# Database Interaction - Exercises

We will use the four simple operations – create, read, update and delete (CRUD) in order to create a mini application. In other words, we will move the previous application in-memory DB to a MySQL DB.

# Move Database to MySQL

## Edit the Shared DB File

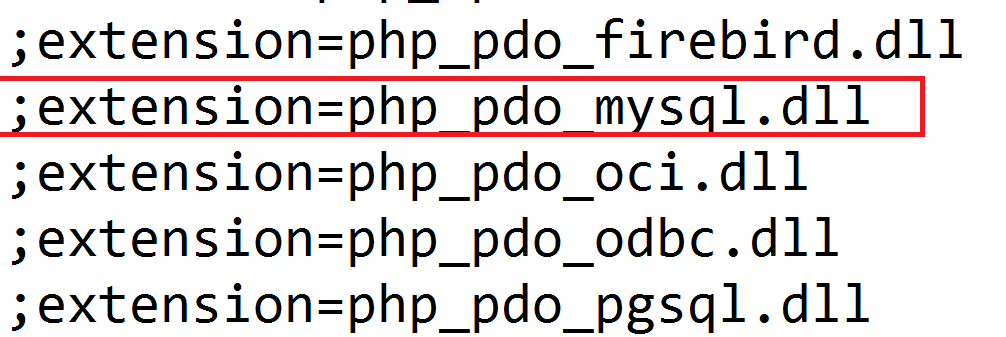
Do you remember the database.php file we have extracted in order to share the Users array? We need to change it, to return use a MySQL Connection.

There are several ways to connect to MySQL database with PHP. One of them is the oldest, deprecated way and **removed in PHP 7** – it’s called the mysql\_\* extension and defines functions **prefixed with** mysql\_ for any database operation.

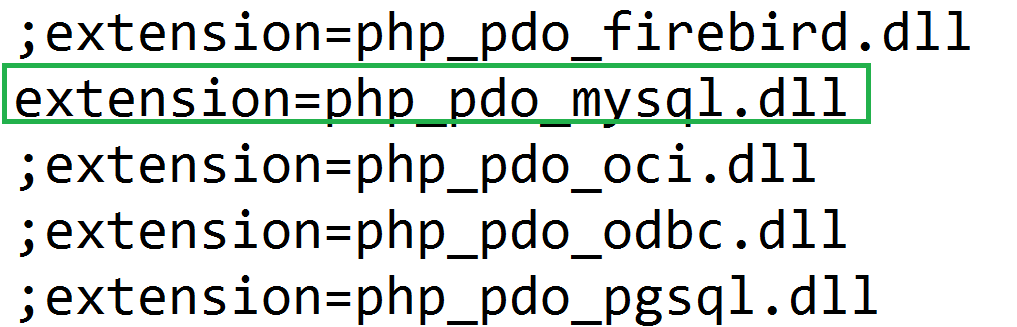
The other way is its upgrade – [mysqli\_\* extension](http://php.net/manual/en/book.mysqli.php). It supports the so called [prepared statements](http://php.net/manual/en/mysqli.quickstart.prepared-statements.php) where you contact the DB with **a valid query and then pass the arguments**, which is a significant difference than concatenating the query and building it manually as it was in the mysql\_\* extension.

The third and more generic way, the one **we will use from now on**, is the [PDO extension](http://php.net/manual/en/book.pdo.php). The PDO extension provides an object oriented generic way to contact a database with a specific database driver. We will use the MySQL driver for PDO.

First of all, we need to check that the lines activating the PDO and the MySQL driver in the **php.ini** configuration file.



If you have the semicolon before the line, like shown here, remove it, save the file and restart the Apache server.



