# MVC in Depth - Exercises

This document defines a set of tasks to be done as a part of the MVC in Depth lecture’s exercises.

Modern MVC Frameworks support a set of mechanisms that increase the ease of use of the framework and speed-up the developing process. Most of them follow the [Convention over configuration](https://en.wikipedia.org/wiki/Convention_over_configuration) paradigm to use the internal tools and also depend on other frameworks and mechanisms such as auto-managing the [Inversion of Control](http://stackoverflow.com/questions/3058/what-is-inversion-of-control).

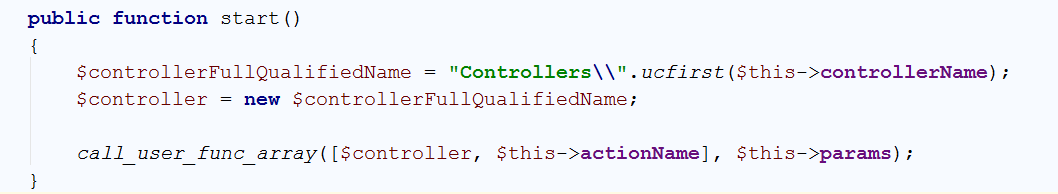
# Application

## Entry Point

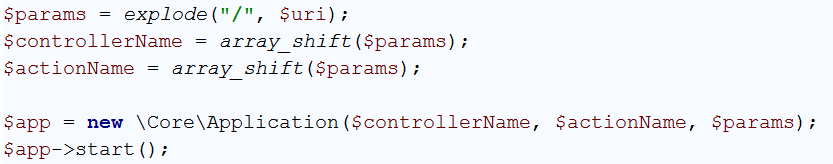
Let’s encapsulate our entry point into a class, so we can use its resources for later usage. For instance, we might need know what controller is invoked, which information is now gone once the controller is invoked.



Let’s move the action calling logic in the Application class:



Now we can just instantiate the class in index.php and pass it needed arguments:



Now test if it works as it worked before ☺

## Views

Our framework still does not support **Views**; our controllers just echo out the desired result. We will continue using part of our Template::render() function, but we will try to move away from the static members in favor of best practices

First of all, let’s rename the Template class, to View and remove the **static** modifier:

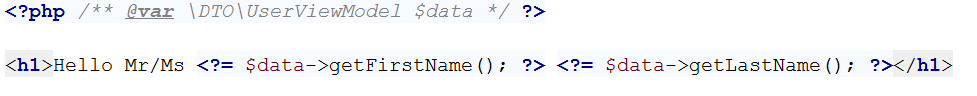


So we can try it in our controller.

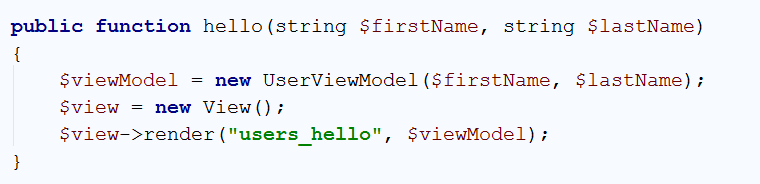
1. Create a DTO (data) called UserViewModel that contains firsName and lastName



1. Create a template in **templates** folder called users\_hello.php that expects the ViewModel and greets the user



1. Go to the controller, instantiate a View object and tell them to render the users\_hello template with this ViewModel.



1. Try it ☺

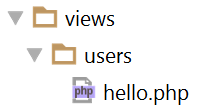


## View Automation

Do we need to specify the view name every time? Can’t we use some convention and only when we need to break the convention, to specify one? Why all of our Views need to float in the templates folder? Can’t we organize them in nested folders?

We can! Here it comes the **Convention over configuration** paradigm. We have already established one: the URI maps to exactly the same class/method pairs. Let’s establish another one: Views for one controller will **reside in a folder named after the controller name**. All of their **filenames will be named after the action name**.

Let’s also rename the whole views folder **from templates to views** ☺



Nice. But if we don’t specify it, how will our View know about the controller name?

Good question. One way is to make the application fields **public static**, then the View class can access them when it’s necessary. But it introduces [high coupling](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) between the modules and will make our codebase in the future one client feature request away from popping.

To be honest, **we have already introduced a coupling**. Our Controller says “I need a View, I will instantiate one”, which is the **core** problem, leading to another problem that might be resolved with **bad design** solutions as the one above.

The solution here is to [invert the control](https://en.wikipedia.org/wiki/Inversion_of_control). Instead of our controller to say “I will instantiate a View”, they will say “Because I’m managed by a generic framework, I will tell I need something that can be rendered and they will give it to me”. This means our methods (hello()) can accept an argument of View type (or even an [interface](http://php.net/manual/en/language.oop5.interfaces.php)).

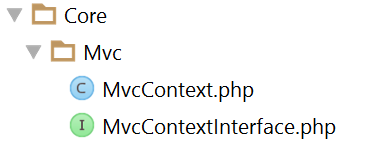
The application (framework) will then spot this needed dependency and provide it by some mechanism. The good part here is that the View class, might need other things, that can be spotted. For instance, a class that knows about the Controller name and the Action name.

These method of asking for dependencies is called [dependency injection](https://en.wikipedia.org/wiki/Dependency_injection). Our framework will recursively find needed dependencies from the deepest part and back, and will check in a configuration which implementation to provide.

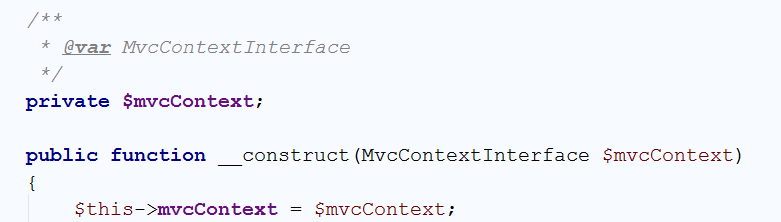
## MVC Context

Let’s move the controller, action and params from the application itself to another module (class) that will be managed by the framework and will be passed as dependency to other modules (e.g. View, Application).

As we have said above, we will create interfaces for each module, so we will register later which interface implementation to live in the application container.



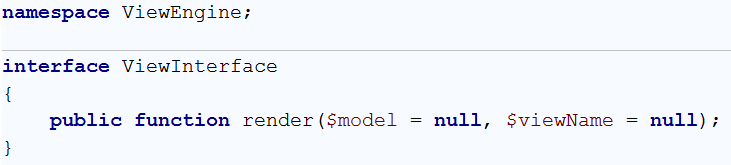
Now our Application should depend on MvcConextInterface instead.



