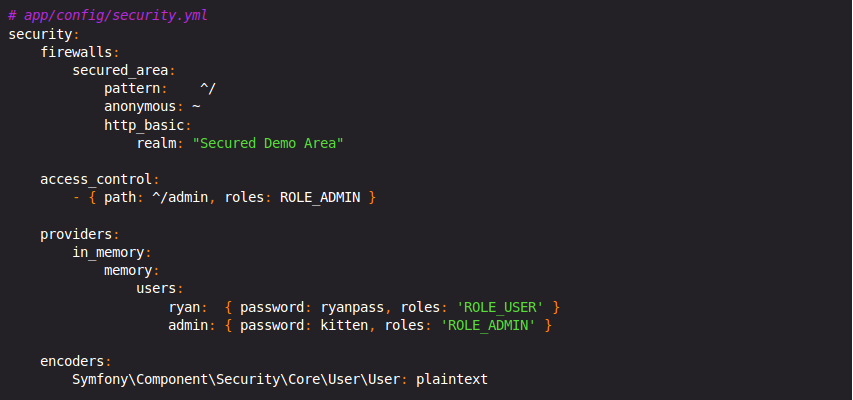
**Symfony 2.1 Security & ACL**

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Author** |
| 12/12/2012 | 1.0 | Quang Tran |

## HTTP Authentication

The security component can be configured via your application configuration. In fact, most standard security setups are just a matter of using the right configuration. The following configuration tells Symfony to secure any URL matching /admin/\* and to ask the user for credentials using basic HTTP authentication (i.e. the old-school username/password box):

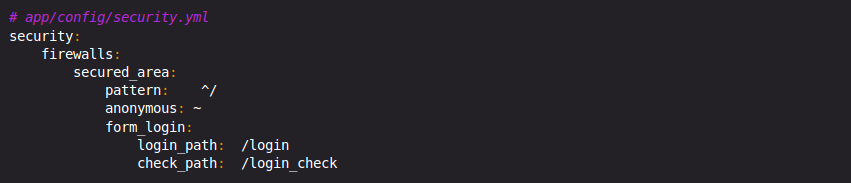
The end result of this configuration is a fully-functional security system that looks like the following:

* There are two users in the system (ryan and admin);
* Users authenticate themselves via the basic HTTP authentication prompt;
* Any URL matching /admin/\* is secured, and only the admin user can access it;
* All URLs not matching /admin/\* are accessible by all users (and the user is never prompted to login).

Let's look briefly at how security works and how each part of the configuration comes into play.

## Using a Traditional Login Form

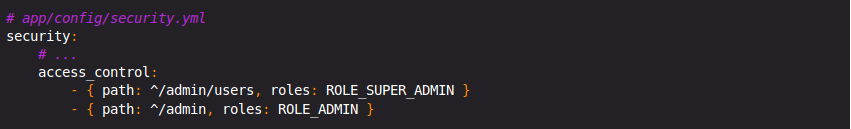
So far, you've seen how to blanket your application beneath a firewall and then protect access to certain areas with roles. By using HTTP Authentication, you can effortlessly tap into the native username/password box offered by all browsers. However, Symfony supports many authentication mechanisms out of the box.  
In this section, you'll enhance this process by allowing the user to authenticate via a traditional HTML login form. First, enable form login under your firewall:

Now, when the security system initiates the authentication process, it will redirect the user to the login form (/login by default). Implementing this login form visually is your job. First, create two routes: one that will display the login form (i.e. /login) and one that will handle the login form submission (i.e. /login\_check):

### A description...Securing Specific URL Patterns

The most basic way to secure part of your application is to secure an entire URL pattern. You've seen this already in the first example of this chapter, where anything matching the regular expression pattern ^/admin requires the ROLE\_ADMIN role.

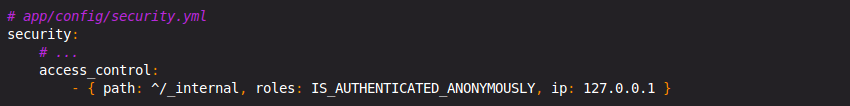
You can define as many URL patterns as you need - each is a regular expression.

