# Blog: PHP and Symfony – Part II

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).

# Creating Articles

## Start MySQL

## Open the Project (Only if you have done 0.)

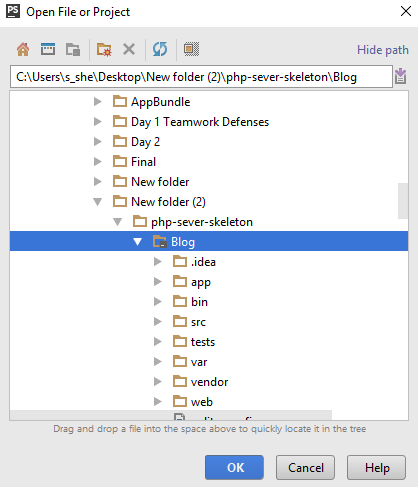
**Skip this step if you have gone through the above III chapters.**

If you are still reading:

For this step, we will open the project with **PhpStorm** or **IntelliJ** Idea. Starting from the home screen, click on “**Open**”:



Locate the skeleton folder that we gave to you and select the “**Blog**” **folder** from the extracted folder (e.g. **c:\project\Blog**):



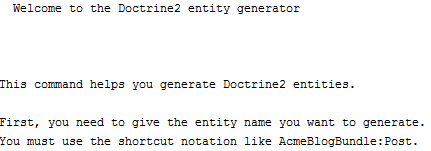
After you click “**OK**” the project should start loading and indexing. After a few seconds/minutes depending on your pc, you will be able to work with the project.

## Create the Article Entity

Open Terminal or Command Prompt (CMD) in the blog project root folder. Let’s model our articles. That means that we are going to create the defining properties of an article. To do that, we need to generate a [Doctrine Entity](http://docs.doctrine-project.org/en/latest/reference/working-with-objects.html). Our entity will describe what are we going to store in our database. The following command will **start entity generator wizard**:

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

You should see this result:



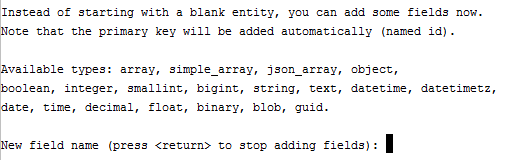
Now we need to choose **appropriate name for our entity**. Use the following name:

|  |
| --- |
| SoftUniBlogBundle:Article |

The result should be the following:



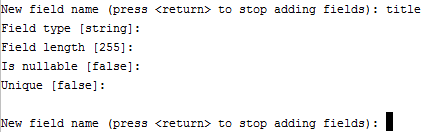
Just press ‘**Enter**’. Now we need to **define the properties** for our entity. You should see this:



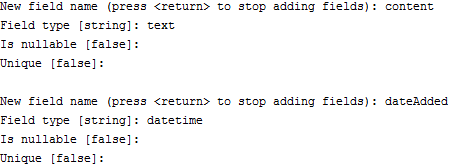
Our first field will be the “Title” of our article. Just write “**title**” and press ‘**Enter**’. You should see this:



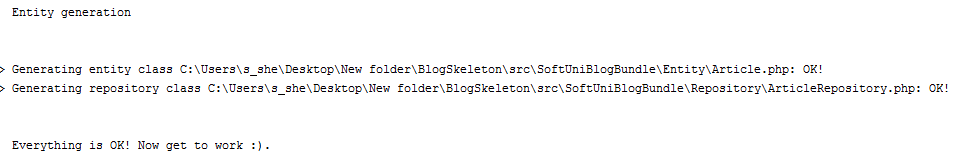
Press ‘**Enter**’. You should see “**Field length [255]**”. Press ‘**Enter**’ again. You will be asked if you want to make the field **nullable**. Press ‘**Enter**’. Finally, you will be asked to make your field **unique**. Just press ‘**Enter**’ one more time. Now you should see this:



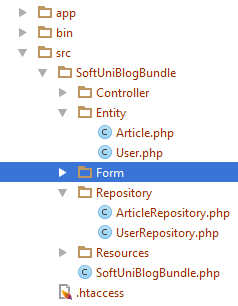
Similar to this, we should create 2 more fields for the “**content**” and “**date**”. Here is how we create them:



Finally, press ‘**Enter**’ one more time to close the wizard. You should see this:



Let’s see the project in **PhpStorm** (IntelliJ Idea):



Okay, Doctrine has created “**Article**” entity and “ArticleRepository”, which is a special type of class. Its job is to manage our data and simplify our work with the database.

## Add Summary to the Article Entity

Let’s go into the “**Article**” entity that Doctrine created in the previous step. It should contain all of the fields, that we created using the terminal, plus one **special** “**id**” **field**. It is the [primary key](http://www.mysqltutorial.org/mysql-primary-key/) for our table. On top of our entity we should see something that looks like a comment:

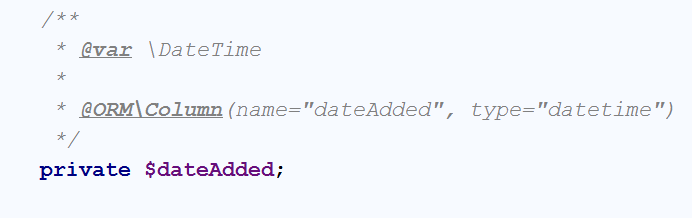


However, this is not just a comment. It is an annotation. More specifically, it is a [Doctrine Annotation](http://docs.doctrine-project.org/projects/doctrine-common/en/latest/reference/annotations.html). It tells Doctrine how are the tables and fields are going to be called in the database. At first glance, we see the annotation

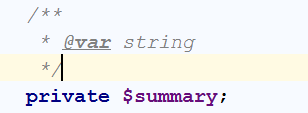
|  |
| --- |
| \* @ORM\Table(name=”article”) |

This defines the name of our table in the database. The names of the tables in the database should be pluralized. For that reason, rename the table to “**articles**”.

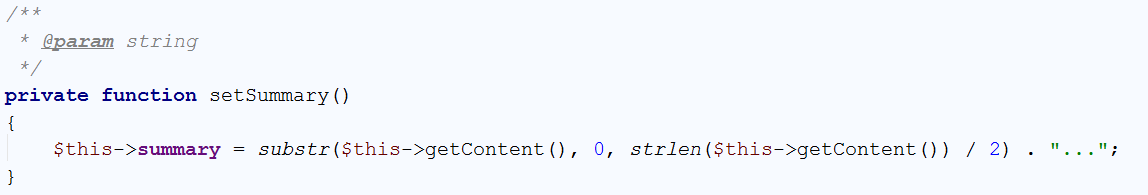
Now we need to create some fields, that will not get saved into the database. Find the “$dateAdded” field. You should see something like this:



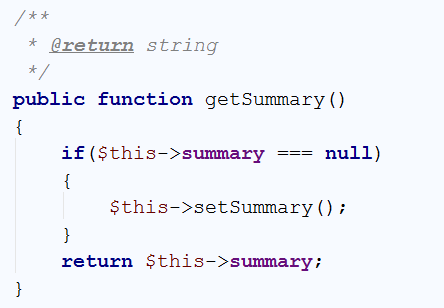
Below that, first add a new private field called “summary”. It will hold the short summary of the article:



Then we need to create [Mutator and Accessor](http://www.refulz.com/mutator-and-accessor-methods-in-php/) (Getter and Setter) methods for the summary. Let’s first start with the **mutator**. Its job is to set the value of summary to half of the article content. The code should look like this:



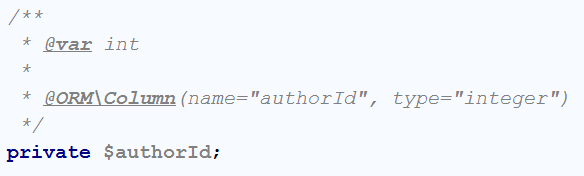
Now we should create the **accessor**. It should simply **return the saved value** in our **summary** variable. However, if summary is empty, it should **call the** **mutator to set the value**:



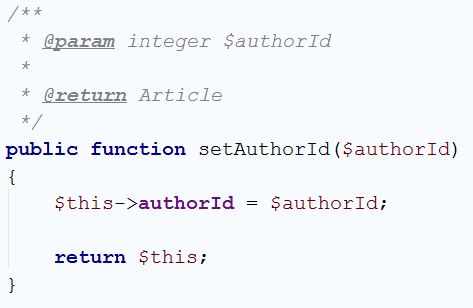
We’re done with the summary variable, but we still have more variable to implement.

## Create a Relationship between the User and the Article

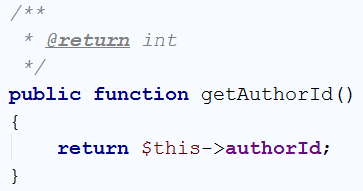
We’ve come to the part where we must connect each user with his articles. To do that, we will create 2 more field in the “**Article**” entity. Just below the private summary field, that we’ve created in the previous step, create new private field called “authorId”. Using that field, each article will know who is its author:



You have probably noticed that we’re going to **save this field in the table** using the @ORM annotation. This will **create a column in the table**, which will keep integer, representing a user. Similarly, to the summary, we need to create **getter and setter** methods for this field. Again, we’re starting with the mutator:



One special thing to note here is that the **mutator** actually returns the object, that it changes. Now simply **create the accessor**:

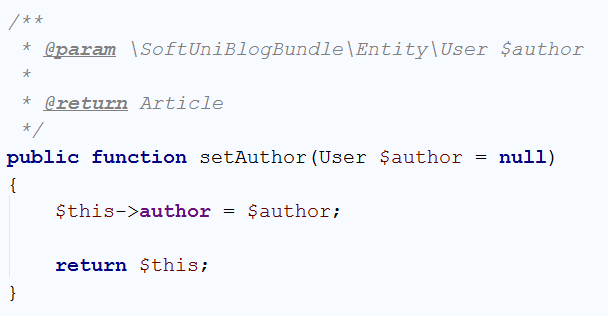


We’re done with the authorId, but the **connection** is **not** **ready** yet. In order for our article to actually have an author, we need to declare a private field of type “**User**”:

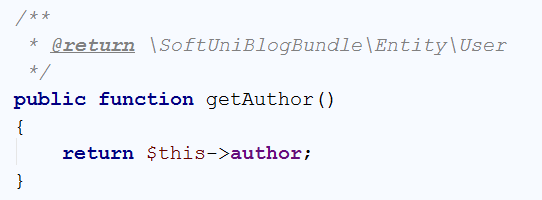


More new stuff! We’re using 2 new annotations. The first one is the “ManyToOne” annotation. Many to one relationship represents [OneToMany](https://en.wikipedia.org/wiki/One-to-many_(data_model)) relationship from the side of the “many”. In our case, we will use “one to many relationship” to tell the program that one user will have many posts. Because we are working with the Article entity, we are telling Doctrine that **many of our articles** will correspond **to one user**. The “inversedBy” parameters tells Doctrine that in the User entity we will have a private field called “**articles**”, which will keep all of the articles of one user. The other annotation is “JoinColumn”, which tells Doctrine how are we going to connect the fields in our entities. Our annotation tells Doctrine that the **field “**authourId**” in our article entity will correspond to the “id” field from the** Userentity.

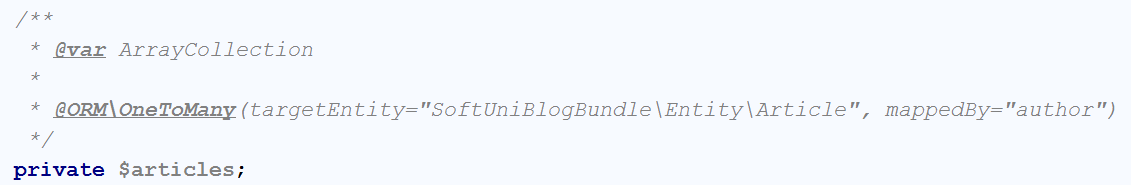
Now we should create the **setter** for the author field:



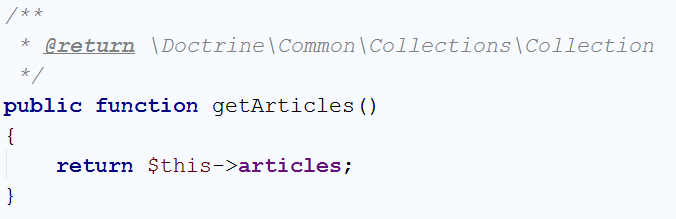
And our **getter**:



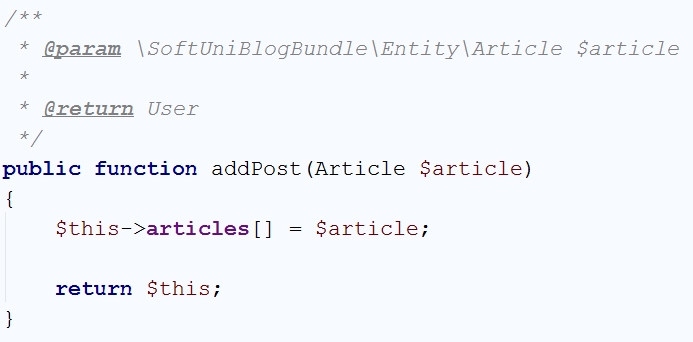
That’s it, we’re done with the Article entity for this step. We need to do the “one to many relationship” on the side of the User entity. Just below the private “password” field, create the following field:



This field will be of type ArrayCollection, that will keep all of the current user posts. As you can see, we define one-to-many relationship with the Article entity, using the author field, we’ve created earlier. For this field, **we won’t create setter** like for previous ones. Firstly, we should create the getter:



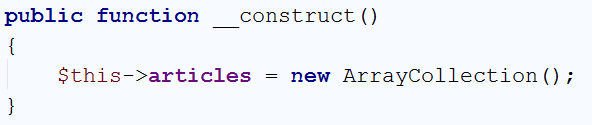
The setter however will be slightly different. It should **add a new post to the current user posts**. To do that, we should write the following code:



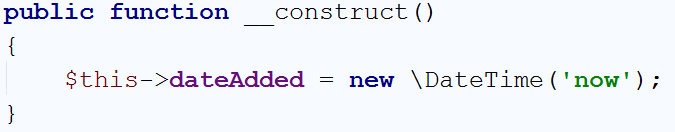
We’re done with the connection for now. Later **we will update the database** schema.

## Set Default Values for our Entities

Our next job is to create the so-called [constructors](http://php.net/manual/en/oop4.constructor.php) for our entities. The constructors are special methods that are called **each time a new object from the entity is created**. Let’s start with the User entity. Its constructor should be the following:



Every time we create a new user, it will receive empty array of articles. The Article on the other hand should look like this:



Each time a new article is created, this constructor will add the current time.

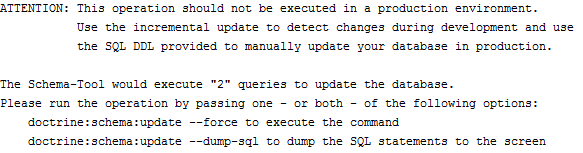
We are ready with this part, now **we can update the database**(schema).

## Updating the DB with our Article Entity

There are many ways to update or create the tables that we need. The first one is to create them **manually**. That will take a lot of time and because of that we won’t do it that way. We will create them **using** [Doctrine](http://docs.doctrine-project.org/en/latest/). **Open** a **Terminal**/**CMD** in the project **root** **folder**. Let’s write the following command:

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

This will result in the following warning:



It basically tells us, that we are doing an operation that is not safe. To do it, **we need to force Doctrine** to execute our command. In order to do that we need to add “**--force**” after our previous command:

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

The result of this command should be the following:



If we take a **look at the DB in HeidiSQL**, we will see that the table “**articles**” is created:



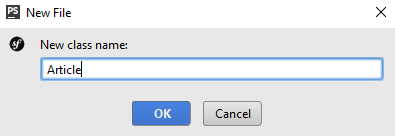
We are ready, to start making our blog.

## Creating the Article Controller

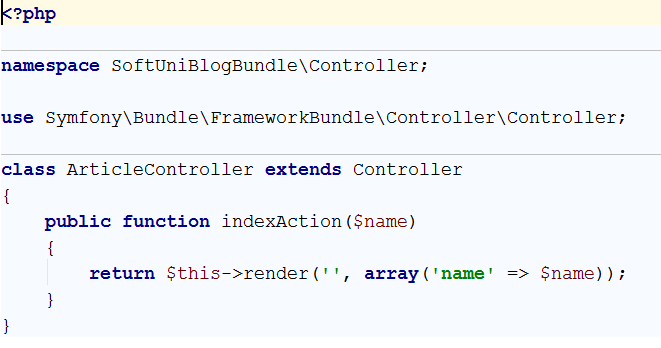
Now we should create a class that will control how the articles are going to be viewed, created, edited and deleted. We will create it in the **Controller** folder. If you are using **PhpStorm or IntelliJ IDEA** and you have the **Symfony plugin installed**, you should see this when you right-click on the Controller folder:



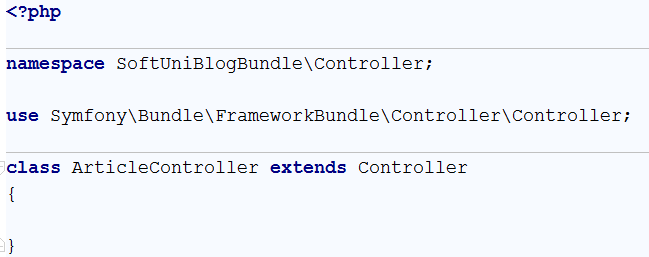
Give it the name Article:



We have just created an ArticleController in the **Controller** folder, that looks like this:



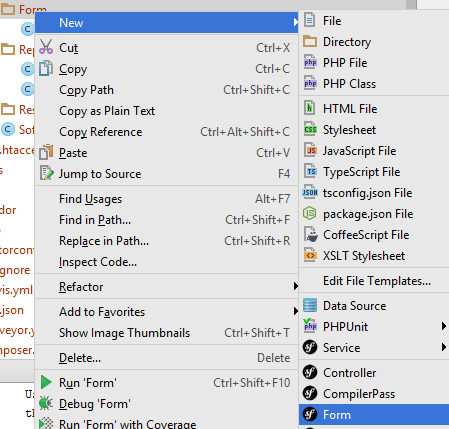
Delete the indexAction method, we won’t need it. We should be happy with the following result:



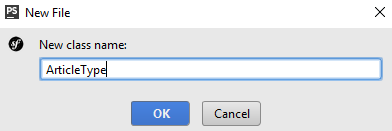
We have a controller, but we need **form template**.

## Creating the Article Type Form

Our next step is to **create a form template**, that we are going to fill, each time when we’re **creating or editing** an article. To create this form, just right-click on the **Form** folder and choose new **Form**:



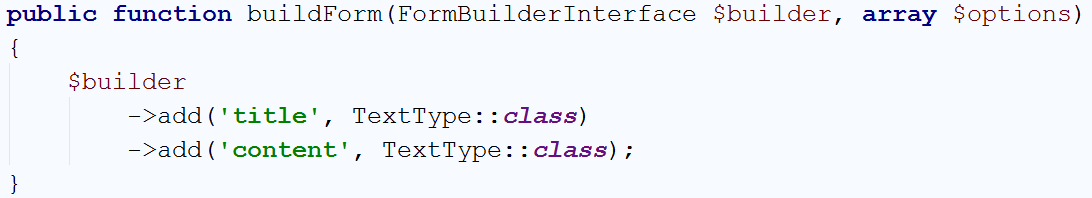
Give it the name “ArticleType”:



We should receive something like this:



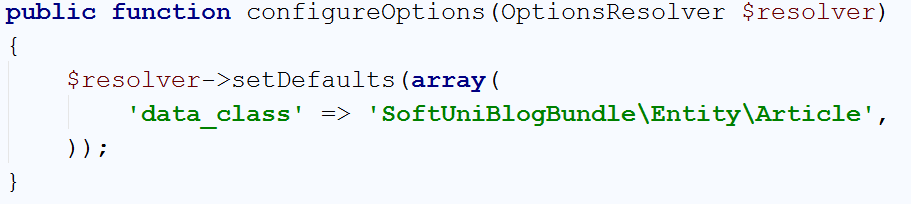
You may notice that we have 2 empty functions. “buildForm” will create our form and “configureOptions” will tell our form that it is for the Article entity. Let’s start with the form creator:



It is pretty simple form. It should only contain **title** and **content** fields, both of type text. You should use specific using for the “TextType” to work. If you have another one **ending in \TextType** already imported – delete it and add:

|  |
| --- |
| **use** Symfony\Component\Form\Extension\Core\Type\TextType; |

Let’s create the logic for our “configureOptions” function:



The default value for our resolver **tells controller that is going to use the form**, in what type of object it should save the date from our form. That’s it.

## Implementing Article Create Function

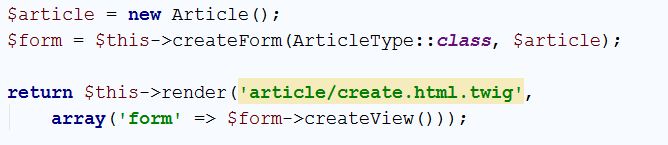
Go back to the article controller, we need to create a new function. We will name it “createAction” and create few annotations for it:



Let’s start from the first annotation. It tells our project that the function will receive **one parameter** of type [Request](http://api.symfony.com/3.1/Symfony/Component/HttpFoundation/Request.html). We will save what request is for some other time. The second annotation is more interesting. It defines a “[Route](http://symfony.com/doc/current/routing.html)”. The **route represents** the **URL**, that the **current** **method will correspond to**. In this case the function will be called when we go to <http://localhost:8000/article/create>. Each time we **use this URL**, the router will **call our function**. To **simplify** the **redirection** between our **pages**, we give a simpler name like “article\_create”. The third annotation is to make sure, that only **logged in users** will **use** our **function**. Without it, every guest **would be able** **to create a new article** and we **don’t** **want** **that**. The final parameter specifies that our **function** will **return** a **response**. We will talk about this later. In order for those annotations to work correctly, make sure you are using the right imports:

|  |
| --- |
| **use** Sensio\Bundle\FrameworkExtraBundle\Configuration\Route; **use** Sensio\Bundle\FrameworkExtraBundle\Configuration\Security; **use** Symfony\Component\HttpFoundation\Request; |

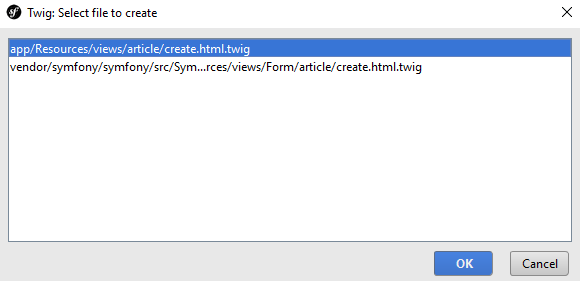
Now let’s write some real code. In the function, write the following:



What is this code doing? It’s simple – it **creates a new article**. Then it **creates** a **new form** from the template we’ve created earlier and tells the **form** that it **should** **fill our new article**. Finally, it **sends the form to a** **view** that we are going to **render** on the screen. Render means draw. Symfony uses [Twig](http://twig.sensiolabs.org/). Twig is a [templating engine](https://en.wikipedia.org/wiki/Template_processor), which job is to **display our data** in an **easier** **way**, than creating the HTML by ourselves. The important part here is that we don’t **have such template yet** and PhpStorm (IntelliJ IDEA) tells us, by making **yellow rectangle** over the name of our template. To create it, just click on the template name and then press ‘**Alt + Enter**’. This will open a context menu in which you call tell your IDE to create the template for you:



Then you need to choose the first option:



Congrats, you are looking at an **empty template**. Write the following code:



This code does 3 things. The **first one** is to ‘extend’ an existing template. What does that mean? It means, that **we’ve** **created the base design of the blog for you**, including all **styles** and **scripts** that you may need. **You** **can** now simply **reuse** this **base** **template** in all of the templates you are going to create. The **second** statement replaces a block called “**main**” in the base template. This means that all of the HTML in the base template for the “**main**” block will be replaced by the code you are going to write in a second.

Just because we don’t want you to focus on HTML and Twig, we will give all of the code, that you need to write in the main block:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'article\_create'**) }}**" method="POST"**>  <**fieldset**>  <**legend**>New Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title"  name="article[title]"**>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_content"**>Content</**label**>  <**div class="col-sm-6"**>  <**textarea class="form-control" rows="6" id="article\_content"  name="article[content]"**></**textarea**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'blog\_index'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

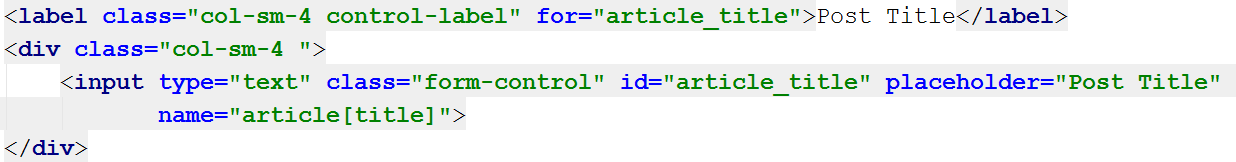
However, let’s explain few parts of that template.

The first part we are going to discuss is:



We are using some css class, this part you should be familiar with. The really interesting parts are the **action** and **method** attributes of our form. First, we are going to talk about the method. This attribute defines what type of [request](http://www.w3schools.com/TAGS/ref_httpmethods.asp) we are going to use. To simplify things, let’s explain the requests shortly. The request we are going to use is “**POST**”. That means that we want to **send** **data** to some place. In our case, it tells the [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) protocol that we want to **send our title and content** to a place in our blog. The other type of request that we’re interested in is “**GET**”. It tells HTTP that we want to **get** **some** **data** from somewhere. There are other types of requests, but we’re not going to bother you with them now. Let’s talk about the **action**. The action attribute defines **from**/**to** where we want to **GET**/**POST** our data. Remember the name of the route we gave our function earlier? Yeah, we want to send a **POST** **request** with our **title** and **content** **back** to the **function** we’ve created earlier. We will see how to use the data from the request later on.

The second part from the template that deserves a quick look is:



The first thing to notice is that the **for** attribute of the **label** and the **id** attribute of the **input** have the same value. Now take a look at the **name** attribute of the **input**. It looks like dictionary value. When you are mapping your entities in the twig templates, it’s important to note that the first part of the **name** is the **name of the entity**. In the square brackets, we put the **name of the field** from the entity we’re going to fill.

Another interesting thing is:



This is a special twig code. It creates a new **invisible** **field** in our form, that validates our form. Without it, our form won’t work. It you want to know more you should check about [CSRF](https://en.wikipedia.org/wiki/Cross-site_request_forgery).

Finally, one more special twig code that we saw earlier as well:



This “path” command uses route name, and redirects to the given route.

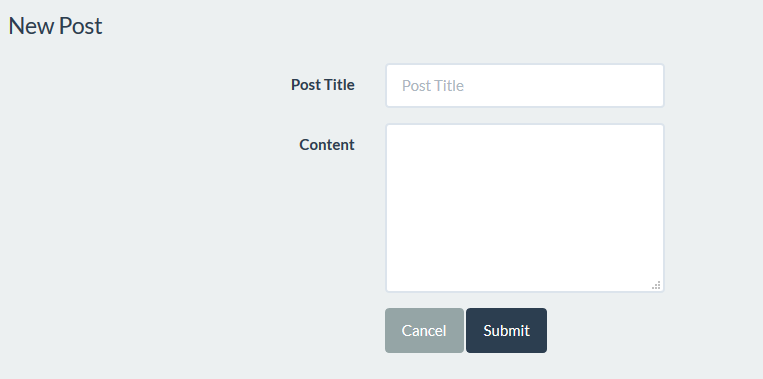
Enough for the templates for now, let’s start the blog and see if it works. To do that we need to open the Terminal/CMD in the root folder of our blog, or use the built-in terminal in PhpStorm (IntelliJ Idea). Don’t forget to start MySQL if you haven’t by now. Enter the following command:

|  |
| --- |
| php bin/console server:run |

If everything works, you should see this:



Open the browser and go to the address. You should see almost empty page. Now you need to register a new user and login. After login, in the URL enter <http://localhost:8000/article/create>. It should redirect you to form like this one:



**Fill the form and click** “Submit”. The **page gets refreshed**, but if we check the table in the **database**, **it is empty**. Let’s fix the problem. Get back to your function in the article controller. The problem is that we’ve never used the data from our form. Add to your function the following code:

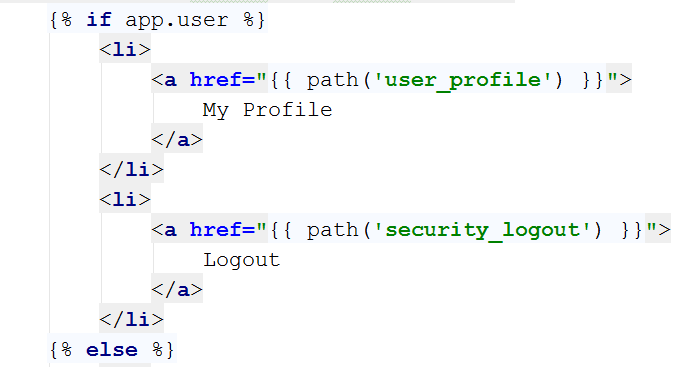


This code **takes the data from request (make sure the imported “use” statement at the beginning of the class is Symfony\Component\HttpFoundation\Request)** and **fills** the **form**. After the form is filled, we check if there is **any data** in the form and if it **is valid**. If everything is okay, then we get the currently **logged** **in** **user** and assign him as **author** of the **article**. Then we get the **entity manager** from **doctrine** and using the “persist” function we **add** our **new article** in the **database**. Finally, we call the “flush” function, which sends the article to our database. **After** the article is **sent** to the database, we **redirect** the **view** to the **index page** of our blog.

While we’re changing the code, open the base template:



Find this part of the code:



Add a new “<li>” element which will redirect to the **create article** page:



Let’s go to our blog and login. Now on the right-side of the navigation bar, we see the new button:



Let’s try to create new article. After pressing the “**Submit**” button, we should get redirected to the home page. Let’s see if we got anything new in the database:



Hooray, we did it! Now we can create articles. The problem is that we can’t see them from our blog. We will fix that.

# Listing Articles

## Listing All Articles

Let’s go to the home controller. When you open it, you will find a function called “indexAction”. Its **only** **job** at the **moment** is to **call** the **index** **view**, without any data. We will change that. **Write** the following **code** in the **beginning** of the **function**:



This will get all of our articles from the database. Let’s pass them to the view. Edit the return statement like this:



We’re done here, go to the view, and examine it:



You should see this:



In the main block, write the following code:

|  |
| --- |
| <**div class="container body-content"**>  <**div class="row"**>  {% **for** article **in** articles %}  <**div class="col-md-6"**>  <**article**>  <**header**>  <**h2**>{{ article.title }}</**h2**>  </**header**>   <**p**>  {{ article.summary }}  </**p**>   <**small class="author"**>  {{ article.author.fullName }}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs"  href="#"**>Read more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  {% **endfor** %}  </**div**> </**div**> |

There are few key moments that we want to take a look at. The first one is:



This is a simple foreach loop in twig. It will traverse the array of articles we’ve sent to the view through the controller. There is also a closing statement few lines below:



Between those two rows, there are a couple of twig calls. The first one is:

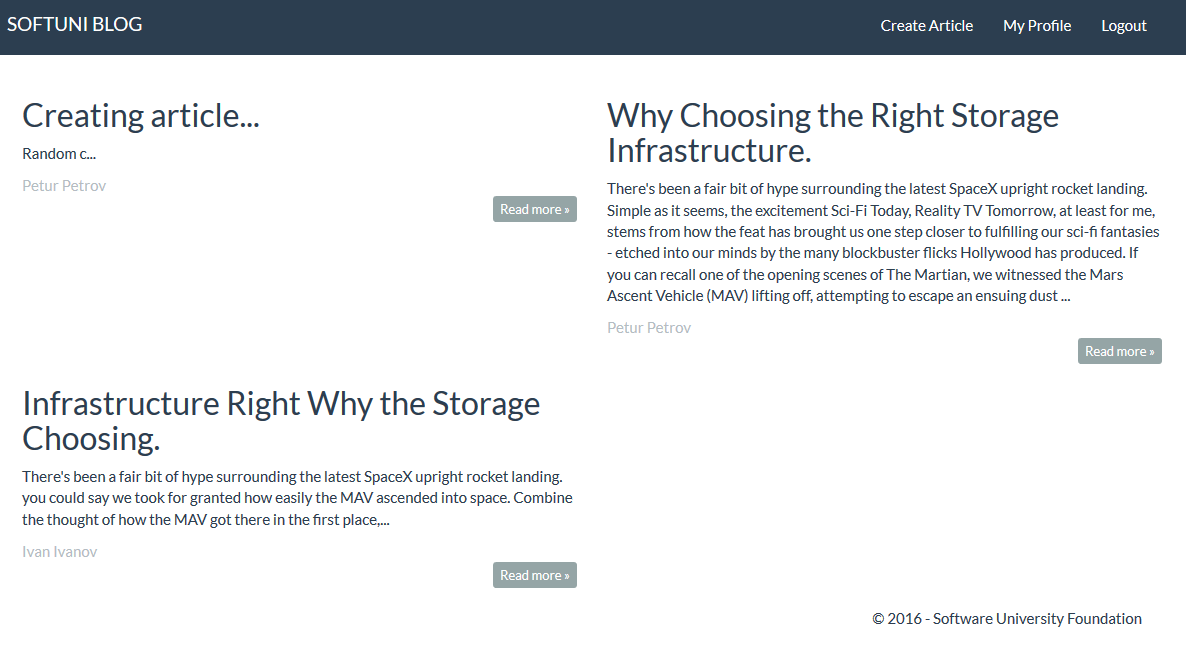


This will print the title for each article. We have the same thing for the summary and author of the article.

For now, let’s start the blog and see what we have:



It works! Let’s create few more articles:



Looks good. The problem is that if we press “**Read more**” nothing happens. We should fix that.

## Showing Single Article

To implement the single article page, we need to go back to the article controller. Create the following method:



Let’s take a look at the annotations. The route annotation is having curly braces (‘{‘, ‘}’) and some parameter inside them. That is the parameter, that the function takes. Everything else is standard. If we take a look at the function, we can see that we are looking for a specific **id** in the database. This row will give us only the article with the given **id**. Then we send it to the view. Create the view, like we did earlier. The generated view will contain the base structure we are already familiar with:



Write the following code in the main block:

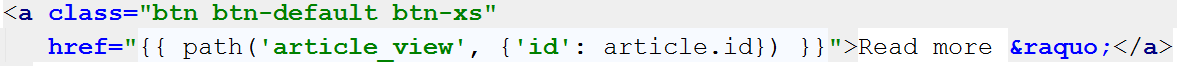
|  |
| --- |
| <**div class="container body-content"**>  <**div class="row"**>  <**div class="col-md-12"**>  <**article**>  <**header**>  <**h2**>{{ article.title }}</**h2**>  </**header**>   <**p**>  {{ article.content }}  </**p**>   <**small class="author"**>  {{ article.author.fullName }}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs" href="**{{ path(**'blog\_index'**) }}**"**>back **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**div**> </**div**> |

This code is really simple, with the only difference from the previous one being that we have only one article and we are printing the content instead of the summary.

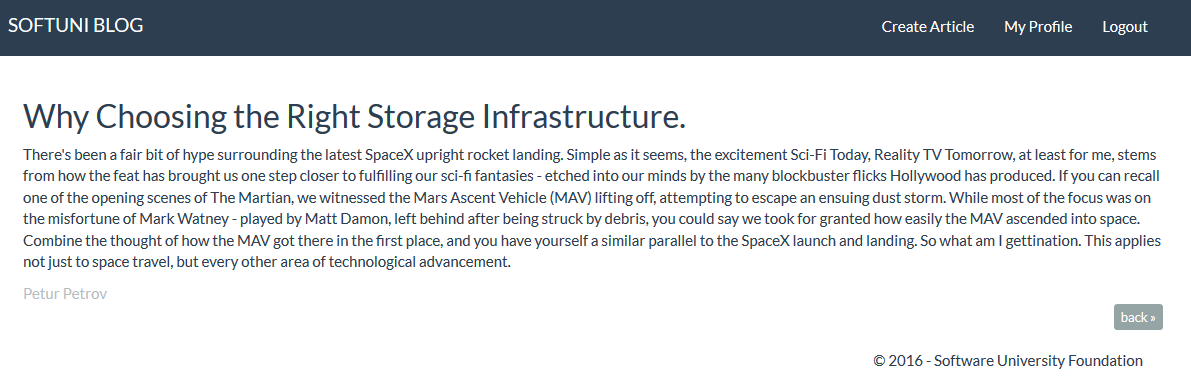
Let’s start the blog and see if it works. The answer is no, it doesn’t. Right now, the button read more doesn’t redirect to the right route. Let’s go back to the index view and find this piece of code:



Change it to:



Let’s try it now:



Another victory! In the next chapter, we will create the functionality to edit articles.

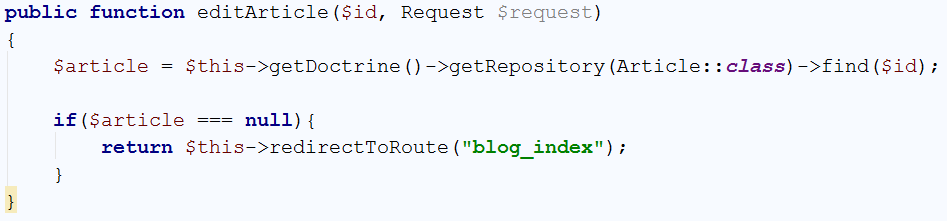
# Editing Articles

## Creating the Action

Let’s create the functionality to **edit articles**. Go to your ArticleController. Create a new action called editArticle():



With the annotations, we **define the route** and we tell that this function is going to be **only accessed** by **logged in user**. Now we need to check **if the article exists** and **if it doesn’t** we are going to **redirect to the home page**:



So far, so good. But what if the article actually exists? We need to **render a form**, which will **allow us to edit the article**:



The form will be the **same form** we’ve used for the create() **function**. Let’s create the view now.

## Creating the View

You should start with this:



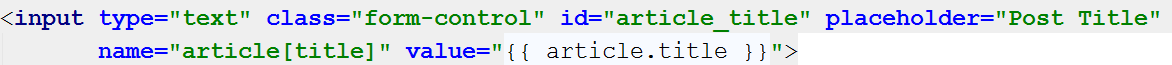
In the main block, write the following code:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'article\_edit'**, {id: article.id}) }}**" method="POST"**>  <**fieldset**>  <**legend**>Edit Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title"  name="article[title]" value="**{{ article.title }}**"**>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_content"**>Content</**label**>  <**div class="col-sm-6"**>  <**textarea class="form-control" rows="6" id="article\_content"  name="article[content]"**>{{ article.content }}</**textarea**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'article\_view'**, {id: article.id}) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-success"**>Edit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

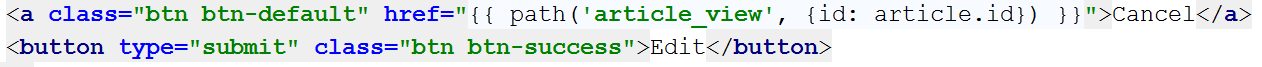
This template may look familiar to you. That is because this is the create view, with few changes. First of all, the action attribute of the <form>, leads to a different route:



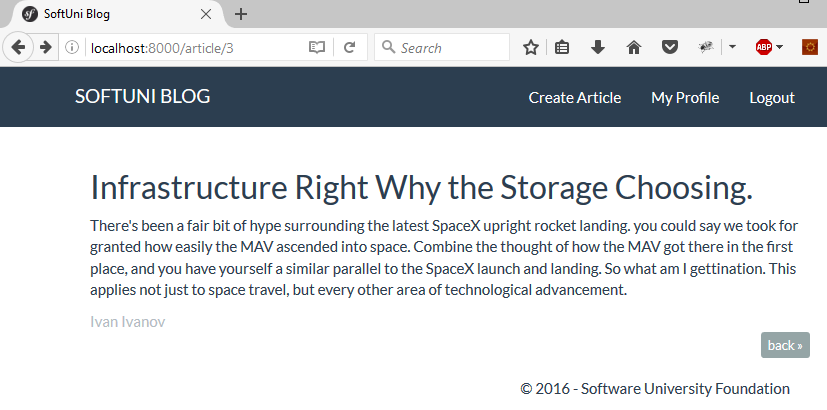
Then, by default every **input field** is **filled** with our **existing article**, using the value attribute:



Finally, the “Cancel” button return us to the single article view and we’ve made the “Edit” button green:

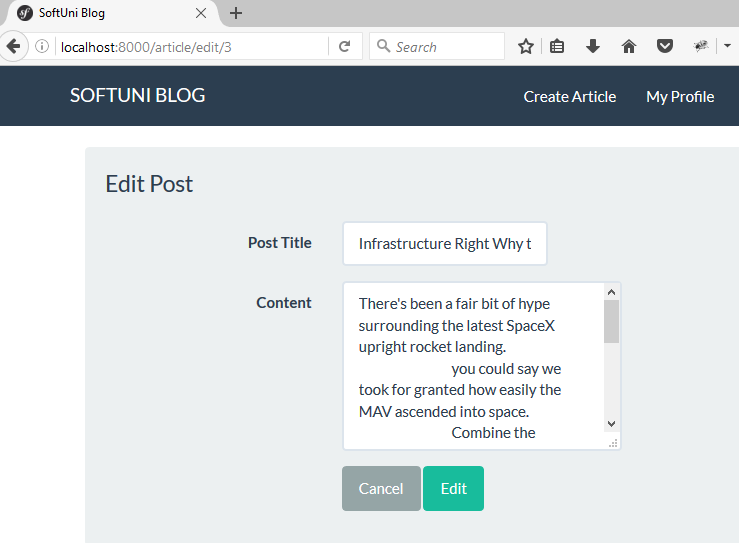


That’s all. Let’s see what we have. Go to **your blog** and open a **single article view**:



The URL for me is <http://localhost:8000/article/3>. I will change it to <http://localhost:8000/article/edit/3>.

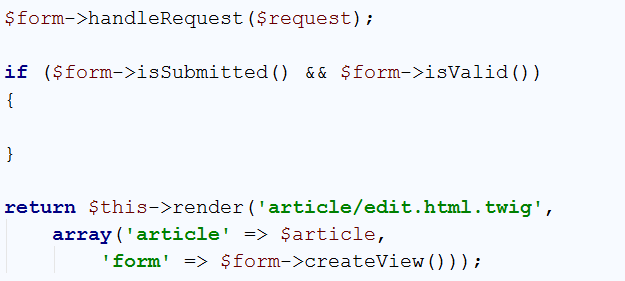
You should see something like this:



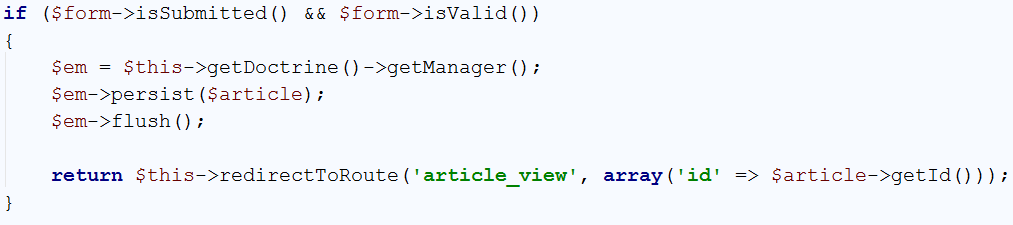
It looks good but **it doesn’t work**. That is because we are **not saving our changes**. We need to fix that.

