Grails Audit Logging Plugin

Robert Oschwald

Version 3.0.0, 2018-07-27

Table of Contents

Chapter 1. Description

The Audit Logging plugin makes it easy to add customizable audit tracking by simply having your domain classes extend the `Auditable` trait.

It also provides a convenient way to add Update and Create user stamping by extending the `Stampable` trait.

1.1. Compatibility

Grails Version	Audit Version
3.3 and later	3.0.x
3.0 to 3.2	2.0.x
2.0 to 2.5	1.0.x
1.3.x	0.5.5.3
< 1.2.0	0.5.3

1.2. GORM Compatibility

This plugin is GORM agnostic, so you can use it with the GORM implementation of your choice (Hibernate 4 or 5, MongoDB, etc.).

Please note, that only Hibernate5 is tested during development. If an issue occurs with your ORM mapper, please file a GitHub issue.

Chapter 2. Change Log

- 3.0.0
  - Major rewrite of plugin to be trait based
  - Removed support for handler callbacks
  - Consolidated and cleaned up configuration
  - Added AuditLogContext to allow configuration overrides at the block level
- 2.0.6
  - Fix #142 Re-introduced truncateLength support (and changed config parameter from *TRUNCATE_LENGTH* to *truncateLength*)
  - Fixed verbose param default description in documentation (Thanks to Matthew Moss)
  - Fix #139 Allow whitelist of properties to be used instead of a ignore list.
- 2.0.5
  - Only pass session to actorClosure if a session actually exists. (Thanks to Dennie de Lange)
  - Updated syncHibernateState to use correct name array (Thanks to Matthias Born)
  - Fix ignore list not used for insert and delete (Backport from 1.x)
  - Fix #147 Document per-datasource auditLog.disabled config key
- 2.0.4
  - Added option to specify createdBy,lastUpdatedBy, dateCreated,lastUpdated fieldnames per domainclass
  - and removed blank constraint for nullable stampable properties.
  - Remove preDelete as stampable event, does not make sense to stamp a delete event. (Thanks to Dennie de Lange)
  - Constraint fixes
- 2.0.3
  - Fix #129 Issue with Hibernate stamping. Stamping was ignored with dynamicUpdate = true and stamping was ignored on cascading saves. (thanks to Dennie de Lange)
  - Fix #130 Docs for verbose mode
- 2.0.2
  - Fix #118, use Grails 3.0.10 internally.
  - Fix #126 Support Many-To-Many logging (thanks to Andrey Zhuchkov)
- 2.0.1
  - Fix #117 Clean build. Version 2.0.0 had issues with Spring Security due to unclean build.
  - Fix #116 (partially). Replacement Patterns do work, but trailing dots are ignored for now due

to Grails 3.0.x limitations.

- 2.0.0
  - First Grails 3 version. Thanks to Graeme Rocher.
  - Added audit-quickstart command to create the AuditLog domain artifact
  - #96 Make identifiers available in the maps during onChange event. Thanks to dmahapatro.
  - branch: master.
  - For 1.0.x plugin version (Grails2), see 1.x_maintenance branch
- 1.0.5
  - Migration of JIRA to GitHub Issues
  - Fix #92 (Support for ignoring certain Events)
  - Starting with this release, the main branch for the 1.0.x series is 1.x_maintenance. Master branch is for Grails 3.0 support, now. Both branches will be tested by Travis-CI.
- 1.0.4
  - GPAUDITLOGGING-69 allow to set uri per domain object
  - GPAUDITLOGGING-62 Add identifier in handler map
  - GPAUDITLOGGING-29 support configurable id mapping for AuditLogEvent
  - GPAUDITLOGGING-70 support configurable datasource name for AuditLogEvent
  - GPAUDITLOGGING-74 Impossible to log values of zero or false
  - GPAUDITLOGGING-75 Support automatic (audit) stamping support on entities
- 1.0.3
  - GPAUDITLOGGING-64 workaround for duplicate log entries written per configured dataSource
  - GPAUDITLOGGING-63 logFullClassName property
- 1.0.2
  - GPAUDITLOGGING-66
- 1.0.1
  - closures
  - nonVerboseDelete property
  - provide domain identifier to onSave() handler
- 1.0.0
  - Grails >= 2.0
  - ORM agnostic implementation
  - major cleanup and new features

