# Blog: PHP and Symfony – Part III

This document defines a complete walkthrough of creating a **Blog** application with the [Symfony](https://symfony.com/) Framework, from setting up the framework through [authentication](http://symfony.com/doc/current/security.html) module, ending up with creating a **CRUD** around [Doctrine](http://www.doctrine-project.org/) entities.

Make sure you have installed [XAMPP](https://www.apachefriends.org/download.html), [HeidiSQL](http://www.heidisql.com/download.php) and added [PHP root folder to the path environment variable](http://php.net/manual/en/faq.installation.php#faq.installation.addtopath).

# Implementing Roles

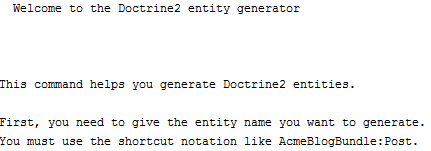
Before we start working with roles we should tell you how can you use them. **Roles are used** to create **user authorization**. For example, if you want a given page from your blog to be **only accessed by admins**, then you should create **admin role**. If you want only the **authors of articles** to be able to **edit** them, you can also do that with roles. Roles are **the natural way to filter** **content** based on **user privileges**.

## Creating the Role Entity

Let’s start by creating the role entity. In the **terminal/CMD** write the following:

|  |
| --- |
| php bin/console doctrine:generate:entity |

You should see this result:



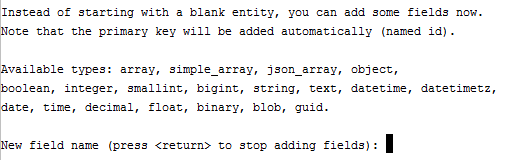
Now we need to choose **appropriate name for our entity**. Use the following name:

|  |
| --- |
| SoftUniBlogBundle:Role |

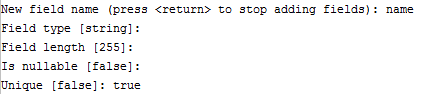
The result should be the following:



Just press ‘**Enter**’. Now we need to **define the properties** for our entity. You should see this:



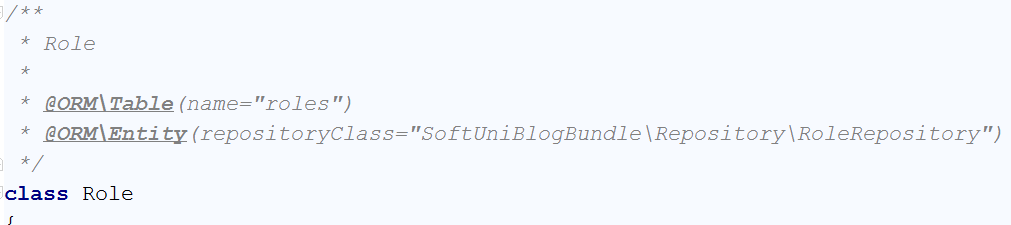
Our **roles** will have **only one column** and that is the **name** of the role:



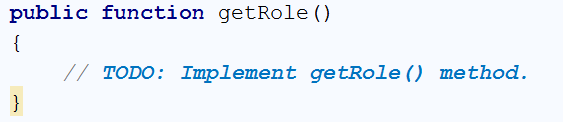
That’s all, just press ‘**Enter**’ again. In the “**Entity**” folder we should have the Role entity.

## Modifying the Role Entity

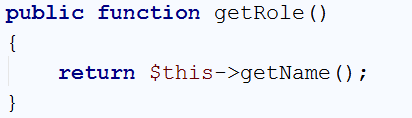
Open the Role entity. First thing to do – pluralize the table name:



Add empty getRole() function to the bottom of our class:

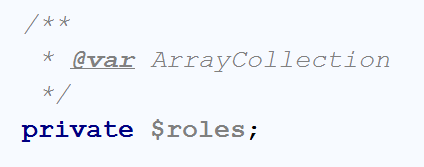


We will just tell it to **return the role name**:

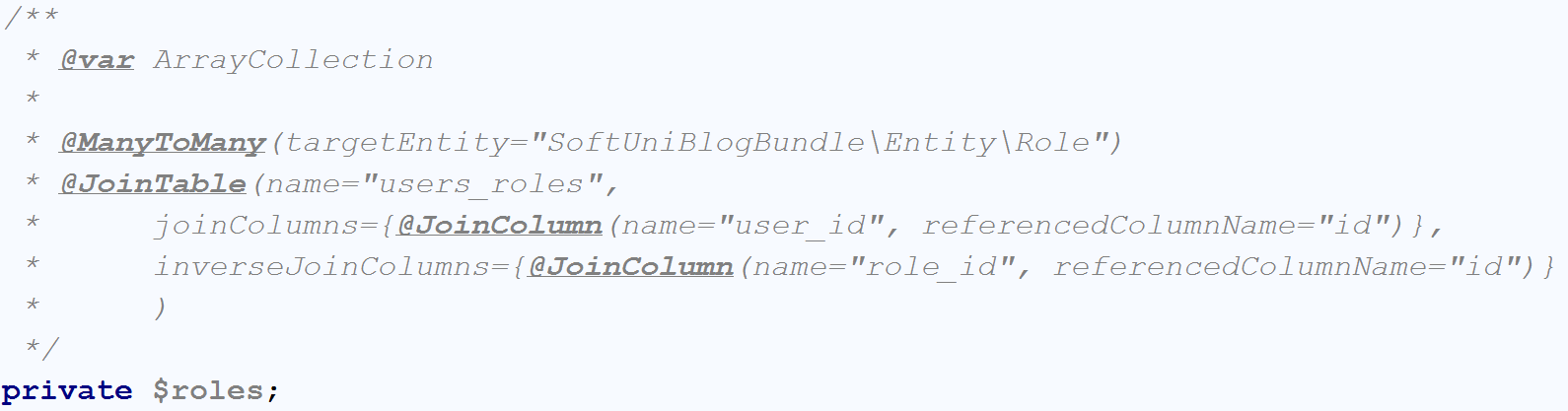


## Create the User-Role Relationship

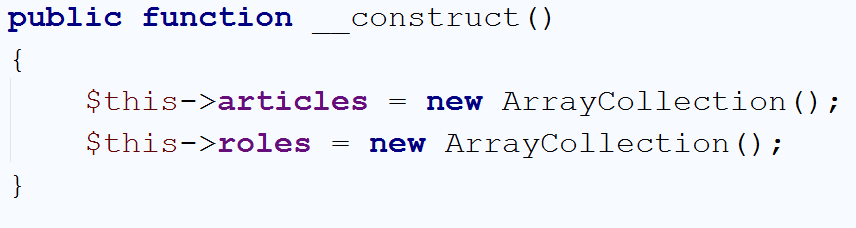
We need to tell the database how are we going to use the roles and users together. Let's create a **new** **private field** in the User entity called "roles":



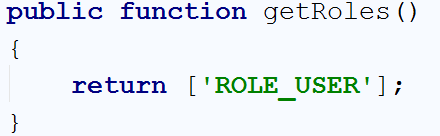
That is not enough. We need to tell Doctrine that our relation is going to be [Many-To-Many](http://www.tomjewett.com/dbdesign/dbdesign.php?page=manymany.php). We're going to do that, by writing the following annotations:



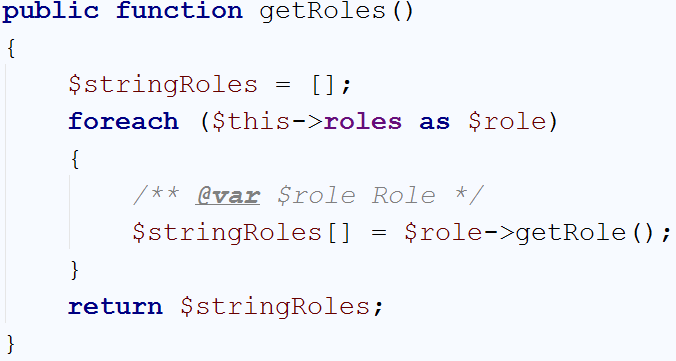
The "@ManyToMany" annotation tells Doctrine that **many users** will have **many roles**. The "@JoinTable" annotation will **create a table** that will **keep our relations**. While we're still in the User entity, let’s **initialize** the $roles ArrayCollection in our constructor:



Now you should find the getRoles() function that looks like this:

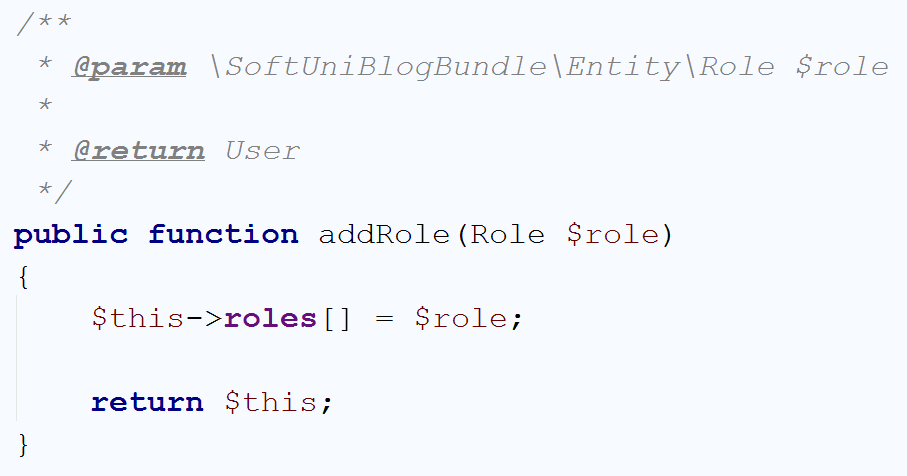


Right now, **every user** is having the "ROLE\_USER" role, which is something we will change. We will **change the function** to **return all of the roles** the current **user has in the database**:



That way it will get the **private field** we've created earlier and **take each role's name**.

One last thing for now is to **create a function** that will **add new role** to the **current user roles**:



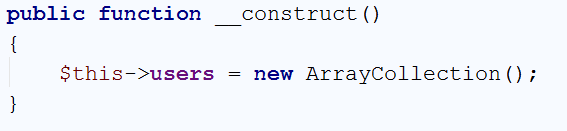
Let's create the relationship from the Role side.

## Create the Role-User Relationship

In the Role entity create the following field:

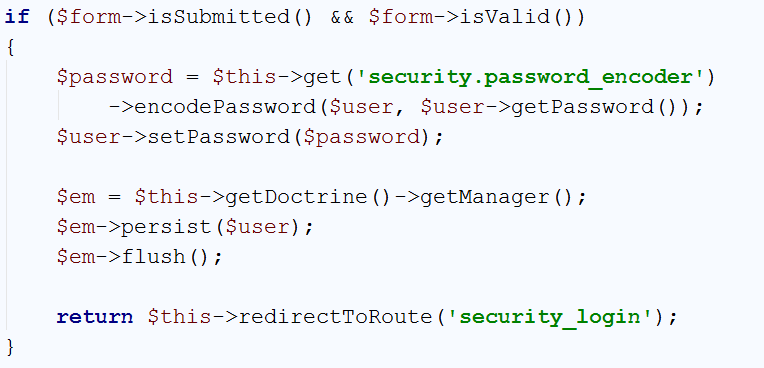


It tells Doctrine that **many roles** will have **many users** and that the relation is **mapped by** roles field in the User entity. Let's create a constructor that will initialize the ArrayCollection:

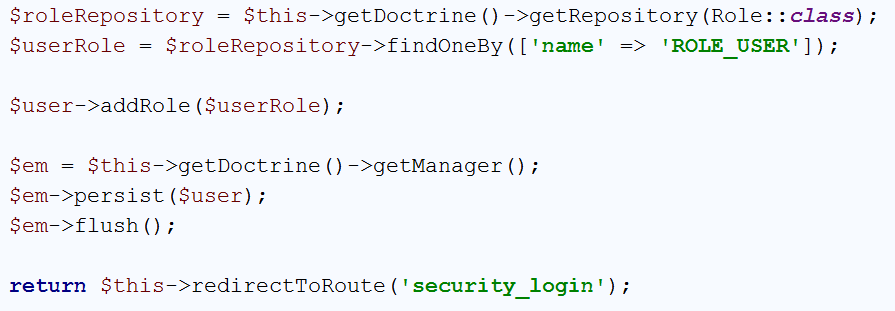


## Edit the User Registration

We've changed how the users are being are initialized and because of that we need to change the registerAction() function in our UserController. Right now, if the user registration form **is valid** and **submitted** we do the following:



We need to add few lines of **code before** we **save** the **user** to the **database**:



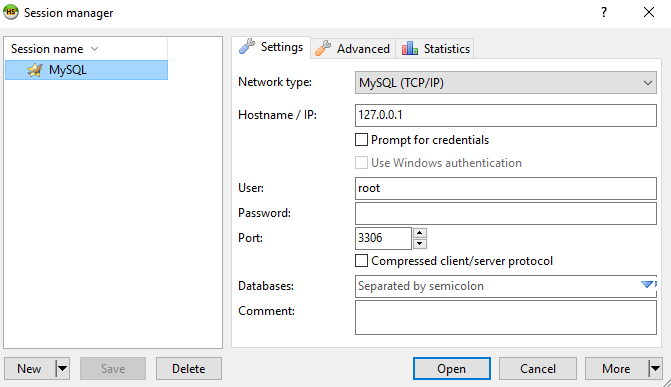
First we get the RoleRepository and then **we get the role** with **name** "ROLE\_USER". Finally, we assign it to the user. It is important that you have the **following using imported on top of your file**:

|  |
| --- |
| **use** SoftUniBlogBundle\Entity\Role; |

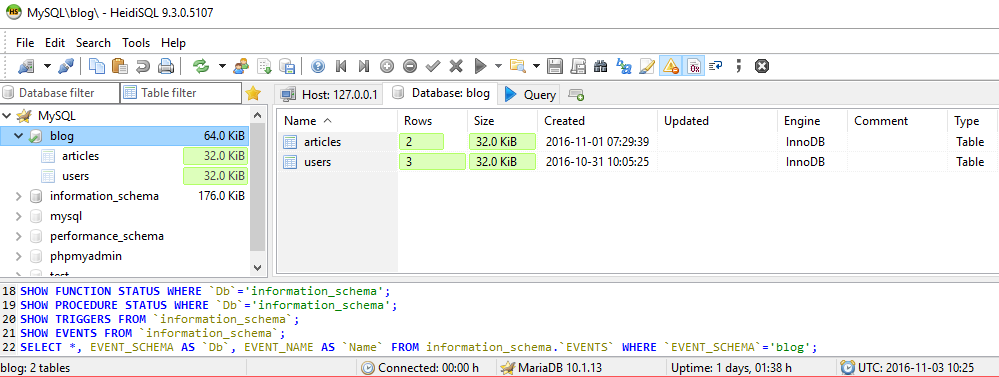
If you have **any other** "use" statement ending with "\Role" **delete** it!

## Update the Database Schema

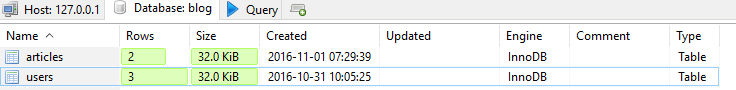
**Before we update the database**, we will **delete all existing tables and data**. We are doing that, because it **will be easier** than **modifying the DB manually**. Open **HeidiSQL**. In the first screen that you see, just click "**Open**":



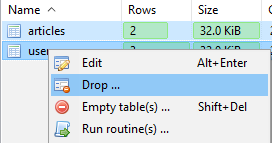
You should see something like this:



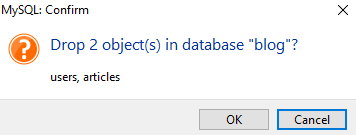
On the **left side** of the **window** you should see the "**blog**" **database**. Click on it. Then in the main part of the window you should see this:



**Select all tables** with your mouse and **right click**. Select the option "Drop…":



You will be asked if you are sure. Click **OK**:



That **will delete our tables and all of the data stored in them**. Now that we've done that, we **can update the database schema using Doctrine**.

