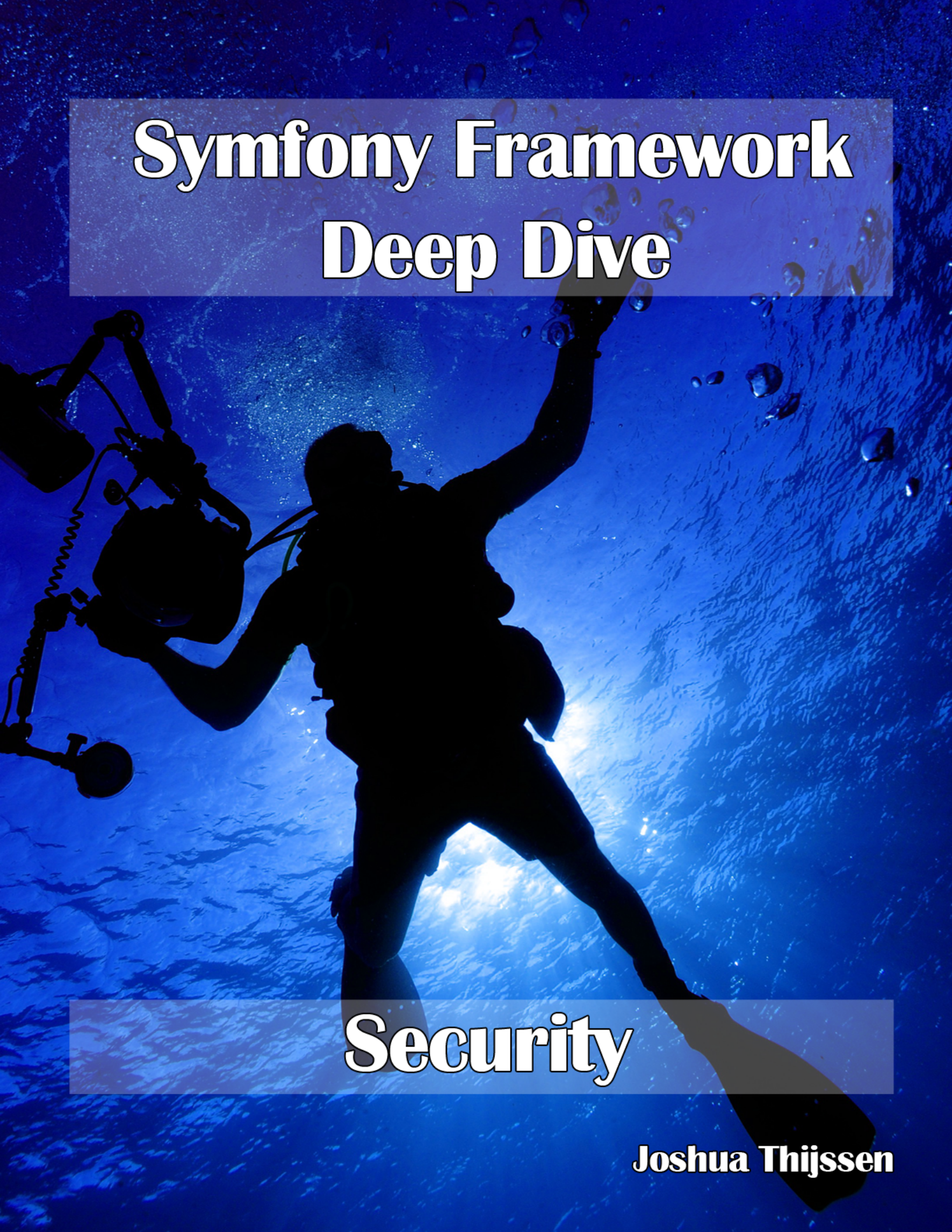
Symfony Framework Deep Dive



Security

Joshua Thijssen

Symfony Framework Deepdive - Security

A deepdive into the Symfony security component

Joshua Thijssen

This book is for sale at http://leanpub.com/symfonyframeworkdeepdive-security

This version was published on 2015-02-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Joshua Thijssen by spreading the word about this book on Twitter!

The suggested hashtag for this book is #symfonyrainbow.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#symfonyrainbow

Contents

Sample edition

This ebook is a sample of security edition of the **Symfony Framework Deep Dive** series. Not all chapters and sections are present in this sample, but nevertheless, it will give you an impression of the full book. Note that the book is still in active development, meaning that chapters, content and even layout can and probably will change.

For more information about the book, please visit our LeanPub website[1].

[1] https://leanpub.com/symfonyframeworkdeepdive-security

Introduction

About the Symfony rainbow series

The Symfony rainbow series is a collection of books based around the different components that make up the Symfony framework. Every book will concentrate on a single component or a set of smaller components. The books will describe their inner workings in detail and give you recipes to solve common problems quickly and efficiently.

[2] http://pixabay.com/en/sea-ocean-water-light-diver-79606/

[3] http://creativecommons.org/publicdomain/zero/1.0/deed.en

About those who will read this book

This book is written for both beginners and experienced developers, who are curious about the internals of the Symfony2 security component. However, it is not an introduction to the Symfony framework itself. Many other good books are available to teach you how to start with Symfony[4], and the Symfony documentation on the website is excellent as well.

Even though everything is explained in great detail, some basic knowledge about using the Symfony framework is assumed. You don't need to be an expert (you will be after reading this book), but you should at least have a basic understanding and experience with writing a (simple) Symfony framework application or some experience with the standalone components.

This book is really about getting to know more about how the components function in-depth, to help you use them in your daily work as a developer. It provides valuable information about understanding the security component and allows you to use it more efficiently.

The first parts of the book will dive deep into the internals and describe in detail how the security component works. We will explain all the security configuration options from our `security.yml` file in detail. The last part of the book contains recipes for some common problems focused around security that can be used or adapted for your own needs.

Note that this book is not meant to be light reading material. In fact, it's anything but light. It is chock full of information about how the components work internally, and almost all of it is explained in great detail, which can make for dry reading. It can be used later as a reference, should you run into a security programming challenge within your symfony application and need to remember exactly how things work or need a push in the right direction in the security component.

A lot of information in this book references to the actual Symfony code. It is wise to have this code close to you when we are talking about explicit classes or methods. You should be able to follow the text better with the code as a reference.

[4] https://leanpub.com/a-year-with-symfony

About the security component

The security component is seen as the most complex component of the whole Symfony2 infrastructure. If you are writing a Symfony2 application and deal with security features, like letting users register and log in to your website, chances are you are using third party bundles like the `FOSUserBundle`. And you're right; the security component by itself doesn't provide a lot of these features directly. By using additional bundles, you save a lot of time and effort on implementing them yourself.

But, even these bundles cannot provide everything you'll need for a full sized application and sometimes diving into the actual Symfony2 security component is needed, in order to accomplish something you need for your project. Maybe you have a specific need on encoding your user passwords, maybe you need to implement some complex business-rules in order to figure out if a user is allowed certain actions, or maybe you just have a custom way on authenticating your users. All this requires a bit more understanding of the security component.

Access managers, token providers, trust deciders, firewalls and access maps, are just a few of the inner parts that make up the security component, but are lesser known terms for many developers. Not too many people outside the Symfony core developers really can tell you how it all works, as most developers will stay away from all the black magic that appears to be happening inside the component.

> Truthfully, the security component isn't all that hard to grasp. It's just a lot of small parts, which all work together and identifying and understanding those smaller parts is the key to understanding the security component.

Even if you are not using the Symfony2 framework, the security component is still an interesting component for using in your application or custom framework. Using the component in a stand-alone fashion does, however, require a bit more in-depth knowledge about how the component works internally, since you are responsible for configuring it properly for your needs. This is something that the Symfony2 framework does for you.

The security component is more than just one single Symfony component. It consists of three different parts: the actual **security component**, the Symfony2 **security bundle** and supporting **bridges**, which add additional security functionality from other Symfony2 bundles.

The component

This is the framework agnostic component that deals with security. This component actually consists of four smaller components: the actual core[5] component, the ACL component[6], the CSRF component[7] and the HTTP component[8]. All parts will be discussed separately in this book, and depending on your needs, you might not need them all.

Anything that happens in these components isn't (and shouldn't be) aware of any kind of framework, not even the Symfony2 framework. They do not know anything about your `security.yml` files, your user databases, your registration forms etc. Some components, however, will depend on other Symfony components. For instance, the HTTP security component ties the security core component to the HTTP request, by using the Symfony `HTTP foundation` and `kernel` components. These components, by themselves, can still be used without using the actual Symfony2 framework.

The bundle

The security bundle can be seen as the "glue" that makes the security component work inside the Symfony2 framework. If you were to use the security component in another framework, this part would need to be created for that framework too. Most of the work done here is actually converting the security configuration (from your `security.yml` file) into things like security listeners, firewall maps, authentication providers etc., which in turn are used within security core component.

The bridges

One of the things that makes Symfony2 so great is that most bundles don't rely on other bundles. For instance, it is possible to have users stored inside a database, which is often handled by Doctrine. But, a very important point to note is, not everybody uses a database (or even if they do, maybe not with Doctrine), so it would be unwise to have the security bundle rely on the Doctrine bundle. We don't want "Doctrine" code polluting our security bundle!