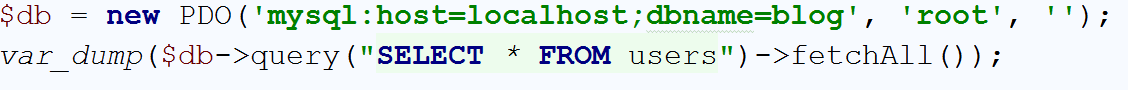
Good. Let’s test if we can connect to the database. To do that we need to know our **host** where the server is running. In this particular scenario the host is the same as the client machine, the so called **localhost.** We also need to know the **username** and the **password** to the db. In the default xampp/wamp installation the **user is “root”** and the **password is empty** (none). Last, but not least is the **database name**. In the previous lecture we have created one called “**blog**”.

When we know that information we can use it in the [DSN](http://php.net/manual/en/ref.pdo-mysql.connection.php) the PDO wants. As the example shows:

mysql:host=localhost;dbname=testdb

This means our DSN will be “mysql:host=localhost;dbname=blog”. The PDO constructor accepts two more arguments – the **username** and the **password**, as you can see [here](http://php.net/manual/en/pdo.construct.php).

With all the prepared information we can add the DB connection in database.php in order to test it.



And the result is:

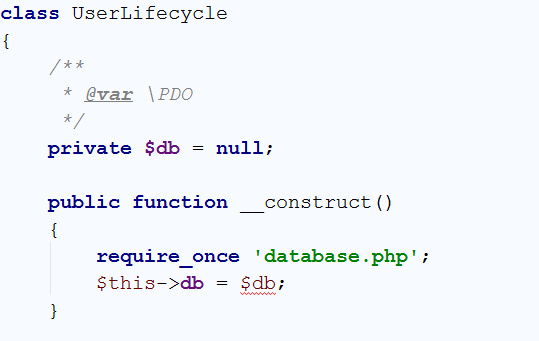


Which means we have successfully extracted our users.

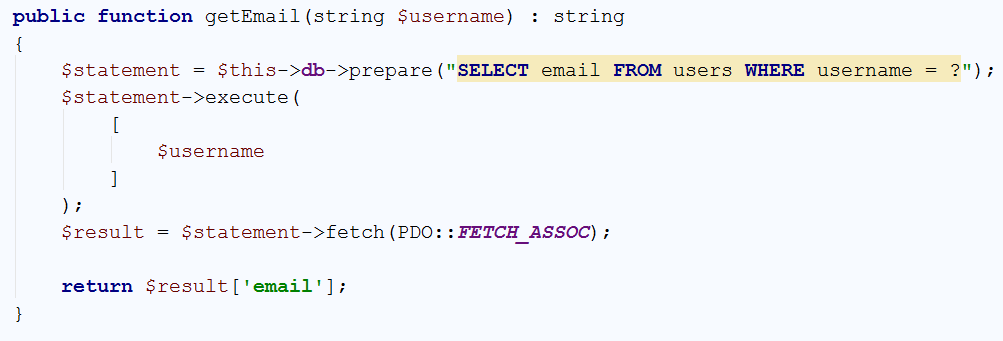
## Edit the Lifecycle Class

But we have a UserLifecycle class and we need to make it use our DB now. This means we can either pass the PDO object to the class or pass the result of all extracted users. The last one will reduce the performance since we will extract all users and perform filtrations in memory, instead of using the database optimization for filtrations.

In other words, we will pass the DB object to the UserLifecycle class. We will stick to the previous approach with declaring a DB object and requiring it in the constructor. Later in the exercises we will decorate the practice with better one.



Good. Now we will use the local field $this->db in order to contact the database. Let’s refactor our getEmail() method:



Let’s examine it statement by statement:

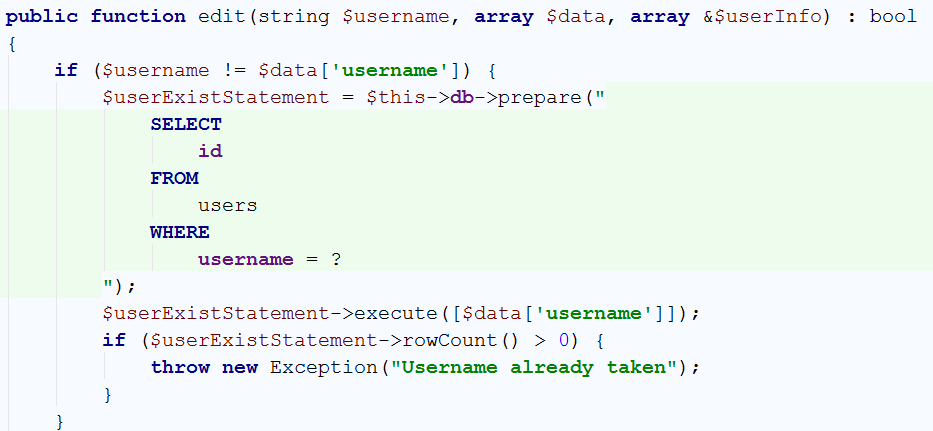
1. We use the prepare() method in order to create a prepared statement. A prepared statement is a valid query with parameters defined as placeholders (in our case we use the question mark). With “username = ?” we have said to the MySQL that we will send later an array with one argument that should map to this question mark and will be the value of the `username` column
2. We send to the MySQL database an array with as much values as placeholders are in the query. In this scenario – only one – the $username value of the argument passed to the function.
3. We have said to MySQL to return the result of the query with the given placeholders. In addition, the PDO::FETCH\_ASSOC constant denotes that we need the result as an **associative array**, this means each column name will be a key of the array and its value respectively a value.
4. We return a value from the associative array – the email column.

Change the other method by analogy.

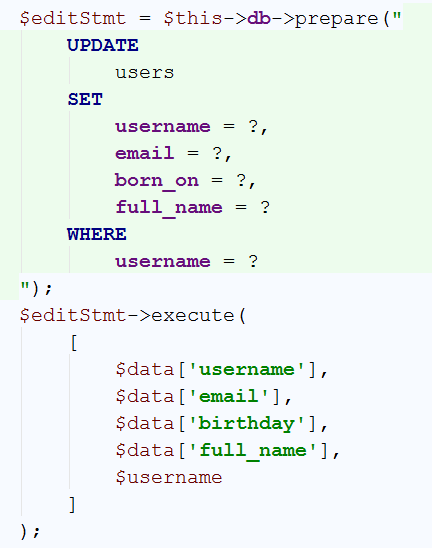
## Migrate edit() Method to Use DB

Let’s now change the edit() method:

1. If the user has tried to change its username we need to check whether the new username is not already taken
   1. This means that if the current username is different from the new one, we check the DB whether the rows returned with the new one are more than zero (exists)



1. If no exception is thrown, we need to edit the current row. We will use an UPDATE statement with multiple columns and WHERE clause. In other words: Set the username with what **username** is in the $data array (e.g. $\_POST), **email**, **birthday** and so forth, where the **username is the current username** (old one) – the $username variable



1. Then we need to preserve the old logic with changing the session username in case it was changed:



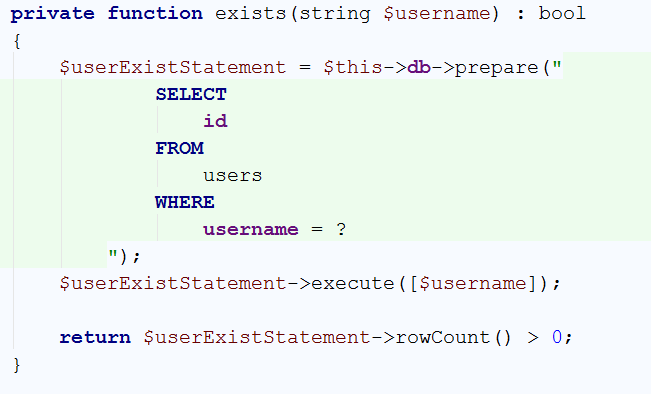
1. And return true/false whether the update was OK:



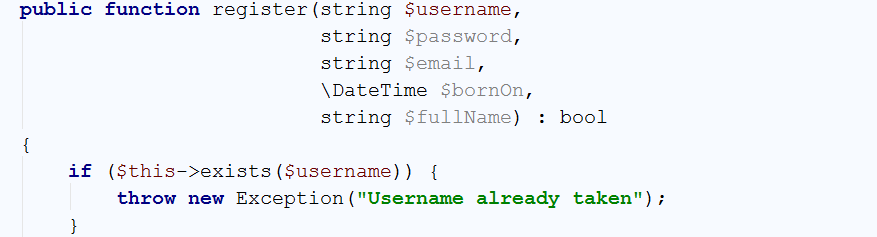
Good. We now may want to move the **login and registration** to work with the class and the db as well.

## Migrate Registration to DB

The registration process has the same check as in the user modification. Whether the username already exists. This means we can extract this to a private method called exists($username) : bool;

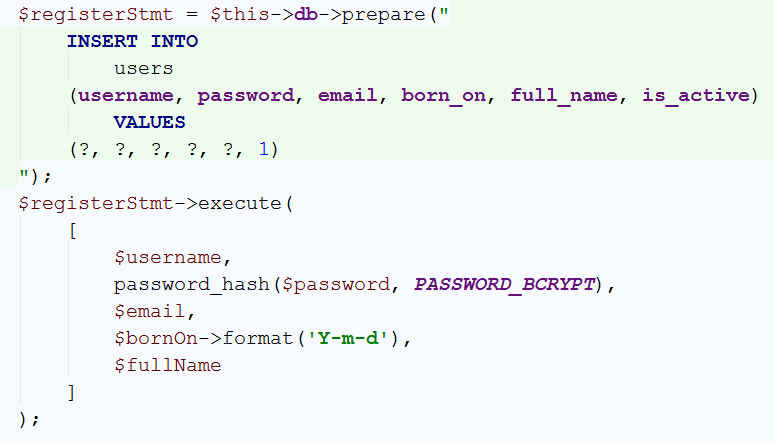


We can use it in both locations and throw an Exception. Example in register():



If everything is OK we can do a natural insertion.

**But beware!!! Would we store the passwords in plain text?! NO, WE WILL NOT! This is highly insecure and even forbidden by the cyberlaw. We will use the** password\_hash() **function to encrypt the password.**

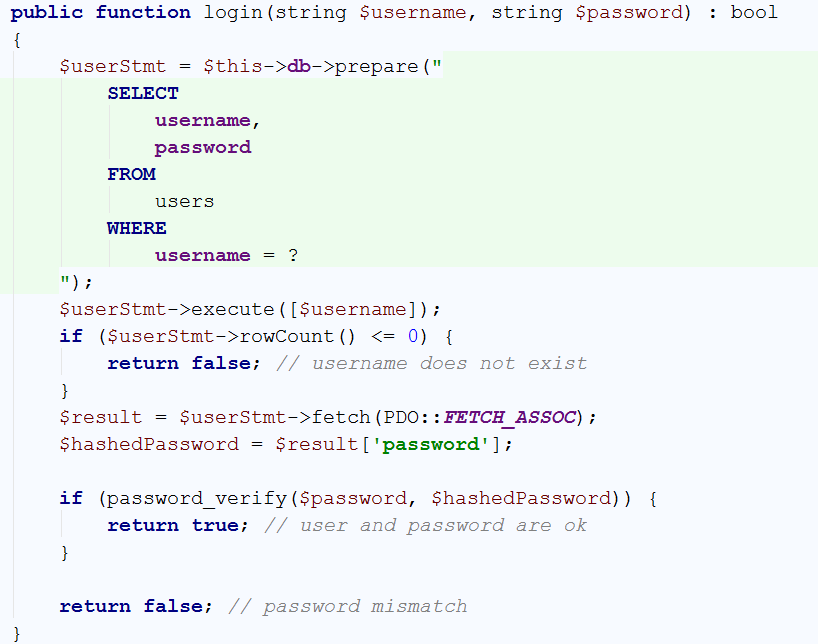
****

And the return whether we succeeded registering the user. E.g. if a new row has appeared:

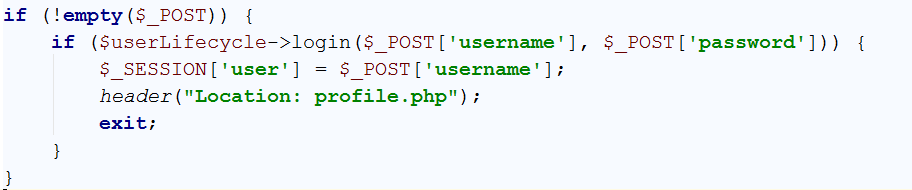


## Migrate Login to DB

The login process needs simply to verify whether certain user and password hash exists. We will extract the user from the DB and verify its password by the password\_verify() function provided by PHP.



Simple usage of the function in our login.php:



## Register Function Usage

By analogy from above make usage of the register function. You may need to create a DateTime object from the user input.

# Improve Code Quality

Some of our code is still a little bit mess. We do include \_frontend.php files, but they have an access to all the variables from the backend file, **superglobals** and so forth. Our DB object is **global** and required in the constructor of the UserLifecycle class. It does work, but by the time passes, the maintenance will become harder and harder.

## Application File

Let’s create a file which will represent our main core logic – the application file. For instance, we will call it app.php. We will extract there the common configurations like DB connection and function we will need from now on.

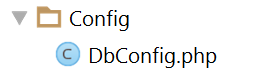
But if we move all the code there, then the application file will become unmaintainable. So let’s separate the things and glue them up in the app.php file.

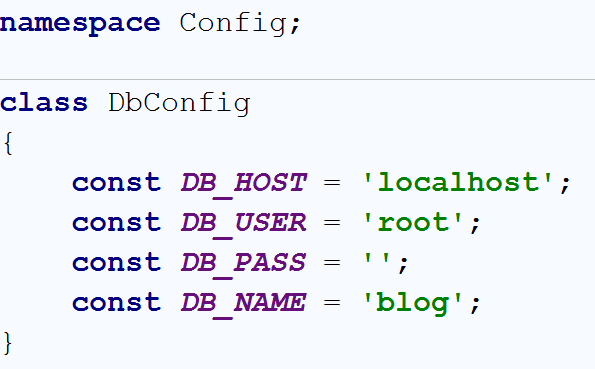
## Database Class

In this chapter you are introduced to [namespaces](http://php.net/manual/en/language.namespaces.php). In order to extract the database class, we may want to extract it in different folder. In another folder **may reside a class with the same name**. In order to **distinguish** them we will need to define different **namespaces**. It is a good practice to follow the **folder structure** for namespaces, because of [autoloading](http://php.net/manual/en/language.oop5.autoload.php) concept.

Our database class will still need the database credentials. It’s **not good to hardcode** them, as we have done, because deploying the application on machine where they are different will make us search where we have hardcoded them in order to change them.

Let’s extract a DbConfig class in Config folder holding constants with our credentials.



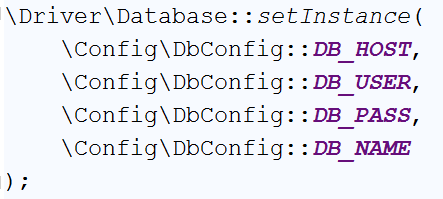


Now let’s define a Database class in Driver folder. It will accept the DB credentials and return a **PDO** object. It will be good to **set the credentials once and use the object as many times we want**. The concept is called [singleton](http://www.phptherightway.com/pages/Design-Patterns.html), but it could easily be abused. In order to not abuse it, we will still pass the object as a **dependency** and not use it as global one.