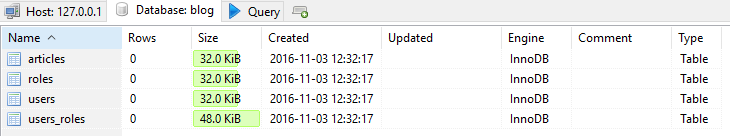
Like we've done before, we **will update the database schema using terminal/CMD**. We will force the update to make sure everything works fine:

|  |
| --- |
| php bin/console doctrine:schema:update --force |

The result of this command should be the following:



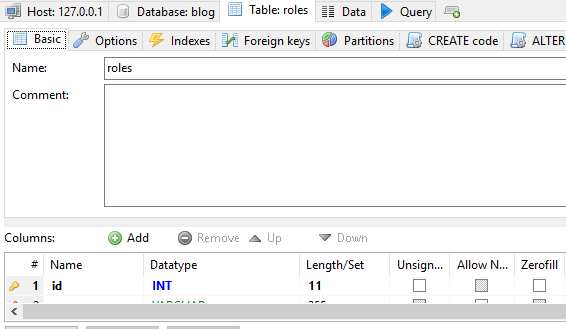
Back in **HeidiSQL** click on the **blog database** and then press "**F5**". In the main part of your window, you should see this:



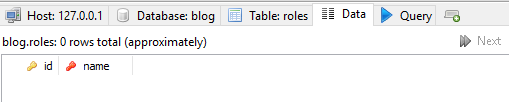
We have successfully created the new tables. Let's start using our roles now.

## Create Roles

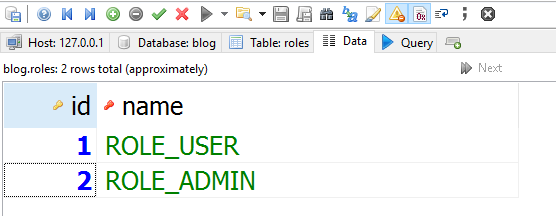
Still in **HeidiSQL**, **double-click** on the roles table. The main screen should change to this:



In the navigation, you will see "**Data**" tab. Open it and you should see this:



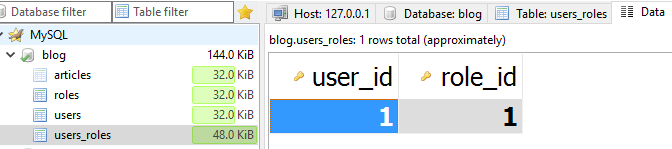
Our database is empty. Click on the "**green plus**" in the **main toolbar**  and you will **be able to enter data** in a **new row**. **Create two roles**, one called "**ROLE\_USER**" and one "**ROLE\_ADMIN**". It should look like that:



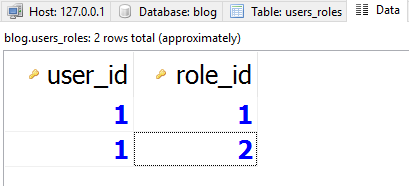
Now we can create new user.

## Register New Admin

Let’s register new user, but **don't login** yet. **After you've created the user**, open the **data tab** for the "users\_roles" table in our database:



Add a **new row**, like you've done earlier and **enter the following data**:



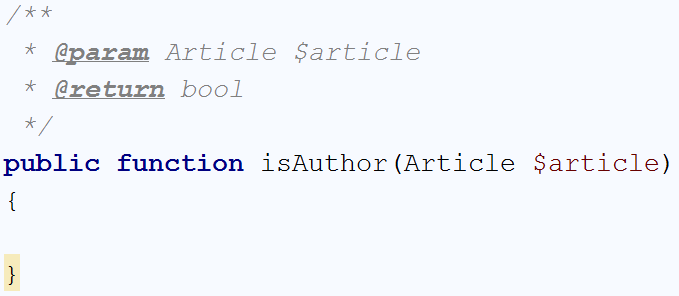
This should make our user a special one – '**admin**'. We will see how we will use that in the next chapter.

# Creating Validations

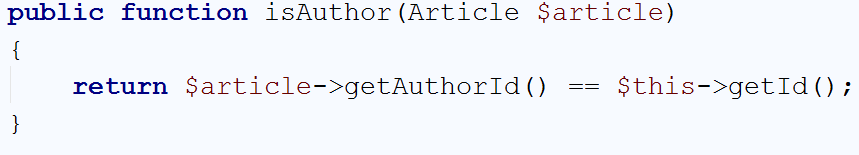
We've already talked about how **every user** is able to **edit** and **delete** **all** **articles**. That shouldn't work like that. Before we start, **create few users** (not admins) and **create at least two articles** with **two users**.