So where do we store the code that loads our users from Doctrine? Inside the Doctrine bundle? This would result in the same issue: we would pollute the Doctrine bundle with security code.

For this kind of code, Symfony uses so-called **bridges**. These bridges connects two bundles together with code that doesn't belong in either bundle directly. These bridges allow us to use additional functionality without cluttering up the main bundles and components.

If you take a look at the Doctrine bridge[9], you'll see that it contains a directory named "security". Herein lies code that forms the actual bridge between the Doctrine bundle and

[5] https://github.com/symfony/security-core
[6] https://github.com/symfony/security-acl
[7] https://github.com/symfony/security-csrf
[8] https://github.com/symfony/security-http
[9] https://github.com/symfony/symfony/tree/2.6/src/Symfony/Bridge/Doctrine

the Security bundle. This is (and should be) the **ONLY** place where the system depends on both the Doctrine and the security bundle (it could also have been a security bridge, with a "Doctrine" directory).

As said before, not everybody uses Doctrine. An alternative ORM named Propel is another common ORM, which is why there is also a propel/security bridge inside Symfony[10]. If you have your own custom database ORM system, you could even write your own bridge to connect it to the security bundle.

But, even though non-specific bundle code should be stored in bridges, if you look closely within the security bundle, you'll see that the bundle DOES rely on another bundle, namely Twig. Inside the bundle there is an extension created for Twig in order to add some additional Twig functions that can be used (the `LogoutUrlExtension` class). It would make a bit more sense to move this into the Twig bridge, which, funny enough, already has some security code. However, the security bundle CAN still function properly without Twig present, as the Twig code just won't be called (but it's a "hard" dependency nevertheless, that shouldn't probably be there).

The third bridge that is available in a standard Symfony2 framework setup is a Twig bridge. It provides a simple Twig extension, which allows you to check for user authorization within your Twig templates.

[10] https://github.com/symfony/symfony/tree/2.6/src/Symfony/Bridge/Propel1

1. The big picture

> This section defines - in a nutshell - the big picture on how the security component works. Read it through, and then re-read it again, once you have read other chapters. As soon as you become more familiar with the internal parts of the security system, the big picture becomes clearer as well.

The security component consists of two phases: the authentication phase and authorization phase. In the authentication phase,where we figure out **WHO** a user is (for which the user needs to give proof of identity, like with a password, a token or a certificate, etc.). In the authorization phase we figure out **WHAT** a user is allowed to do. This can be very general, like if a user is allowed to view a certain section of your site, or maybe a user can look at a blog post, but is not allowed to edit it, while another user can view, edit and even delete the same blog post. This part, the authorization part, can be simple and straight forward or very complex, depending on your business rules.

Each of these two phases is handled by a manager. The authentication phase is handled by the **AuthenticationManager**, and the authorization phase is handled by the **AccessDecisionManager**. Both of these managers delegate the actual work to other parts of the system. The authentication manager delegates the hard work of authentication to **authentication providers** and the access decision manager to **voters**.

The authentication phase

Starting the authentication phase is probably the hardest part of the whole security system within Symfony2. We must know where on your site authentication must take place (in `/admin` or in `/api` or even for your whole site), and we must know what kind of authentication is used (through a login form, through HTTP Basic authentication or even with OAuth2, for instance). This will be handled by both the Symfony2 bundle and the security HTTP component. Ultimately, the goal is to call the authentication manager with the correct information.

Depending on how authentication must be done, the authentication manager must be called with a **token**. This token holds information about who is trying to authenticate. For instance, it will hold the username and password that has been entered on a login form or through basic HTTP authentication, or it will hold info about the the user found in a x509 certificate or single sign-on system. This token is (normally) at this point an **unauthenticated token**. It cannot be used for authorization, but it can be converted by the authentication providers into an **authenticated token**.

The authentication manager will try to authenticate the token by calling all the configured authentication providers. Each authentication provider will decide if it's able to authenticate the token. This is normally based on the class of the token or information found within the token. This means that the authentication provider for authenticating single sign-on info will not try to authenticate tokens, which are created when logging in through a login form.

If an authentication provider is not able to authenticate the token, it will return it directly, so the authentication manager can try the next authentication provider. If an authentication provider can authenticate the token, then several other steps can happen, before the authentication takes place. For instance, some authentication providers look up the given username from the token inside a database. This is done by querying a **user provider**, which is configured in the authentication provider. These user providers can look up the username and return an actual user entity from the database. If an authentication provider must also check if a given password is valid, it will do so through a **password encoder**. This password encoder knows how the passwords of the users are encoded, which depends on the actual user entity. Thus, an authentication provider by itself will not have any knowledge about where to find users, or how to check for their passwords.

Other types of authentication providers do less work with tokens. The pre-authentication provider, for instance, does not have to check passwords, since this is already done outside of the application (through a x509 certificate, or a single sign-on system that is configured in front of the web server). In these cases, the authentication provider only needs to load the user entity from the user provider. The username for this will come from the incoming HTTP request.

Once authentication is completed, the authentication provider will create a new token, based on the token class that has been passed to the authentication provider and turns this token into an **authenticated token**. This token will hold information about who is authenticated, which is normally the user entity that has been loaded through the authentication providers.

If a user provider cannot find a user entity for the given username, or if the given credentials are incorrect for that user, the user provider will throw an exception. These exceptions can be used to display information to the user, like the username was not found or that the given password was invalid. This is normally handled by exception handlers within the framework or application.

If none of the authentication providers happen to return a token, the manager will throw an exception, meaning you probably have not configured your authentication manager properly with the correct authentication providers.

The authorization phase

Starting the authorization phase is - in contrast to the authentication phase- fairly easy. Authorization is done by the **AccessDecisionManager**. This manager will be configured with one or more voters which will do the actual authorization and will have access to the authenticated token created in the authentication phase.

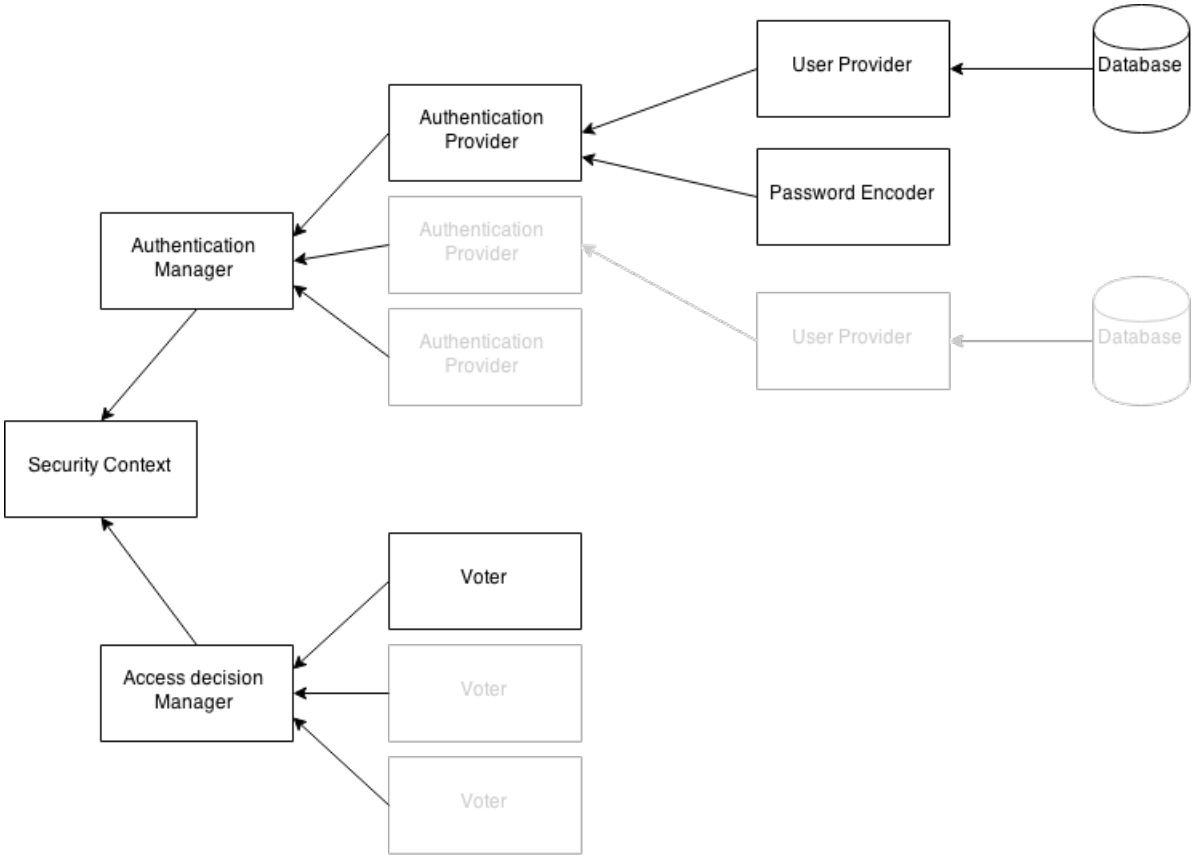
In order to authorize something, you call the `isGranted()` method from the authorization manager with a set of **attributes** and optionally an **object** parameter. The attributes are **roles** most of the

time, which are strings in the form of `ROLE_ADMIN`, `ROLE_USER` etc. If the optional object parameter is specified, voters can check authorization against the given object, provided the voters know about the object (this is used for checking ACLs, but you can use it for creating custom voters as well).