    - fix #99 Plugin not working with MongoDB as Only Database
    - Changed issue management url to GH.
    - #13 Externalize AuditTrailEvent domain to user
- 0.5.5.3
    - Added ability to disable audit logging by config.
- 0.5.5.2
    - Added issueManagement to plugin descriptor for the portal. No changes in the plugin code.
- 0.5.5.1
    - Fixed the title. No changes in the plugin code.
- 0.5.5
    - collections logging
    - log ids
    - replacement patterns
    - property value masking
    - large fields support
    - fixes and enhancements
- 0.5.4
    - compatibility issues with Grails 1.3.x
- 0.5.3
    - GRAILSPLUGINS-2135
    - GRAILSPLUGINS-2060
    - an issue with extra JAR files that are somehow getting released as part of the plugin
- 0.5.2
    - GRAILSPLUGINS-1887 and GRAILSPLUGINS-1354
- 0.5.1
    - fixes regression in field logging
- 0.5
    - GRAILSPLUGINS-391
    - GRAILSPLUGINS-1496
    - GRAILSPLUGINS-1181
    - GRAILSPLUGINS-1515
    - GRAILSPLUGINS-1811

    - changes to AuditLogEvent domain object uses composite id to simplify logging
    - changes to AuditLogListener uses new domain model with separate transaction
    - for logging action to avoid invalidating the main hibernate session.
- 0.4.1
    - repackaged for Grails 1.1.1 see GRAILSPLUGINS-1181
- 0.4
    - custom serializable implementation for AuditLogEvent so events can happen inside a webflow context.
    - tweak application.properties for loading in other grails versions
    - update to views to show URI in an event
    - fix missing oldState bug in change event
- 0.3
    - actorKey and username features allow for the logging of user or userPrincipal for most security systems.
    - Fix #31 disable hotkeys for layout.

Chapter 3. Installation

3.1. Dependencies

Add to your build.gradle project dependencies block:

```
dependencies {
   compile 'org.grails.plugins:audit-logging:{version}'
}
```

Then run the following to refresh gradle dependencies:

```
gradle classes
```

After installing the plugin, you must perform the following command to let the plugin create the audit-logging domain class within your project.

3.2. Create Audit Domain Artifact

```
grails audit-quickstart <your.package.name> <YourAuditLogEventClassName>
```

For example:

```
grails audit-quickstart org.myaudit.example AuditTrail
```

This will create the Audit Domain class as well as configure:

```
grails.plugin.auditLog.auditDomainClassName = 'org.myaudit.example.AuditTrail'
```

Once created, you should open the generated file to adjust the `mappings` and `constraints` to suit your needs.

Be sure to respect the existing nullability constraints or the plugin may not work correctly

3.3. Prepare your Domain classes for Auditing

For every Domain class you want to be audited, implement the Auditable Trait.

For example:

```
import grails.plugins.orm.auditable.Auditable

class MyDomain implements Auditable {
    String whatever
    ...
}
```

If you additionally want to enable stamping, implement the Stampable Trait:

```
import grails.plugins.orm.auditable.Auditable

 class MyDomain implements Auditable, Stampable {
     String whatever
     ...
 }
 ```

== Upgrading

The 3.0.0 version of the plugin is a major rewrite of the codebase. Therefore you need
to upgrade your code for this version.

=== Domain Classes
Prior versions of the plugin used a static property syntax to enable and customize
logging:
[source,groovy]

// Legacy enable/disable
static auditable = true

// Legacy whitelist
static auditable = [auditableProperties: ['name', 'famous', 'lastUpdated']]

Starting with version 3.0.0, the plugin uses an `Auditable` trait and/or a `Stampable`
trait as both a marker as well as a way to configure and override auditing behavior.