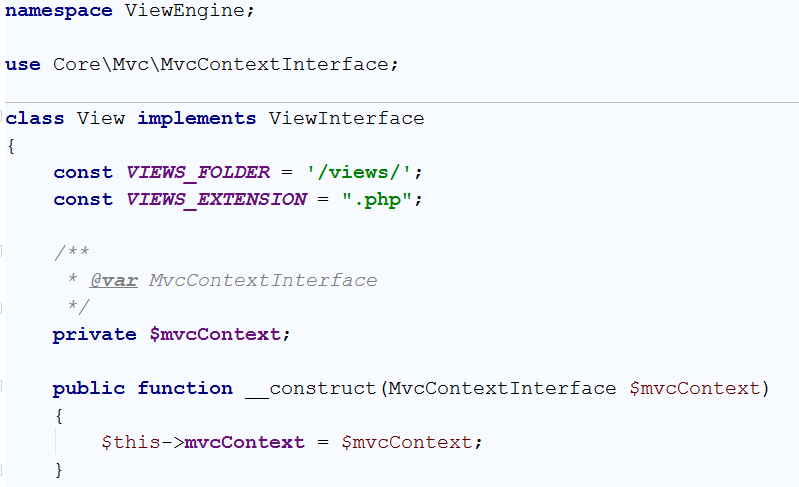
And our index.php could provide it:



## View Logic

Let’s first create a ViewInterface which holds one and only method – render(). This method needs two optional arguments: $model (data) and $viewName 

Our View class should implement it. But as we said before, we will need to know about the **controller name** and **action name**. This means our View class will depend on MvcContextInterface too.

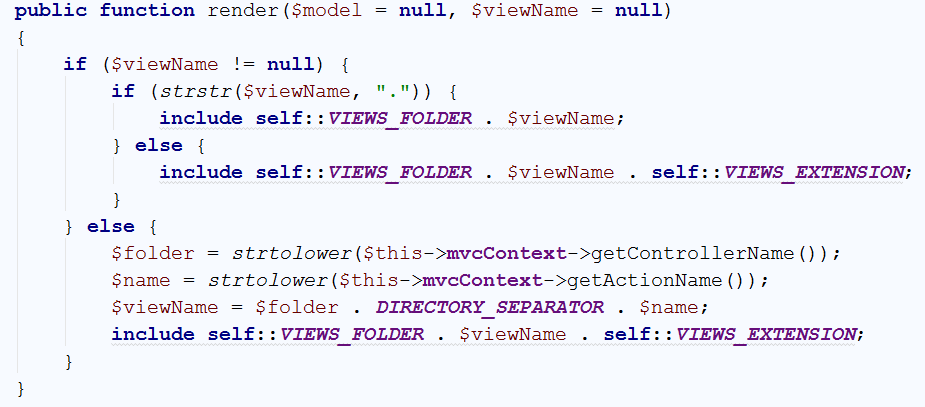


From the picture above you can also see we have changed the view’s folder to **“/views/”** and the extension is also extracted in a constant. It’s a small change, but still needed to be mentioned.

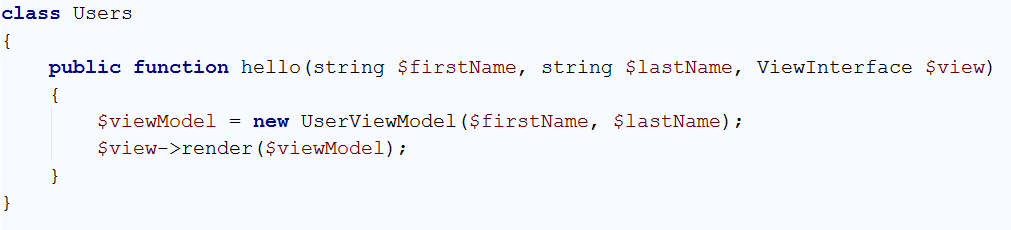
Now we need to implement our render() method.

If a ViewName is provided, then we need to include this view. Additionally, we need to check whether an extension is provided. A simple check for containing a dot (.) could be enough. E.g. when “**users/all**” is provided we need to include **“/views/users/all.php**”, but so does if “**users/all.php**” is provided.

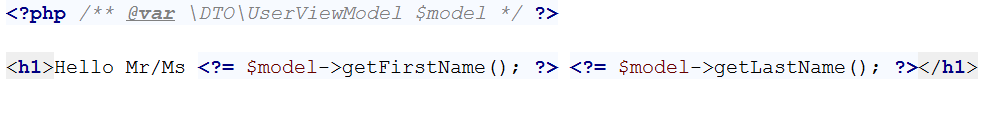
If no ViewName is provided, then we need to compose one with an extension from the controller name and action name.



Now we can introduce dependency of ViewInterface in our hello() method and call the rendering method.



And rename the $data variable in our views to $model ☺



# Dependency Injection

## Container

So far we have built our architecture around **interface dependencies**, but there’s not mechanism that manages them. Our application should know how to resolve these dependencies.

In order to know this, someone should register the abstraction and implementation pairs. We can keep these pairs in associative array. Let’s call it $dependencies.

It will be filled with **abstraction** names as a key and its corresponding **implementation** name as a value. For instances:

$dependencies["\\Core\\ViewEngine\\ViewInterface"] = "\\Core\\ViewEngine\\View";

But we also need to **keep track of already resolved** dependencies. Because we will try to scan them forever if we have the so called **cyclic dependency**, e.g. if **“A” depends on “B”**; **“B” depends on “C”** and **“C” depends on “A”**, we will try to resolve “A” forever, but if we save it as resolved once, we will not try to resolve it second time.

Let’s keep this track in another array called $resolvedDependencies

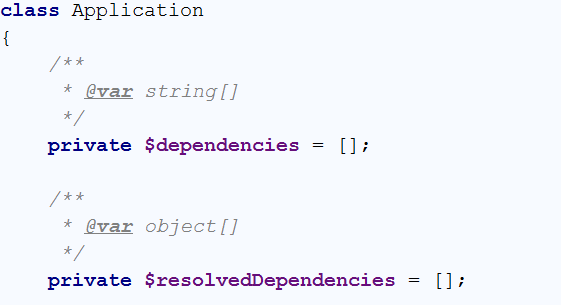
It will be filled with **implementation** name as a key and its corresponding **object created** as a value. For instance:

$view = new View();

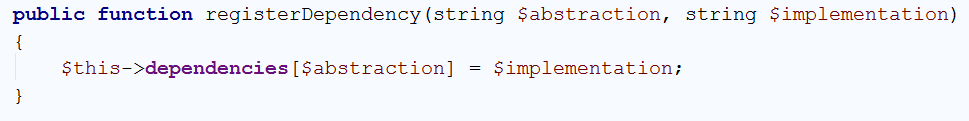
$resolvedDependencies["\\Core\\ViewEngine\\View"] = $view;

Once we try to resolve “ViewInterface” we will look it up in $dependencies, find that it needs to be “View” implementation, then lookup in $resolvedDependencies and find it’s already instantiated so we will take that instance. If it is not already instantiated we will instantiate it and save it there ☺