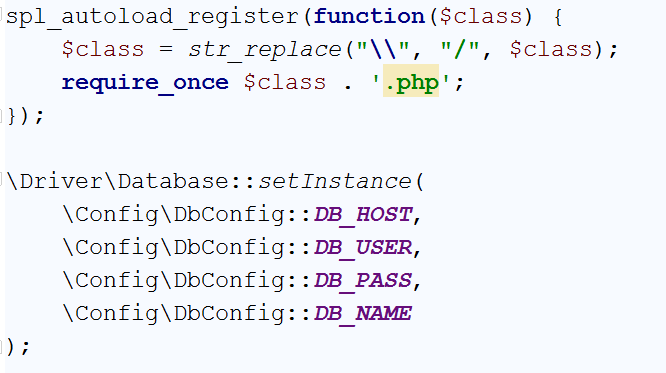


It is simple wrapper around the PDO object. Stores an array of PDO objects by $dsn + $user + $pass association and retrieves them back.

Now we can set that instance in our app.php file:



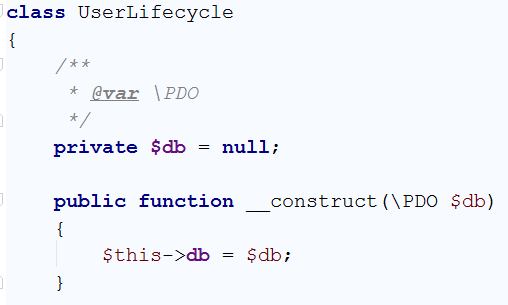
The problem here is PHP will not know where this Database class resides. We need to include it. So we need the autoloader concept here.



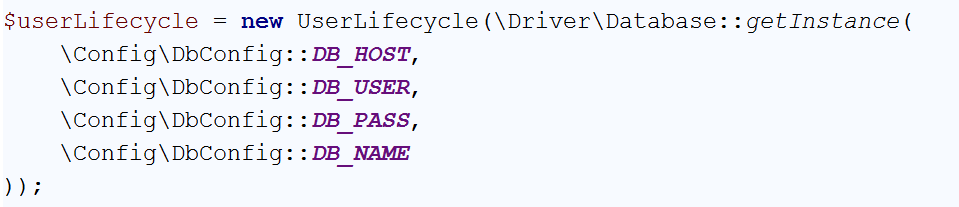
\Driver\Database will become /Driver/Database.php which is a physical file, thus the require\_once will invoke it.

## UserLifecycle Using PDO Class

Now we can move the inclusion in the constructor and introduce a dependency – a \PDO parameter in the UserLifecyce’s constructor



And instantiate it in app.php

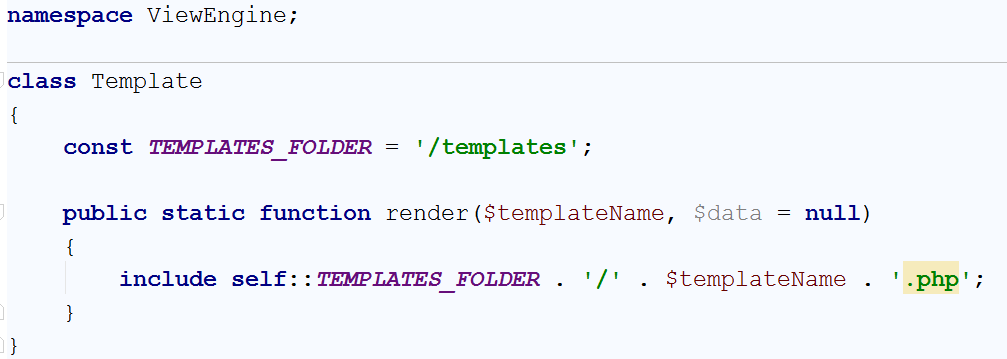


Now we can include app.php in our files instead.

## Template Rendering Method

Instead of directly including templates with global scope, let’s include them in a function and give them information via data object. A data object is a class that we will make specially for this template containing the information needed for the frontend.

Let’s define a class Template in ViewEngine folder:

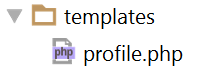


The constant defined says we need to move all of our templates in a folder names templates.

