Informe Refactorización

Sistema de identificación y registro

En este documento se especifican los cambios realizados al código de la aplicación para su mejor entendimiento ademas de las nuevas funcionalidades integradas.

## Fecha: 18 de Febrero de 2015

## Autor: Ricardo Sánchez

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Índice

[Requerimientos de funcionamiento del sistema 1](#_Toc329186826)

[Code Smells encontrados 2](#_Toc329186827)

[Patrones de diseño aplicados 6](#_Toc329186828)

[Problemas Encontrados 9](#_Toc329186828)

[Arquitectura Hexagonal 10](#_Toc329186828)

[Resultados 11](#_Toc329186828)

[Conclusión 12](#_Toc329186829)

# Funcionalidades Requeridas

## Registro e identificación

* Vía consola de comandos: Se requiere que los usuarios puedan registrarse e identificarse via consola
* Via web: Se requiere que los usuarios puedan registrarse e indetificarse via web mediante un formulario. En este formulario se enviaran los datos a la aplicación por medio de una request POST por temas de seguridad.

## Cifrado de Contraseña

A continuación se especifican dos sistemas de cifrado de contraseña:

* Cifrado mediante el nuevo sistema de HASHING de php 5.5: Se ha implementado esta nueva funcionalidad para el cifrado de las contraseñas de los nuevos usuarios que se registren en la aplicación
* Cifrado mediante el sistema original ya implementado: Se ha mantenido dicha funcionalidad para permitir a los antiguos usurios poder identificarse sin tener que modificar el cifrado de sus actuales contraseñas.

# Code Smells detectados

## Descripción

En este apartado se describen todas las partes del código que se han considerado como malas prácticas ya que ensucian el código, dificultan su comprensión y dificultan aumentar las funcionalidades de la aplicación.

## Nombres de variables poco descriptivos

Se han detectado nombres de variables con nombres poco descriptivos, lo que conduce a una gran dificultad a la hora de entender el código

Ejemplo:

**if**($check){

**echo** "<div>Access granted.</div>";

}

El nombre de la varible check es poco descriptivo y no sabemos que comprueba,

Una buena solución sería renombrarla como “isValidUser”.

**if**($isValidUser){

**echo** "<div>Access granted.</div>";

}

## Excesivos Comentarios

Los Comentarios En el codigo no termina de ser una mala practica del todo si es para casos que consideramos que es dificil enter una parte del codigo, pero por norma general y debemos utilizar comentarios es por que el nombre de nuestras variables, metodos o clases no son suficientemente descriptivas, ademas de hacer el programador utilizar mas tiempo para tener que leer todos esos comentarios.

La solución aportada ha sido eliminar todos los comentarios y crear un código que sea facil de interpretar.

## Uso excesivo de hardcodeos (Magic numbers y Magic Strings)

En muchas partes del código se dejan texto o números donde los cuales se repiten o si hay que modificarlos es incomodo tener que ir al código, y ver donde se usan y cambiarlos.

Ejemplo:

$host = "localhost";

$db\_name = "practica";

$username = "root";

$password = "root";

En este ejemplo la solución aportada pasa por sacar los parámetros de conexión a un fichero de configuración donde si fuera necesario cambiarlos, de una sola vez podríamos aplicarlos a todas las partes de nuestro programa donde sean necesarios.

Se han dado otras situaciones como texto y números puestos dentro del código directamente, la solución ha pasado por sacarlos a constantes en la parte superior del fichero.

## Bloques de condicionales demasiado grandes

En esta situación nos hemos encontrado con bloques muy grandes de condicionales, eso suele pasar por que se hacen demasiadas cosas dentro de ellos:

Ejemplo:

**if**($num==1){

$row = $stmt->fetch(PDO::FETCH\_ASSOC);

$storedPassword = $row['password'];

$salt = "ilovecodeofaninjabymikedalisay";

$postedPassword = $\_POST['password'];

$saltedPostedPassword = $salt . $postedPassword;

$hasher = **new** PasswordHash(8,**false**);

$check = $hasher->CheckPassword ($saltedPostedPassword,$storedPassword);

**if**($check){

**echo** "<div>Access granted.</div>";

}**else**{

**echo** "<div>Access denied. <a href='login.php'>Back.</a></div>";

}

}

En este ejemplo la solución aportada pasa por sacar poner cláusula de guarda donde comprobar al principio de lo condicionales para asegurarnos que no nos afectaran durante la ejecución del resto del programa, además de separar la lógica en pequeñas funciones fuera del condicional, de esta manera quedaría todo muchas más legible.

