Grails Platform Core - Reference Documentation

Authors: Marc Palmer (marc@grailsrocks.com), Stéphane Maldini (smaldini@vmware.com)

Version: 1.0.0

Table of Contents

1 Overview

The Plugin Platform provides APIs and utilities that provide the glue required for advanced Grails plug and integration across multiple Grails versions, and turbo-charge the development of a new generation of p

The founding principle is that these platform features should not be part of Grails core because this would use the APIs to specific Grails versions.

This relative freedom from Grails versions means that plugins that use the platform should remain compa Grails versions for longer, and that new features used by all plugins can be added outside of the Grails rele

1.1 The APIs

The features include in this release include:

- Configuration API - Plugin-namespaced config, Config merging, validation and default values
- Security API - An abstraction of basic security features that most applications require, with implen provided by plugins or your application
- UI Extensions - A set of tags and helper properties and functions
- Navigation API - A standard artefact and DSL for navigation and tags for accessing this
- Events API - A standard event bus that can be plugged in to any event implementation, with a ligh provider for in-app messaging

Each feature is covered in more detail later in this documentation.

The platform is very self-referential - it uses own APIs - for example to provide event hooks for Gr lifecycle, to declare dynamic methods on your controller and service artefacts, and to declare the configura it uses.

All of these APIs are designed to be as simple as possible.

Use this platform to add tighter and more consistent integration to your own plugins.

1.2 Change log

1.0.RC1

- Injected methods will not overwrite existing methods now, will warn instead
- Added userExists method to security API
- Unified i18N text and body handling in UI Extension tags
- Improved dev-mode UI at /platform
- Bug fixes and improvements to Events API

1.0.M3

Was: 1.0.M2 but due to release error we had to roll to M3.

- Events API
- New UI Extension tags p:text and p:textScope
- New UI Extension properties pluginFlash, pluginSession, pluginRequestAttributes
- Documentation for Events API, Navigation API, new UI Extensions
- A little Grails treat for those browsing /platform/ on Mac

1.0.M2-SNAPSHOT

- Refined and documented Navigation API
- There are no longer any "g" namespaced tags. All `g:` tags have move to `p:` namespace
- Added "site.url" Config setting for siteLink tag to use instead of grails.serverURL if the two differ for
- Refactored Injection, Conventions and Navigation implementations into public interface + implement
- Config reloading supported now - all plugin configs and constraints etc. are re-applied
- legacyPrefix support in doWithConfigOptions - automatically copies over old Config values to y namespaced config
- Added cssPrefix attribute to displayMessage

1.0.M1

First public release with Config and Security APIs and some UI Extensions. Work-in-progress APIs for E Conventions and Navigation.

1.3 Known Issues

1.0.M3

- Automatic convention controller navigation includes all actions, not just those with GET allowedMetl
- Platform "dev" navigation scope items do not render when browsing /platform in your own applicatio

1.0.M1

- Config API - false validation errors with platform-ui due to no x.y.'*' support yet
- Navigation API - controllers do not auto-register in Grails 1.3.x, no DSL artefact, no reloading
- Conventions API - API not for public use. Not fully implemented / TBD
- Events - API not for public use. Scopes not fully implemented / TBD
- Injection - injection may not re-apply dynamic methods and properties to reloaded or new artefacts public use at all yet

2 Getting Started

To get started you need to install the platform-core plugin.

Add the plugin platform as a dependency to your application or plugin, either directly or transitively. T edit your `BuildConfig.groovy` to look something like this:

{docx} grails.project.dependency.resolution = { // inside your plugin dependencies block plug ':platform-core:1.0.M1' } } {docx}

You can run your code now and browse to `http://localhost:8080/<appname>/platform/` available only in the development environment. From there you can explore some of the plugin platform debug interface.

There is no default security implementation for the Security API, this will need to come from the security or your application. See [implementing a security bridge]

3 Configuration API

The Configuration API adds the following features:

- A way to declare the Config properties that a plugin supports
- Automatic namespacing of plugin Config values to avoid clashes
- Validation of Config values
- Merging of config from plugins into main Application config
- The ability for plugins to configure other plugins
- An injected "pluginConfig" variable in all artefacts containing the plugin's configuration
- Automatic merging of legacy Config settings into the new plugin namespace

All of this adds up to more powerful integration and less frustration and confusion for developers.

3.1 Changing Application and Plugin Config Values

To change application or plugin configuration from within a plugin you need to declare the **doWithConf** plugin descriptor.

The code uses a simple DSL that is identical to normal Config except:

1. The top level nodes are plugin names or "application" to determine what scope of config you are chan
2. The closure is passed the existing config as its first and only argument

The application Config is loaded first. All the doWithConfig blocks are evaluated and the results merged i

{docx} def doWithConfig = { config -> platformUi { ui.Bootstrap.button.cssClass = 'btn' ui.Bootstra 'tab-pane' ui.Bootstrap.field.cssClass = 'input' }

application { // set something based on another config value that has already been // by the applicat config.p.q == 'something' ? true : false } } {docx}

See doWithConfig for more details.

3.2 Declaring Configuration Options

To make use of the plugin configuration features and make life easier for developers, your plugin configuration options it accepts.

This allows the platform to warn users when they mistype a config name or supply and invalid value - a definition of default values rather than a plugin merging in default values.

Do declare the options your plugin supports, add the `doWithConfigOptions` closure to your plugin de

{docx} def doWithConfigOptions = { 'organization.name'(type: String, defaultValue: 'M plugin.platformCore.organization.name)') 'site.name'(type: String, defaultValue: 'Our plugin.platformCore.site.name)') } {docx}

This block, from the platform core, defines two configuration values of type String, with a default value.

You can also supply a custom validator:

{docx} def doWithConfigOptions = { 'concurrentConnections'(type: Integer, defaultValue: 10, validator: null : 'concurrent.connections.too.big' } } {docx}

Behaving just like constraint validators, your validator returns null for "ok" or an i18n message string for tl

When implementing this config in your plugins, you may want to use the `legacyPrefix` value so that a use your existing Config settings will continue to work, and users will be warned to update their Config.

See doWithConfigOptions for more details.

