Enforcer

Plugin Documentation

Table of Contents

Chapter 1. Introduction

Intro Video

The Enforcer Plugin, gives you the tools to enforce business rules, and or permissions. Enforcer is light weight, easy to maintain, extend and, use. The plugin works off of a EnforcerService in conjunction with the Enforce, Reinforce, and ReinforceFilter Annotations(AST transforms).

For Enforcement the service takes up to 3 closures, a predicate, a failure(defaults to an EnforcerException if not specified) and a success(defaulted to a closure that returns true). The predicate is evaluated, if it returns true, the the success closure is evaluated, else the failure closure is evaluated. With this you can enforce any business rule you need. Reinforce does the same, but is injected at the end of the method before the return statement, rather than the beginning. The ReinforceFilter allows you to filter the returned value of any method.

The EnforcerService, by default implements two traits, RoleTrait, and DomainRoleTrait. This is so you can use the methods in the traits without injecting any extra calls to services, which makes the calls to the service less verbose, and easier to read. The preferred method of extending the Enforcer Service is to add new methods to the traits, modify the existing methods in the traits, or add a new trait. These traits make the EnforcerService, and by extension the annotations into a DSL for enforcing business rules/permission

The annotations Enforce and Reinforce can be applied to any class or method. When applying the annotation to a class, it will then be applied to all the methods in the class, and be overridden if it is also applied at any of the methods. The ReinforceFilter can be applied to any method that returns a valueThe. The Annotations are AST transforms so they are applied at compile time, without any third party annotation processor. The annotations make it clear when you are enforcing business rules which gets out of the way of your logic code.

The Enforcer plugin was inspired by the limitations, and rigidity of other frameworks, and the thought that a better alternative should exist.

Also Zed Shaw's The ACL is Dead, take this with a grain of salt, it tends to veer off course, but does bring up some valid points:

The ACL is Dead

Chapter 2. Version History

- 0.2.1 and 0.3.1
  - Initial Release including the Enforce annotation that can only be applied to methods. Documentation is gdoc.
- 1.2.1 and 1.3.1
  - Upgraded Enforce so that it can be applied both at the class and method level: https://github.com/virtualdogbert/Enforcer/issues/7
  - Added Reinforce and ReinforceFilter annotations
  - Updating documentation to Asciidoctor
  - Update video documentation
  - Fixed a bug with the default domainRole trait: https://github.com/virtualdogbert/Enforcer/issues/9

Chapter 3. Getting Started

Getting Started Video

1. Have the Spring Security Core plug-in installed
2. Make sure you've run the quick start(e.g. grails s2-quickstart com.security User Role) note the package name
3. Add the Enforcer dependency:

In Grails 2 add the following to your buildConfig.groovy under the plug-ins section:

```
compile ":enforcer:1.2.1"
```

In grails 3 add the following dependency to your gradle.build:

```
compile "org.grails.plugins:enforcer:1.3.1"
```

4.Run grails enforcer-quickstart <name of the package you installed spring security core under>

5.Check the Usage section and start using Enforcer.

3.1. Example applications

For Grails 2:

GitHub testEnforcer2

For Grails 3:

GitHub testEnforcer3

GitHub Grails3Tutorial

```
The version scheme is Major.GrailsVersion.Minor
```

Chapter 4. Usage

Usage Video

Here are some code examples of what you can do with the Enforcer plug-in:

Check to domain role for the role owner on the Sprocket domain instance for a user:

```
def enforcerService
enforcerService.enforce({ hasDomainRole('owner', sprocket, testUser) })
```

Checks to see if the test user has the Role ROLE_USER:

```
def enforcerService
enforcerService.enforce({ hasRole('ROLE_USER', testUser) })
```

Those same checks on a method using @Enforce:

```
@Enforce({hasRole('ROLE_USER', testUser)  && hasDomainRole('owner', sprocket,
testUser)})
def someMethod(){
    //some logic
}
```

Or the same check using Reinforce:

```
@Reinforce({hasRole('ROLE_USER', testUser)  && hasDomainRole('owner', sprocket,
testUser)})
def someMethod(){
    //some logic
}
```

An example of ReinforceFilter which takes the original value that would be returned, and the result of the closure will take that returned values place, be careful that that returned value of the closure matches the returned value of the method, if you are not using def:

```
@ReinforceFilter({ Object o -> (o as List).findResults { it % 2 == 0 ? it : null } })
    List<Integer> reinforceFilter() {
        [1, 2, 3, 4, 5, 6, 7, 8, 9]
    }
```

For More examples check out the unit tests:

- EnforcerServiceSpec.groovy

- EnforcerAnnotationSpec.groovy
- ReinforceAnnotationSpec.groovy

Chapter 5. Why use Enforcer?

Why User Enforcer Video

So why would you want to use Enforcer?

With Enforcer I try to set you up to be successful, in enforcing business rules/permissions, in a way that is easy to read,is expendable without the limitations I see in other frameworks. So lets take a look at alternatives, and how Enforcer addresses some of there limitations.