For each incoming request, Symfony2 tries to find a matching access control rule (the first one wins). If the user isn't authenticated yet, the authentication process is initiated (i.e. the user is given a chance to login). However, if the user is authenticated but doesn't have the required role, an AccessDeniedException exception is thrown, which you can handle and turn into a nice "access denied" error page for the user.

Since Symfony uses the first access control rule it matches, a URL like /admin/users/new will match the first rule and require only the ROLE\_SUPER\_ADMIN role. Any URL like /admin/blog will match the second rule and require ROLE\_ADMIN.

### Securing by IP

Certain situations may arise when you may need to restrict access to a given route based on IP. This is particularly relevant in the case of Edge Side Includes (ESI), for example, which utilize a route named "\_internal". When ESI is used, the \_internal route is required by the gateway cache to enable different caching options for subsections within a given page. This route comes with the ^/\_internal prefix by default in the standard edition (assuming you've uncommented those lines from the routing file).  
Here is an example of how you might secure this route from outside access:

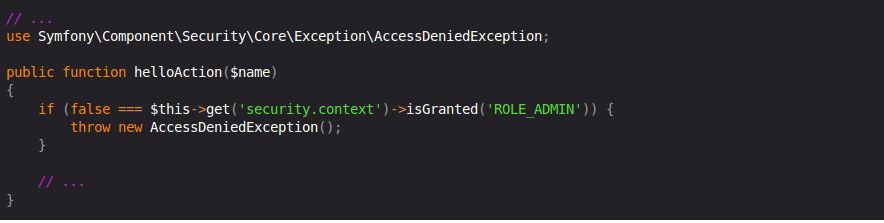


### Securing by Channel

Much like securing based on IP, requiring the use of SSL is as simple as adding a new access\_control entry:

### Securing a ControllerA description...

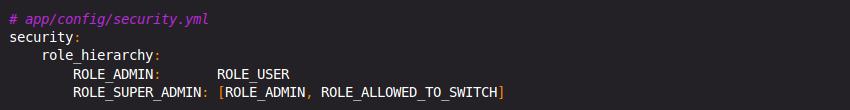
Protecting your application based on URL patterns is easy, but may not be fine-grained enough in certain cases. When necessary, you can easily force authorization from inside a controller:



You can also choose to install and use the optional JMSSecurityExtraBundle, which can secure your controller using annotations:

### A description...Hierarchical Roles

Instead of associating many roles to users, you can define role inheritance rules by creating a role hierarchy:



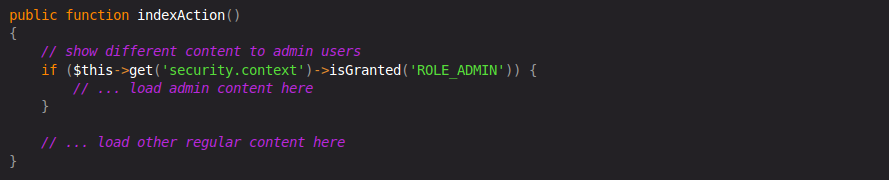
In the above configuration, users with ROLE\_ADMIN role will also have the ROLE\_USER role. The ROLE\_SUPER\_ADMIN role has ROLE\_ADMIN, ROLE\_ALLOWED\_TO\_SWITCH and ROLE\_USER (inherited from ROLE\_ADMIN).

## Access Control in Templates

If you want to check if the current user has a role inside a template, use the built-in helper function:

**Access Control in Controllers**

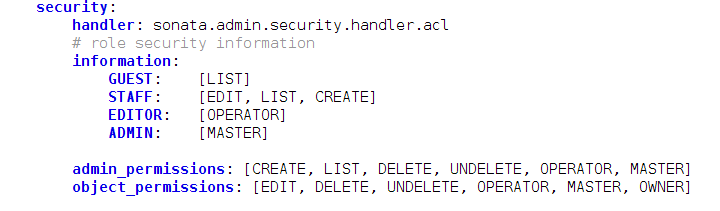
If you want to check if the current user has a role in your controller, use the [isGranted()](http://api.symfony.com/2.1/Symfony/Component/Security/Core/SecurityContext.html" \l "isGranted()) method of the security context:



**ACL Sonata Admin configuration:** The security part is managed by a SecurityHandler, the bundle comes with 3 handlers

* sonata.admin.security.handler.role : ROLES to handle permissions
* sonata.admin.security.handler.acl : ACL and ROLES to handle permissions
* sonata.admin.security.handler.noop : always returns true, can be used with the Symfony2 firewall

We use ACL and ROLES to handle permission: *# app/config/sonata.yml*

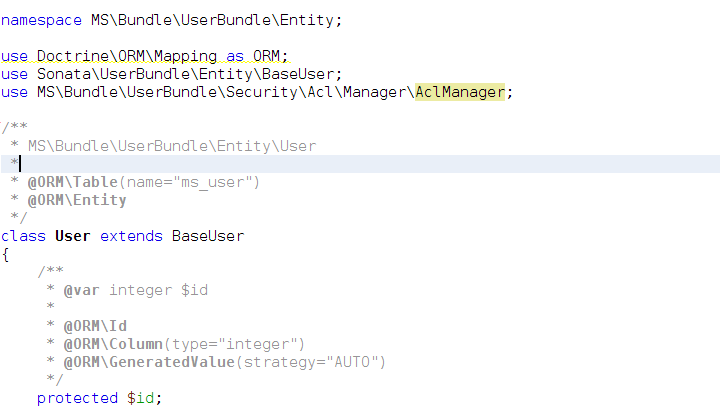


information: *# acl security information*

admin\_permissions:

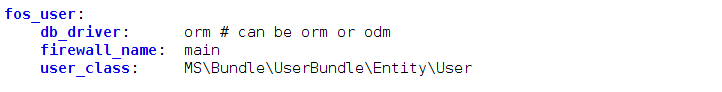
object\_permissions: *# permission related to the objects*

**Setup ACL with the FOSUserBundle:**Before you can use FriendsOfSymfony/FOSUserBundle you need to set it up as described in the documentation of the bundle.



*Note: Sonata\UserBundle\Entity\BaseUser extends FOS\UserBundle\Entity\User*

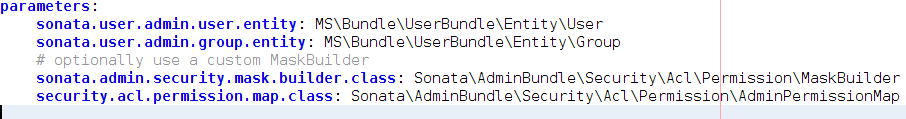
In your **app/config/config.yml** you then need to put the following:



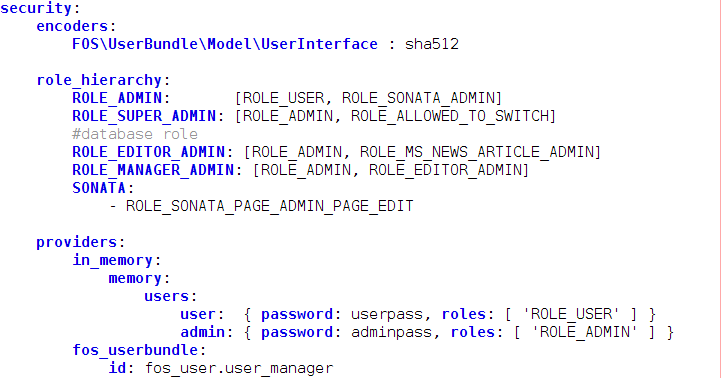
The following configuration for the SonataUserBundle defines:

* the FriendsOfSymfony/FOSUserBundle as a security provider
* the login form for authentification
* the access control : resources with related required roles, the important part is the admin configuration
* the acl option to enable the ACL.
* the AdminPermissionMap defines the permissions of the Admin class

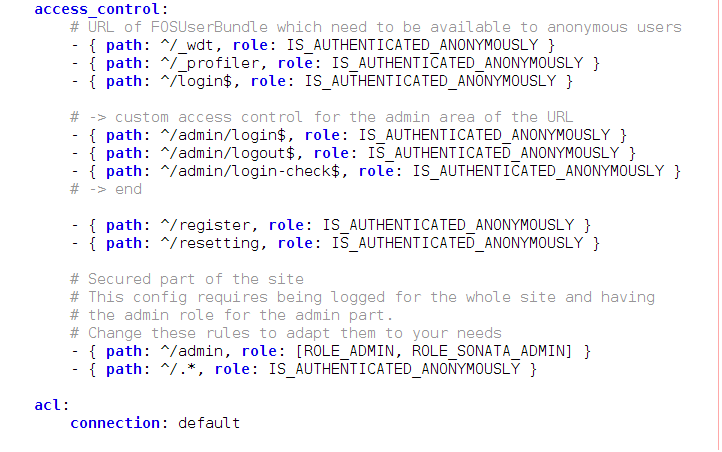
In your **app/config/sonata.yml** you then need to put the following:



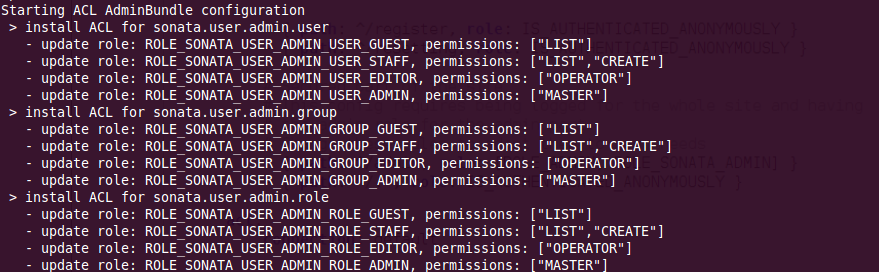
In app/config/security.yml:



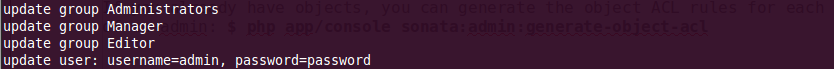




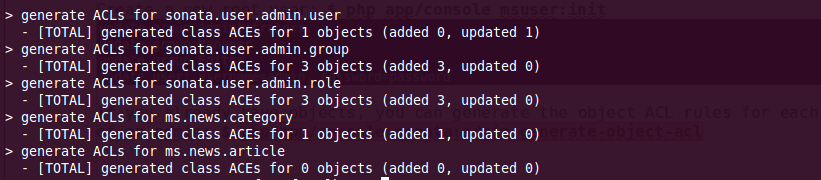
Install the ACL tables: **$** **php app/console sonata:admin:setup-acl**



Create a new root user and groups: **$** **php app/console msuser:init**



If you already have objects, you can generate the object ACL rules for each object of an admin: **$ php app/console sonata:admin:generate-object-acl**



### **Roles and Access control lists**

A user can have several roles when working with an application. Each Admin class has several roles, and each role specifies the permissions of the user for the Admin class. Or more specifically, what the user can do with the domain object(s) the Admin class is created for.

By default each Admin class contains the following roles, override the property $securityInformation to change this:

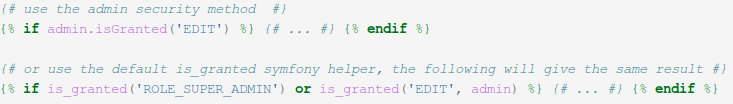
* ROLE\_SONATA\_...\_GUEST : a guest that is allowed to view an object and a list of objects;
* ROLE\_SONATA\_...\_STAFF : probably the biggest part of the users, a staff user has the same permissions as guests and is additionally allowed to EDIT and CREATE new objects;
* ROLE\_SONATA\_...\_EDITOR: an editor is granted all access and, compared to the staff users, is allowed to DELETE;
* ROLE\_SONATA\_...\_ADMIN : an administrative user is granted all access and on top of that, the user is allowed to grant other users access.

### **Usage**

Everytime you create a new Admin class, you should start with the command **php app/console sonata:admin:setup-acl** so the ACL database will be updated with the latest roles and permissions.