3.3 Accessing Plugin Config

Plugins that declare their configuration with doWithConfigOptions can get access to their "slice" of the C pluginConfig variable.

The `pluginConfig` variable is automatically injected into all artefacts of your plugin, automatically your plugin using the `plugin.<pluginName>.` prefix.

So in a service you can trivially access this config inside a service or controller for example:

{docx} class MyPluginService { def doSomething() { if (pluginConfig.enabled) { println "It worked!" } }

4 Security API

The Security API provides common security related functionality so that plugins and in some cases applic need to be tied to a specific security implementation.

It is deliberately minimal to place few requirements on implementations.

It is not intended to be a complete security abstraction, but "just enough" for the needs of most applicati that do not require advanced security manipulation - which will likely always require direct knowledge o provider.

This API provides the most basic security features to enable this interoperability, using a bridging interfa plugins must implement to actually provide these services.

Security plugins must implement the "provider" bean - out of the box there is no security implementation.

Dynamic methods

These methods and properties are added to Services, Controllers, TagLibs and Domain Classes:

- securityIdentity - The string used to identify the current logged in user. Typically a user id, user address. The nature of this value is dependent on your security implementation
- securityInfo - The object representing the current logged in user's information.
- userHasAnyRole - test if the current user has any of the roles
- userHasAllRoles - test if the current user has all of the roles
- userIsAllowed - test if the current user has a specified permission on an object
- withUser - run a section of code as another user

See the reference guide for Security Properties and Security Methods for further details on these.

All of these features and more can be accessed by any code in your application by using the grailsSecurity

Tags

Here's an example of some of the security tags available:

{docx} <s:identity/> <s:info property="email"/> <s:ifPermitted role="ROLE_ADMIN"> … < <s:ifNotPermitted role="ROLE_ADMIN"> … </s:ifNotPermitted> href="${s.createLogoutLink()}">Log out Sign up {docx}

See the **Tags Security** reference section for details.

grailsSecurity bean

The Security bean provides access to all the basic security functions. These are passed through to the implementation.

This includes methods for applications and plugins to use such as:

- String getUserIdentity()
- def getUserInfo()
- boolean userHasRole(role)
- boolean userIsAllowed(object, action)
- def ifUserHasRole(role, Closure code)

You simply auto wire this bean into your classes using the name "grailsSecurity"

For more details see grailsSecurity.

4.1 Implementing a Security Bridge

To use the Security API, an application must have a Security Bridge bean that implements the SecurityBri

Typically this bean will be provided by the security plugin you are using. However you can easily implem your own plugin or application.

Simply implement the interface and register the bean as "grailsSecurityBridge" in your Spring context eit `doWithSpring` or your application's `resources.groovy`:

{docx} grailsSecurityBridge(com.mycorp.security.MySecurityProvider) { // wire in any other depe grailsApplication = ref('grailsApplication') } {docx}

The interface is defined here:

{docx} interface SecurityBridge {

/

- Implementations must return the name of their security provider
- @return A name such as "Spring Security"

*/ String getProviderName()

/

- Get user id string i.e. "marcpalmer" of the currently logged in user, from whatever
- underlying security API is in force
- @return the user name / identity String or null if nobody is logged in

*/ String getUserIdentity()

/

- Get user info object containing i.e. email address, other stuff defined by the security implementation
- @return the implementation's user object or null if nobody is logged in

*/ def getUserInfo()

/

- Return true if the current logged in user has the specified role

*/ boolean userHasRole(role)

/

- Can the current user access this object to perform the named action?
- @param object The object, typically domain but we don't care what
- @param action Some application-defined action string i.e. "view" or "edit"

*/ boolean userIsAllowed(object, action)

/

- Create a link to the specified security action
- @param action One of "login", "logout", "signup"
- @return Must return a Map of arguments to pass to g:link to create the link

*/ Map createLink(String action)

/

- Execute code masquerading as the specified user, for the duration of the Closure block
- @return Whatever the closure returns

*/ def withUser(identity, Closure code) } {docx}

5 Events Bus API

Why an events bus ? Today's applications rely more and more on non-blocking processing, elasticity and events bus loosely couples modules, enabling different codes and frameworks to work together, because **th the right purpose** paradigm is becoming a reality. The bus may also support **publish/subscribe** pattern, same message across the handling modules and giving an excellent opportunity to deploy the same ap cluster or *cloud* .

Within the bus, an event is often as simple as a "callback" with no parameters, but usually there is some ex as an event object. An event can be sent to multiple listeners, and any result returned from any listeners is the original sender of the event. An event belongs to a "topic" and often has a "subject". The topic is lik identifies the kind of events. The optional subject is the object that the event "happened to". So for examp started" notification has no subject but may have topic "grails", but a "user logged in" even may have topi subject set to the user principal supplied by the security plugin you are using.

With Platform-Core plugin we have implemented a couple of features and artifacts to let you simply mana and get maximum flexibility when required :

- Sending **Events methods** injected in your **Domains**, **Controllers** and **Services**
- **@Listener** annotations for your **Services** methods.
- Events mapping **DSL artifact** to select and control events topics
- **Spring beans** with access to underlying API
- **More cool stuff** with Events Spring Integration and Events Push
- Simple **config keys**

In a nutshell, you will :

- Send events :

{docx} class UserController{

def registration(){ def user = new User(params).save() if(user){

//non-blocking call, will trigger application listeners for this topic event('mailRegistration', user)

//blocking call : //event('mailRegistration', user).waitFor()

//can also be written like that //event topic:'mailRegistration', data:user

//and if you need to reuse the current thread //event topic:'mailRegistration', data:user, fork:false

render(view:'sendingRegistrationMail') }else{ render(view:'errorRegistration') } } } {docx}

- Write listeners (or event handler, or event reactor or whatever you call it):

{docx} class UserService{

//use method name 'mailRegistration' as topic name //can also use custom topic name us @Listener(topic='test') @grails.events.Listener def mailRegistration(User user){ sendMail{ to us "Confirmation" html g.render(template:"userMailConfirmation") } }

//Can also receive an EventMessage to get more information on this parti
@grails.events.Listener(topic="mailRegistration") def mailRegistration2(org.grails.plugin.platform.event
msg){ sendMail{ to msg.data.mail subject "Confirmation" html g.render(template:"userMailConfirmation"

5.1 Sending Events

Sending an event is simple. You only need to remind 1 method name and 2 different signatures:

- **event**(topic, [data, params, callbackClosure])
- **event**(Map args, [callbackClosure])

We recommend using the former signature if you don't have any params, otherwise the latter is more elega

Let's see what the key arguments are doing:

- Topic argument is a **String** which represents channel subscribed by listeners.
- *optional* Data argument is an **Object** - *preferrably Serializable for IO facilities* - which represents the event such as a domain class.
- *optional* Params argument is a **Map** which represents sending behaviors including **namespace**.
- *optional* callbackClosure is a **Closure** triggered after an event completion.
- The map notation allows you to reuse the same arguments than params plus **topic** for topic, **data** f (shortcut for 'namespace'). If you specify **params**, it will use it for the **params** argument otherwis map is used as **params**.