The access decision manager will call all configured voters and ask for a vote based on the authenticated token, the given attributes and the optional object. Each voter will return an answer which can either be `GRANTED`, `DENIED` or `ABSTAIN`.

The manager decides authorization based on different strategies. The `affirmative` strategy, where the manager will authorize as soon as a voter returns `GRANTED` (no other voters will be checked after the first `GRANTED`), the `consensus` strategy, where the majority of the votes decides authorization, and the last one is `unanonimous`, where **ALL** the voters must return `GRANTED` in order to authorize. You should configure the strategy depending on your own security needs.

Ultimately, the `isGranted()` method will return either true or false based on the votes and the strategy used.



Schematic of the Symfony security system

Tying into HTTP

The security system is tied into HTTP in the form of **firewalls**. Each firewall is defined as a "part" of your website. This could be a certain path (for instance, everything starting with `/admin`), it could be a hostname (like `admin.my-application.com`), or it could even be based on the IP address of the client or the HTTP method of the client.

Each firewall can be configured independently. One firewall can be configured to only accept users with a certain role, while another firewall also allows anonymous users. A firewall can define how the users will authenticate themselves. Some would authenticate through a pre-authentication system like a single sign-on system, others through a login form, OAuth2, or even a mix of different authentication systems. All these different firewall configurations are stored in the **firewall map**.

When you configure a firewall, you must also populate the authentication manager and access decision manager with the correct providers and voters. If you configure a firewall with a login form authentication, you must add the authentication provider that will authenticate this login information. If you use OAuth2 authentication, an authentication provider that handles OAuth2 must be added as well.

Each firewall configuration will also include a set of **security listeners**. These listeners all act on incoming HTTP requests. One security listener, for instance, will see if a username and password have been POSTed through a form on your login page. If so, it will create a specific token (a `UsernamePasswordToken`), sets the username and password inside the token and calls the authentication manager, which will authenticate the token. Other listeners can extract information from the HTTP request to check pre-authentication headers through a single sign-on system or X509 certificate. In turn, it creates a token from another class (`PreAuthenticationToken`) and fills the token with the username, so another authentication provider can validate it.

> Often, authentication providers, authentication tokens and security listeners will come in triplets: the token itself, the security listener to create the unauthenticated token and the authentication provider to convert the token to an authenticated token.

Not all security listeners are about authenticating users. Some of them will do other things, like dealing with logging users out, redirecting user to HTTPS, and there is even a security listener that stores the authenticated token into a session, to load it again for the next request.

By adding the correct authentication providers and voters to your managers, and by adding the correct security listeners to the firewall maps, you can control every aspect of your security firewalls.

Once we know which parts of your application uses which firewall, we have a global listener called the **firewall listener** that will try and match the current incoming HTTP request against the firewall

map. When a match has been found, it will call all the security listeners for that firewall, which will start the security features (start the authentication phase, letting a user log out, switch a user to HTTPS, etc.).

Tying into the Symfony2 framework

Using the Symfony2 security components inside the framework is really easy; you only need the firewall listener to listen to the `kernel.request` event. This is automatically configured as soon as you use the security bundle within your Symfony2 framework application.

The hardest part is to properly configure all the authentication providers, voters, security listeners, user providers, etc. Since it is a bothersome task for most developers to do this manually, the Symfony2 security bundle assists developers through the configuration. With the settings in your security configuration (`security.yml`), you are in fact creating firewalls, adding authentication providers, voters, and configuring the setting for your authentication manager and access decision manager. If, for instance, you create a firewall, where you can login through a login form (by setting options in the `form_login` section), Symfony will automatically add the `DaoAuthenticationProvider` to the authentication manager and will add the `UsernamePasswordFormAuthenticationListener` to the security listeners for that firewall. Many simple settings can trigger a whole range of complex processes internally.

In this manner, Symfony hides all the complexity that comes with using the security components and firewalls, by letting a user customize a fairly simple configuration file.

It's always possible to override some of the decisions that Symfony makes based on your configuration. With the help of the dependency injection system and configuration builder, you can override security services, and add or replace services that you need. Symfony hides the complexity, but still allows you to handle things yourself, if you want to.

The Security Core Component

2. Security context

2.1 The security context in Symfony 2.5

The main access point to communicate with the security component is the **security context** which can be found in the `Symfony\Component\Security\Core\SecurityContext` class. This is a surprisingly small class compared to the complexity that lies behind it. The class consists of just three simple methods: `getToken()`, `setToken()` and `isGranted()`.

getToken & setToken

The `getToken()` and `setToken()` methods are simple getters and setters for authentication tokens. We will discuss tokens later on, but for now they can be seen as information about **who** is authenticated and **how** somebody is authenticated. They take care of the **authentication** part of the security component, and they are used for **authorization** later on.

isGranted

This small method actually wields a lot of power. How it works really depends on how it is called and how the security component is configured. The method can be called with two arguments: with `$attributes` and the `$object`.

Attributes

The first argument of `isGranted()` should be an array, but it's not type hinted as an array. If it isn't an array, it will be automatically turned into one by `isGranted()`. So a single string will become an array of one element. These items are called the `attributes` and normally take the form of simple strings, but could be anything you like, even complex objects.

The reason why `isGranted()` is not type hinted as an array, is because it will be easier to use `isGranted` this way. Most of the time you will be passing just a single string, and it saves you the hassle of placing this string in an array first. It's just a convenience thing.

Most of the time, these attributes are strings that start with `ROLE_`, like `ROLE_ADMIN`, `ROLE_USER` etc. We will talk about how attributes are processed later. For now, we can see attributes as things, which authenticated users must have in order to be authorized.

Specifying multiple attributes will act mostly as an OR-relation. The user must have at least one of these attributes, but not necessarily ALL of them. It's possible however to specify AND-relations, where you want to check if a user has all of them. This requires a bit more configuration and is somewhat complex in setup. In most situations, however, OR-relations are enough.

Objects

The second argument of the `isGranted()` method is called `$object` and defaults to `null`. Since this argument isn't type hinted either, we could add any kind of object in it (even arrays with objects). Unlike the `$attributes`, the `$object` argument will not be converted to an array, but used as-is.

This object can be used to specify a context on what to authorize upon. For instance, somebody might be allowed to edit blog post A, but not blog post B. This could happen, when the user is the owner of blog post A, but not of blog post B. By adding the blog post(s) as the object(s) argument to `isGranted()`, the security component can use this information to check authorization on the specific blog post(s) instead of using a more global authorization. This process to authorize data access is one of the main uses for ACLs (access control lists) and one we'll cover fully later on.

Examples on calling isGranted()

```
1  if ($securityContext->isGranted('ROLE_ADMIN')) { ... }
2
3  if ($securityContext->isGranted(array('ROLE_ADMIN'))) { ... }
4
5  if ($securityContext->isGranted('ROLE_EDIT', $entity)) { ... }
6
7  if ($securityContext->isGranted('ROLE_VIEW', new FieldVote($entity, 'email'))) { ... }
8
9  if ($securityContext->isGranted(array('ROLE_VIEW', 'ROLE_ADMIN'))) { ... }
```

2.2 The security context in Symfony 2.6

In Symfony 2.6, the `securityContext` is replaced by two smaller classes: the `TokenStorage` and the `authorizationChecker`. The three methods by themselves have not changed, they are just split into these separate classes. This allows for better separation of concerns, and it also makes it easier to use these classes as dependencies. More information about this change can be found on the Symfony blog[1].

Changes in the security context

```
// Symfony 2.5
$user = $this->get('security.context')->getToken()->getUser();
if ($this->get('security.context')->isGranted('ROLE_ADMIN')) { ... }

// Symfony 2.6
$user = $this->get('security.token_storage')->getToken()->getUser();
if ($this->get('security.authorization_checker')->isGranted('ROLE_ADMIN')) { ... }
```

If you don't use the Symfony2 framework but still want to use the security component within your own application or framework, you must create a separate authentication checker and token storage class, and tie them together:

Changes in the security context

```
// Symfony 2.5
$securityContext = new SecurityContext($authenticationManager, $accessDecisionManager, true);
if ($securityContext->isGranted('ROLE_ADMIN')) { ... }
$token = $securityContext->getToken();

// Symfony 2.6
$tokenStorage = new TokenStorage();
$AuthorizationChecker = new AuthorizationChecker($tokenStorage,$authenticationManager,
                                                 $accessDecisionManager, true);
if ($AuthorizationChecker->isGranted('ROLE_ADMIN')) { ... }
$token = $tokenStorage->getToken();
```

Symfony 2.6 still provides the `SecurityContext` class in order to maintain backward compatibility with Symfony 2.5. Internally, the security context will instantiate a `tokenStorage` and `authorizationChecker` objects, and proxies the main methods to the correct objects. This `securityContext` class will be removed in Symfony 3.

