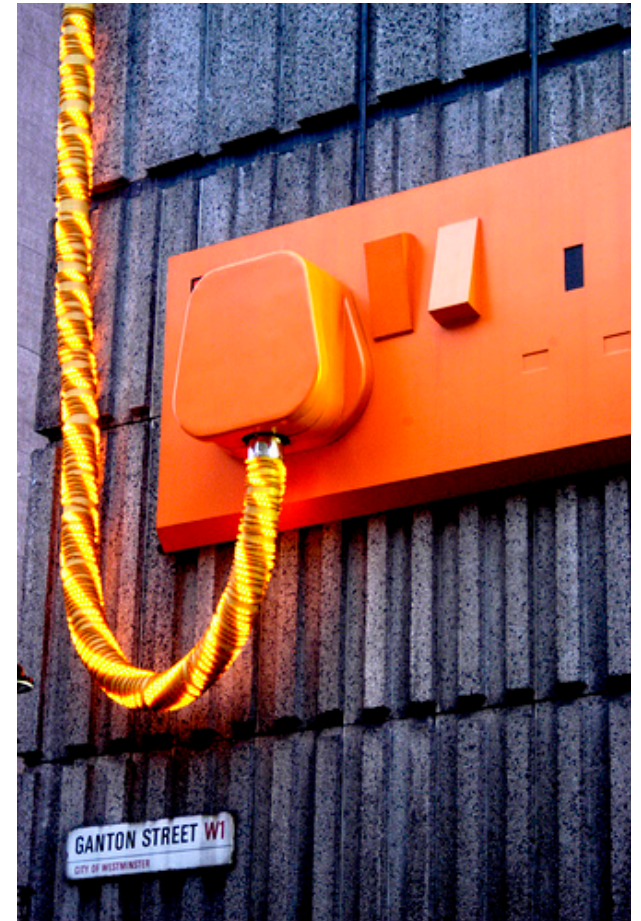
# The Background

- Grails is designed to wire together different libraries and make them easy to use
- In this sense it can be seen as a "platform for runtime configuration"
- De-coupling those components was hard without a well defined system