There are several **params** arguments :

Key	Type	Default	Description
fork	`Boolean`	`false`	Force the event to reuse the caller thread, there the method synchronously and propagating any
namespace / for	`String`	`'app'`	Target a dedicated topic namespace. To avo topic names, the events bus supports a scoping namespace. E.g. 'gorm' is used by gorm event is used for Javascript listeners in **events-push**
onReply	`Closure{EventReply reply}`		Same behavior than *callbackClosure* argument both are defined.
onError	`Closure{List<Exception> errors}`		If exceptions has been raised by listeners, thi be triggered. If undefined, exceptions will be EventReply.getValue(s).
gormSession	`Boolean`	`true`	Opens a GORM session for the new thread event execution.
timeout	`Long`		Define a maximum time in millisecond execution.
headers	`Map<String, Serializable>`		Additional headers for the event message enve

The event method returns **EventReply** which implements Future<Object> and provides usefuls methods :

- **List<Object> getValues()** : Returns as many values as listeners has replied.
- **Object getValue()** : First element of getValues().
- **int size()** : Invoked listeners count.
- **List<Throwable> getErrors()** : Available errors.
- **boolean hasErrors()** : Scans for any errors.
- **EventReply waitFor()** : blocks current thread and return this reply.
- **EventReply waitFor(long time)** : blocks current thread for T milliseconds and returns this reply.

Events workflow

Events can be sent from domains, services and controllers artefacts by using *EventReply event(String topi* Platform-core Events bus provides a non-blocking way to send events by default, however you can **bl methods** from **EventReply** :

- **size**
- **waitFor**
- **get**
- **getValues**
- **getValue**

Therefore you have the control on the execution flow if you want. Just keep in mind it does not block for after event() call, which seems to be a sensible default for the bus. Eventual **Exceptions** will be raised af the mentioned blocking methods except if **onError** parameter is used.

{docx} class SomeController{

def logout(){ def reply = event("logout", session.user) //doesn't wait for event execution

render reply.value //wait and display value

event(topic:"afterLogout").waitFor()

//Only triggered when "afterLogout" finished def errorHandler = {errs -> } //Use a dedicated event(topic:"afterAfterLogout", onError:errs) } } {docx}

Non forked events

If you want to reuse the current thread and force synchronous processing, use the fork param. Be a exception will be directly propagated to caller even without using blocking methods except if **onError** par

{docx} class SomeController{

def logout(){ def reply = event('logout', session.user, [fork:false]) //block for processing

//no need to wait for reply since it has been populated on event call. render reply.value } } {docx}

Assigning a namespace

All listeners get a property called namespace which prevents topic naming collisions and undesired eve they are all assigned to **app**. This is the same default used when you send an event, but what if you want namespaced listeners, like 'browser' ones if you use **events-push** plugin ? Simply use **namespace** argum you stick with Map notation.

{docx} class SomeController{

def logout(){ //we use the Map form, the namespace argument is identified by the 'for' key ever topic:'logout', data:session.user } } {docx}

> It's mandatory to declare namespace when using events bus from a plugin in order to avoid conflicts.

Wildcard support

It's possible to call multiple topics/namespaces in a single shot using **wildcard as the last character**.

{docx} class SomeController{

def logout(){ /* We send to every listeners starting with "chat-" on every namespaces starting with ' for:'role-**', topic:'chat-**', data:session.user

//Here we can trigger every listeners in the default namespace 'app' event '*' } } {docx}

> This feature will probably evolve to a smarter implementation behaving like UrlMappings authorizing substring captures

5.2 Listening Events

Listening for events simply requires registering the method that should receive the event notifications.

There are few ways to register events.

Defining listeners at compile time

Within Grails services you can use the **@Listener** annotation. It takes a **topic** string, but you can omit **method name** as the topic to listen for:

{docx} class SomeService{

@grails.events.Listener(topic = 'userLogged') def myMethod(User user){ }

//use 'mailSent' as topic name @grails.events.Listener def mailSent(){ } } {docx}

Event methods can define a **single argument**, and the value is the object sent with the event. Usually this of the event. However an event is carried by an enveloppe called EventMessage which contains several like additionnal headers, current topic :

{docx} class SomeService{

@grails.events.Listener(topic = 'userLogged') def myMethod(org.grails.plugin.platform.events userMessage){ println userMessage.headers // display opt headers println userMessage.event // display println userMessage.data // displays data } } {docx}

If a listener argument type is not assignable to an event data type, the event **silently skips the mismatcl** you want to catch every event types, use Object type or if the argument is not necessary, do not declare it.

> Filtering on the EventMessage<D> generic type doesn't work, e.g. EventMessage<Book> w prevent EventMessage<Author> invokation. For such fine grained control, you can rely Events Artifact

Namespacing

Your declared events belongs to the **app** namespace, unless you tune it using the **namespace** argument or we will introduce later.

{docx} class SomeService{

@grails.events.Listener(topic = 'userLogged', namespace = 'security') def myMethod(User user){ }

//will subscribe this method to topic 'afterInsert' on namespace 'gorm' @grails.events.Listener(namespac afterInsert(User user){ } } {docx}

Remember that you will need to specify the scope when triggering events if you customize it with a diffe **app** :

{docx} class SomeController{ def myAction(){ event for:'security', topic:'userLogged', data:session.user }

> It's mandatory to declare namespace when using events bus from a plugin in order to avoid conflicts.