## Código Duplicado

En algunas partes del código se puede ver código duplicado.

Ejemplo:

// $check variable is false, access denied.

**else**{

**echo** "<div>Access denied. <a href='login.php'>Back.</a></div>";

}

}

// no rows returned, access denied

**else**{

**echo** "<div>Access denied. <a href='login.php'>Back.</a></div>";

}

Pasa solucionarlo se ha procedido a sacar el código duplicado a métodos

Public function printError(){

**echo** "<div>Access denied. <a href='login.php'>Back.</a></div>";

}

## Concentración de responsabilidades

Las Ficheros de Login y Register contienen casi toda la lógica de la aplicación. Según el principio “SRP” cada una debería tener una única responsabilidad

La solución ha pasado por extraer la lógica en clases donde cada una tiene usa única responsabilidad. Separando de manera individual las siguientes clases e interfaces:

* RequestMethodinterface -> Interfaz encargada de definir los métodos de las clases que se encargaran de manejar el tipo de request y así saber cómo tratar los datos recibidos del usuario.
* ResponseMethodInterface -> Interfaz encargada de definir los métodos de las clases que se encargaran de manejar el tipo de respuesta y así saber cómo tratar los que han de ser enviados al cliente..
* UserStorageInterface -> Interfaz encargada de definir las funciones de manejo de la persistencia de datos, de esta manera todas las clases utilizadas para el acceso a datos implementaran esta interfaz aislando totalmente la lógica de acceso a datos del resto de la lógica del programa.
* UserLoginUseCase -> Esta clase de encarga de la lógica principal de la parte del login de usuario, según la arquitectura hexagonal se corresponde con la capa de los casos de uso define los métodos para validar la contraseña además del método “execute()” encargado de ejecutar la lógica principal del login de usuario.
* UserRegisterUse Case -> Esta clase se encarga de la lógica principal de la parte de registro del usuario, define los métodos preparar la contraseña para su guardado y además de ejecutar el método principal “execute()” que ejecuta toda la lógica principal del registro de usuario

# Patrones de diseño aplicados

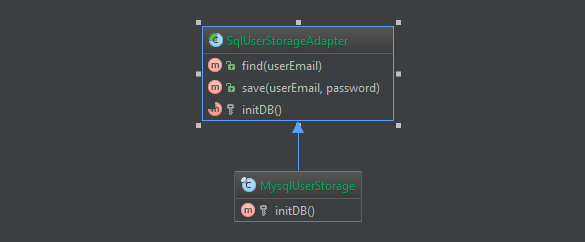
## Template Method Pattern

Este patrón nace de la necesidad de extender determinados comportamientos dentro de un mismo algoritmo por parte de diferentes entidades. Es decir, diferentes entidades tienen un comportamiento similar pero que difiere en determinados aspectos puntuales en función de la entidad concreta.

La solución que propone el patrón Template Method es abstraer todo el comportamiento que comparten las entidades en una clase (abstracta) de la que, posteriormente, extenderán dichas entidades. Esta superclase definirá un método que contendrá el esqueleto de ese algoritmo común (método plantilla o template method) y delegará determinada responsabilidad en las clases hijas, mediante uno o varios métodos abstractos que deberán implementar.

En nuestro Caso hemos aplicado este patrón en la lógica de persistencia, el diseño ha pasado por diseñar una clase abstracta (SqlUserStorageAdapter) el cual se encarga de definir e implementar las siguientes funciones:

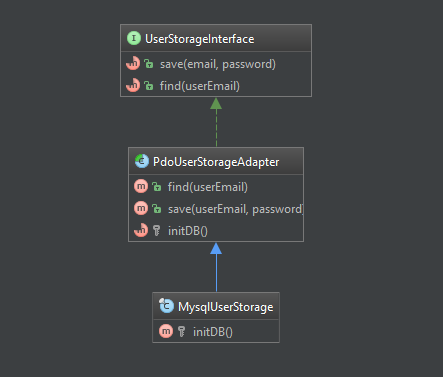
* **Find()** -> Esta función se encarga de buscar a un usuario almacenado a partir de su email y devolverlo,
* **Save()** -> Este método se encarga de almacenar un usuario en base de datos con su email y su contraseña.
* **initDB()** -> Este función ser implementada por las clases hijas las cuales se encargan