## Create User Helper Functions

Before we start creating **validations**, we should make our life easier. Open the User entity and let's **create the following function**:

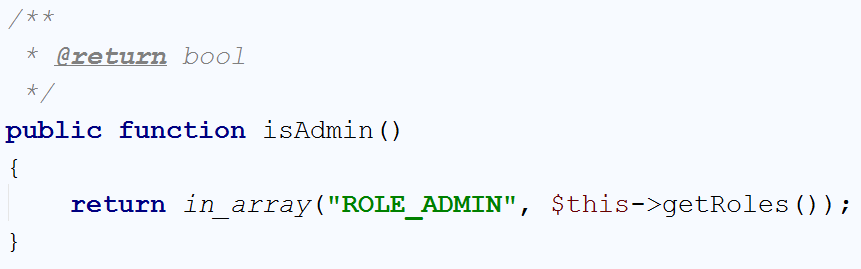


It will check if the **currently logged-in user** **is** the **author** of a **given article**. You should write the following code:



That's all we need to check for that function. If the authorId from the **article** matches the **current user** Id, we will return true.

We need one more **function** that will tell us if the currently logged-in **user** **is admin**:

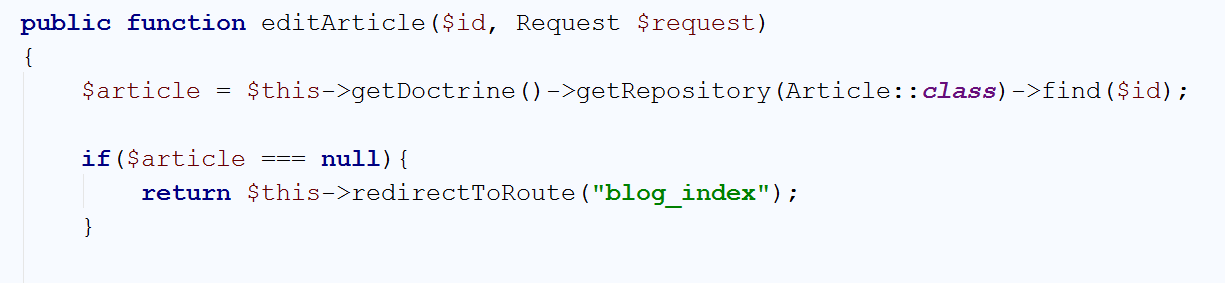


The in\_array() function in PHP is the Contains() method in C#. We check if the roles ArrayCollection **contains** "ROLE\_ADMIN" and if it does, we return true.

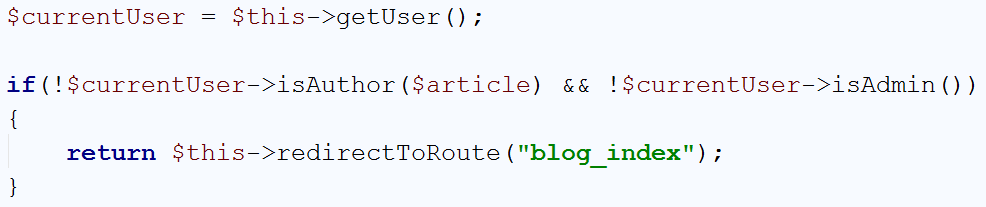
We can now start making **validations**.

## Validating the Edit Article Function

Find the editArticle() function in the ArticleController. At the moment, it should start like that:



The code we are **going** **to** **write now** will be **right after** the existing if statement. Our code should take the **current user** and **check** if he **is the author** of the **article**. If he **isn't**,we will **check** if he **is admin**. If **both** are false, we will **redirect him to the home page** of the blog. Write the following code:

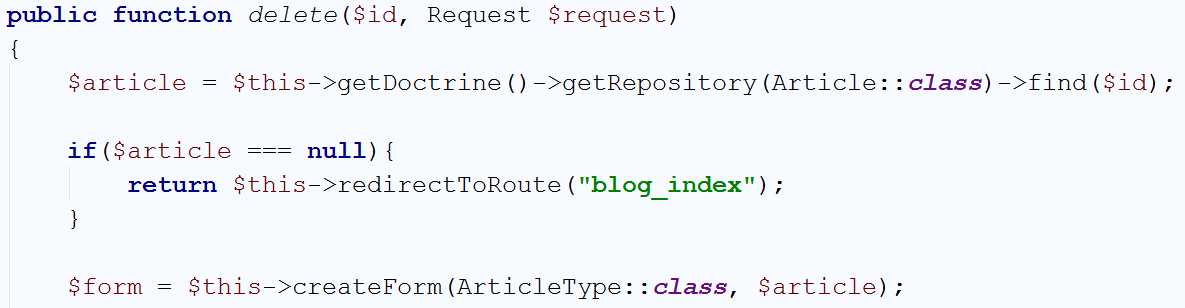


If you've created 2-3 users with 2-3 articles each, you can test if it works. Simply **try to edit an artic**le that is **authored** by another user. If you are **logged with the admin user** we've created earlier, you should be able to **edit all posts**. If you are logged with a **regular user**, you should be **able to only edit your own posts**.

We need to create the same check for the delete() action.

## Validating the Delete Article Function

The delete() function in the ArticleController should begin like this:



Just like in the edit() function, we need to write the **exact same code after the** null check **statement**.

In the end, it should look like this:



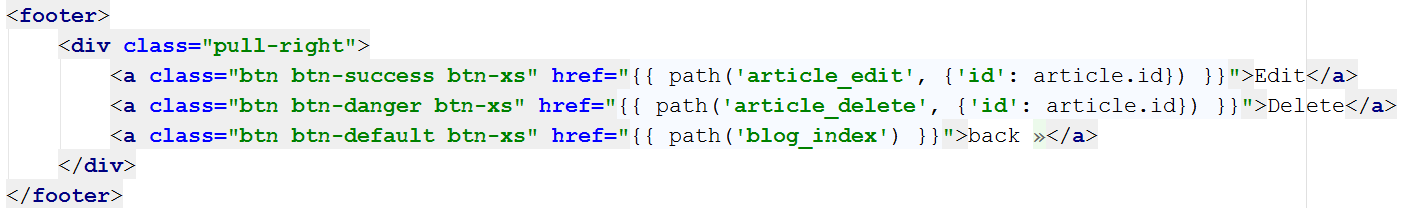
Test it and see if it works. If everything is working as it should, **the admins will be able to delete all posts**, and only the **authors will be able to delete their own posts**.

We have another problem. Do we need to **allow all users to see** the "**Edit**" and "**Delete**" buttons? **No**, they should be **visible only to the authors and admins**. We will deal with that in the next chapter.

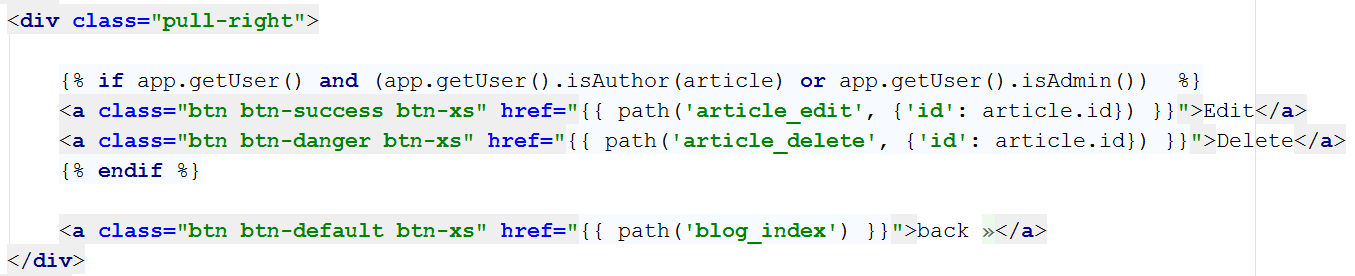
# Creating View Helper Validations

## Create View Helper Validation

Currently, **every user** that opens the **single article view**, will be able to see the "**Edit**" and "**Delete**" buttons. In order to fix that, we will open the article.html.twig template. You should remember the buttons we've added earlier:



In order to **make them invisible** for users that are **not the article author or admins**, we will create an if statement with **twig**:



Let's split that check in two parts. The first part is:



This part will secure that the other check will be **executed** **only if there is a logged-in user**. Without it, if the user is **guest** (not logged-in), **Twig** will **throw** Runtime\_Error. The second part of the check is:



As you've probably figured it out by now, this will check **if the user is author** of the article **or admin**.