For the remainder of the book when I talk about the security context, I'm referring to the authenticationChecker and/or the tokenStorage if you are using Symfony 2.6 or higher.

[1] http://symfony.com/blog/new-in-symfony-2-6-security-component-improvements

3. Tokens

Tokens are small objects, which tell us about who has authenticated and in what way. These tokens are created during the authentication phase and used during authorization. There are multiple types of tokens, but all of them must implement the `tokenInterface`. Every kind of token will manage at least the following properties:

The user
: The user entity class or just a string with a username that is authenticated.

Authentication
: This is a boolean flag, which tells us if the current token is an authenticated token or not. Only authenticated tokens can be used for authorizing users.

Credentials
: A token can hold credentials like the user's password entered from a login form. During the authentication phase, this data can be checked to see if the credentials supplied are correct.

Roles
: A list of roles for the user of this token. With some tokens, in order to authenticate a user, not only must their credentials be valid, but the user must also have at least one role.

Attributes
: Tokens can also hold additional attributes, which would not necessarily need to be part of the `tokenInterface`, but could be valuable for others components, which will use the tokens. Symfony doesn't use these attributes itself, but you can use them for your own purposes.

3.1 Token authentication

Tokens can be either in an authenticated or unauthenticated state. The `isAuthenticated()` method of a token will return either `true` or `false` depending on that status.

Authenticated tokens are tokens, which have been validated during the authentication phase. Once the authentication manager returns an authenticated token, it will be stored in the security context. It can be used to authorize the user, who is also stored in the token.

During the authentication phase, multiple security listeners will be called. Each listener will handle a small aspect of the whole security infrastructure. Some security listeners will deal with users who try to login in through a login form, other security listeners deal with remember-me cookies and others with redirecting users to HTTPS, etc.

If, for instance, you submit your username and password, then a security listener will trigger, which creates an unauthenticated token with the submitted username and password. In turn, the triggered listener will call the authentication manager, in order to convert the unauthenticated token into an authenticated token.

Authenticated and unauthenticated tokens can be very confusing for developers, who deal with the security system for the first time. Each kind of token has its own way of creating unauthenticated and authenticated tokens. Some tokens do not allow you to manually set the authentication flag of the token, but uses other means like checking the number of roles a user entity has stored in the token.

3.2 Different tokens

There are different types of tokens. The reason for this is due to the security system's need to decide which kind of authentication should happen, which is based on the type of token. For instance, when you use an HTML login form or use HTTP's basic authentication system, the token that is stored in the security context will be a `UsernamePasswordToken`. Or, if you use X509 certificates (often called SSL certificates) to authenticate, the token class will be the `PreAuthenticatedToken`. And, if you authenticate using remember-me cookies, a `RememberMeToken` will be used.

Symfony provides several different token classes in the `Symfony\Component\Security\Core\Authentication\Token` namespace. Often, each authentication mechanism has their own token class, their own security listener and own authentication provider.

AbstractToken

The `abstractToken` class is an abstract class that can be used for implementing your own tokens. You will see that many tokens within Symfony extend from this class, as it contains all kinds of generic functionality.

For instance, the constructor makes sure that all roles, which you add to the token, are based on the `Role` class or `RoleInterface`. Roles are normally stored in a database as strings in the form of `ROLE_ADMIN`, `ROLE_USER` etc, so they are automatically converted to these `role` objects by the constructor. By doing so, we have a much cleaner interface when dealing with roles in the different parts of the security component later on.

User information inside a token can vary on the type of token and the type of users you are using. A user must be an object that implements the `UserInterface`, an object that has a `__toString()` magic method to convert it to a string, or it can actually be a string, in case you don't really use user entities.

When we are finished with processing the credentials of an unauthenticated token, we almost always want to clear sensitive data inside our authenticated tokens. This way a token is safe to be stored inside a session or database (this happens automatically when you use a login form, for instance). If you store user entities from your database for instance, additional sensitive information like the encrypted password, salts, etc., are removed from the token. The `ereaseCredentials` method from the `abstractToken` class will make sure that such information is also cleared by calling the `eraseCredentials` method of the user entity (provided that the user entity implements `UserInterface`).

The `abstractToken` also provides methods for getting and setting attributes on tokens and also for serializing/unserializing the token.

UsernamePasswordToken

The `UsernamePasswordToken` is used when you try to authenticate through a login form or through HTTP basic or digest authentication methods.

The token itself extends the `AbstractToken` class and doesn't have a lot of additional features. The only strange thing about this token is, it cannot call the `setAuthenticated()` method with the argument `true` (telling the token that it is an authenticated token).

To actually create an authenticated `UsernamePasswordToken`, you must make sure you instantiate the token with **at least** one role. If you don't provide any roles, the `usernamePasswordToken` will automatically become an unauthenticated token.

Creating UsernamePasswordTokens

```
1  // Create an unauthorized token
2  $token = new UsernamePasswordToken("johndoe", "secret", "main", array());
3  var_dump($token->isAuthenticated()); // bool(false)
4
5  // Create a authorized token
6  $token = new UsernamePasswordToken("johndoe", "secret", "main", array('ROLE_ADMIN'));
7  var_dump($token->isAuthenticated()); // bool(true)
```

This is a common error Symfony developers tend to make: creating users without roles, and having trouble logging in, even when the password given is correct.

RememberMeToken

This token class is used for authenticating users based on a remember-me cookie. These cookies store (in an secure fashion) information about the user, so when the user comes back later on, the

authentication phase can fetch the user directly from the cookie and store it inside this token. This allows users to login, without needing to supply their credentials again.

A `rememberMeToken` is always instantiated as an authenticated token, even if the user stored does not have any roles.

PreAuthenticatedToken

A `PreAuthenticatedToken` is used when your authentication is done outside of the web application. This could happen if you use a single-sign-on (SSO) system placed before the web server, use x509 (SSL) certificates, etc.

AnonymousToken

Strange as it may seem, an `AnonymousToken` is a simple token that is always set to be authenticated. This causes a lot of confusion with developers, because it begs the question: How can we be authenticated, when we are anonymous and haven't even provided any credentials or when we even don't know WHO the user is?

The `AnonymousToken` is used within the Symfony2 framework as a "fallback" token and Symfony will register a fall-back security listener for it. In other words, when no other security listeners provide a token, Symfony will automatically create an `AnonymousToken` instead. This happens, for instance, when somebody visits the login or registration page for the first time. At this point, the user isn't logged in yet, but we are already inside the "secure" part of the website.

Because there is no user stored in anonymous tokens, the user stored is actually a simple string called `Anon.` with no attached roles. You can see this, when you run a Symfony2 application in development mode and look at the web debug toolbar. Here you will see a user named `Anon.` appear. If you click on this, you will see the details, the token being an authenticated token, the user having no roles and the token of the `AnonymousToken` class.

These `AnonymousToken` tokens play an important part to allow anonymous users inside your secure areas. As said before, this might be needed when you want users to be able to reach your login or registration forms.

4. A complete implementation

This chapter describes, in detail, a complete implementation of the security core component. It does not use the Symfony2 framework at all, but will show you how the security core is setup and used internally.

Note that this example ONLY uses the security core component. This means that it is not tied to HTTP and thus cannot authenticate automatically. In this example, we do this manually by supplying a manually created token.

security_example_01.php

```php
<?php

use Symfony\Component\Security\Core\Authentication\AuthenticationProviderManager;
use Symfony\Component\Security\Core\Authentication\Provider\DaoAuthenticationProvider;
use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorage;
use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;
use Symfony\Component\Security\Core\Authorization\AccessDecisionManager;
use Symfony\Component\Security\Core\Authorization\AuthorizationChecker;
use Symfony\Component\Security\Core\Encoder\PlaintextPasswordEncoder;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\User\InMemoryUserProvider;
use Symfony\Component\Security\Core\User\UserChecker;

require "vendor/autoload.php";

// Create a user provider with in-memory users
$userProvider = new InMemoryUserProvider(
    array(
        'john' => array(
            'password' => 'password',
            'roles' => array('ROLE_USER'),
        ),
        'admin' => array(
            'password' => 'secret',
            'roles' => array('ROLE_USER', 'ROLE_ADMIN'),
        ),
    )
);

$userChecker = new UserChecker();

// Our user credentials are "encoded" as plaintext.
$encoderFactory = new \Symfony\Component\Security\Core\Encoder\EncoderFactory(array(
    'Symfony\Component\Security\Core\User\User' => new PlaintextPasswordEncoder(),
```

```
));

# Create a authentication provider that can authenticate our users for our "main" context.
$providers = array(
    new DaoAuthenticationProvider($userProvider, $userChecker, 'main', $encoderFactory, true),
);

# Create the authentication manager
$authenticationManager = new AuthenticationProviderManager($providers, true);


# Create a voter that check our roles
$voters = array(
    new \Symfony\Component\Security\Core\Authorization\Voter\RoleVoter('ROLE_'),
);

# Create an access decision manager
$accessDecisionManager = new AccessDecisionManager(
                                $voters,
                                AccessDecisionManager::STRATEGY_AFFIRMATIVE,
                                false,
                                true);

# Create a token storage, an authorization checker and tie everything together
$tokenStorage = new TokenStorage();
$authorizationChecker = new AuthorizationChecker($tokenStorage,
                                            $authenticationManager,
                                            $accessDecisionManager,
                                            false);




# Instead of automatically authenticate a user, do this manually by creating
# a (unauthenticated) token for our "main" firewall context filled with our credentials.
$token = new UsernamePasswordToken('john', 'password', 'main', array());

# Try and authenticate the token
try {
    $token = $authenticationManager->authenticate($token);
} catch (AuthenticationException $e) {
    print $e->getMessage();
    exit(1);
}

# Store the (authenticated) token inside the security context
$tokenStorage->setToken($token);


# Now, check if the user has authorization to see the page.
```

```
86 if ($authorizationChecker->isGranted('ROLE_ADMIN')) {
87     print "Here is the admin page..\n";
88 } else {
89     print "Access denied\n";
90 }
```

That is a lot of code in order to accomplish something fairly trivial! But, let's take it all apart and see what each individual block of code is all about. When using the Symfony2 framework, most of the work we do here manually is handled by the framework.