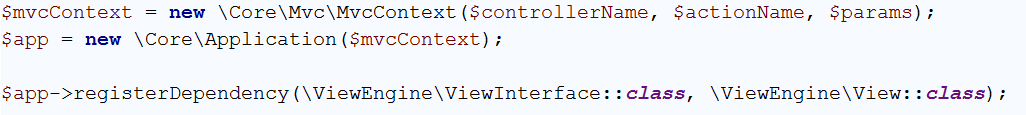
Let’s define both arrays in the Application class:



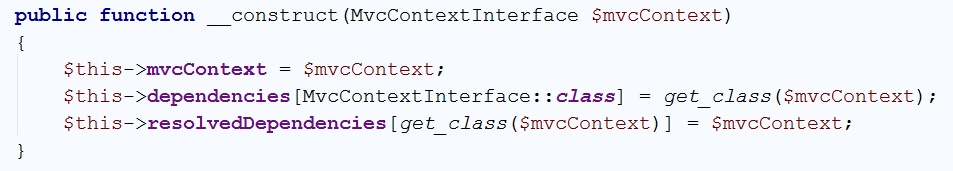
And define a method that fills the first array from the outside:



Let’s register in our index.php the view interface and implementation names. We can use pure strings or the predefined “class” constant that gives that string.

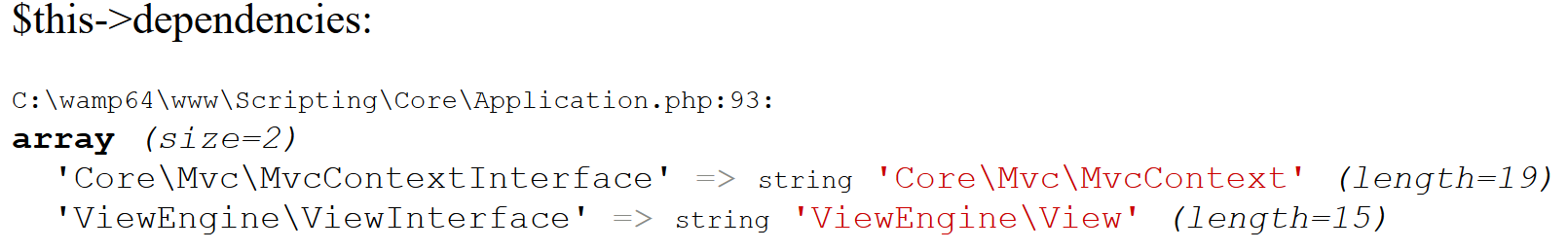


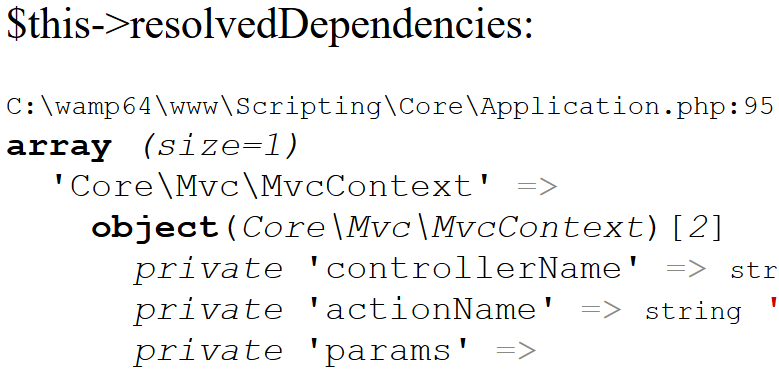
The Application class is receiving the $mvcContext instance manually, so we need to take care of its registering in both arrays for further use. Let’s do it in the **Application constructor.**



The get\_class() built-in function returns the class name of an instance, in this case “\Core\Mvc\MvcContext”

Now in our arrays we have the following information registered:





So we are ready to start scanning dependencies and resolve them.

How we can introspect what dependencies a class or a method has? We can use the [ReflectionClass](http://php.net/manual/en/class.reflectionclass.php) built-in object to introspect class methods and constructors, so a general dependency for class is placed in the constructor arguments.

The [getConstructor()](http://php.net/manual/en/reflectionclass.getconstructor.php) method returns a [ReflectionMethod](http://php.net/manual/en/class.reflectionmethod.php) instance which holds informaton about its [Parameters](http://php.net/manual/en/class.reflectionparameter.php).

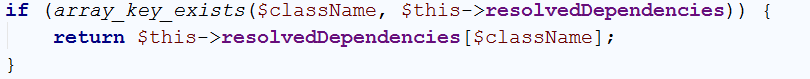
If the class uses the default constructor (in other words, there’s no constructor defined), the getConstructor() method will return **null**, which means we should stop using the Reflection\* classes and just instantiate it with the new keyword, as we have done before

Define a method resolve() that accepts a class name argument in our Application class.



This function takes the class name and returns an object of this class, but before that – scans all of its constructor dependencies and resolve them too ([calls itself](https://en.wikipedia.org/wiki/Recursion_(computer_science)) with each argument)

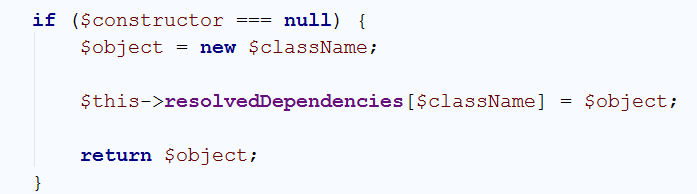
Before all of these operations, it will be very handy to **check if the class name is not already resolved** and just return it from the array



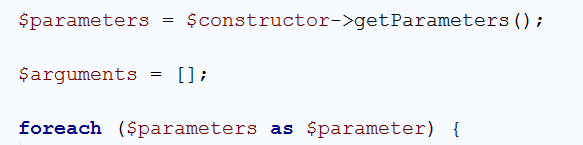
But if it’s not resolved, we need to prepare the magic. First of all we need to extract the **constructor**



But if there’s **no constructor**, we can safely say there are **no obvious dependencies**, so just instantiate the class with new and add it to the already resolved objects.



Otherwise continue with the magic and **extract its parameters**. Each of them should be an **interface**. We will try to resolve them and add it to a general $arguments array, which lately we will pass to the constructor



Each $parameter knows its class (interface) by the getClass() method



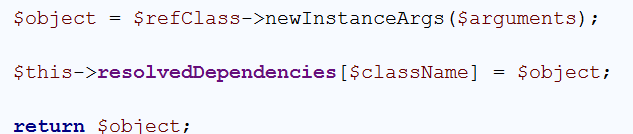
The **corresponding class** of this interface we have in our $dependencies mapping (remember, abstraction->implementation pairs?)



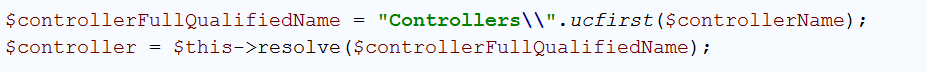
As we have the class name for the dependency **we can perform the same operation** (recursion) for it and when the result is returned (the recursion ends) we need to add it this general $arguments array



When the foreach call ends, we have all arguments added in the $arguments array, and via the ReflectionClass built-in method called newInstanceArgs() we can instantiate the object, add it to the $resolvedDependencies and return it.