## Try Template Render Class

Let’s try it in the profile.php

Move the profile\_frontend to templates folder. Even more, name it “profile.php” again:

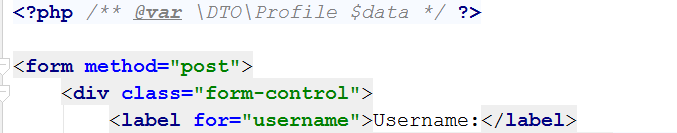


Our profile template needs information for the current **username, password and email, birthday**. Let’s create a class named Profile in DTO folder (DTO stands for Data Transfer Object):

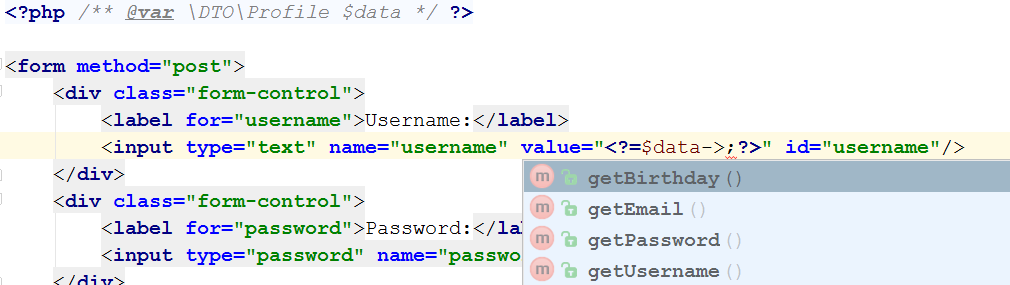


And so forth getX() methods. Use **ALT + Insert** to auto-generate them.

Now let’s define in the first line of our **template (profile)** that our $data variable will be an object of that class. This will help the autocomplete ☺



Let’s check the magic ☺



Wohooo. Let’s use it ☺



Let’s go back in the profile backend and make it eligible to render the profile template

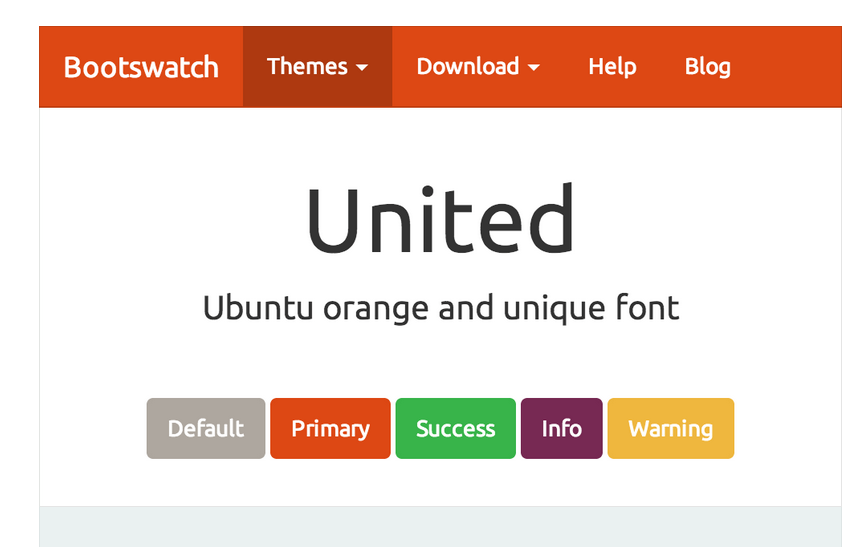


# Styling

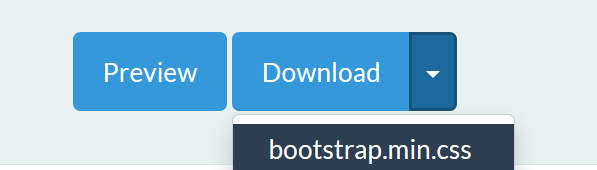
In order to style our pages we will use already defined styles in a framework - [Bootstrap](http://www.tutorialrepublic.com/twitter-bootstrap-examples.php)