Proxy (AOP) support

By default, listeners try to call the original method (unproxified bean). Using **proxySupport** you can twea

{docx} class SomeService{

static transactional = true

//Will invoke transactional logic, similar to someSerice.myMethod() @grails.events.Listener(proxySu myMethod(User user){ }

} {docx}

Dynamic listeners

Some edge cases need runtime registration. If you meet this use case, use the injected on method :

{docx} class SomeController{

def testInlineListener = { //register with 'logout' topic on 'app' default namespace def listener = on("logout println "test $user" } render "$listener registered" }

def testInlineListener2 = { //register a 'gorm' namespaced handler on 'afterInsert' topic. def listene "afterInsert") {Book book -> println "test $book" } render "$listener registered" } } {docx}

Wildcard support

Capturing a wider group of events can be useful, specially for monitoring purposes. It's possible to lis topics/namespaces in a single shot using **wildcard as the last character**.

{docx} class SomeService{

@grails.events.Listener(namespace='role-', **topic = 'chat-'**) myMethod(org.grails.plugin.platform.events.EventMessage userMessage){ println userMessage.nam userMessage.event } } {docx}

> ⚠ This feature will probably evolve to a smarter implementation behaving like UrlMappings authorizing substring captures

Listener ID

Registered listeners generate a unique id (**ListenerId**) applying the following `[namespace://]topic[:package.Class][#method][@hashcode]`

The above square brackets determine each optional part of the sequence id thus allowing to target gr depending of the known arguments: namespace, class, method, hashcode.

This pattern is useful when using **countListeners**, **removeListeners** or **extensions**. For instance, overrid **channel** with **events-si** plugin requires to use `namespace://topic` if *namespace* is different from example to count listeners: {docx} //count every listeners subscribed to 'mytopic' insid countListeners("mytopic:my.TestService")

//count every listeners using gorm namespace countListeners("gorm://*")

//remove every listeners in TestService removeListeners("*:my.TestService") {docx}

Reloading in Development mode

It works.

5.3 Replying from Listeners

Usually, an event is *fired and forgot* . In some cases, you may expect an answer to transform your messag into a controlled flow. For instance, a negative reply can be used in GORM events to veto database writin subjet. Another usual example is the aggregation of multiple workers products.

Simple reply

Replying is a simple matter of returning an object from the listener method :

{docx} class SomeService{ @Listener def logout(User user){ Date disconnectDate = new Date()

//do something with user

return disconnectDate } } {docx}

If listeners return non null objects, the caller can access them through the EventReply enveloppe returned i calling **event** method. The other option is the use of a **reply handler** :

{docx} class SomeController{

def logout(){ def reply = event topic:"logout", data:session.user, fork:false render reply.value //display valu

//Using callback closure def replyHandler = {EventReply reply-> } event topic:"logout", da onReply:replyHandler

//Or as last argument event(topic:"logout", data:session.user){ EventReply r-> }

//EventReply object is a Future implementation def reply_future = event topic:"logout", data:sess reply_future.get(30, TimeUnit.SECONDS) } } {docx}

> Whenever an event is triggered, a task is submitted into the events bus thread pool and a Fu returned, wrapped into EventReply. It's also planned to fully support reply-address pattern future version (replyTo parameter) which brings interesting features out of the box : non bloc response, streaming handler response one by one, forwarding using topic name instea closure…

Multiple replies

Multiple listeners can return values for the same topic/namespace. In this case, EventReply will wait before returning any value. Remember that a valid result is a non null value, hence why even if 3 handle but only 2 did return something, then you will only see 2 values in the **EventReply.values**.

{docx} class SomeController{

def logout(){ def reply = event topic:"sendMails", data:session.user

//wait for all listeners and then display the first value from the aggregated results render reply.value

//display all results as List render reply.values } } {docx}

Exceptions

Because no code is perfect, exceptions can happen in the event process for 3 reasons :

- RuntimeException in one or more handlers
- InterruptedException if the process has been cancelled
- TimeoutException if the maximum process time has been reached (timeout parameter)

An **onError** parameter is available and accepts a **Closure{List<Throwable> errors}**. If non set, propagated to the caller when blocking the EventReply object (*getValue* etc) and/or when **fork == false**.

Exceptions in multiple listeners scenario don't interrupt the execution flow and leave a chance to oth execute as well. The return value from a failing listener becomes the raised exception.

{docx} class SomeController{

def logout(){ on('test'){ sleep(5000) throw new MyException('haha') }

def reply = event topic:"test" reply.values //throws MyException after 5s

def errorsHandler = {println it} reply = event topic:"test", onError:errorsHandler reply.values //calls er returns values which contain at MyException

event(topic:"test", onError:errorsHandler, timeout:1000){ //executes both this and errorsHandler c TimeoutException }

reply = event(topic:"test", onError:errorsHandler, timeout:1000) reply.cancel() //executes errorsHandler InterruptedException

event(topic:"test", fork:false) //wait 5s and raises an exception in the caller thread

} } {docx}

Waiting replies

In domains, services and controllers artefacts you can wait for events using "EventReply waitFo eventReplies)". It accepts as many events replies you want and returns the same array for functiunal pro EventReply also have a waitFor method for one-line waiting.

{docx} class SomeController{

def logout(){ def reply = event('logout', session.user) def reply2 = event('logout', session.user) def reply3 session.user)

waitFor(reply,reply2,reply3).each{EventReply reply-> render reply.value +'</br>' }

//same with 20 seconds timeout on each reply waitFor(20, TimeUnit.SECONDS, reply,reply2,reply3).e reply-> render reply.value }

//other style : event('logout', session.user).waitFor() //blocks event event('logout', session.user).waitFo event for maximum 2 seconds

} } {docx}

5.4 Routing configuration -- The XxxEvents Artifact

An extensible Events DSL is available in **grails-app/conf** for routing configuration. This artifact does det **event** method by selecting topics and namespaces to apply :

- Filtering
- Disabling
- Sending behaviors
- *Extensions*
- *Security*
- *Declarations*

> ⚠ The DSL is intended to evolve. One of the most wanted features is topic/namespace declarat assigning a definition to a property would generate an injectable eponym bean with strean methods.

The DSL requires to assign a closure to an **events** variable. Each call is a **definition**, the method name is p name and key/value arguments are definitions attributes. Wildcard topics/namespaces are supported as wel

An **Events** artifact is a script with some bound variables:

Variable	Description
grailsApplication	Grails application object, retrieves artifacts, context etc.
ctx	Spring context, useful for beans access, e.g. *ctx.myService.method()*
config	Configuration object

Each **definition** supports the following attributes:

Attribute name	Type	Default	Description
namespace	String	*"app"*	Define which namespace the current definition is bou
filter	Closure(Object) Closure(EventMessage) Class		If a closure is passed, the return value matched as th e v e n t p r c If a class is passed, the subject data type must match.
disabled	boolean	false	Disable event propagation
fork	boolean	false	Use the current thread for event processing (blocking
onError	Closure(List<Throwable>)		Default onError handler for the current topic(s)
onReply	Closure(EventReply)		Default onReply handler for the current topic(s)
timeout	Long		Default timeout for execution time, throwing a Tin and calls handlers
*	*		Any attributes can be written to be used by pl EventDefinition.othersAttributes

{docx} events = { //prevents any events in gorm namespace '*' namespace:'gorm', disabled:true

//filters any events on 'testTopic' where data <= 2 testTopic filter:{it > 2}

//filters any events on 'testTopic2' where data is not a TestTopic class type testTopic2 filter:TestTopic

//filters any events on 'testTopicX' using boolean method from service testTopicX filter:ctx.myService.&sc

//only if using events-push plugin, allows client-side listener on this topic testTopic3 browser:true

//Default Error Handling, Global Reply Handling, timeout and fork testTopicD onError:{}, onReply:{} testTopicD2 fork:false

//

---

roadmap

//not yet implemented: Assigning and merging definitions //myTopic = testTopic4(filter:{i>2} testTopic4(filter:{i<4})

//not yet implemented: Enabling security context for target listeners //testTopic5 secured:true

//not yet implemented: Topic Forwarding //testTopic6 to:'anotherTopic'

//not yet implemented: Topic Handlers //testTopic9 onError:'anotherTopicErrors', onReply:'anotherTopicR

} {docx}

Reloading in Development mode

It works.

5.5 Listening GORM events

Starting **from Grails 2**, the Events Bus supports GORM events.

GORM Listeners

To listen for GORM, simply declare listeners on the **gorm** namespace using the following supported topics

Event Type	Target Topic
PreInsertEvent	beforeInsert
PreUpdateEvent	beforeUpdate
PreDeleteEvent	beforeDelete
ValidationEvent	beforeValidate
PostInsertEvent	afterInsert
PostUpdateEvent	afterUpdate
PostDeleteEvent	afterDelete
SaveOrUpdateEvent	onSaveOrUpdate

Same listeners behaviors apply, e.g. using EventMessage for the argument type, using wildcard topi listeners are called if there are **no arguments** or the argument **type is assignable to current event data ty** domain class is the only required step to filter domains events.

{docx} class SomeService{

@Listener(namespace = 'gorm') void afterInsert(Author author) { println "after save author - $author.name

@Listener(topic = 'beforeInsert', namespace = 'gorm') void beforeInsertBook(Book book) { println "wi $book.title" }

//Will catch everything since we don't filter on the subject by using EventMessage @Listener(topic = 'befo = 'gorm') void beforeEachGormEvent(EventMessage message) { println "gorm event $message.ev $message.data.class" }

} {docx}

Filtering with Events Artifact

Setting a filter through an Events artifact allows more fined control and efficient selection since it preve propagated :

{docx} events = { 'afterInsert' namespace:'gorm', filter:Book 'afterDelete' namespace:'gorm', filter:{it.id > namespace:'gorm', filter:{it in Book || it in Author} 'beforeDelete' namespace:'gorm', disabled:true } {docx

> ⚠ GORM may generate tons of events. Consider using it wisely, combine it with routing filter You can also totally disable gorm bridge by using `events.gorm.disabled` configura key.

Threading behaviors

GORM Listeners are executed in the same thread than the caller in order to reuse the current opened Avoid blocking logic if possible or use the listener body to call another event.

Vetoing changes

If a listener handles one of the *before** topics and returns a boolean value, it becomes part of the vetoing cl

- Returning **false** will cancel the current database write
- Returning **true** will just let the chain continuing

{docx} class SomeService{

//veto any Book insert @Listener(topic = 'beforeInsert', namespace = 'gorm') boolean beforeInsertBool false } } {docx}

5.6 Spring Beans

Plugin developpers and any crazy tweakers may need to override one or more Events Bus beans, l **Integration plugin** does. The **grailsEvents** bean is also useful to inject events methods into unhandled than **domain**, **service**, **controller**).

Bean Name	Type	Default Implementation	Des
grailsEvents	*org.grails.plugin.platform.events .Events*	*org.grails.plugin.platform.events .EventsImpl*	Mai gate cont met artif
grailsEventsPublisher	*org.grails.plugin.platform.events .publisher.EventsPublisher*	*org.grails.plugin.platform.events .publisher.DefaultsEventsPublisher*	Publ trigg be i exte requ **even**
grailsEventsRegistry	*org.grails.plugin.platform.events .registry.EventsRegistry*	*org.grails.plugin.platform.events .registry.DefaultsEventsRegistry*	Reg store rout impl exte requ **even**
gormTopicSupport	*org.grails.plugin.platform.events .dispatcher.GormTopicSupport*	*org.grails.plugin.platform.events .dispatcher.GormTopicSupport2X*	Tran even nam proc
grailsEventsGormBridge	*org.grails.plugin.platform.events .publisher.GormBridgePublisher*		Liste eve publ righ gorn
grailsTopicExecutor	*org.springframework .core.task.TaskExecutor*	*org.springframework.scheduling .concurrent.ThreadPoolTaskExecutor*	Carr exec

5.7 Securing events

> ⚠ To be implemented. You can still use headers or data to pass security context for instance. release will bring platform-security abstraction ready for events.

5.8 Extensions

Writing extensions is one of the greatest habbits of **grails** developpers. Groovy and Grails communit pragmatic and pleasant.

Having seen the referenced beans in the previous chapter should already give you ideas to improve or needs. There are two available examples of extensions:

- events-si : This plugin overrides the publisher and registry beans in order to replace the default mech much

more flexible Spring Integration channels and endpoints.

- events-push : This plugin registers new Listeners from your cool browsers using javascript, authorize

new Events DSL attributes *browser* and *browserFilter*

5.9 Configuration properties

Based on Platform-Core configuration mechanism, the plugin provides the following Events-Bus related k

Configuration Key	Type	Default	Description
grails.plugin.platform.events.disabled	Boolean	false	Fully disable Events Bus mechan methods will be injected
grails.plugin.platform.events.poolSize	Integer	10	Allow X concurrent workers to pro
grails.plugin.platform.events.gorm.disabled	Boolean	false	Disable GORM bridge, stopping from being published
grails.plugin.platform.events.catchFlushException	Boolean	true	Catch any GORM flushing exc could be noisy specially when vetoi

In addition, you can override beans values such as gormTopicSupport {docx} beans{ gormTopicSuppo gorm Events Objects types into topics translateTable = 'PreInsertEvent': 'beforeInsert', 'PreUpdateEvent': ' **'PreLoadEvent': 'beforeLoad',**/ 'PreDeleteEvent': 'beforeDelete', 'ValidationEvent': 'beforeValidate', 'P 'afterInsert', 'PostUpdateEvent': 'afterUpdate', 'PostDeleteEvent': 'afterDelete', /**'PostLoadEvent':** 'SaveOrUpdateEvent': 'onSaveOrUpdate' } } {docx}

6 Navigation API

The Navigation API provides a standard way to expose information about the menus available in your plugins.

Aside from application navigation, plugins can expose their controllers and actions so that application ca their own navigation structure. Applications can also add items to the navigation structure of plugins, to n the UI of plugins.

6.1 Concepts

There are only three concepts to understand in the Navigation API - items, scopes and the activation path.

Out of the box, scopes are created for all your application and plugin controllers automatically by conve created in these scopes for every action on the controller.

You will typically move from this to using the navigation DSL artefact for more control over the navigatio

What is a navigation item?

An item is a place the user can reach in your navigation structure. Every item results in a menu item and is visible or enabled can be determined at runtime.

Items are always inside one scope.

Items can have child items.

Items must be resolvable from a controller/action pair, so the navigation API can always tell where th structure if the current controller/action is known and you have an item declared for them.

What is a Scope?

A scope is a name that identifies one or more navigation items. Top-level scopes are called root scopes an main groupings of navigation items. For example you may have your application navigation for regul admin root scope for backend administration.

Example of scope names:

```
app                     // typically your default app navigation root scope
app/messages            // the "messages" item in the "app" root scope
admin/scaffolding/book  // the "book" item under "scaffolding" item in the "admin
plugin.cms/admin        // the "admin" item supplied by the "CMS" plugin
plugin.socialFeed/feeds // the "feeds" item supplied by the "social-feed" plugin
```

Root scopes do not generate any menu links themselves, they are merely containers for your top level n They enable you to have multiple sets of navigation for different contexts.

The items that scopes refer to can be nested arbitrarily. It is however generally recommended that you use of navigation, sometimes three if really necessary. This is purely because of the user experience is navigation.

Usually you should factor out deep navigation into separate root scopes. For example most applications "app" scope, a "footer" scope for footer links like Terms of Use, Support etc., and a "user" scope for log in

What is an Activation Path?

An activation path is a string that represents the currently active navigation item. This may be a few leve navigation structure and represents the breadcrumb trail the user would see to get to the location the viewing.

> ⚠ Breadcrumbs themselves represent a navigational superset of your app's primary naviga structure. They are not supported in this release of the API, because the work has not yet b done to declare breadcrumbs that represent non-navigational items i.e. nested content insi multi-page document is not part of your regular site navigation.

The activation path is set on the current request and indicates which node is currently active. By default attempts to identify the correct activation path in your structure using the current controller and action, m URL mapping.

However you can explicitly set the activation path using a tag or some code, for cases where you need to example if your action performs some odd redirection, or the endpoint is simply a GSP view which cannc a location in the structure.

6.2 Getting Started

The first thing you need to do is install the platform-core plugin if you haven't already.

If you then run your application and you have some existing controllers you'll find that if you add the na one of your sitemesh layouts or GSP pages you will see top-level navigation for each of your controllers.

6.3 Navigation by convention

To get you started quickly, all your controllers will be automatically registered in the "app" scope and eac sub-items for each of it actions.

All the tags default to the "app" scope if you don't supply a scope and the current controller/action are w so it just works out of the box for the simple cases. So add the following to your sitemesh layout or GSP:

{docx:xml} <nav:primary/> <nav:secondary/> {docx}

This will render one or two <ul> tags for the "app" scope based on the currently active controller/action pa

By default all your controllers are automatically declared for you inside the "app" scope if they are not ex in a navigation DSL artefact and the navigationScope property is not set on them.

These controller scopes have a nested item for each action defined on the controller, including the default a as the link for e controller scope itself).

Moving some controllers from the default app navigation scope

You often have some controllers that you don't want to appear in the main navigation of the application. these to appear in an admin interface for example. To do this with convention based navigation you can navigationScope property to controllers.

{docx} class BookController { static scaffold = Book

static navigationScope = 'admin' } {docx}

This allows you to push controllers into another scope. Note that plugin controllers are automatically nai scope under "plugin.<pluginName>", in a scope beneath this with the value of the navigationScope propert

You will not need to change your tags to render the admin navigation - if the controller/action the user is v to an item inside the admin scope, the nav:primary tag will render the admin scope.

6.4 What is primary and secondary navigation?

The primary navigation is the top level application the user sees, and the secondary is the context-sensiti the currently active primary item.

Contemporary site styles typically separate out the primary and secondary navigation.

The primary and secondary tags are geared up for this and automatically lookup up the scope and activati out what to render.

Normally you will only use these once in a page.

6.5 Rendering other menus

You can render any part of your navigation structure as a menu as many times as you like anywhere in yo the menu tag.

Rendering multiple navigation scopes on the same page

A typical contemporary application will have something like three separate menus used on most pages; footer.

The main menu would use primary & secondary tags.

You would then render the user and footer navigation using the menu tag, and passing the user and footer s

{docx:xml} <html> <body> <nav:primary/> <nav:secondary/>

 <nav:menu scope="user"/>

<g:layoutBody/>

 <nav:menu scope="footer"/> </body> </html> {docx}

This results in a page where there are actually for navigation renderings, showing different scopes.

6.6 Using the Navigation DSL

To declare navigation items you use navigation DSL artefacts to determine the items in each scope. Scopes can be nested to provide a hierarchy.

Navigation artefacts are groovy scripts end in the name "Navigation" in `grails-app/conf`.

Here's an example for the various ways to use the DSL to declare scopes and items:

Example contents of `grails-app/conf/AppNavigation.groovy`: {docx} navigation = { // De scope, used by default in tags app {

// A nav item pointing to HomeController, using the default action home()

// Items pointing to ContentController, using the specific action about(controller:'content') contact(con help(controller:'content')

// Some user interface actions in second-level nav // All in BooksController books { // "list" action in "b list() // "create" action in "books" controller create() }

// More convoluted stuff split across controllers/locations support(controller:'content', acti faq(url:'http://faqs.mysite.com') // point to CMS makeRequest(controller:'supportRequest', action:'create') }

// Some back-end admin scaffolding stuff in a separate scope admin { // Use "list" action as default item default action // and create automatic sub-items for the other actions books(controller:'bookAdmin', act search')

// User admin, with default screen using "search" action users(controller:'userAdmin', action:'search') { // alias so "create" is active for both "create" and "update" actions create(action:'create', actionAliases:'update

Using tags such as the primary and secondary navigation tags you can render all the page elements you nee

The Navigation DSL Definition

The script must return a Closure in the `navigation` variable in the binding.

This closure represents the DSL and method invocations have a special meaning within the DSL.

The name used in method calls is used to construct the activation path of each item. So a call to "app" t "messages" which has a closure that calls "inbox" will create the following:

- A scope called "app"
- A top-level item in the "app" scope, called "messages", with activation path `"app/message"`
- A nested item under "messages" called "inbox" with activation path `"app/messages/inbox"`

Top level method invocations (root scopes)

The top-level method calls that pass a Closure define root scopes in the navigation structure.

The "app" scope is a prime example of this:

{docx} navigation = { app { home controller:'test', data:icon:'house' } } {docx}

By default scopes defined by Navigation artefacts within plugins are automatically namespaced to preven application namespaces.

Thus the scope "app" in a plugin called "SpringSecurityCore" would become the scope "plugin.springSe If a plugin defines the scope with the `global:true` argument, this will not happen:

{docx} // Example of a plugin exposing a root scope without namespacing navigation = { app(global controller:'test', data:icon:'mail' } } {docx}

Nested method calls - defining navigation items

The DSL supports the following arguments when defining a navigation items.

Linking arguments

These are `controller`, `action`, `uri`, `url` and `view`. These are passed to `g:link` to create lin
attribute is handled internally and removed and converted to "uri" for the purpose of calling g:link

These values are passed through to the navigation tags for link rendering just as you would expect when ca

There are some special behaviours however:

Argument	Usage
`controller`	Optional - it will be inherited from the parent node if the parent uses `controlle` link, or failing that it will use the name of the DSL method call
`action`	Optional - it will fall back to the name of the method call if the controller is specified the controller was not specified either (and hence "uses up" the method call name), t default action of the controller or "index" if none is set. The `action` value ca comma-delimited string. If it is, the first element is the action used to generate the i any other actions listed will have sub-items created for them, in alphabetical order.
`actionAliases`	Optional - list of actions that will also **activate** this navigation item. The link is `action` defined for the item in the DSL, but if the current controller/action resolves this alias list, the navigation item will appear to be active. Used for those situations v multiple actions presenting the same user view i.e. create/save, edit/update

Visibility and Status

You can control per request whether items are visible or enabled, or set this in the navigation structure stati

The arguments:

Argument	Usage
`visible`	Determines whether the item is visible and can be a boolean or a Closure. If it is a Closure, i delegate that supplies request and application properties (see below)
`enabled`	Determines if the item is enabled or not and can be a boolean or a Closure. If it is a Closure, i delegate that supplies request and application properties (see below)

Typically you will want to hide items if the user is not permitted to see them. An example of doing t
Security Core:

{docx} import org.codehaus.groovy.grails.plugins.springsecurity.SpringSecurityUtils

def loggedIn = { -> springSecurityService.principal instanceof String } def logged
!(springSecurityService.principal instanceof String) } def isAdmin =
SpringSecurityUtils.ifAllGranted('ROLE_ADMIN') }

navigation = { app { home controller:'test', data:icon:'house' … }

admin { superUserStuff controller:'admin', visible: isAdmin … }

user { login controller:'auth', action:'login', visible: notLoggedIn logout controller:'auth', action:'logout', v
signup controller:'auth', action:'signup', visible: notLoggedIn profile controller:'auth', action:'profile', visib
{docx}

Note how the Closures are "def"'d in the script to make them reusable and reachable within the DSL

The closures receive a delegate which resolves the following standard Grails properties:

- grailsApplication
- pageScope
- session
- request
- controllerName
- actionName
- flash
- params
- item (current NavigationItem instance being tested)

Any unresolved properties will resolve to the model (pageScope) and failing that, to the application's bean can resolve service beans etc by just accessing them by name.

Title text

The title of an item is the text used to display the navigation item.

Two arguments are used for this:

Argument	Usage
`title`	Optional. Represents an i18n message code to use. It defaults to "nav." plus the the item's with "/" converted to "." so path `app/messages/inbox` becomes the `nav.app.messages.inbox`
`titleText`	Optional. represents literal text to use for the navigation item title if the i18n bundle do anything for the value of `title`

For automatically created action navigation items, the titleText defaults to the "human friendly" form of t i.e. "index" becomes "Index", "showItems" becomes "Show Items".

Application custom data

Each item can have arbitrary data associated with it - but note that this data is singleton and should not cha

Typically you would use this to associate some extra data such as an icon name, which you then use i rendering code.

Just put the values into the "data" Map:

{docx} navigation = { app { home controller:'test', action:'home', data:icon:'house' } } {docx}

Ordering of items

Items are ordered naturally in the order they are declared in the DSL.

However you may wish to manually order items, for example so that plugins (or the application) can i certain positions in your navigation.

Just pass the integer value in the `order` argument:

{docx} navigation = { app { home controller:'test', action:'home', order:-1000 about controller:'test' order:100 contact controller:'test', action:'contact', order:500 data:icon:'mail' messages(controller:'test', d order:10) { inbox action:'inbox' archive action:'archive' trash action:'trash', order:99999999 // always last }

6.7 The Navigation Tags

There are a few Navigation tags available, all detailed in the reference section.

The most common tags you will use are explained here.

It is important to understand that all the tags work by default using the current scope and activation path a the request - but you can override scope and path on all of these tags to render anything you like.

Navigation is rendered by default as an HTML `<ul>` tag with an `<li>` containing a single link for ea Nested items are rendered as nested `<ul>`.

All navigation rendering tags support attributes for CSS class, id and custom rendering of items if requirec always rendered within `<ul>`.

nav:primary

Use this tag to render the primary user navigation of your site:

{docx:xml} <nav:primary scope="admin" id="nav" class="admin"/>

<%~~- With custom item renderi~~ng %> <nav:primary scope="admin" id="nav" class="admin" custo <p:callTag tag="g:link" attrs="${linkArgs + class:'nav button'}"> <nav:title item="${item}"/> </p </nav:primary> {docx}

This supports custom rendering in the same way as the menu tag.

See the primary tag reference for full details.

nav:secondary

Use this tag to render the second-level navigation based on the selected item within the current primary scope resolved by `nav:primary` is stored in the request so that this tag knows which scope to use:

{docx:xml} <nav:secondary id="secondary-nav" class="admin"/> {docx}

This supports custom rendering in the same way as the menu tag.

See the secondary tag reference for full details.

nav:menu

The menu tag is used internally by the primary/secondary tags and can be called directly to render navigation structure, with any activation path.

{docx:xml} <nav:menu id="main-nav"/>

<% ~~Render the admin nav 3 deep, including all nested descendents whether active or not~~ %> <nav:menu depth="3" forceChildren="true"/>

<% ~~With custom item rendering~~ %> <nav:menu scope="admin" id="nav" class="admin" custom="true"> tag="g:link" attrs="${linkArgs + class:'nav button'}"> <nav:title item="${item}"/> </p:callTag> </li {docx}

See the menu tag reference for full details.

nav:title

This renders the i18n title of a specific navigation item passed to it; for use in custom menu rendering.

{docx} <nav:primary scope="admin" id="nav" class="admin" custom="true"> <nav:title item="! </nav:primary> {docx}

See the menu tag reference for full details.

nav:set

You can call this tag from inside a controller or GSP if you need to define request-specific parameters.

You can "fudge" the current request's activation path or set the default scope to be used by navigation tags.

You may need to do this inside an error.gsp for example, or inside admin pages to reuse a generic the navigation using `nav:primary`.

{docx:xml} <html> <body> <!-- pretend we are in messages/inbox even though we are in a GSP with ~~<nav:set path="app/messages/inbox"/> <nav:set scope="admin"/>~~

~~<!~~ or set those together ~~> <nav:set path="app/messages/inbox" scope="admin"/>~~

~~<!~~ these will use whatever the current active path and scope are --> <nav:primary/> <nav:secondary/>

Something went wrong! </body> </html> {docx}

See the set tag reference for full details.

7 UI Extensions

Several simple UI Extensions are included in platform-core.

The tags supplied make it trivial to render links to controllers and actions using i18n messages, display
end user, and render buttons and labels in i18n friendly ways.

7.1 Tags

Linking tags

The smartLink tag renders links for controllers and actions, automatically working out the text of the link u

{docx:xml} <% ~~Link to default action of BooksController~~ %> <p:smartLink controller="books"/>

<% ~~Link to list action of current controller~~ %> <p:smartLink action="list"/> {docx}

These will use i18n messages located by convention of the form: `action.controllerName.action`

Label tag

The label tag will render a <label> with the text optionally loaded from i18n:

{docx:xml} <p:label text="field.user.name"/> {docx}

See label for full details of the attributes, which include passing arguments to the i18n message.

Button tag

The button tag will render a text-based button using either a <button>, <input type="submit"> or <a> ta
optionally loaded from i18n:

{docx:xml} <p:button text="button.save"/> {docx}

See button for full details of the attributes, which include setting the kind of button rendered and passing a
i18n message.

Display message tag

The displayMessage tag works with the `displayMessage` and `displayFlashMessage` controller n
it easy to render messages to the user in a uniform way.

{docx:xml} <p:displayMessage/> {docx}

See displayMessage for full details of the attributes and the displayMessage and displayFlashMessage cont

The tag will render both request and flash messages, and wraps them in a div with CSS classes accordin
message.

Branding tags

There are several simple but useful site branding tags included. Commonly to be used in site footers and er

- organization - Renders the name of the business, taken from Co `plugin.platformCore.organization.name`
- siteName - Renders the name of the site/product, taken from Config var `plugin.platformCore`
- siteURL - Renders an absolute URL for the root of the site
- siteLink - Renders an absolute link to the site, with the site name as the link text
- year - Renders the current year, for use in copyright footers

7.2 Properties

New auto-namespaced equivalents of `session`, `request` and `flash` attributes are added to all **control** exposed by plugins.

These properties are pluginRequestAttributes, pluginSession and pluginFlash.

They allow you to access these attributes from plugin code without having to worry about key name cla plugins or the application:

{docx} class MyController { def beginPasswordReset = { pluginSession.resetMode = true pluginFlas resetTokenFactory.newTicket() } } {docx}

7.3 Beans and utilities

There are a some UI utility classes and beans available:

- grailsUiExtensions - Provides methods for setting and getting displayMessages
- TagLibUtils - Provides helper functions for manipulating attributes, CSS class name lists etc.

8 Injection API

This API is partly implemented but for internal use only at this time.

9 Convention API

This API is partly implemented but for internal use only at this time.