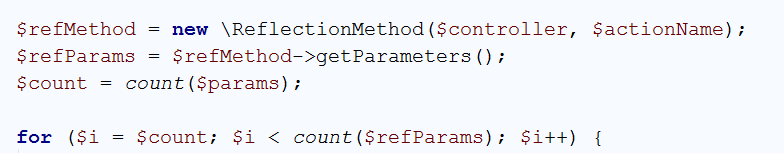
Nice. We have just written the mechanism for **recursive dependency resolving**. Now we need to use it. The first place we will use it, is the **controller instantiation**. So far we are using the new operator, but it will not trigger the dependency resolving. Go to the start() method and change the instantiation via new, with resolve();



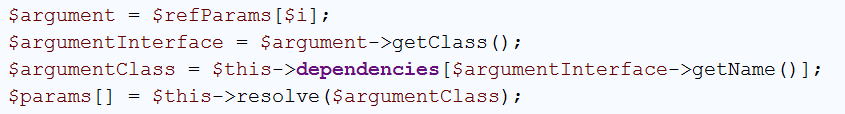
We have introduced dependency of ViewInterace in a controller action (e.g. hello()), remember?

So we need to introspect the action parameters as well. But there some non-object arguments which are coming from the request (e.g. $firstName and $lastName). The good thing here is that we have an array with the arguments coming from the request, called $params and we know how many the arguments exactly are.

This means, while we are scanning the parameters via reflection, we can skip as many arguments as the $params array size is.



Now we can perform the resolving method upon the next arguments and add them to $params in order to invoke the action with the right number of arguments.



Once the loop exits, the logic for calling the method stays the same



We can now run our application to see that everything works just fine



## Model Binding

We have removed the need to use $\_GET in a controller, by passing the query string arguments as arguments of the function. But if there’s a form submission, we still need to use the $\_POST array. There several problems here:

* **Coupling** again! Seriously! Using a **superglobal does not say to the outside world what does this function need**. How can I know, I need to submit a form with X, Y, Z fields? Only by opening the method and scan all the code in it.
* Possible **human errors**. Using strings is never good idea. What if I misspell $\_POST[‘actionName’] with $\_POST[‘actioName’]? I will kill hours in debugging it.

But sometimes the Form data could be long (let’s say 30 fields), maintaining function with 30 arguments will also be pain.

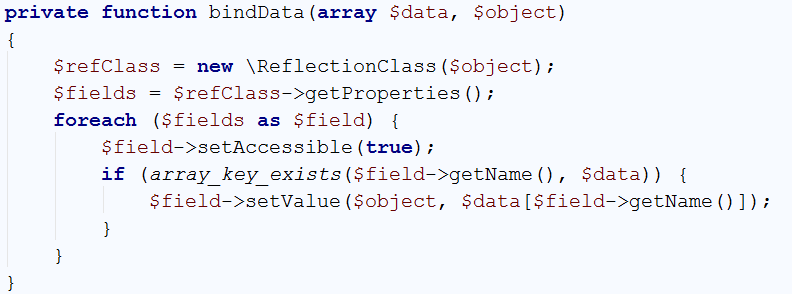
The solution here is to create a class (just like the view models in DTO folder) which represents the Form data. Let’s call them **BindingModels**.

This means that if I have a function (let’s say edit()) that edits the user profile by ID, I can define it as:

* edit($id, ProfileEditBindingModel $model);

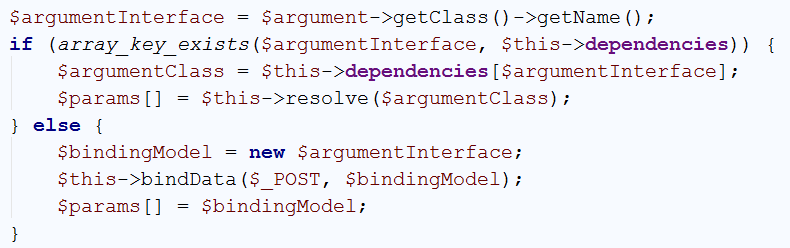
But in order this to happen, we need to change our framework a little bit. While scanning the method arguments, if one of them is not in the dependencies array, we will try to map it to the form data. The **keys** in the $\_POST array will be the **property names** and the **values** respectively – **their values**.

Let’s create a method in our Application class, called bindData($data, $object)



It scans via Reflection all the fields, and no matter whether they are private or not, by setAccessible(true) we can modify them

Now we need to go back to start() method and check whether there is a binding model, to perform the data binding instead of dependency resolving.



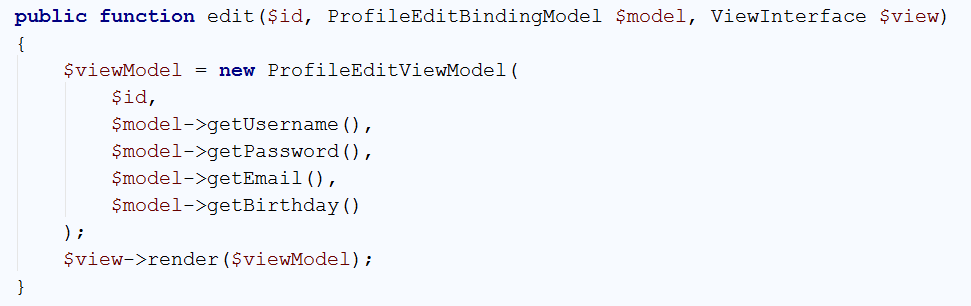
Now let’s test it. Remember the Profile DTO we have from previous exercises? Just rename it to ProfileEditBindingModel and remove the constructor. And let’s create in our Users controller a method edit()

This method will not perform an action edit, but rather test if all the data passed is OK, by just printing it on the screen. If it renders the view, then the ViewInterface dependency is OK, if **the form data** is displayed then the data binding is OK ☺

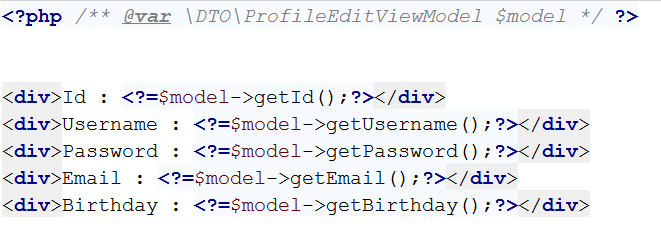
It’s good practice to separate BindingModels from ViewModels, so if we need to display the data to the view we can create an analogical ViewModel, but it will hold the ID too (and can have a constructor).