First of all, in order to authenticate, we need something to authenticate against. This could be users inside a database, but Symfony also provides a way to create "in-memory" users. These users are just simple arrays with user information: the username, the password and additional roles. This way, we don't have to setup a complete database in order to use the security component.

Passwords can be stored in different ways: through different algorithms, different ways of hashing, salting etc. The `encoderFactory` is a class that creates encoders based on our user classes. So for instance, we could say that some users have a strong password (salted, and hashed with `sha512`) and other users use something simple like hashing through `md5` (don't do this, it's insecure to use plain `md5`). This example takes things a bit simpler and will be using a `PlainTextEncoder` for our users. This encoder doesn't really encode passwords, but leaves the passwords as-is.

Since we are using the in-memory user provider, our users are automatically created as a `Symfony\Component\Security\Core\User\User` class. We "link" the `plainTextEncoder` to this user class by specifying this inside the array in the constructor of the `EncoderFactory`. This factory will check what kind of class the given user is and will try to encode the given user passwords accordingly.

The `$userchecker` is nothing more than a simple class, which is called to see if a user is actually allowed to log in by checking more advanced features. For instance, a user could be locked out, or their credentials could be expired, or maybe something more complex, like a user that attempted too many failed logins. As we are using in-memory users, they do not have many advanced features, but we need to add a user checker nevertheless.

Now we can create our actual authentication provider list. We actually use one type of authentication provider, the `DaoAuthenticationProvider`. This authentication provider is based on the `UserAuthenticationProvider`, meaning it will check username/passwords. The string `main` is the **provider key** which is a key that defines the security firewall (in case you want to use the same authentication provider for different firewalls).

Once we have the authentication providers ready, we can actually create our `AuthenticationProviderManager`. The additional `true` value specifies that we want to erase the credentials in our tokens after we have authenticated successfully. It's a common precaution to make sure we don't accidentally leak sensitive information.

Next up is creating the `$accessDecisionManager`. This is much easier, as we only need a list of voters. We don't need much for our example, just a single `RoleVoter`. Note that we can specify how the

roles should be prefixed. In our case, we use `ROLE_` (also used as default by the Symfony framework), meaning that every attribute we pass in `isGranted()` starting with `ROLE_`, will be checked by this voter. The `accessDecisionManager` is created by adding the voters and configuring the strategy.

On the next few lines, we setup a `tokenStorage`, which is the class that holds our current authenticated security token and the actual `AuthorizationChecker`, from which we can actually call `isGranted` on. Older Symfony 2.5 systems might setup a `securityContext` instead.

At this point, the configuration is done, and we can start using our system. Instead of automatically authenticating, we do this manually and create a `UsernamePasswordToken` token with the credentials, a provider key and an empty array for roles, in order to make this an unauthenticated token. The provider key we use must be the same key we used in our `DaoAuthenticationProvider`.

If the authentication of the token goes well, an authenticated token will be returned and stored in the tokenStorage. If something happens, for instance, an invalid password has been used, an exception will be thrown. We display the message and exit in that case.

Once we have an authenticated token in our "token storage"/"security context", we can finally start authorizing. The `isGranted()` will authenticate the token by checking against all registered voters (we only registered one: the `roleVoter`). The roleVoter will see if the given attribute (`ROLE_ADMIN`) is present in the roles stored in the token. If so, the voter will grant access and `isGranted()` returns true.

Instead of the `PlainTextEncoder`, try and use another encoder. What must be done in order to make it work?

Try and add more voters, like the `RoleHierarchyVoter` to the access decision manager.

Try and create a user provider that stores users in a YAML or JSON configuration file and load the users from there. Make sure your user provider implements the `UserProviderInterface`, and take a look at the `InMemoryUserProvider` how it's done.

The Security HTTP Component

5. The HTTP component

Where the security core component by itself is protocol agnostic, the HTTP component is available to tie the security component to the HTTP protocol. Much of the code you find in the HTTP component library is about handling HTTP requests, creating HTTP responses or somehow dealing with HTTP in general.

Although the core component only requires PHP 5.3.3 or higher, the HTTP component has more requirements. It needs the Symfony security core component, the Symfony event dispatcher, the Symfony HTTP foundation and the Symfony HTTP kernel. More information about the dependencies can be found in the `composer.json` file[1].

[1] https://github.com/symfony/security-http/blob/2.6/composer.json

The Security CSRF Component

6. Cross Site Request Forgery (CSRF)

CSRF stands for `Cross Site Request Forgery`, which is an attack that allows third parties to take over control of a user on a website. It can be used to send unauthorized commands on behalf of a user to a website. Maybe not always as interesting on small sites, but suppose an attacker takes control of your bank account and transfers money to an account without your knowledge?

We will not describe the attack in detail. For more information about how this attack works, you can visit this wikipedia page[1].

Even though it's not only directed to forms alone, forms are the most vulnerable items for this attack. Without any precaution, every form you will create is potentially vulnerable, but fortunately, Symfony2 adds counter measurements to forms created through the Symfony Form component to prevent these attacks. It does so by adding a hidden `_token` input field to each form. This token will get checked automatically by Symfony upon submission and will throw an exception when the `_token` field is missing, or when the value in it doesn't correspond with the value that was added to the form. Attackers do not know the value of this `_token` field, as it will change on each form, and this makes a CSRF attack impossible on the form.

[1] http://en.wikipedia.org/wiki/Cross-site_request_forgery

7. The CSRF security component

The CSRF security code was originally located in the the form component[1], since this was the place where CSRF protection must be added. In Symfony 2.4, the CSRF functionality has been removed from here and placed inside this dedicated CSRF security component. The main reason for this was because both the form component and the security HTTP component use this CSRF functionality.

By separating the CSRF code into a separate component, the security HTTP component does not need to rely on the whole form component just for the CSRF functionality.

The CSRF tokens are automatically added to your forms generated by the Symfony form component. However, your login and logout actions are by default **NOT** protected by CSRF tokens, unless you explicitly turn this on in the security configuration. This can be done by configuring the `csrf_token_generator` in your `logout` configuration and the `csrf_provider` for your firewalls to enable CSRF protection for logging in. More information can be found in the security configuration chapter, where we discuss all the settings in detail.

[1] https://github.com/symfony/symfony/tree/2.3/src/Symfony/Component/Form/Extension/Csrf

The Security ACL Component

8. Permissions made easier

As mentioned earlier, dealing with bits isn't something that most PHP developers are comfortable doing (mostly because PHP doesn't really do a lot with single bits), the ACL component provides an excellent way to use bit masks in a comfortable way with the help of the `MaskBuilder`.

There is no need to use the `MaskBuilder` class. If you are comfortable enough dealing with bit masks manually, feel free to do this within your own permission maps. However, your code will be more maintainable, flexible and easier to understand and debug, when you use the `maskBuilder`.

Using the default mask builder

```
<?php

use Symfony\Component\Security\Acl\Permission\MaskBuilder;

$builder = new MaskBuilder();
$builder
    ->add('view')
    ->add('edit')
    ->add('delete')
    ->add('undelete')
;
print $builder->get()."\n";        // int(29)
print $builder->getPattern()."\n"; // ............................UDE.V

$builder->remove('delete');
print $builder->getPattern()."\n"; // ............................U.E.V
```

Inside the mask builder, there is a mask constant called `MASK_IDDQD`. This mask will set all permission bits to one and therefore will match anything. It's an inside joke about a secret cheat code in the game 'Doom' that enables 'god mode'.

8.1 Extending the mask builder

It's easy enough to extend the default mask builder to supply your own custom permissions. The default mask builder uses reflection on itself to find all permissions, so you only need to define a few constants in an extended class.