In the templates, or in your code, you can use the Admin method isGranted() :

* check for an admin that the user is allowed to EDIT :



* check for an admin that the user is allowed to DELETE, the object is added to also check if the object owner is allowed to DELETE :

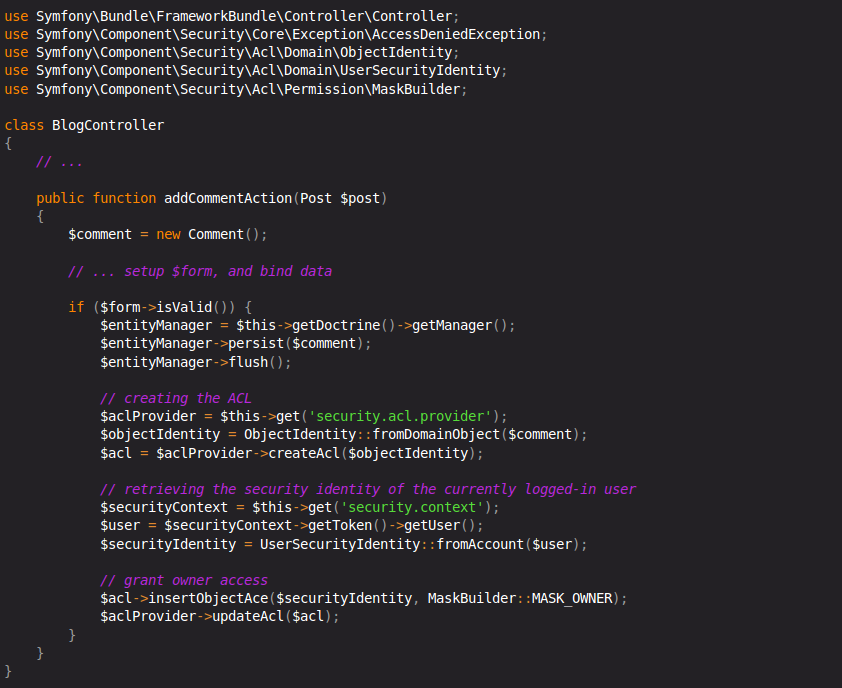
### A description...

### **ACL on Frontend**

Now, before you can finally get into action, you need to do some bootstrapping. First, you need to configure the connection the ACL system is supposed to use: # in **app/config/sonata.yml** or in **app/config/config.yml**



### *Creating an ACL and adding an ACE*

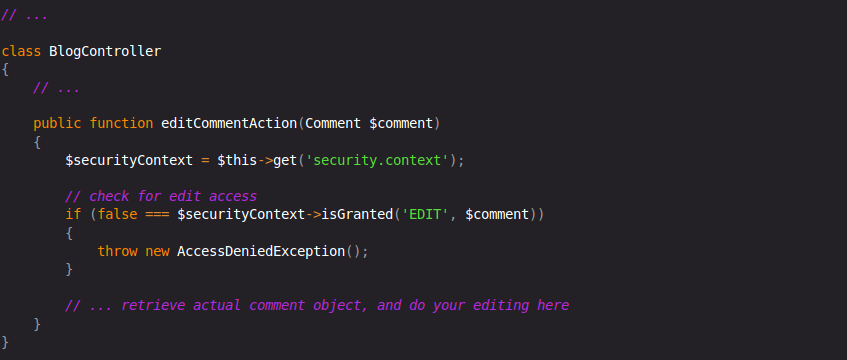


There are a couple of important implementation decisions in this code snippet. For now, I only want to highlight two:

First, you may have noticed that ->createAcl() does not accept domain objects directly, but only implementations of the ObjectIdentityInterface. This additional step of indirection allows you to work with ACLs even when you have no actual domain object instance at hand. This will be extremely helpful if you want to check permissions for a large number of objects without actually hydrating these objects.

The other interesting part is the ->insertObjectAce() call. In the example, you are granting the user who is currently logged in owner access to the Comment. The MaskBuilder::MASK\_OWNER is a pre-defined integer bitmask; don't worry the mask builder will abstract away most of the technical details, but using this technique you can store many different permissions in one database row which gives a considerable boost in performance.

*Checking Access*

In this example, you check whether the user has the EDIT permission. Internally, Symfony2 maps the permission to several integer bitmasks, and checks whether the user has any of them

## *Cumulative Permissions*

In the first example above, you only granted the user the OWNER base permission. While this effectively also allows the user to perform any operation such as view, edit, etc. on the domain object, there are cases where you may want to grant these permissions explicitly.

The MaskBuilder can be used for creating bit masks easily by combining several base permissions:

**

This integer bitmask can then be used to grant a user the base permissions you added above:

The user is now allowed to view, edit, delete, and un-delete objects.A description...