For the first example, just adding `implements Auditable` will enable the default
behavior.

[source,groovy]
----
import grails.plugins.orm.auditable.Auditable
class MyDomain implements Auditable {
  ..
}

For the second example above, you simply override the appropriate method on the
```
```

`Auditable` trait, in this case the method is `getLogIncluded()`:

```
[source,groovy]
----
@Override
Collection<String> getLogIncluded() {
    ['name', 'famous', 'lastUpdated']
}
----
```

NOTE: The getter methods on `Auditable` all follow the `getLog*()` format to minimize collisions with user methods. Typically they are `getLog<Configuration Name>()` followed by whatever the configurable value (included, excluded, mask) with a few exceptions.

=== Configuration Changes

To support unifying the configuration, trait, and context usage, some of the configuration properties have been renamed. The following table should help to map old values and types to the new ones:

```
[width="100%",options="header,footer"]
|====================
| Prior Name | New Name | New Type
| auditableProperties
| included
| Collection<String>

| defaultIgnore
| excluded
| Collection<String>

| ignore
| excluded
| Collection<String>

| defaultMask
| mask
| Collection<String>

| nonVerboseDelete
| verboseEvents
| Collection<AuditEventType>

| transactional
| *Removed*
| Inherits existing transactionality

| actorClosure
| *Removed*
| Use AuditRequestResolver

| actorKey
```

```
| *Removed*
| Use AuditRequestResolver

| sessionAttribute
| *Removed*
| Use AuditRequestResolver

| stampAlways
| *Removed*
| Use a `TraitInjector` with `Stampable`
|====================

=== Transactional Behavior
Previously, the audit logging plugin had a `transactional` flag that indicated whether
to include the audit log saves in a transaction. However, the audit logging plugin
should really just participate (or not) in existing transactions and not make any
attempts to control transactionality at that grain. For most usage, you want your
Audit domain instances to be atomically created in the same transaction as the changes
that are triggering them. If there's no existing transaction for the changes, it's not
clear why there __should__ be a transaction just for the audit events or vice versa.

The Audit instances are still saved in a new session within any existing transaction.

=== Handlers Removed
The handler callbacks such as `onChange`, `onSave`, etc. have been removed starting
with version 3.0.0.

This behavior is already provided by Grails using the `before*` and `after*` callback
methods.

For example, you could do something like:
[source,groovy]
----
def beforeInsert() {
    def dirtyAudit = getAuditableDirtyPropertyNames()

    // Do something special if certain properties are dirty, etc.
}
----
Obviously, the `handlersOnly` configuration is also no longer relevant.

=== Actor Closure
The actor closure has been replaced with a more formalized `AuditRequestResolver`
strategy for resolving actor and URI information.

By default, the plugin uses the `DefaultAuditRequestResolver`, which gets Principal
information from the current Grails web request context. This is essentially the same
as the prior default actor closure.

If your application uses Spring Security, the plugin will register an instance of the
`SpringSecurityRequestResolver` which will use the `springSecurityService` to resolve
```

```
the current principal.

For other security frameworks, you can implement the `AuditRequestResolver` interface
and register a bean named `auditRequestResolver` to override the resolver provided by
the plugin.

=== Stampable Trait
The `@Stamp` annotation was removed in favor of a `Stampable` trait. This keeps the
usage more in line with the direction of Grails in general as well as simplifying the
implementation and usage.

However, there are some limitations to implementing this as a trait:

* Property names are `dateCreated, lastUpdated, createdBy, lastUpdatedBy` and are not
configurable
* If you want to customized the constraints, you must do so in your domain class
* The values are technically populated on the `ValidationEvent` since they are non-
nullable and must be populated prior to the default property validation.

The `stampAlways` has been removed. If you want to mark *all* of your domain objects
as stampable, you could define the following `TraitInjector`:

[source,groovy]
----
@CompileStatic
class StampableTraitInjector implements TraitInjector {
    @Override
    Class getTrait() {
        Stampable
    }

    @Override
    String[] getArtefactTypes() {
        ['Domain'] as String[]
    }
}
----

== Configuration

The plugin configuration can be specified in the application.groovy file.
It supports Environments blocks for environment-specific configuration.

[WARNING]
====
Since version 2.0.0, the configuration key has changed from "auditLog" to
"grails.plugins.auditLog".
If you use the old configuration key, the plugin will log a notice.
====

[cols="30,30,40"]
```

```
|====================
| *Property* | *Default Value* | *Meaning*

|grails.plugin.auditLog.auditDomainClassName
|*(Requred)* Use `grails audit-quickstart` to create
|Domain Class Name for the Audit Domain class.

|grails.plugin.auditLog.disabled
|`false`
|If set to true, audit logging is completely disabled (no listeners are registered).

|grails.plugin.auditLog.verbose
|`true`
|Column by column change logging in insert and delete events is enabled by default.
You can disable it if you are not interested in value changes.

|grails.plugin.auditLog.verboseEvents
|`[INSERT, UPDATE, DELETE]`
|Set of `AuditEventType` enums. Enables verbose logging on a per-event type basis. If
`verbose = true`, all event types are logged verbosely.

|grails.plugin.auditLog.ignoreEvents
|`[]`
|Set of `AuditEventType` enums that can be used to disable logging on a per-event type
basis. For example, setting `[AuditEventType.DELETE]` would disable audit logging for
all DELETE events only.

|grails.plugin.auditLog.failOnError
|`false`
|If the save of the Audit domain fails, enabling this will cause the failure to be
thrown out of the listener potentially failing the entire transaction.

|grails.plugin.auditLog.logIds
|`true`
|If set to true, object-ids of associated objects are logged.

|grails.plugin.auditLog.logFullClassName
|`true`
|Log the full entity name including the package name

|grails.plugin.auditLog.defaultActor
|`'SYS'`
|The default actor (user) to log if no user context can be found using the configured
`AuditRequestResolver` implementation (see below).

|grails.plugin.auditLog.excluded
|`['version', 'lastUpdated', 'lastUpdatedBy']`
|These properties are not logged and will not trigger verbose logging when changed.

|grails.plugin.auditLog.included
|`null` (all persistent properties)
```

```
|This is a global whitelist of properties to log and will override anything in the
excluded list. A null value indicates that all persistent properties not in the
`excluded` list specifically should be logged.

|grails.plugin.auditLog.mask
|`['password']`
|Any property with this name will be masked using the `propertyMask` attribute (below
).

|grails.plugin.auditLog.propertyMask
|`pass:[**********]`
|String to use when masking properties.

|grails.plugin.auditLog.truncateLength
|`maxSize` of `newValue`/`oldValue`
|Allow overriding the maximum length allowed in the `oldValue` and `newValue` fields
before truncating. This property defaults to the max size allowed by the `constraint`,
so it's best to just control it that way.

|grails.plugin.auditLog.stampEnabled
|`true`
|Enable stamping support for entities with the `Stampable` trait (dateCreated,
lastUpdated, createdBy, lastUpdatedBy).
|====================
NOTE: You can view the above defaults in the `DefaultAuditLogConfig.groovy`

=== Verbose mode
When enabled, per-property logging is enabled for all operations.

    grails.plugin.auditLog.verbose = true

You can enable verbose logging for specific event types using:

    grails.plugin.auditLog.verboseEvents = [AuditLogEvent.UPDATE]

[WARNING]
====
Verbose logging will insert 1 row of data per property of your domain object for
insert and delete operations. This can result in a large amount of additional data,
sometimes far more than the actual operation itself. Consider disabling verbose mode
for bulk operations using the `AuditLogContext.withoutVerboseAuditLog { }` closure.
====

=== Logging of Associated Ids
You can log the object-ids of associated objects. Logging will be performed in the