De esta manera solucionamos el problema de en caso de requerir otro tipo de conexión sql (tipo mysqli) tan solo habría que crear otra calse hija de SqlUserStorageAdapter la cual implementara el método initDB de manera especifica de mysqli. Y en el caso de necesitar otro tipo de persistencia que no fuera sql (tipo redis) habría que crear otra clase abstracta que implementara la interfaz “userStorageInterface” y nuestra lógica de persistencia seguiría funcionando perfectamente estando desacoplada de la lógica de nuestra aplicación respetando el principio de “OPEN CLOSE PRINCIPAL”

## Bridge Pattern

El patrón Bridge es un patrón estructural para desacoplar una abstracción de su implementación, de manera que ambas pueden ser modificadas independiente sin necesidad de alterar por ello la otra.



El problema nos surge a la hora de tener que utilizar las funciones find() y save(), ya que tendriamos que replicar su su utilidad practicamente en todos los casos de persistencia de nuestra aplicación.

En nuestro caso la solución es sencilla, realizamos una clase base abstracta para los tipos de conexión y en la interfaz los metodos que irian a duplicarse. La magia de esto es que la abstracta tiene la propiedad “conexión a BD” lo cual cualquier clase que extienda de ella va a poder utilizar sus propiedades, de esta manera, la logica de inicializar esa conexión es la unica que difiere de los tipos de conexión, asique el metodo de inicializarlo lo implementan las clases hijas.

# Problemas Encontrados

## Compatibilidad php 5.5 y php 5.4

Según los requerimientos de la aplicación, Se requería utilizar las nuevas funcionalidades de hashing de PHP 5.5 para el cifrado de contraseñas con el inconveniente de que algunos de los servidores que se podrían utilizar aun no dispondrían de esa versión de PHP, y seguirían usando PHP 5.4.

Con esto nos surge el problema de poder compatibilizar las librerías de php 5.5 para hashing de cifrado de contraseñas con servidores con php 5.4

La Solucion elegida para solvertar este problema ha sido incluir una librería llamada “password\_compat”, descargada de la siguiente dirección:

<https://github.com/ircmaxell/password_compat>

Según su documentación esta librería consigue compatibilizar las nuevas funciones de hashing de contraseñas de PHP 5.5 con versiones a partir de la versión 5.3.7 de PHP.

Para su correcto funcionamiento tan solo es necesario incluir la librería en Composer así nos aseguramos de incluir esa dependencia para el correcto funcionamiento de la aplicación.

Para comprobar su funcionamiento se han realizado pruebas en un Sistema Operativo CentOs con Apache y PHP 5.4.

# Arquitectura Hexagonal

## Aplicación en nuestro proyecto



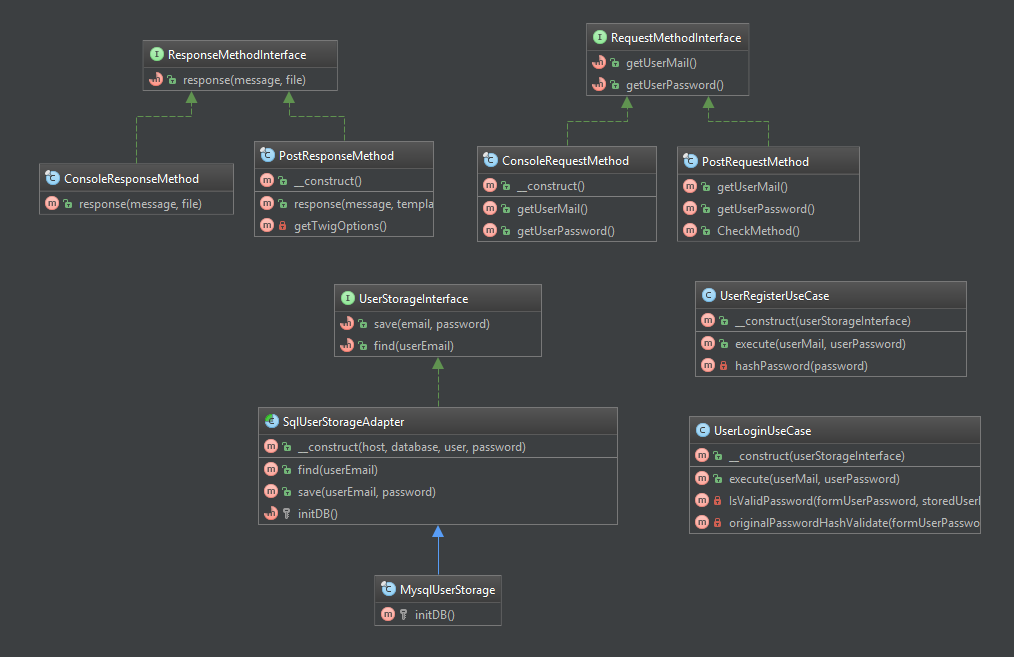
Se ha decidido utilizar la arquitectura hexagonal ya que nos permite aislar la lógica de negocio de las capas externas haciéndolas agnósticas de la infraestructura.