## Get Bootstrap

There are already defined themes made in bootstrap. Let’s take a look at [bootswatch](https://bootswatch.com/) and catch one. For instance:



Let’s open the CSS url:

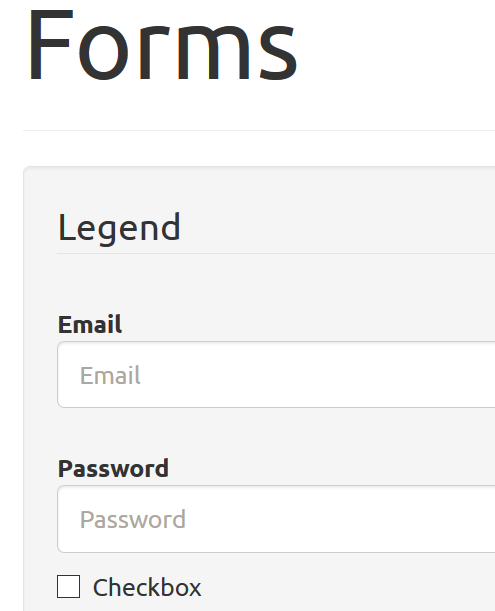


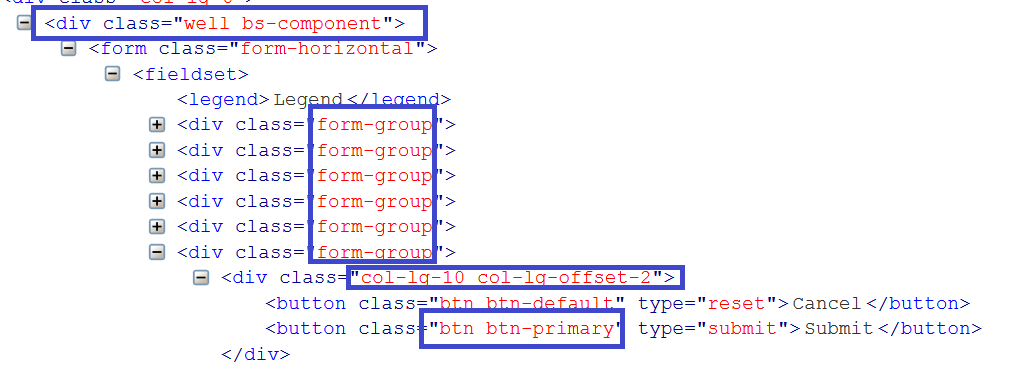


And relate the CSS stylesheet in our profile template:



Let’s see how do they make forms. Open the theme, navigate to forms and inspect what we need:

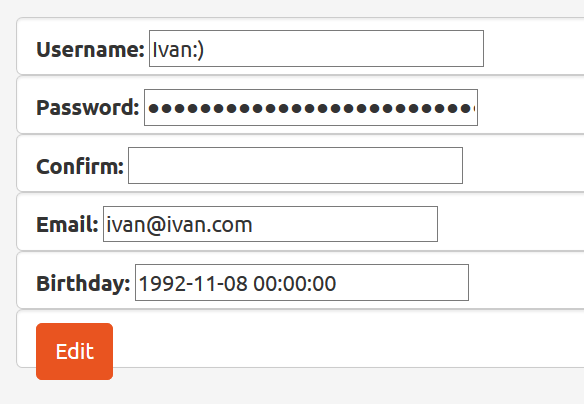




Let’s apply some of these styles. Button class btn btn-primary and wrap the form with div class well bs-component



Result:



Of course it could be much better. Go deep in bootstrap to make it look good ☺

# Do It By Yourself ☺

## Categories

When the user is logged it, render them two hyperlink: Add Category, View Categories. Clicking on Add Category renders a form that inserts a category in the Database.

Clicking on View Categories renders a list with all categories. Each category name is a hyperlink leading to, let’s say category.php?id={id}

There the user can find a list with all topics in this category

## Topics

In the topics listing in category.php render a button **“Add New Topic**”. It navigates to a form where the user can insert new topic to this category.