Extending the default mask builder

```
<?php

use Symfony\Component\Security\Acl\Permission\MaskBuilder;

class CustomMaskBuilder extends MaskBuilder {
    const MASK_DOWNLOAD = 1048576;  // 1 << 20;

    const CODE_DOWNLOAD = 'X';
}

$builder = new CustomMaskBuilder();
$builder
    ->add('VIEW')
    ->add('DOWNLOAD')
;
print $builder->get()."\n";         // int(1048577)
print $builder->getPattern()."\n"; // ...........X..................V
```

From PHP 5.6 onwards, it's possible to use expressions when defining constants, meaning you don't have to calculate the numbers by yourself and you can use the shift-left expression directly:

```
    const MASK_DOWNLOAD     = 1 << 8;       // # PHP 5.6 or above
```

There is a limited amount of room for permissions to be stored. Depending on your platform and database, this could be either 30 or 32 permissions. Don't use more than 30 permissions, if your application needs to stay portable.

The Security Bundle

9. The glue that binds them

As you have read in the previous chapters, the security component consists of a lot of small parts, which all fit together to make the security system as flexible as possible. This however, means that a lot of configuration must take place to construct all the separate parts with the right values and the right dependencies, in order for an application to work with the security component properly. If this type of configuration would need to do be done manually for every application created with Symfony2, quite frankly, nobody would use the security system, despite how powerful it may be.

Luckily, a lot of the configuration is done by the Symfony2 security bundle. This bundle makes it possible to use the security components (core, ACL, HTTP and CSRF) in your Symfony2 application.

As the security HTTP component already ties most of the security of other components together, not a lot needs to be done within the Symfony Security Bundle. Most of the work done in this bundle, is trying to make configuration of the components as painless as possible. And even though it might seem that they have failed by looking at your default `security.yml`, which looks large, bulky and complex, they actually did a very good job, because despite the large configuration, it's fairly easy and understandable.

10. The security configuration

The security configuration file is the place where you, as a Symfony2 developer, can control all aspects of the security component. Its task is to create a simple enough configuration, yet give you control over all the flexibility, which the security component has to offer.

The configuration is found in `app/config/security.yml`, which is imported by your main `app/config/config.yml`. Since the security configuration can become large, having it in a separate file makes it easier to maintain, but is not necessary.

The following sections will explain in detail **ALL** of the configuration settings, which can be found in the `security.yml` file. For each setting, we will provide the global setting name (prefixed with the parent sections), and the default value, which will be used, when the setting is not specified in your configuration.

The Symfony configuration settings are very dynamic and third party bundles can add more configuration settings, which aren't discussed here. We'll only address the configuration settings from the default Symfony framework setup.

It's possible that you won't be able to see all the configuration settings displayed here, most notable the 'form_login', 'x509', 'simple_preauth' and other authentication mechanisms. In order to view these, you must clear the cache **and** skip the warming up by issuing the following command: `app/console cache:clear --no-warmup`.

Default security configuration as outputted by ‘app/console config:dump security

```
# Default configuration for extension with alias: "security"
security:
    access_denied_url:    null # Example: /foo/error403
    # strategy can be: none, migrate, invalidate
    session_fixation_strategy:  migrate
    hide_user_not_found:  true
    always_authenticate_before_granting:  false
    erase_credentials:    true
    access_decision_manager:
        strategy:             affirmative
        allow_if_all_abstain:  false
        allow_if_equal_granted_denied:  true
    acl:
        # any name configured in doctrine.dbal section
        connection:           null
        cache:
            id:                   ~
            prefix:               sf2_acl_
        provider:             ~
        tables:
            class:                acl_classes
            entry:                acl_entries
            object_identity:      acl_object_identities
            object_identity_ancestors:  acl_object_identity_ancestors
            security_identity:    acl_security_identities
        voter:
            allow_if_object_identity_unavailable:  true
    encoders:
        # Examples:
        Acme\DemoBundle\Entity\User1: sha512
        Acme\DemoBundle\Entity\User2:
            algorithm:           sha512
            encode_as_base64:    true
            iterations:          5000
        # Prototype
        class:
            algorithm:            ~
            # Name of hashing algorithm for PBKDF2 (i.e. sha256, sha512, etc..)
            # See hash_algos() for a list of supported algorithms.
            hash_algorithm:       sha512
            key_length:           40
            ignore_case:          false
            encode_as_base64:     true
            iterations:           5000
            cost:                 13
            id:                   ~
    providers:            # Required
        # Examples:
        my_memory_provider:
```

```
            memory:
                users:
                    foo:
                        password:            foo
                        roles:               ROLE_USER
                    bar:
                        password:            bar
                        roles:               [ROLE_USER, ROLE_ADMIN]
        my_entity_provider:
            entity:
                class:               SecurityBundle:User
                property:            username
        # Prototype
        name:
            id:                   ~
            chain:
                providers:            []
            memory:
                users:
                    # Prototype
                    name:
                        password:             5491dc5c9c982
                        roles:                []
            entity:
                class:                ~ # Required
                property:             null
                manager_name:         null
    firewalls:            # Required
        # Prototype
        name:
            pattern:              ~
            host:                 ~
            methods:              []
            security:             true
            request_matcher:      ~
            access_denied_url:    ~
            access_denied_handler:  ~
            entry_point:          ~
            provider:             ~
            stateless:            false
            context:              ~
            logout:
                csrf_parameter:       _csrf_token
                csrf_token_generator:  ~
                csrf_token_id:        logout
                path:                 /logout
                target:               /
                success_handler:      ~
                invalidate_session:   true
                delete_cookies:
                    # Prototype
```

```
                    name:
                        path:                null
                        domain:              null
                handlers:            []
            anonymous:
                key:                 5491dc5d3ef97
            switch_user:
                provider:            ~
                parameter:           _switch_user
                role:                ROLE_ALLOWED_TO_SWITCH
            x509:
                provider:            ~
                user:                SSL_CLIENT_S_DN_Email
                credentials:         SSL_CLIENT_S_DN
            simple_preauth:
                provider:            ~
                authenticator:       ~
            form_login:
                provider:            ~
                remember_me:         true
                success_handler:     ~
                failure_handler:     ~
                check_path:          /login_check
                use_forward:         false
                require_previous_session:  true
                username_parameter:  _username
                password_parameter:  _password
                csrf_parameter:      _csrf_token
                intention:           authenticate
                post_only:           true
                always_use_default_target_path:  false
                default_target_path:  /
                login_path:          /login
                target_path_parameter:  _target_path
                use_referer:         false
                failure_path:        null
                failure_forward:     false
                failure_path_parameter:  _failure_path
                csrf_provider:       ~
            simple_form:
                provider:            ~
                remember_me:         true
                success_handler:     ~
                failure_handler:     ~
                check_path:          /login_check
                use_forward:         false
                require_previous_session:  true
                username_parameter:  _username
                password_parameter:  _password
                csrf_parameter:      _csrf_token
                intention:           authenticate
```

```
                post_only:            true
                authenticator:        ~
                always_use_default_target_path:  false
                default_target_path:  /
                login_path:           /login
                target_path_parameter:  _target_path
                use_referer:          false
                failure_path:         null
                failure_forward:      false
                failure_path_parameter:  _failure_path
                csrf_provider:        ~
            http_basic:
                provider:             ~
                realm:                'Secured Area'
            http_digest:
                provider:             ~
                realm:                'Secured Area'
                key:                  ~ # Required
            remember_me:
                key:                  ~ # Required
                token_provider:       ~
                user_providers:       []
                name:                 REMEMBERME
                lifetime:             31536000
                path:                 /
                domain:               null
                secure:               false
                httponly:             true
                always_remember_me:   false
                remember_me_parameter:  _remember_me
    access_control:
        requires_channel:     null
        # use the urldecoded format
        path:                 null # Example: ^/path to resource/
        host:                 null
        ips:                  []
        methods:              []
        allow_if:             null
        roles:                []
    role_hierarchy:
        # Prototype
        id:                   []
```

10.1 Global configuration

This section defines global security settings. They are valid for all firewalls, which you might configure.

access_denied_url: null

This is the URL to display, when access to a page has been denied. When you configure a firewall without a custom access denied url (or handler), this URL will be used instead. The value does not have to be an absolute URL but can also be a route name. It cannot be an external URL though (it cannot start with `http://`).

Note that the URL will be retrieved within a Symfony sub-request. This means that you will see the normal URL of the page you requested in your browsers address bar.

Use a custom `access_denied_handler` within your firewall configuration and use a `redirectResponse`, if you want to specifically redirect a user or want to use an external URL.

The security configuration only allows you to define a global access denied URL, but not a global access denied **handler**. If you want to use the same access denied handler in multiple firewalls, you must enter them manually in each firewall with the `firewalls.<name>.access_denied_handler` configuration setting.

session_fixation_strategy: migrate

This setting controls what will happen with session data, when a user logs in. This setting can be either `migrate`, `invalidate` or `none` and defaults to `migrate`.

`migrate`
: A new session will be created and the data from the old session will be copied into this new session. From the user perspective, all data in the session will be kept, even though a new session (and PHP session id cookie) has been created.

`invalidate`
: The session will be invalidated/destroyed. All the data, which was stored in the session, will be lost after logging in, including items like flash notifications. A complete new session will be created instead.

none Nothing will be done with regards to the session. The old session will be kept with the same PHP session ID.

You most likely want to keep this on `migrate`, so that data in your session will be kept after a user logs in.

This setting controls how session data is handled during login. The `firewalls.<name>.logout.invalidate_session` controls how sessions are handled during logging out.

hide_user_not_found: true

When entering an invalid username during the log-in process, the default exception, which will be thrown, is the `UsernameNotFoundException`. If a user enters a correct username, but an incorrect password, it will throw a `BadCredentialsException`. This however, "leaks" information about which users are present in your application. When setting `hide_user_not_found` to `true`, it will throw the same `BadCredentialsException` whenever a username is invalid, just like when you entered an invalid password. This will avoid attackers deducing if either the password is invalid, or if the user is not found in the system.

Note though that this might not always be a very good countermeasure[1].

always_authenticate_before_granting: false

This option controls if authentication must always take place during every call to `isGranted`, EVEN if the token has already been authenticated. Since authenticated tokens can be stored in sessions, they are usually not re-authenticated unless a user explicitly logs out and in again. If, for instance, you want to change the roles of a user while they are logged in, set this option to `true`. It will re-authenticate tokens on every request, and automatically sets the current roles inside the security token.

erase_credentials: true

When set to `true`, the credentials of users will be erased from the token, before it is persisted to a session. This adds a bit more security, as plain-text passwords are never stored in your sessions.

This option does nothing with the (encrypted) passwords stored in the database records. It only erases credentials from tokens in order to store them inside sessions.

[1] https://kev.inburke.com/kevin/invalid-username-or-password-useless/

10.2 Access control configuration

This section allows you to configure the access control for the different parts of your firewalls. For instance, you might have a firewall configured on your whole site, but you only want to let certain type of users in (those with `ROLE_ADMIN` to access the path `/admin`). You could theoretically create a second firewall or make sure all controllers and paths within `/admin` are secured. However, the access control configuration makes it much easier to configure this.

Each entry consists of a number of optional configuration settings, which are used to find a match on the incoming request. When a request has been found (based on the `path`, `host`, `ips` or HTTP `methods` configurations specified), it will check to see if authorization is possible by checking against the `roles` and/or the `allow_if` values.

This access control works similar to the `isGranted()` method from the security context, but you can only specify attributes, like roles, but not additional objects, as is possible with the normal `isGranted()` method.

A very common practice is to allow certain users access to small parts of your firewall, like the login or registration page or to allow access to users from a specific IP address without the need to log in. This can be easily achieved with the `IS_AUTHENTICATED_ANONYMOUSLY` attribute. See the `AuthenticatedVoter` chapter for more information about these special roles.

Examples

The example below only allows users with the `ROLE_ADMIN` in any of the `/admin` pages.

Only allow specific users on a path

```
access_control:
  # Only authenticated users with the ROLE_ADMIN role can enter the `/admin` pages.
  - { path: ^/admin, roles: ROLE_ADMIN }
```

If you specify multiple conditions, all of them must be met before authorization is checked. The following example will check for `ROLE_ADMIN`, when a user enters an `/admin` page **and** the user comes from the 127.0.0.1 IP address. If the user enters the `/admin` page, but from a different IP address, no checks against `ROLE_ADMIN` will be done, which means that all users can enter the admin pages. To make sure this works as intended, provide a "fallback" rule, which will always match and denies access as the **last** rule of the access control. Also in the example below, we provide a fallback rule, which always denies access to the `/admin` pages, provided that none of your users will have the `ROLE_NO_ACCESS` role.

Correctly allow users on certain paths, based on IP

```
access_control:
  - { path: ^/admin, roles: ROLE_ADMIN, ips: [ 127.0.0.1 ] }
  # Fallback rule
  - { path: ^/admin, roles: ROLE_NO_ACCESS }
```

More information about how the access control managers work and some more detailed examples can be found in the official Symfony2 documentation[2].

access_control.requires_channel: null

When set to `https`, you will be redirected to HTTPS, whenever you try to access the given path from HTTP. This is not really an authorization mechanism by itself, but comes in handy to make sure users can only access certain parts of your site from HTTPS only. It allows you to switch from HTTP to HTTPS, or vice versa.

If you want to make sure your website is only reachable from HTTPS only, a better way to achieve this is to configure your web server to redirect directly to HTTPS.

access_control.path: null

When provided, this will be the URL path to match against. Note that this value is used as a regular expression, so it will be common to start this value with a `^/` to indicate to match the beginning of the URL. Paths will already be url-decoded, so values like `%20` inside the URL are already converted.

Using ^/ to match beginning of a URL

```
access_control:
  # Will match /admin, /admin/foo, but not /foo/admin
  - { path: ^/admin, roles: ROLE_ADMIN }

  # Will match /admin, /admin/foo and even /foo/admin
  - { path: /admin, roles: ROLE_ADMIN }
```

[2] http://symfony.com/doc/2.0/book/security.html#understanding-how-access-control-works

access_control.host: null

When provided, the access control will check against this hostname. This value is used as a regular expression and is checked case insensitive so `FOO.COM` matches `foo.com`.

access_control.ips: []

When provided, this will be either a single IP address (either IPv4 or IPv6), a subnet range, or an array of IPs/subnets, which will be checked against. Note that Symfony understands the concepts of proxies and will try and use the clients actual IP address, instead of the requesting proxy IP address (through the `X-FORWARDED-FOR` header).

access_control.methods: []

Only allow the given HTTP methods to be authorized. Even though the most common are `GET`, `POST`, `PUT` and `DELETE`, you can use less common and even custom HTTP methods here as well. The given value can be either an array or comma separated list of methods.

Even though not needed, specify the HTTP methods in upper case for consistency. Symfony will automatically convert them to upper case if you don't.

Different ways to define HTTP methods in access controls

```
access_control:
  - { path: ^/admin, roles: ROLE_ADMIN, methods: "GET,POST" }
  - { path: ^/admin, roles: ROLE_ADMIN, methods: [ GET, PUT ] }
  - { path: ^/admin, roles: ROLE_ADMIN, methods: GET }
```

access_control.allow_if: []

The `allow_if` setting allows you to use the Symfony expression language within your matcher. The following variables are present to be used: `token`, `user`, `object`, `roles` and `request`.

Access control expression language

```
access_control:
  - { path: ^/admin, allow_if: "request.getClientIp() == '127.0.0.1' or has_role('ROLE_ADMIN')" }
```

access_control.roles: []

Only authorize access, when the user has one of these specific roles. This setting can be either an array or comma separated list of roles. This is an OR-relation. The user must have at least one of the given roles before access is granted.

10.3 Encoders configuration

There can be some confusion about the term 'algorithm' in this section. There are two types of algorithms, which we use: the **encoder algorithm**, which specifies how a password will be encoded (`digest`, `bcrypt`, `plaintext` etc), and the **hash algorithm**, which is an algorithm that **some** of the encoder algorithms use internally. The hash algorithms can be any value that can be found in the output of the PHP `hash_algos()`[3] function.

This section specifies how passwords are encoded in your user entities. These encoders are specified as an array, where each key is the class name of the user entity, which will use the specific encoder.

The remainder of the array will be the encoder configuration. Either it will be an array of configuration values, OR it will be a single argument. When it's a single argument, Symfony will assume you will be using the `digest` encoder algorithm, and the given value will be used as the hash algorithm.

With Symfony 2.5 and above, it is also possible to use 'named encoders'. In these cases, you can use any key you'd like, but your user class must implement the `EncoderAwareInterface`, which has a method called `getEncoderName()`. This method returns the name of the encoder, which will be used and must be configured in your security configuration. More information about named encoders can be found in the Symfony Cookbook[4].

Example of different encoder configurations

```
encoders:
    # Use the message digest encoder algorithm with a sha512 hash algorithm.
    Symfony\Component\Security\Core\User\User: sha512

    # This is exactly the same as the previous configuration.
    Symfony\Component\Security\Core\User\User:
        algorithm: sha512

    # Use the bcrypt encoder algorithm. It doesn't need a hash algorithm.
    Rainbow\SecurityBundle\User:
        algorithm: bcrypt
        cost: 16

    # Create a named encoder, which uses the pbkdf2 encoder algorithm.
    my_userclass_encoder:
        algorithm: pbkdf2
        hash_algorithm: sha512
        encode_as_base64: true
```

[3] http://php.net/manual/en/function.hash-algos.php

[4] http://symfony.com/doc/current/cookbook/security/named_encoders.html

```
19          iterations: 10000
20          key_length: 40
```

encoders.<name>.algorithm:

The encoding algorithm to be used. This class can be one of the following: `plaintext`, `pbkdf2`, `bcrypt`. If any other value is used, the encoder algorithm will default to `digest` and use the given value as the `hash_algorithm`.

The following encoders are available:

`plaintext`
: A simple encoder that doesn't actually encode the password, but leaves it as plain text. Do not use this encoder in production!

`pbkdf2`
: Encodes passwords conforming to the `pbkdf2` standard.

`bcrypt`
: This encoder uses the PHP password API to encode passwords.