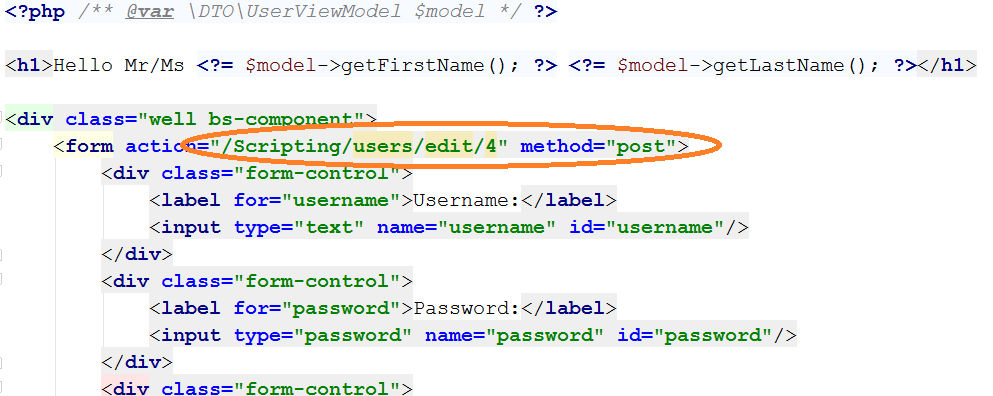
So this is our edit() method now



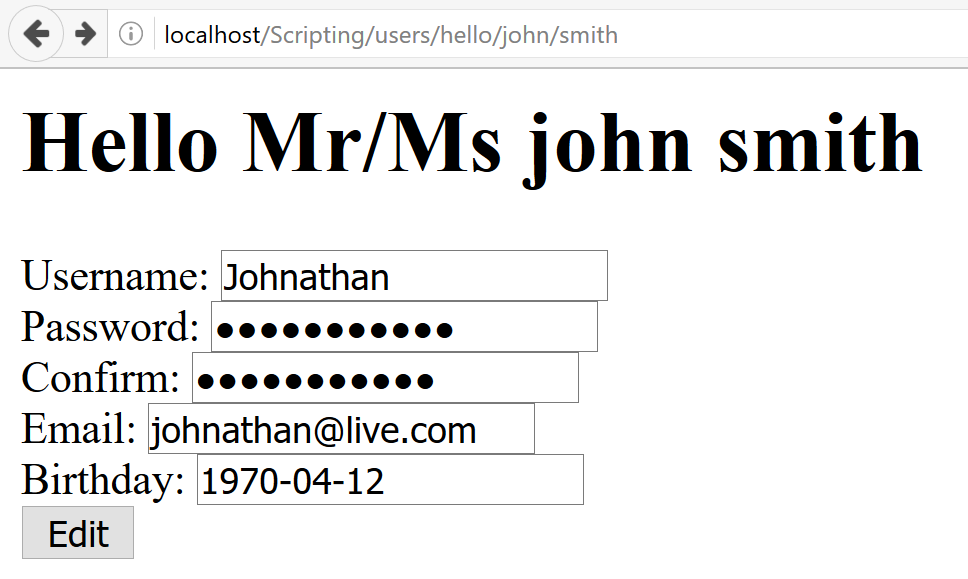
And our edit view (just for debug reasons)



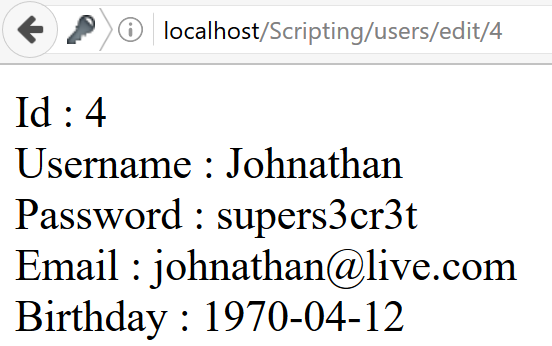
Now we need a form that posts to this controller. Let’s just create one form in users/hello that posts to /users/edit/4 (where 4 is the ID).



And try it out ☺



After clicking [**Edit]**:

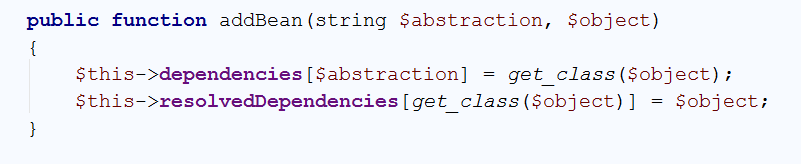


Yeah! We have successfully bound the data to the BindingModel (which we later passed to a ViewModel), resolved the ViewInterface dependency and received the $id from the URL

1. **Database Abstraction Layer**
2. **DI Changes**

The **Database Abstraction Layer** (DBAL) will be our set of classes which will **serve the logic around the database** **operations**. These classes might be needed in series of other classes (hereinafter called “**Services**”). In order to do that, we need to **initialize the classes once** and register them as **already resolved dependencies**, so the DI mechanism could handle them.

We still don’t have a public method that register already resolved dependency, let’s create one in our Application class. It needs to handle the same logic as in the Application’s constructor: The **interface name** should be registered as **key** in $dependencies array, and the **class name** that implements it – as a **value**. Additionally, the **class name** should be registered as **a** key in $resolvedDependencies and the already initialized **object** – as a **value**. Let’s call this method addBean():

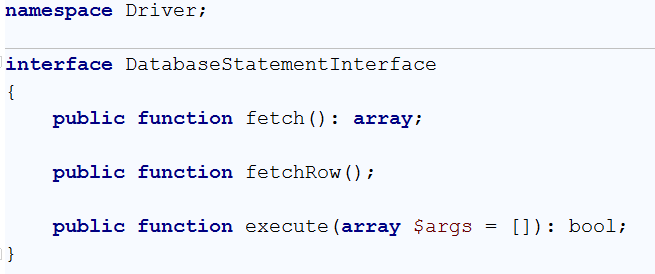


## Defining Common Operations

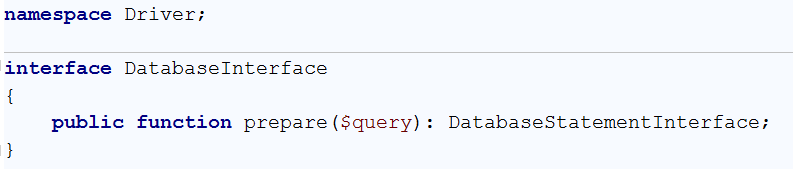
In order our DI mechanism to work, we need one or more interfaces for each of our injected classes. Let’s define the following operations

* fetch() – returns an array with all rows
* fetchRow() – returns one row
* execute() – executes a statement and returns whether it happened
* prepare() – returning an object containing the operations above