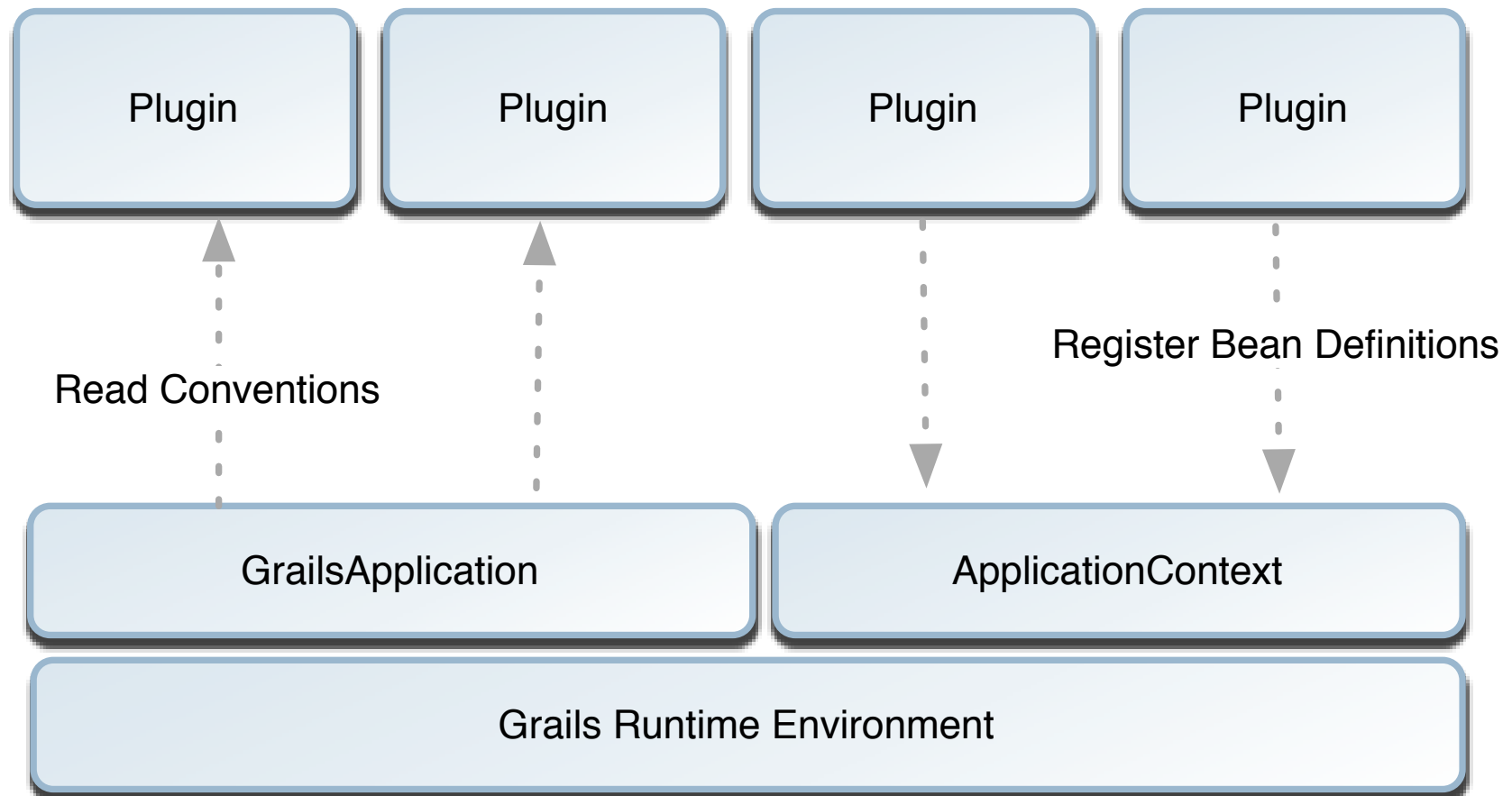


# The Extension Points

- The Build System
- Spring Application Context
- Dynamic method registration
- Auto Reloading
- Container Config



# Plug-in Architecture



# Plug-in Goals

- The system should embrace convention-based approaches and DRY
- All required extension points should be satisfied
- Plug-ins should be easy to distribute, install with zero configuration



# What is a Plug-in?

- Just like a normal Grails project!
- The only difference is the presence of a  
`*GrailsPlugin.groovy` file
- Use `grails create-plugin` to create one!

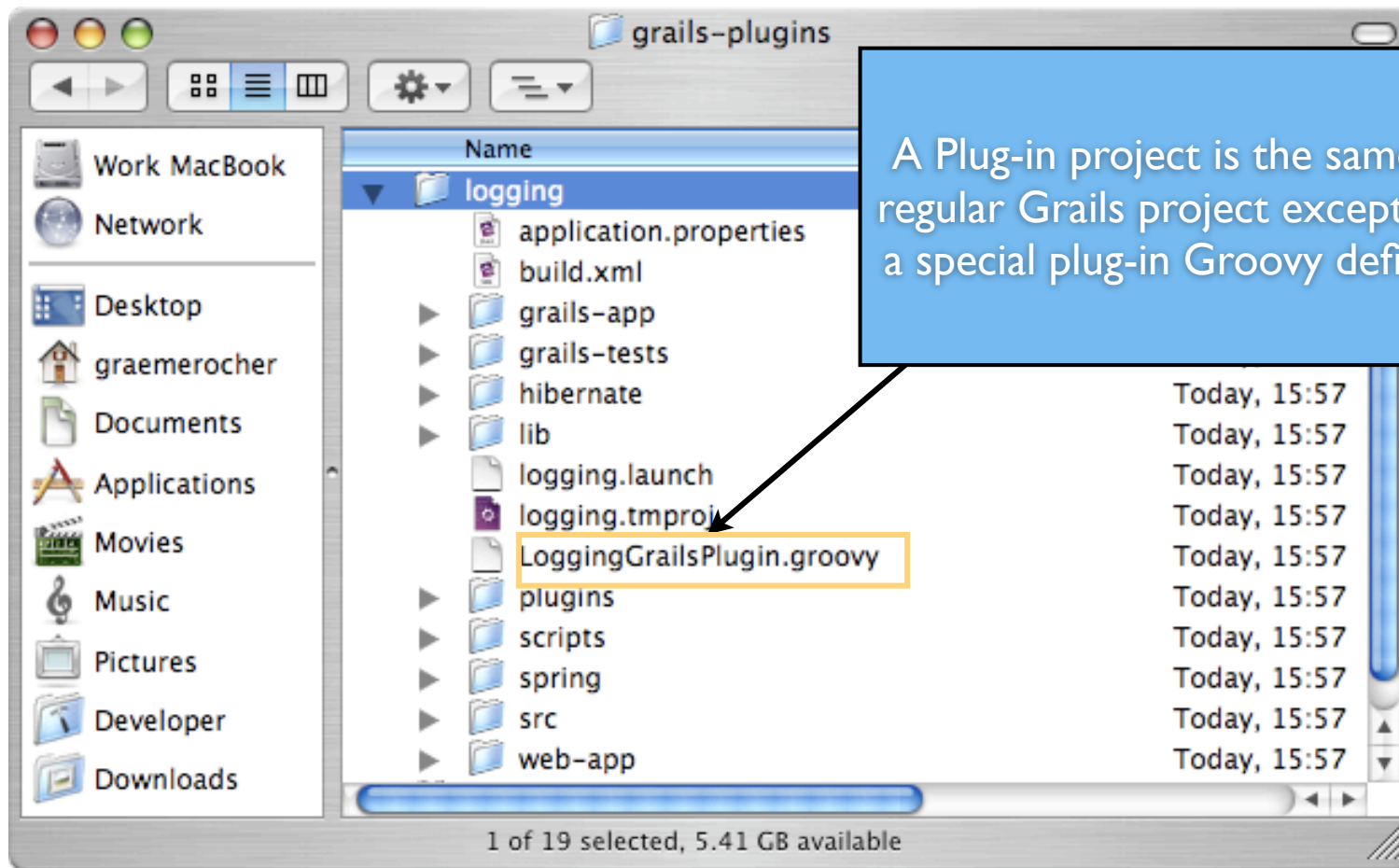


# What can a Plug-in do?

- Add new methods, constructors, properties etc. to any class at runtime
- Perform runtime Spring configuration
- Modify web.xml on the fly
- Add new controllers, tag



# A Plug-in Project

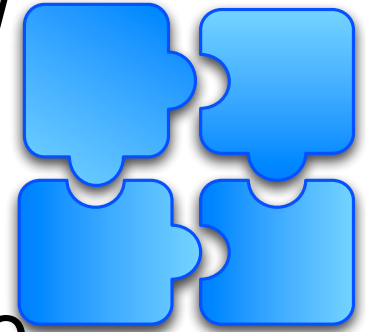


A Plug-in project is the same as a regular Grails project except it has a special plug-in Groovy definition



# Plugging In Dynamic Methods

- We know the conventions, but how do we easily take advantage of them to modify types?
- Grails users needed an easy way to add dynamic methods, properties, constructors etc.
- No out-of-the box solution in Groovy so we created **ExpandoMetaClass (EMC)**



# Overview of Plug-in Closures

- A Plug-in can specify multiple closures each of which can manipulate Grails:
  - ▶ **doWithSpring** - Participate in Spring configuration
  - ▶ **doWithApplicationContext** - Post ApplicationContext initialisation activities
  - ▶ **doWithWebDescriptor** - Modify the XML generated for web.xml at runtime!
  - ▶ **doWithDynamicMethods** - Add methods!