De esta manera hemos podido aislar la parte de las solicitudes de los usuarios para registrarse o identificarse de los casos de uso como es validar al usuario como de permitirle registrarse.

Y por último completamente desacoplado, está la parte de la persistencia de datos donde solo se hace uso de ella a través de los controladores.

# Resultado

Como resultado final este es el diagrama de la aplicación.



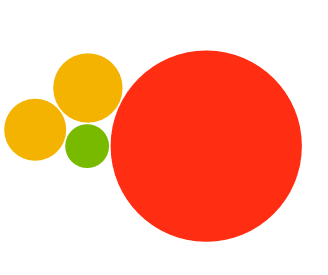
Como se puede observar en el diagrama se ha desacoplado tres módulos además de dos casos de uso:

* **Request** -> El objetivo de este módulo pasa con recoger la request del usuario donde deben venir el usuario y contraseña tanto desde el navegador o por consola de comando, esta responsabilidad pasa por definir dos clases diferentes donde una trate el formulario web y la otra la entrada de datos por consola, ambas dos implementando una interfaz que les obliga a definir tanto el email como la contraseña que son los dos campos requeridos para poder realizar la identificación o el registro.
* **Response** - >El objetivo de este módulo pasa por definir la lógica de devolver una respuesta al usuario indicando el resultado de su solicitud, en función de si está realizándolo a través de consola o de formulario web. , las clases constan de un único método encargado de enviar la respuesta, el cual están obligados a implementar ya que tienen que cumplir el contrato con la interfaz.
* **UserStorage** - > El objetivo de este módulo es poder tratar la persistencia de los datos de usuario, la cual consta de dos principales objetivos, comprobar si existe el usuario a la hora de identificarse y guardar el usuario en el caso de que quiera registrarse, estos dos métodos están definidos en una interfaz.
* **UserLoginUseCase** .-> El objetivo de este módulo cosiste en validar el login de usuario, utilizando el módulo de persistencia comprueba que existe el usuario y que su contraseña en correcto, el caso de que sea incorrecto simplemente lanza una excepción si no, no devuelve nada simplemente permite la continuidad de la ejecución del programa.
* **UserRegisterUseCase** -> El objetivo de este módulo consiste en registrar un usuario y almacenarlo, utilizando el módulo de persistencia hashe la contraseña para ls seguridad y la almacena justo al email.

# Conclusión

Tras haber refactorizado la aplicación completamente hemos querido analizar la calidad del código mediante PHPMETRICS y así poder ver en detalle cuales han sido los las mejoras en la calidad del código frente a la versión original de sistema de registro y validación.

VERSION ORIGINAL: En la versión original se puede observar en el grafico, la aplicación estaba compuesta por ficheros con una gran cantidad de líneas de código donde toda la responsabilidad recaía en un unico fichero



VERSION FINAL: Como se puede observar en el gráfico, la actual versión ha mejorado en consideración ya que reparte las responsabilidades del programa permitiendo así añadir más funcionalidades si fuera necesario de manera sencilla y dejando la menor deuda técnica posible.