`digest`
: Hashes passwords by using the PHP `hash()` method.

encoders.<name>.hash_algorithm: sha512

The hashing algorithm used in the specified encoding algorithm. It can be one of the algorithms, which is outputted by the PHP `hash_algos()` function. This setting is only used in `pbkdf2` and in the `digest` encoder.

encoders.<name>.key_length: 40

The length of the key, in bits, which is generated by the encoder. This setting is only used in the `pbkdf2` encoder.

encoders.<name>.ignore_case: false

Ignore the case of the password, while validating the password. when this is set to `true`. This means

that `password` will match `PASSWORD`. This setting is only used in the `plaintext` encoder.

encoders.<name>.encode_as_base64: true

Returns the encoded hash as a base64 encoded string instead of a binary string. This might make it easier to store the password in a database. This setting is only used in the `digest` and `pbkdf2` encoders.

encoders.<name>.iterations: 5000

Defines how many times the encoder will be iterated, before returning the encoded password. The more iterations used, the longer it takes to encode a password, which can help against brute-force attacks. This setting is only used in the `digest` and the `pbkdf2` encoders. The `bcrypt` encoder has a similar system built in with the `cost` configuration setting (see below).

encoders.<name>.cost: 13

Just like `encoders.<name>.iterations`, this setting defines how long it takes to encode a password. The higher the number, the longer it takes. Note that this number is exponential and should be between 4 and 31. This configuration setting is only used by the `bcrypt` encoder. Other encoders can use the `iterations` configuration setting instead.

encoders.<name>.id: ~

If you want to use a custom encoder, you can set this value to your custom encoder service. This service must implement the `PasswordEncoderInterface` class. Note that this encoder cannot use any of the configuration settings like `cost`, `iteration`, `hash_algorithm` etc.

Using a custom password encoder service

```
encoders:
    # Users will be encoded with the digest encoder and sha512 algorithm
    Rainbow\SecurityBundle\Entity\User: sha512

    # "company users" are encoded with a custom encoder service
    Rainbow\SecurityBundle\Entity\CompanyUser:
        id: rainbow.securitybundle.companyuser_encoder.service
```

A custom encoder service

```php
<?php
namespace Rainbow\SecurityBundle\Service\Encoder;

use Symfony\Component\Security\Core\Util\StringUtils;
use Symfony\Component\Security\Core\Encoder\PasswordEncoderInterface;

class CompanyUserEncoder implements PasswordEncoderInterface
{
    public function encodePassword($raw, $salt)
    {
        // Your custom encoding scheme here. Don't use this example in production!
        $encrypted = str_rot13($salt.$raw);

        return $encrypted;
    }

    public function isPasswordValid($encoded, $raw, $salt)
    {
        // Use the StringUtils class for protection against timing attacks
        return StringUtils::equals($this->encodePassword($raw, $salt), $encoded);
    }
}
```

The Security Bridges

There are three different bridges available that tie into the security component / bundle. These bridges can be found in the `Symfony\Bridges` directory. Each bridge defines additional features for other bundles, including the security bundle. This part of the book describes the security features of these bundles in detail.

11. The Doctrine bridge

The Doctrine bridge provides two additional features to the security component: A `remember-me` function, which allows us to let users authenticate automatically, when a certain cookie has been set. And an entity user provider, which allows users to be loaded and stored from a database through Doctrine.

11.1 DoctrineTokenProvider

The "normal" remember-me functionality of the security component stores and retrieves all information inside a cookie, which is sent to the user's browser. As this cookie is the key for logging into the system without any authentication (since having the remember-me cookie IS the authentication), the data inside the cookie must be encrypted, so it cannot be tampered with.

The Doctrine bridge provides a `DoctrineTokenProvider` that doesn't rely solely on the content of the cookie, but uses that cookie to load and save information into a database. Based on that data, it creates a `PersistentToken`. It behaves this way more or less like a PHP session ID. The only value that is stored onto the client side is the identifier of the record in the database. Theoretically, this improves security as information is stored server side instead of client side, however, on the server side, this information is NOT stored encrypted.

This tokenProvider does have some drawbacks. It only saves the data inside a table called `rememberme_token`. There is no way to change the name of this table. The pull request for the `DoctrineTokenProvider`[1] shows there is initial work done on creating a schema-system (just like the ACL has inside the security bundle), but it wasn't finished nor merged. For most users, this isn't really an issue, unless you already have a table named `rememberme_token` (highly unlikely) or when you have a good reason to use a different table-name (when you need to prefix your tables for instance).

11.2 EntityUserProvider

A more useful feature, which this bridge provides, is the `EntityUserProvider`. This provider allows us to load users by querying a Doctrine repository. A goal of even the simplest applications would be to store this data inside a database. Using this provider makes it much easier to do so with the help of Doctrine.

The user provider is a service that relies on two dependencies: a Doctrine manager, as this will be used for querying the user. And, a "User class", which is the class name of the entity that will

[1] https://github.com/symfony/symfony/pull/6057#issuecomment-16734538

be loaded from the provided Doctrine manager (for instance, `Rainbow\SecurityBundle\User`, if you have named your entity like this). The provider will collect information from this class directly from the Doctrine manager, so it can figure out things like primary keys etc.

There are also two optional arguments, which can be injected: a `property` and a `managerName`. The `managerName` is just the name of the entity manager we are using. If not given, it will use the default entity manager from your Doctrine configuration, but you can use this to load the users from a different entity manager, in case you have multiple databases or connections.

The `property` argument is quite interesting. When the repository of the user entities we use in this entity provider implements the `UserProviderInterface`, the repository will have a `loadUserByName` and `refreshUser` method. These methods are called by the user provider, so the actual entity repository class will handle the loading and refreshing of users.

When the user repository does not implement this interface (although it would be trivial to make this happen I guess), you can use the `property` argument to tell the user provider which of the fields of the entity maps to the actual username. If your entity has a field named "username", you should set the `property` to `username`. If you use email addresses to identify users, you could set this to `email` etc. If the property has been set, this user provider will directly search users based on `property` within Doctrine. To be a bit more exact: it will do a `findByOne` on the `property` on the repository of your user entities. So basically, instead of the repository class doing the actual work, the entity user provider can do it for you. However, it's much nicer to keep this kind of code within your repository classes.

You'll notice that when refreshing the user through `refreshUser()`, it will not use the `property` to find the user, but uses the primary key instead. This means that you **MUST** have a primary key on your entities (and it's very unlikely that you wouldn't have one). The reason for this is nicely commented in the actual code: all data (including the property you used to load the user), might change. The user **could** have changed their username, email address or any another property. The only thing that cannot change is the primary key. This is why we are loading users by its property, but refresh them with their primary key.

12. The Propel1 bridge

12.1 EntityUserProvider

The Propel1 bridge is a very simple bridge. It consists of only an `EntityUserProvider`, but not the `remember-me` functionality, as we saw in the Doctrine bridge. It would be easy enough to add this remember-me functionality to Propel as well, but it seems nobody really has the need for it.

The Propel1 bridge code for the `EntityUserProvider` seems a bit less advanced than the code from the Doctrine bridge. It has the option to use the `property` argument, but loading of a user without a `property` argument will automatically load the user through `filterByUsername`. To be honest, I am not an expert on the topic of Propel, so I am not certain how to add interfaces to Propel's entities, since there are no `UserProviderInterface` checks.

The `refreshUser` method works pretty much the same way as in the Doctrine bundle: it uses the `primaryKey` for refreshing instead of the actual property. Again, this is due to the fact that data, including the actual property, might have changed.

13. The Twig bridge

13.1 SecurityExtension

The Twig bridge found in `Symfony\Bridge\Twig\Extension\SecurityExtension` does nothing more than add a new Twig function called `is_granted()`. This function allows us to check authorization within Twig directly, just like the normal `isGranted()` you can use within your controllers.

Usage of the is_granted() Twig extension

```
{% if is_granted('ROLE_ADMIN') %}
  <a href="{{ path('item_delete') }}">Delete item</a>
{% endif %}
```

Using the `is_granted()` Twig function only takes care of displaying data. You must make sure that you actually secure the underlying action as well by calling `isGranted` within your controller actions, otherwise it might be possible for somebody to reconstruct the link manually in the browser and gain access to unauthorized parts of your application.

There is a small difference between this `is_granted()` function and the normal `isGranted()` method from the security context. When you want to check against an ACL for an object field in the regular `isGranted()` method, you must create a `FieldVote` object that you pass along as the 'object'. This `FieldVote` object consists of your entity and a field name which you want to check.

Since you cannot instantiate objects directly from Twig, the `is_granted()` Twig function adds a third argument: the actual field. When set, the twig extension will automatically use the `object` and `field` arguments and creates a `FieldVote` object for you, which is internally passed to `isGranted()`. This allows you to easily use field ACLs within Twig as well:

Usage of the is_granted() Twig extension with field ACL check

```
{% if is_granted('VIEW', comment, 'ip_address') %}
  The user's IP address: {{ comment.ip_address }}
{% endif %}
```

If you provide your own custom code that bridges between the security bundle and your own bundles, think about placing this code within a separate bridge instead of your own bundle. This makes your own bundle smaller, and keeps your code flexible.

Recipes