format: "[id:<objId>]objDetails".
[source,groovy]
----
grails.plugin.auditLog.logIds = true
----
```

```
This setting is enabled by default.

=== Property Value Masking
You can configure properties to mask on a per-Domain-Class base. If properties are
defined as masked, their values are not stored into the audit log table if verbose
mode is enabled. Instead, a mask of "**********" will be logged.

By default, "password" properties are masked. You can mask property fields in domain
classes like this:

[source,groovy]
----
@Override
Collection<String> getLogMask() {
    ['password', 'otherField']
}
----

=== Verbose Log Truncation Length
If you enabled verbose mode, you can configure the truncation length of detail
information in the oldValue and newValue
columns (Default is 255). Configure the truncateLength in application.groovy:

[source,groovy]
----
truncateLength = 400
----

[WARNING]
====
When you set truncateLength to a value > 255 you must ensure that oldValue and
newValue fields in your audit-log domain class are large enough. Example setting with
the same maxSize constraints as the former "largeValueColumnTypes" setting:

[source,groovy]
----
static constraints = {
    // For large column support (as in < 1.0.6 plugin versions)
    oldValue(nullable: true, maxSize: 65534)
    newValue(nullable: true, maxSize: 65534)
}
----
When you forget to set the constraints in your AuditLog class while setting
truncateLength > 255,
a truncation warning may occur and only partial information is logged.
====

=== Disable All Auditing
You can disable auditing by config. If you disable auditing, event handlers are still
triggered but no changes are comitted to the audit log table. This can be used e.g. if
you need to bootstrap many objects and want to programmatically disable auditing to
```

```
not slow down the bootstrap process or if you want to audit log by Enviroment.

[source,groovy]
----
grails.plugin.auditLog.disabled = true
----

This setting is "false" by default (auditing is enabled).

=== Log Full Class Name
By default, only the entity class name is logged. If you want to log the entity full
name (including the package name), you can enable full logging. Thanks to tcrossland
for this feature.

[source,groovy]
----
grails.plugin.auditLog.logFullClassName = true
----

This setting is "true" by default (full name is logged).


=== Ignoring Specific Events
To ignore certain events on a per-domain base, override the `getLogIgnoreEvents()`
method:
[source,groovy]
----
@Override
Collection<AuditEventType> getLogIgnoreEvents() {
    [AuditEventType.INSERT]
}
----
You can also ignore them either globally with:
[source,groovy]
----
grails.plugin.auditLog.ignoreEvents = [AuditEventType.INSERT]
----
Or for a specific logging context by using:
[source,groovy]
----
AuditLogContext.withConfig(ignoreEvents = [AuditEventType.INSERT]) {
    //
    // Anything here will only log UPDATE and DELETE events
    //
}
----
=== Runtime Overrides
See the <<index#context-overrides,Context Overrides>> for help using `AuditLogContext`
to override configuration for a block.

=== Example Configuration
```

````
Example `application.groovy` configuration with various settings as described above:

[source,groovy]
----
// AuditLog Plugin config
grails {
    plugin {
        auditLog {
            auditDomainClassName = 'my.example.project.MyAuditTrail'
            verbose = true
            failOnError = true
            excluded = ['version', 'lastUpdated', 'lastUpdatedBy']
            mask = ['password']
            logIds = true
            stampEnabled = true
        }
    }
}
----

== Usage
You can use the grails-audit-logging plugin in several ways.

=== Basic
To enable auditing using the default configuration, simply implement the `Auditable`
trait.

For example:
[source,groovy]
----
class Author implements Auditable {
    String name
    Integer age

    static hasMany = [books: Book]
}
----
This will enable auditing for the domain using your configuration in `grails.plugin
.auditLog` merged with the defaults from `DefaultAuditLogConfig.groovy` (in the plugin
jar).

=== Context Overrides
There are many scenarios where you want to use a default configuration for most