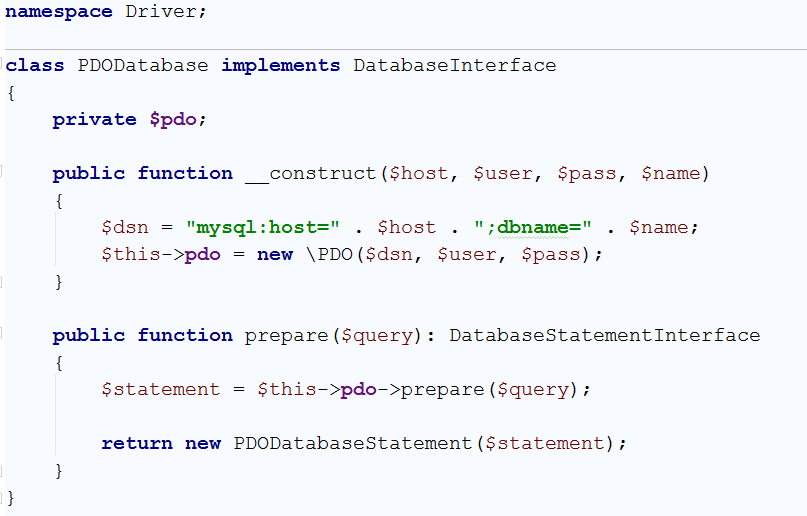
So let’s create an interface with the first three operations, called DatabaseStatementInterface



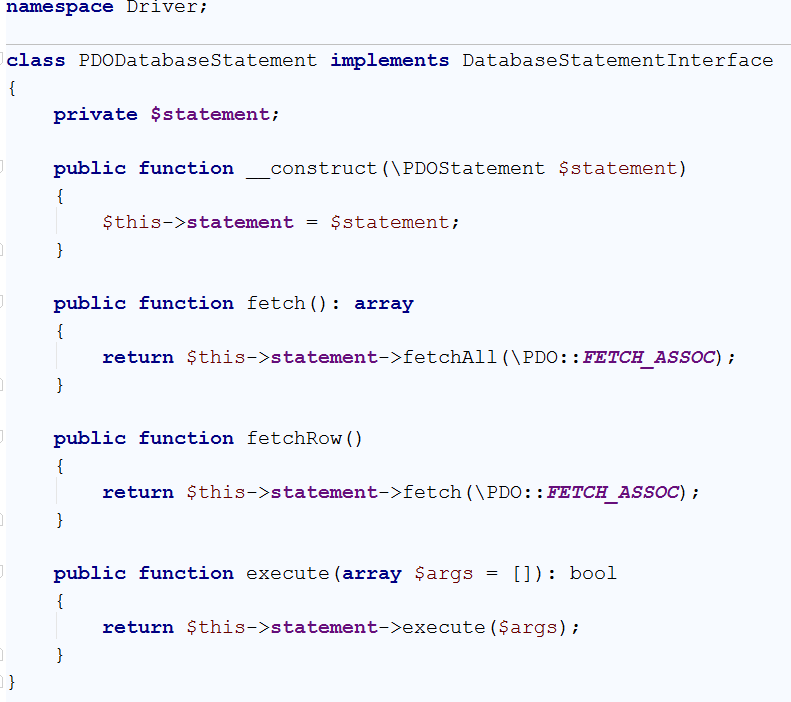
And the other interface (DatabaseInterface) that will have the **prepare** operation returning the first one. Both of them are in the Driver namespace:



We can just create a simple **PDO Wrapper** in order to implement these interfaces:

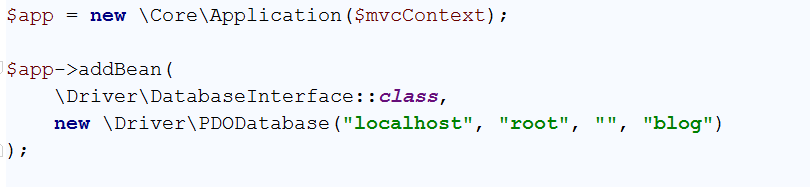


And the **Statement** respectively:



## Register the Database Class

Our PDODatabase class is now eligible for dependency injection, we just need to register it. Create an object of it in the index.php and add it as a bean via addBean() method



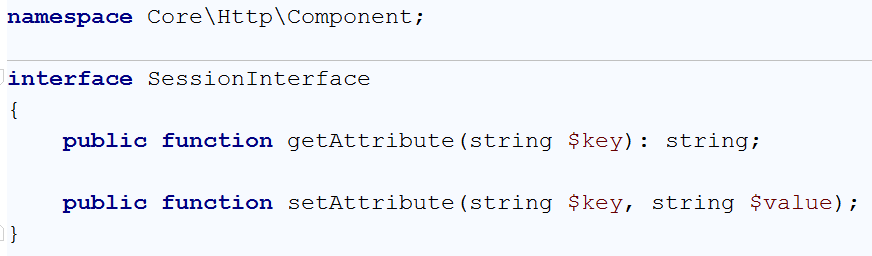
Now, wherever you try to ask for DatabaseInterface you will receive an instance of the PDODatabase class

1. **Authentication**

## Session Abstraction Layer

The authentication layer will need by all chances **session operations,** because that’s the natural way to authenticate a user in the application. As we have discussed before, **using the Superglobals is bad practice** and we need to **wrap the Session operations in a class**, which will later be eligible for dependency injection and used in our authentication service.

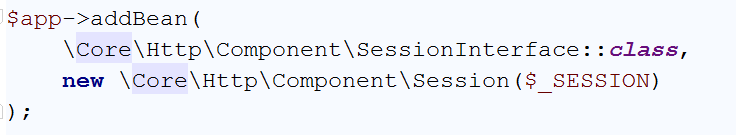
We need a simple interface that provides method for **extracting and saving session attributes**. Let’s define a SessionInterface in a namespace called Core\Http\Component



And it needs its corresponding **implementation**. The class will operate with an **array reference** (in the ideal scenario the $\_SESSION superglobal) and set/get key-value pairs in it.



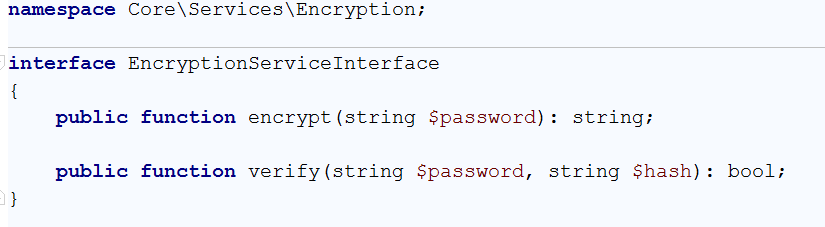
The only left thing here is to **add the Session bean to the application dependency graph**



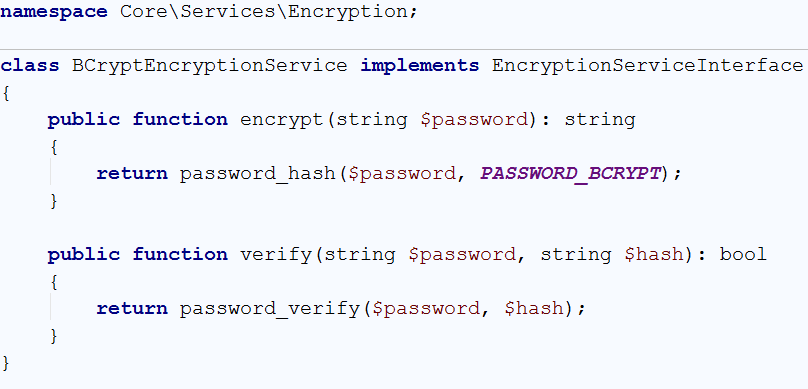
## Encryption Service

In order to authenticate the user, we need not only to use the session, but to **encrypt and verify passwords**, so we don’t store it as plain texts. We can use password\_verify/password\_hash in all places it’s used, but **once the encryption algorithm changes, we need to change it everywhere**. So we will abstract these operations in another wrapper class.