5.1. Spring Security ACL

1. It's annotations use the Spring EL language, in the form of a string, so... no compiler help, no IDE help(newer versions of Intellij may help with this), no syntax highlighting, and EL isn't as flexible as groovy.
2. It uses a "bit mask" for storing permission, while this is "efficient", but hard to read, also we're not in the 80's anymore, and disk is cheap.
3. It uses a highly normalized set for domains/tables for storing permission. This means that querying the db will involve many joins, which will slow down the writing and running of queries.
4. Updating permission especially in bulk can be really slow, because of the built-in caching. If you use direct SQL you have to invalidate the plug-ins internal cache.
5. The annotation is a Spring Proxy, so it only gets called if you are calling from outside the class, the annotation is in. This can lead to a false sense of security, or false negatives.
6. If you are not using a hierarchy for your permission, which I don't see a config option for, you will end up with a lot of permissions in the db. I've seen it as bad and 400,000 permission entries for just over a 1000 containers.
7. Extending the plug-in is not straight forward. Adding new permissions isn't bad but you have to deal with the "bit mask". However if you want to add functions that can be called in the EL expression... good luck. Adding new functions that you can call would involve some been overrides, and extending classes. I've tried it twice and it didn't work either time. I do thing however you maybe able to call services from within the EL, with the current version, but don't quote me on that.
8. It's somewhat hard to setup tests your annotations.

5.2. Shiro

1. I'll be honest I don't really get Shiro, and I haven't taken the time to understand it. The permissions aren't as readable as I would like.
2. I know the library of Shiro not the plugin does have annotations, but I suspect that they would have the same issue that the spring security one have being proxies.

5.3. Drools

1. I never used it, but I haven't heard good things, and from what I can see the plugins are not being actively maintained.

5.4. GContracts

1. It seems the last commit was 4 years ago, and it wan't meant to work with Grails:

https://github.com/andresteingress/gcontracts/wiki/GContracts-in-Spring-and-Grails-Applications

5.5. Enforcer in response:

1. Use a closure, so you have the full flexibility of groovy, your IDE, will help you and will do syntax highlighting.
2. The default DomainRole, uses a string which is easy to read and query.
3. The DomainRole is denormalized, so that it will be quicker, and involves less queries to read, and write.
4. As said before DomainRole is denormalized so it's quick to update.
5. The Enforce AST transform/annotation runs every time a method it's annotation is called, unless your are running from the test environment.
6. DomainRole uses a hierarchy by default so you'll have less entries in the db, which will be easier to query, and migrate if you have too. That 400,000 for 1000 contains was reduced to 20,000 entries using a hierarchy.
7. Extending Enforcer is easy, by adding methods to either of the traits the EnforcerService uses, or adding your own trait, to just calling any service from with in the closure(s) you pass in.
8. Testing is easy see the EnforcerServiceSpec for examples.

For the rest I don't see them as viable competition at this point, mainly, because most of them aren't being maintained. All of these tools had there usage, but I think it's time for something better, and in the future maybe a better plugin will be written, than Enforcer.

Chapter 6. Extending Enforcer

Extending Enforcer Video

After you run the Enforcer quick start script, in your services you will have the EnforcerService, the RoleTrait and the DomainRole Trait. You can add function to either of the traits, or add your own traits. The traits should just be seen as a suggested starting point. You could add whatever you need like project, group, or feature flag checks.

Chapter 7. Testing

Testing Video

The quick start will install the unit tests EnforcerServiceSpec.groovy, EnforcerAnnotationSpec.groovy and, ReinforceAnnotationSpec.groovy, which will allow you to test the Enforcer service and the Enforce Reinforce, and ReinforceFilter annotations(AST Transforms)*[]:

By default the test for a DomainRole is commented out, you will have to comment it back in and replace the Sprocket domain, with one from your own application. You will also have to add that domain class to the mock section. Here are some example implementations of the enforcer unit test:

- EnforcerServiceSpec.groovy
- EnforcerAnnotationSpec.groovy
- ReinforceAnnotationSpec.groovy

Chapter 8. The Enforcer AST Transforms

Enforcer How The ASTs Work Video

The Enforcer AST transform Enforce allows you to put enforcer checks on methods, and classes, making then distinguishable from the main logic as business rule/security checks.

8.1. How the Enforce AST Transform was built

The Enforce Transform was built in the testAst project:

Enforcer testAST github

Using the ./scripts/_Events.groovy eventCompileStart hook to precompile the Enforce transform. If you import this project into Intellij You will be able to set break points in the AST transform, and Intellij will pick them up, which make debugging AST transforms a lot easier. I also used the Groovy/Grails Console, and the Inspect AST mode to see, what statements and expressions make up the code I wanted to write. At some point I will upgrade the project to use Grails 3, and use gradle rather then the _Events.groovy.