operations, but need to explicitly override for certain scenarios.

One fairly common example is the need to disable audit logging completely for a
particular set of changes. This is often the case for bulk operations or operations
that might be performed "by the system" such as cleanup, purge, expiration jobs, etc.

==== Audit Log Context
You can use the `AuditLogContext` object to override almost any configuration
````

```
parameter for a given block

For example:
[source,groovy]
----
// Enable verbose logging with full class name excluding only properties named 'ssn'
AuditLogContext.withConfig(verbose: true, logFullClassName: true, excluded: ['ssn']) {
... }

// Disable verbose logging
AuditLogContext.withConfig(verbose: false) { ... }

// Disable all logging
AuditLogContext.withConfig(disabled: true) { ... }
----
The last two are so common, there's a method that does them by name:

[source,groovy]
----
// Disable verbose logging
AuditLogContext.withoutVerboseAuditLog { ... }

// Disable all logging
AuditLogContext.withoutAuditLog { ... }
----

WARNING: The AuditLogContext stores the context using ThreadLocal, so configuration is
only present for the current thread. If you start something asynchronously, be sure
you setup the context within the new execution thread.


=== Domain Overrides
The context gives you programmatic control over configuration of auditing but you can
go even finer grained, if needed, to control auditing on a per-entity basis.

To override auditing for a specific entity or based on runtime values, just override
the trait method:

[source,groovy]
----
// Assuming this domain has a status attribute, disable auditing for anything not
active, for example
@Override
boolean isAuditable() {
    this.status == ACTIVE
}
----

Another example is to override which properties get ignored for a specific entity:
[source,groovy]
----
```

```
@Override
Collection<String> getLogExcluded() {
    ['myField', 'version']
}
----

Control the specific whitelist of attributes logged:
[source,groovy]
----
@Override
Collection<String> getLogIncluded() {
    ['whiteListField1', 'whiteListField2']
}
----

Customize the id that is logged for the entity, for example instead of using Author's
`id` property, log the Author's name and age as `Tolkien:85`:
[source,groovy]
----
@Override
String getLogEntityId() {
    "${name}:${age}"
}
----

The ability to override and augment the default behavior provided by the trait is what
makes the trait implementation so flexible.

NOTE: Most of the `Auditable` methods rely on the AuditLogContext for defaults. If you
override the trait method, ensure you consider the keeping the default behavior if you
are supplementing.

=== Customized Auditable Trait
You could also extend the trait entirely to override the default auditing behavior:
[source,groovy]
----
trait MyAuditable<D> extends Auditable<D> {
    // Customize formatting behavior for new and old values
    @Override
    String convertLoggedPropertyToString(String propertyName, Object value) {
        if (value instanceof MySpecialThing) {
            return ((MySpecialThing)value).formatAsString()
        }
        super.convertLoggedPropertyToString(propertyName, value)
    }

    // Customize to populate custom attributes on the audit log entity
    @Override
    boolean beforeSaveLog(Object auditEntity) {
        auditEntity.naturalKey = getNaturalKey()
    }
```

```
}
----

=== Request Resolvers
Audit logging usually requires auditing the user and/or request context for a specific
action. The plugin supports a pluggable method of resolving request context by
registering an `auditRequestResolver` bean that applications can override if needed.

==== Audit Request Resolvers
The plugin ships with two resolvers:
[source,groovy]
----
class DefaultAuditRequestResolver implements AuditRequestResolver { ... }
----
and
[source,groovy]
----
class SpringSecurityRequestResolver extends DefaultAuditRequestResolver { ... }
----

The default resolver uses the `principal` in the active GrailsWebRequest to resolve
the Actor name and Request URI for logging purposes.

If a bean named `springSecurityService` is available, the second resolver is
registered which uses the `currentUser()` method to resolve the user context.

For other authentication strategies, you can implement and override the
`auditRequestResolver` bean with your own implementation of:
[source,groovy]
----
interface AuditRequestResolver {
    /**
     * @return the current actor
     */
    String getCurrentActor()

    /**
     * @return the current request URI or null if no active request
     */
    String getCurrentURI()
}
----
Just register your resolver in `resources.groovy`:
[source,groovy]
----
beans = {
    auditRequestResolver(CustomAuditRequestResolver) {
        customService = ref('customService')
    }
}
----
```

```
Below are a few examples for other common security frameworks.

===== Acegi Plugin
[source,groovy]
----
/**
 * @author Jorge Aguilera
 */
class AcegiAuditResolver extends DefaultAuditRequestResolver {
    def authenticateService

    @Override
    String getCurrentActor() {
        authenticateService?.principal()?.username ?: super.getCurrentActor()
    }
}
----

===== CAS Authentication
[source,groovy]
----
import edu.yale.its.tp.cas.client.filter.CASFilter

class CASAuditResolver extends DefaultAuditRequestResolver {
    def authenticateService

    @Override
    String getCurrentActor() {
        GrailsWebRequest request = GrailsWebRequest.lookup()
        request?.session?.getAttribute(CASFilter.CAS_FILTER_USER)
    }
}
----

===== Shiro Plugin
[source,groovy]
----
@Component('auditRequestResolver')
class ShiroAuditResolver extends DefaultAuditRequestResolver {
    @Override
    String getCurrentActor() {
        org.apache.shiro.SecurityUtils.getSubject()?.getPrincipal()
    }
}
----

== Stamping
Stamping adds the following attributes to a domain object via the `Stampable` trait:

[source,groovy]
```

```
----
trait Stampable<D> extends GormEntity<D> {
    Date dateCreated
    Date lastUpdated

    String createdBy
    String lastUpdatedBy
}
----

The attributes will assume the default constraints, which should be fine for most
cases.

You can configure the `constraints` and `mappings` block as usual in your domain class
to customize the stampable properties.

=== Stamping Configuration
Previous versions used an AST transformation to apply the stamping and could allow
more configuration.
For now, the stamping support has been boiled down to just the basic trait attributes
and the ability to disable the plugin globally.

[source,groovy]
----
// Enable or disable stamping
grails.plugin.auditLog.stampEnabled = true
----

WARNING: If you disable stamping but have `Stampable` entities, they will likely fail
validation since the plugin will not be populating the fields which are still added to
the domain objects. The main reason to disable stamping is to prevent the listener
registration in the case that you just aren't using the `Stampable` support.

If you want to mark *all* of your domain objects as stampable, you could define the
following `TraitInjector`:

[source,groovy]
----
@CompileStatic
class StampableTraitInjector implements TraitInjector {
    @Override
    Class getTrait() {
        Stampable
    }

    @Override
    String[] getArtefactTypes() {
        ['Domain'] as String[]
    }
}
----
```

```
== Implementation
Most of the plugin code is marked as @CompileStatic.

=== AuditLogEventListener
The Audit Logging plugin registers a PersistenceEventListener (`AuditLogListener`)
bean per datasource, which listens to GORM events.

=== StampEventListener
The plugin registers a separate StampEventListener that responds to validate events.
Ideally, we would stamp when an insert or update event occurs, but since the
attributes are non-null by default, we need to populate them before the default
validation is triggered.

=== Plugin Descriptor
The Plugin Descriptor (AuditLogListenerGrailsPlugin) configures the plugin during
startup.

 * Configures the plugin either by default values - see DefaultAuditLogConfig.groovy -
or by user configured settings.
 * Registers a PersistenceEventListener bean per datasource

=== Auditable trait
Enabling auditing on a Domain class is done by implementing the `Auditable` trait.

=== Stampable trait
Enabling stamping on a Domain class is done by implementing the `Stampable` trait.
This trait adds the properties dateCreated, lastUpdated, createdBy, lastUpdatedBy
to the domain class.
```