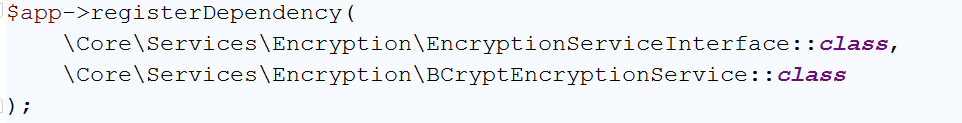
Let’s define an **interface** that provides operations for **hashing and verifying password hashes**. This will be the EncryptionServiceInterface in Core\Services\Encryption namespace



And its corresponding implementation which we will call BCryptEncryptionService



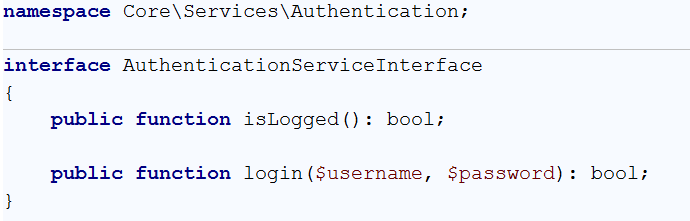
The only thing left is to **add it to the dependency graph**. As it is **context-independent** we don’t need to instantiate it (resolved dependency), so we don’t need to use the addBean() method and can simply use registerDependency()



## Authentication Service

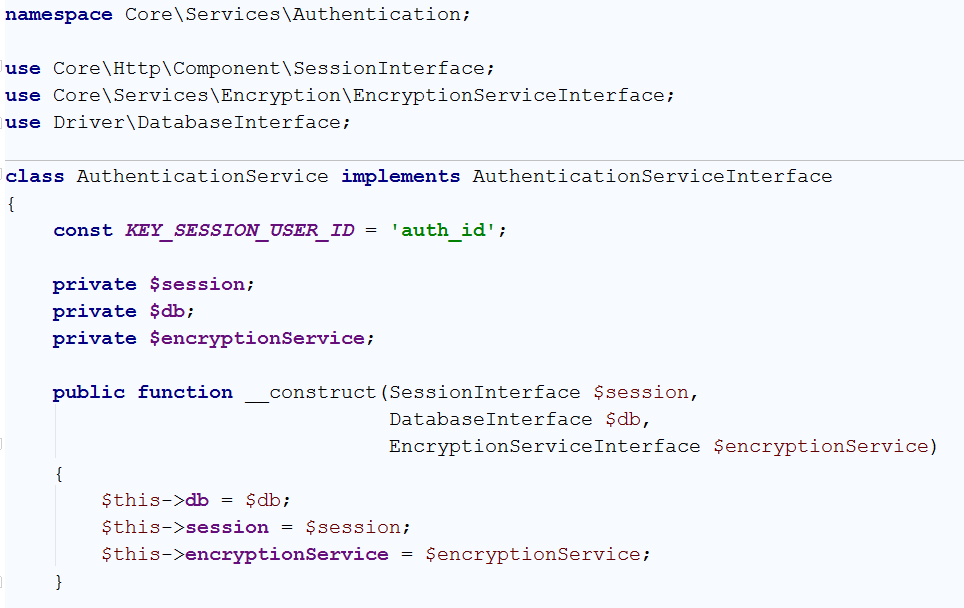
Good. All **dependencies** needed for the authentication services are already done. **The Database abstraction layer**, the **session abstraction layer** and the **encryption service.**

We are ready to define an **authentication service** that **logs in the user** **and check whether a user is logged in**. Let’s define an interface AuthenticationServiceInterface in Core\Services\Authentication namespace

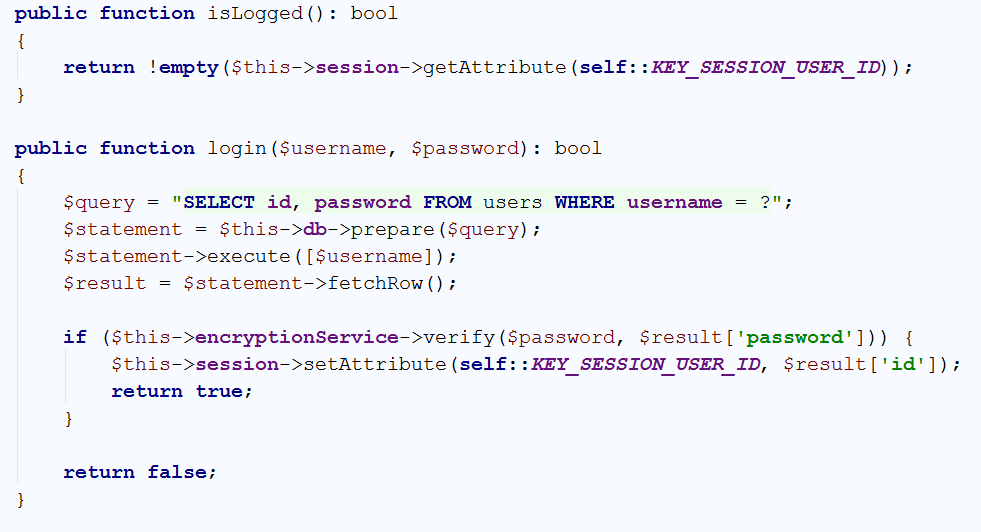


In order our implementation to login the user we need to **extract its information from the database using our DBAL**. Then we need to **check its password** if matches the password hash **using the** EncryptionService and finally **add the user id in the session using the Session Abstraction Layer**.