## Saving the Edit Changes

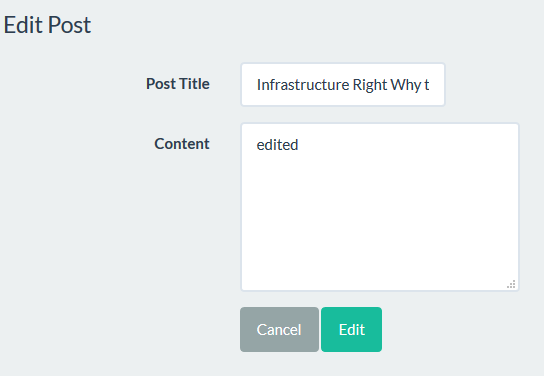
Let’s go back to the ArticleController and our edit() function. **Above** the final return statement, we need to **get the data** from our **form**:



Now we’re **taking the data** from our **form** and the only thing left is to **save the changes**:



Let’s try to **edit the post**, we’ve opened earlier. I will change the **content** to “edited”:



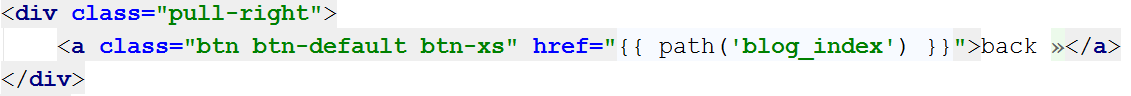
After I **press** “**Edit**” I should be **redirected** to the **article view**. And here it is:



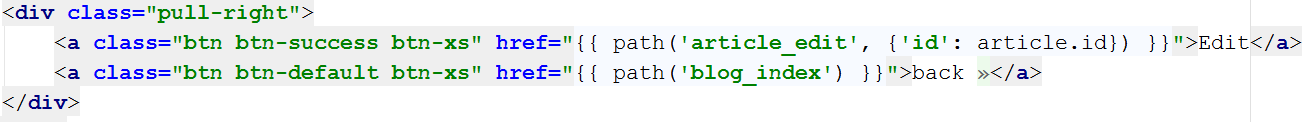
We’ve created the edit functionality, but it’s hard to use it.

## Adding Edit Button to the Single Article View

Open the “article.html.twig” file. **Find** the **back** **button**:



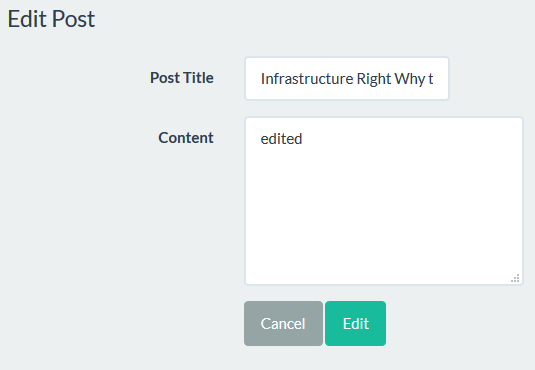
Let’s **add another button** next to it in the <div>. We will make it green, and it will **redirect to the** edit() **function**:



Let’s see if it works:



After we **click** it, we get **redirected to the edit page**:



That is everything when it comes to editing. Next chapter – **deleting**.

# Deleting Articles

## Creating the Action

Let’s create a new delete() function in our ArticleController. It should look like that:



The contents of the function will be **very similar** to our edit() function:



There are very few differences. First and **most important** is the function we are going to **call** when we **submit our form**. For this action, the function we are going to call is:



This function will **delete** the **article** with the given id. Then we are going to **redirect** the user to the home page. Now we must create the view.

## Creating the View

You should start with this:



In the main block, write the following code:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'article\_delete'**, {id: article.id}) }}**" method="POST"**>  <**fieldset**>  <**legend**>Delete Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title"  name="article[title]" value="**{{ article.title }}**" disabled**>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_content"**>Content</**label**>  <**div class="col-sm-6"**>  <**textarea class="form-control" rows="6" id="article\_content"  name="article[content]" disabled**>{{ article.content }}</**textarea**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'article\_view'**, {id: article.id}) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-danger"**>Delete</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

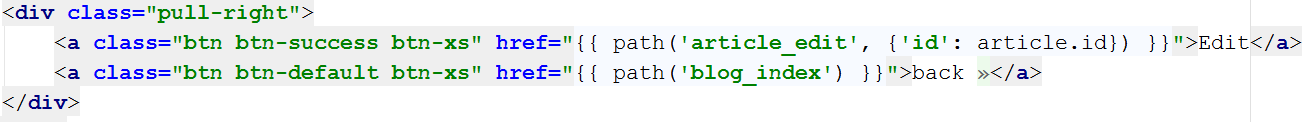
This code may look familiar to you. This is the **edit view** with only 3 changes. The first change is that our form action redirects to the delete function. The second one, is that we’ve changed the **delete** **button color** to **red**. The third and final one is that our **text inputs** have the disabled attribute:



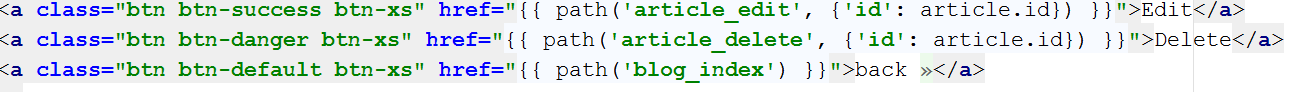
The only thing left is to **create a button** in the **single article view**, which will call our delete() function.

## Adding Delete Button to the Single Article View

In the “article.html.twig” file find the place where we’ve added the edit button:



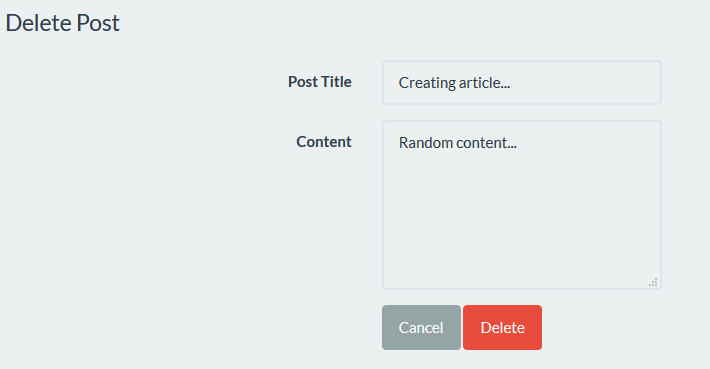
You’ve guessed it, we should create the delete button here:



Everything should be working. Let’s see how it looks:



**Click** the **delete** button. You should be **redirected** to the **delete view**:



If you try to delete the article, you will see that it **works**! Now we have another problem. **Everyone can delete any article**. Even the **articles** that **belong to other users**. We will **fix** that in the following chapters.