# A Basic Plug-in

The Plug-in Version

Plug-in Dependencies

```
class LoggingGrailsPlugin {  
  
    def version = 0.4  
    def dependsOn = [core:"0.4.2 > 1.0"]  
  
    def doWithDynamicMethods = {  
        application.allClasses.each { c ->  
            c.metaClass.getLog = {->  
                LogFactory.getLog(c)  
            }  
        }  
    }  
}
```

# DEMO

Creating a Plug-in

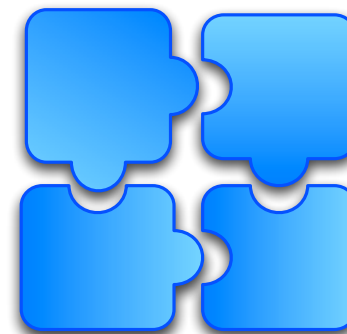
# Packaging & Installation

- Installation of Grails plug-ins can then be achieved with a few simple commands:

```
grails package-plugin  
  
cd ../my-project  
  
grails install-plugin  
    ../logging/grails-Logging-0.4.zip  
  
// or remotely  
grails install-plugin  
    http://myserver/logging/grails-Logging-0.4.zip
```

# Adding Basic Artefacts

- A Plug-in can add new tag libraries, controllers and services simply by creating them in the plug-in project
- Since a plug-in project is just like any project you can run and debug a plug-in in its own project before distributing it
- Once you're done package and distribute it!




# Configuring Spring

```
// Configuring Spring
class JcrGrailsPlugin {
  def version = 0.1
  def dependsOn = [core:0.4]

  def doWithSpring = {
    jcrRepository(RepositoryFactoryBean) {
      configuration =
        "classpath:repository.xml"
      homeDir = "/repo"
    }
  }
}
```

Bean name is method name, first argument is bean class



Set properties on the bean



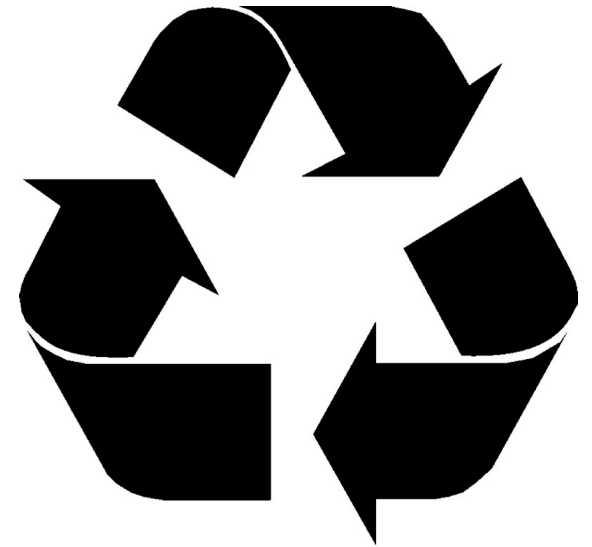
# Advantages of Spring DSL

- Since it is just Groovy code it can contain complex logic that is dependant on
  - ▶ The Environment
  - ▶ The Conventions in the project
  - ▶ Anything!
- Thanks to Groovy's neat syntax for lists and maps it is very easy to specify complex bean definitions



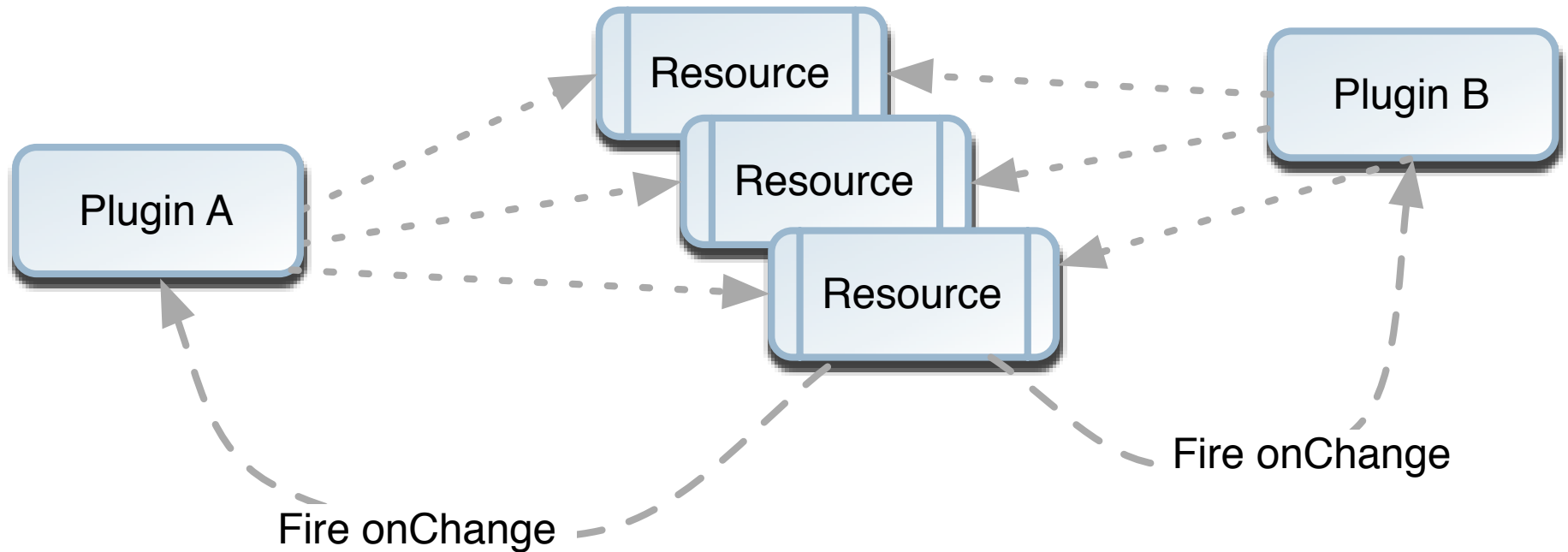
# Reload Events

- Grails applications must be reloadable during development
- Plug-ins can be defined **watchedResources** that fire **onChange** events when modified



# Watching Resources

Watched Resources



# Example Reloading Plug-in

```
class I18nGrailsPlugin {
  def version = "0.4.2"
  def watchedResources =
    "file:../grails-app/i18n/*.properties"

  def onChange = { event ->
    def messageSource =
      event.ctx.getBean("messageSource")

    messageSource?.clearCache()
  }
}
```

Defines a set of files to watch using Spring resource pattern

When one changes, event is fired and plug-in responds by clearing message cache

# The Event Object

- **event.source** - The source of the change which is either a Spring Resource or a `java.lang.Class` if the class was reloaded
- **event.ctx** - The Spring `ApplicationContext`
- **event.application** - A `GrailsApplication`
- **event.manager** - A

# Finding Plug-ins

- Grails has a central plug-in repo where you can access plugins:

```
                                // lists all plugins
grails list-plugins

                                // install latest version of searchable
grails install-plugin searchable

                                // install version 0.3 of searchable
grails install-plugin searchable 0.3
```

# Becoming a Plugin Developer

- Talk about your idea here:
  - ▶ <http://grails.org/Mailing+lists>
- Get access to the plug-in repo here:
  - ▶ <http://xircles.codehaus.org/projects/grails-plugins/members>
- Build your plug-in and type:
  - ▶ grails release-plugin
- You're done!

# Summary

- The Grails plug-in system is flexible and powerful
- Plug-ins can do just about anything from adding new methods to configuring Spring beans
- Grails has a growing plug-in community



Q & A