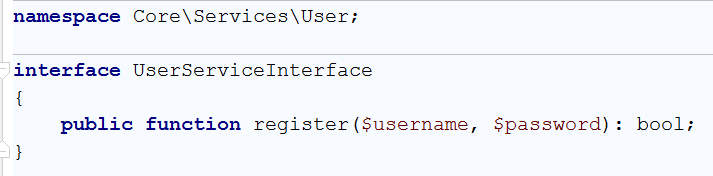
This will be possible, because we have made them **eligible for dependency injection**, which means once we require them in our constructor, **the DI mechanism will give them to us**



And we are ready to implement the methods from the interface



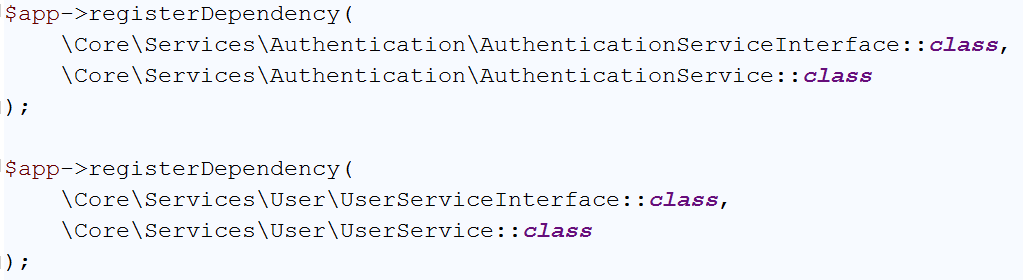
We can make the same thing in order to register users. In normal applications the **Authentication service differs from the service which registers users** (as the first comes from the generic framework and the second is user defined). So we will define a separate UserService which registers users by analogy



-



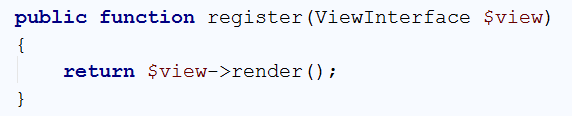
Now we need to register both Services in the dependency graph



## Registration Process

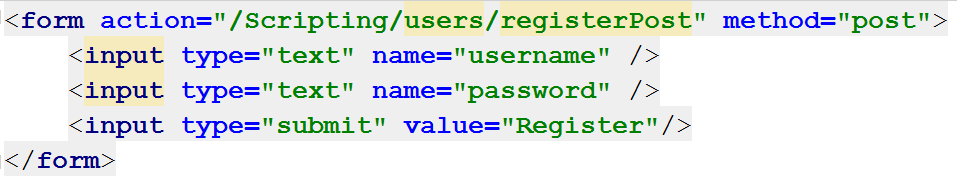
We are here to **define action in our controller which will render a view with registration form**, and another action that will handle the form posting and register the user via the services

Define a register() method in the Users controller which renders the view with the form. The method needs the ViewInterface in order to render a view

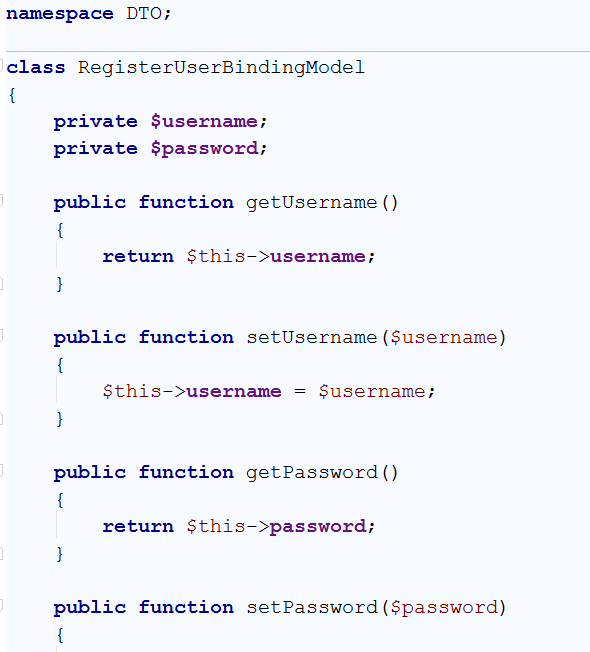


Now we need to create our html form. The form needs to post to certain URL which will be later handled. As we will **separate the GET and the POST** action, we will post to registerPost (which will be another method in the controller)

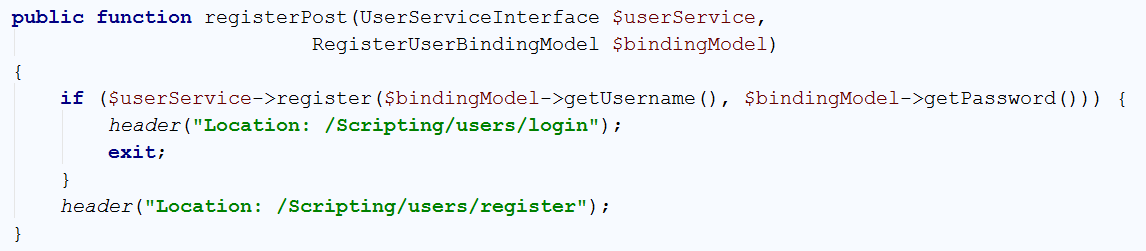
Our view resides in views/users/register.php



The form posts two fields **namely username and password**, thus we need to define a class that will have these fields for model binding. Let’s call it RegisterUserBindingModel



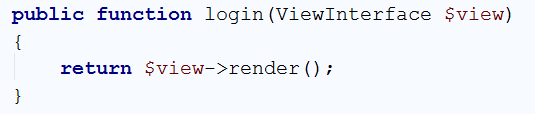
Then we need to define the so called registerPost() method which will accept the **binding model from above and the user service** in order to register it



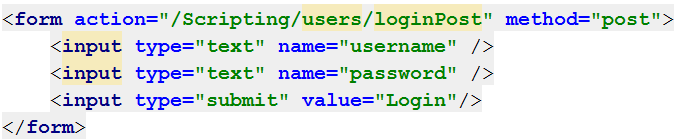
## Login Process

By analogy, we need a login form and two methods for get and post.

The GET method rendering the form:

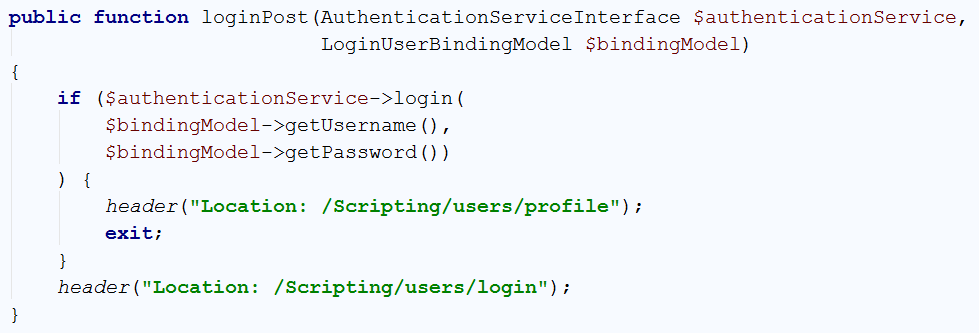


The form (views/users/login.php):



Create a binding model for login (the same as for register).

Then create a method which will handle the post, take the bound model and use the authentication service in order to authenticate the user



## Secured Page

The profile page, which we redirect from successful login should be secured and cannot be accessed from non-logged in users. So we need the authentication service once again to ask it if the user is authenticated:

