Ejercicio 3:
Estructura del Proyecto

Acerca del ejercicio

En este ejercicio partiremos de una posible solución del ejercicio 2 y organizaremos y mejoraremos el código del proyecto con el objetivo de lograr una estructura adecuada para un proyecto PHP de tamaño medio.

Paso 1: Reorganización de archivos públicos y privados

*NOTA PREVIA: Descarga el archivo **01-inicio.zip** del campus virtual.*

Las convenciones habituales en el desarrollo con PHP (ver PSR1 y PSR-12: https://www.php-fig.org/psr/) establecen que es necesario hacer una distinción en los scripts PHP (al menos) en varios tipos:

- **Scripts de vista**: aquellos que generan salida para el usuario.
- **Scripts de apoyo para vistas**: Son scripts que se incluyen exclusivamente en los scripts de vista, normalmente con la finalidad de reutilizar o bien HTML o funciones de apoyo para generar el HTML de las vistas (cabeceras, menús, etc.)
- **Scripts de lógica o definiciones**, que exclusivamente contienen definiciones de funciones, clases, etc.

Esta separación, además de permitir organizar el código en una primera aproximación, permite dar los primeros pasos para poder reutilizar el código entre diferentes proyectos y restringir el acceso a los usuarios finales a aquellos archivos de la aplicación que no deban verse externamente (e.g. los que tienen datos sensibles como contraseñas de la BD).

Para organizar los scripts también es recomendable crear varias carpetas dentro del proyecto. Al menos deberían existir las siguientes carpetas:

- **includes**: Como el propio nombre indica, esta carpeta está pensada para que contenga scripts PHP que se van a incluir (include / require) en otros scripts.
- **mysql**: En esta carpeta podemos almacenar los diferentes archivos .sql necesarios para crear, borrar y cargar datos de la BD.

Una posible organización para el proyecto podría ser la siguiente:

`X:\.....\ejercicio3\`

- **`includes\`** : Aquí estarán los scripts que no son de vista
  - **`vistas\:`** Aquí estarán los scripts de las vistas
    - **`vistas\comun\`** : Aquí estarán los scripts de apoyo para las vistas
- **`mysql\`** : Aquí estarán los .sql
- **`index.php`**

- … otros ficheros que se invocarán en el sitio y que generan las vistas

El objetivo de este apartado es:

1. Crear las carpetas necesarias para tener una organización como la anteriormente descrita: `includes, includes\vistas y mysql`
2. Colocar los scripts, atendiendo a su tipo, en las diferentes carpetas (en este paso los de cabecera, pie y sidebars en la carpeta `vistas\comun`).
3. Realizar las modificaciones oportunas a los scripts para que la aplicación siga funcionando.
4. Importar la base de datos usando la información incluida en la carpeta **mysql**.

Adicionalmente, habrá que restringir el acceso a las carpetas a las que no debe tener acceso el usuario final (includes y mysql), para lo cual se puede crear en cada uno de estos directorios un archivo **`.htaccess`** de configuración de Apache y que incluya las directivas de Apache necesarias para restringir el acceso.

Paso 2: Plantillas para las vistas

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **02-inicio.zip**.*

Pese a que en el ejercicio2 has practicado como organizar las vistas para simplificarlas, sigue existiendo bastante contenido HTML que está repetido en todos los scripts de vista.

Además, si nos fijamos bien podemos ver que la diferencia en el HTML en las diferentes vistas varía exclusivamente en la etiqueta `<title>` y en el contenido de la etiqueta `<article>`.

Una manera de paliar este problema consiste en usar un script PHP a modo de plantilla (en entornos más avanzados se utilizaría una librería de plantillas como Twig) donde simplemente seguimos la convención de utilizar una serie de variables que alojarán el contenido de las secciones que cambian en la plantilla. Por ejemplo:

```
// plantilla.php
<!DOCTYPE html>
<html>
  <head><title><?= $tituloPagina ?></title></head>
  <body>
    …
   <?= $contenidoPrincipal ?>
  …
</html>

// index.php
<?php
```

```
  $tituloPagina = 'Portada';
  $contenidoPrincipal = <<<EOS
    <p>Ejemplo de contenido</p>
  EOS;

include 'plantilla.php';
```

Esta plantilla sirve a modo de layout (distribución) de las páginas de la aplicación. En algunas aplicaciones más complejas es necesario tener más de un layout para las páginas, simplemente se crearían más archivos plantilla1.php, plantilla2.php, etc., y se incluiría en la vista la plantilla que se quiere utilizar como base.

Por otro lado, el uso de las plantillas también permite repensar como organizar las vistas y lógica de la aplicación y evitar uno de los problemas más habituales en PHP el error "Warning: Headers already sent" que es debido habitualmente a que se genera HTML antes de terminar de procesar la lógica asociada a la petición, por ejemplo, un formulario. El uso de las plantillas nos fuerza a que vayamos acumulando en variables y acumulando poco a poco los diferentes componentes de las vistas.

El objetivo de este apartado es:

1. Crear el script "`includes/vistas/plantillas/plantilla.php`" donde se defina el contenido de la única plantilla que es necesaria para este ejercicio. Asegúrate de utilizar `$tituloPagina` y `$contenidoPrincipal` en la plantilla.
2. Modifica los scripts PHP para inicializar `$tituloPagina` y `$contenidoPrincipal` y finalmente incluir la plantilla.

Paso 3: Configuración e inicialización de la aplicación

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **03-inicio.zip**.*

Es habitual que antes de poder procesar la petición del usuario se realice algún tipo de inicialización o configuración de la aplicación, por ejemplo, para poder utilizar la sesión del usuario, invocar

```
session_start();
```

Otros aspectos relevantes a configurar son el soporte UTF-8 de PHP para gestionar las peticiones del usuario. El código PHP necesario es:

```
/**
 * Configuración del soporte de UTF-8, localización (idioma y país)
 */
ini_set('default_charset', 'UTF-8');
setLocale(LC_ALL, 'es_ES.UTF.8');
```

Por otro lado, si el proyecto gestiona fechas y horas, es recomendable también indicar la zona horaria de referencia del servidor. El código PHP necesario es:

```
date_default_timezone_set('Europe/Madrid');
```

El objetivo de este apartado es:

1. Crear el script `includes\config.php` para incluir todo el código necesario para inicializar la aplicación antes de procesar la petición del usuario.
2. Modificar los scripts PHP de la raíz del ejercicio3 para que como primera instrucción se requiera el archivo `includes\config.php` y eliminar el código que ya no es necesario en estos scripts.

Paso 4: Centralización de la gestión de conexiones a BD y otras operaciones de la aplicación: clase Aplicacion

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **04-inicio.zip**.*

Si revisas los scripts `procesaLogin.php` y `procesaRegistro.php` observarás que hay varias instrucciones PHP que son iguales. Estas instrucciones son las necesarias para crear una conexión a la BD e inicializarla de manera adecuada para que utilice la codificación de caracteres `utf8mb4` necesaria para almacenar cualquier carácter UTF-8 en MySQL.

Por otro lado, hay otro problema y es que los datos de la conexión a la BD (usuario, contraseña, etc.) se incluyen en todos los scripts. Este problema es potencialmente peligroso ya que, si existe un error en nuestro script, el usuario final de la aplicación podría llegar a ver los datos de conexión, y si la configuración del servidor no es la adecuada o tenemos un phpMyAdmin instalado en el mismo servidor, un potencial atacante podría destrozar nuestra BD.

Finalmente es recomendable ser cuidadosos con la gestión de conexiones a la BD. Las conexiones a la BD requieren bastantes recursos (memoria) en el servidor (tanto en Apache como en MySQL), además el servidor de BD suele tener configurado un límite máximo de conexiones. Finalmente, cabe destacar que el establecer una conexión entre PHP y MySQL requiere de cierto tiempo que hay que añadir al tiempo de ejecución de nuestro código por lo que es interesante no abrir una nueva conexión a la BD cada vez que queremos realizar una operación contra la misma. Por tanto, es recomendable que por cada petición del usuario se mantenga como mucho 1 conexión por petición.

Una manera de solucionar este problema es crear una clase `Aplicacion` que implemente el patrón Singleton[1] de modo que la única instancia de `Aplicacion` se encargue de gestionar la conexión a la BD.

El objetivo de este apartado es:

[1] http://www.phptherightway.com/pages/Design-Patterns.html

1. Crear una clase `Aplicacion` que implemente el patrón Singleton. Para ello, crea una variable privada static donde almacenes la instancia de la clase y crea una función static getInstance, que devolverá la instancia de la clase si existe la instancia, y en caso de que no exista creará una instancia de la clase.
2. Implementar un método `init($bdDatosConexion)` que permita inicializar la instancia de la aplicación.
    - Se sugiere que `$bdDatosConexion` sea un array con todos los datos necesarios para crear una conexión a la BD. Por ejemplo:

```
array('host'=>'xxxx', 'bd'=>'yyyy', 'user'=>'zzzz', 'pass'=>'uuuu')
```

    - Puedes incluir la llamada a session_start() en este método, o dejarla en config.php.
3. Implementar un método getConexionBd() que se encargue de crear una conexión con el servidor MySQL y devolverla, o si ya existe una devolver la que ya exista.
4. Modificar `procesarLogin.php` y `procesarRegistro.php` para utilizar la clase `Aplicacion` que acabas de crear. Recuerda que debes llamar a `init($bdDatosConexion)` en algún momento. Por ejemplo, un buen sitio puede ser en config.php, ya que así nos ahorramos tener que inicializar la aplicación explícitamente en varios scripts diferentes.

Paso 5: Reorganización de la funcionalidad asociada a la gestión de los usuarios: clase Usuario

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si te atascas descarga el archivo **05-inicio.zip**.*

Llegado este apartado del ejercicio, la funcionalidad asociada a la gestión de los usuarios (dejando aparte la gestión de los formularios PHP) en al menos dos scripts `procesarLogin.php` y `procesarRegistro.php`. En una aplicación real esta situación se complicará aún más al añadir otra funcionalidad como la edición del perfil del usuario, etc. Además, cambios en los detalles de implementación asociados a los usuarios como un cambio de diseño en la BD o en la gestión de las contraseñas, requeriría tocar varios scripts PHP.

Para evitar esta situación que es poco deseable, la solución consiste en definir la clase Usuario que se encargue de aglutinar toda la funcionalidad propia de la gestión de los usuarios (e.g. acceso a las tablas de la BD de usuarios, gestión de passwords, etc.).

El objetivo de este apartado es:

1. Crear la clase Usuario con al menos los siguientes métodos:

- `public static function buscaUsuario($nombreUsuario).` Devuelve un objeto Usuario con la información del usuario $nombreUsuario, o false si no lo encuentra.
- `public function compruebaPassword($password).` Comprueba si la contraseña introducida coincide con la del Usuario.
- `public static function login($nombreUsuario, $password).` Usando las funciones anteriores, devuelve un objeto Usuario si el usuario existe y coincide su contraseña. En caso contrario, devuelve false. Para esta función te será útil la función de PHP "password_verify".
- `public static function crea($nombreUsuario, $nombre, $password, $rol).` Crea un nuevo usuario con los datos introducidos por parámetro. Para esta función te será útil la función definida en PHP5 "password_hash".

2. Modificar los scripts procesarLogin.php y procesarRegistro.php para que utilicen la clase Usuario.

Paso 6: Reorganización de la gestión de formularios: clase Formulario

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si te atascas descarga el archivo **06-inicio.zip**.*

Las etapas principales de gestión de un formulario HTML son muy concretas, de modo que es posible utilizar el patrón TemplateMethod[2] para definir esta lógica una única vez y simplificar la creación y gestión de formularios de toda la aplicación.

Descarga el archivo **06-Formulario.zip** del CV y descomprímelo dentro de \includes para que puedas utilizar la clase **Formulario** dentro de tu aplicación. El método plantilla es el método `public function gestiona()` de dicha clase. Es recomendable que analices el código de éste y el resto de métodos (especialmente `generaFormulario()` y `generaErroresCampos()`) para que comprendas como funciona, y en particular, cómo se gestionan los errores del formulario.

Para utilizar la clase Formulario dentro de la aplicación, además de crear una subclase por cada formulario que tengas que gestionar en la aplicación, también es necesario cambiar la metodología de trabajo. Llegado este punto del ejercicio la mayoría de scripts de vista están organizados por pares (login, procesarLogin), (registro, procesarRegistro). Al utilizar las subclases de la clase Form ya no es necesario utilizar los scripts procesarXXXX sino que el mismo script gestiona las peticiones GET (cuando el usuario lo visita por primera vez) o las peticiones POST (cuando el usuario envía el formulario).

El objetivo de este apartado es:

1. Crear la clase FormularioLogin que hereda de Formulario e incluye toda la lógica de gestión del formulario. La creación de esta clase debería de ser

[2] https://sourcemaking.com/design_patterns/template_method

inmediata a partir de los scripts login.php y procesarLogin.php, simplemente debes copiar el código HTML que sea necesario a la nueva clase.

2. Crear la clase FormularioRegistro que hereda de Formulario e incluye toda la lógica de gestión del formulario. La creación de esta clase debería de ser inmediata a partir de los scripts registro.php y procesarRegistro.php, simplemente debes copiar el código HTML que sea necesario a la nueva clase.
3. Modifica los scripts login.php y registro.php para que utilicen las clases que acabas de crear.
4. Crea una carpeta "Eliminados" y mueve los scripts procesarLogin.php y procesarRegistro.php a esta carpeta.

Paso 7: Reducción de include / require en los scripts

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **07-inicio.zip**.*

Como resultado de la reorganización en archivos separados entre los diferentes tipos de script y con mayor énfasis en el caso de las clases, el número de instrucciones require_once es elevado.

Aunque no es demasiado problemático, sí que resulta incómodo a la hora de desarrollar y desplegar la aplicación. Una opción podría ser hacer un require_once por cada clase en el archivo `includes\config.php` de la aplicación y suponer que no es necesario hacer ningún require_once dentro de los .php asociados a nuestras clases. Esta solución tiene dos problemas principales:

- Es frágil. Si se nos olvida incluir el require_once la aplicación fallará estrepitosamente.
- Es ineficiente. Debemos recordar que PHP es un lenguaje interpretado y, aunque existen optimizaciones, por cada petición que ejecuta debe analizar todo el código PHP de nuestros scripts. Normalmente no es necesario utilizar todas las clases para poder implementar una funcionalidad de nuestra aplicación, por tanto, el motor de PHP tiene que trabajar más de lo necesario.

El segundo problema podría evitarse realizando require_once selectivos, pero el primer problema sólo podríamos mitigarlo parcialmente.

Por suerte PHP tiene un mecanismo que nos permitirá hacer require_once selectivos según hagamos referencias a nuestras clases PHP. Se trata de la función spl_autoload_register. Todavía mejor, existe la convención PSR-4 que nos proporciona

unas pautas a la hora de definir nuestras clases de modo que si las seguimos podemos utilizar una función de carga de clases ya probada[3].

El objetivo de este apartado es:

1. Añadir una sentencia namespace a todas las clases de modo que todas estén dentro del espacio de nombres **`es\ucm\fdi\aw`**.
    - Nota: La sentencias `namespace` y `use` tienen una utilidad similar a las sentencias `package` e `import` de Java respectivamente.
2. Incluir dentro de includes\config.php una función de carga de clases. Utiliza como referencia el ejemplo proporcionado en PSR-4[4].
    - Nota: presta particular atención a las variables $prefix y $base_dir del código.
3. Elimina los require_once que ya no son necesarios en todos los scripts.

Entrega del trabajo

La entrega de este ejercicio se puede hacer de dos formas: en pareja y de forma individual.

- Entrega en pareja:
    - Guarda todos los archivos en una carpeta con nombre "ej3_p".
    - Una vez terminado el ejercicio, comprime la carpeta con nombre "ej3_p"y añade un archivo TXT, nombrado "pareja.txt", con los nombres de los estudiantes que entregan el trabajo. **Sólo uno de los miembros debe subir el ejercicio**.
    - Contenido de la carpeta "ej3_p<numPareja>":
        - Carpeta "Paso6": todos los ficheros y carpetas necesarios para el paso 6.
        - Carpeta "Paso7": todos los ficheros y carpetas necesarios para el paso 7.
- Entrega individual:
    - Guarda todos los archivos en una carpeta con nombre "ej3_<Apellido>", pero no utilices tildes, ni "ñ" en tu apellido, en su lugar puedes utilizar "nn".

[3] https://www.php-fig.org/psr/psr-4/examples/

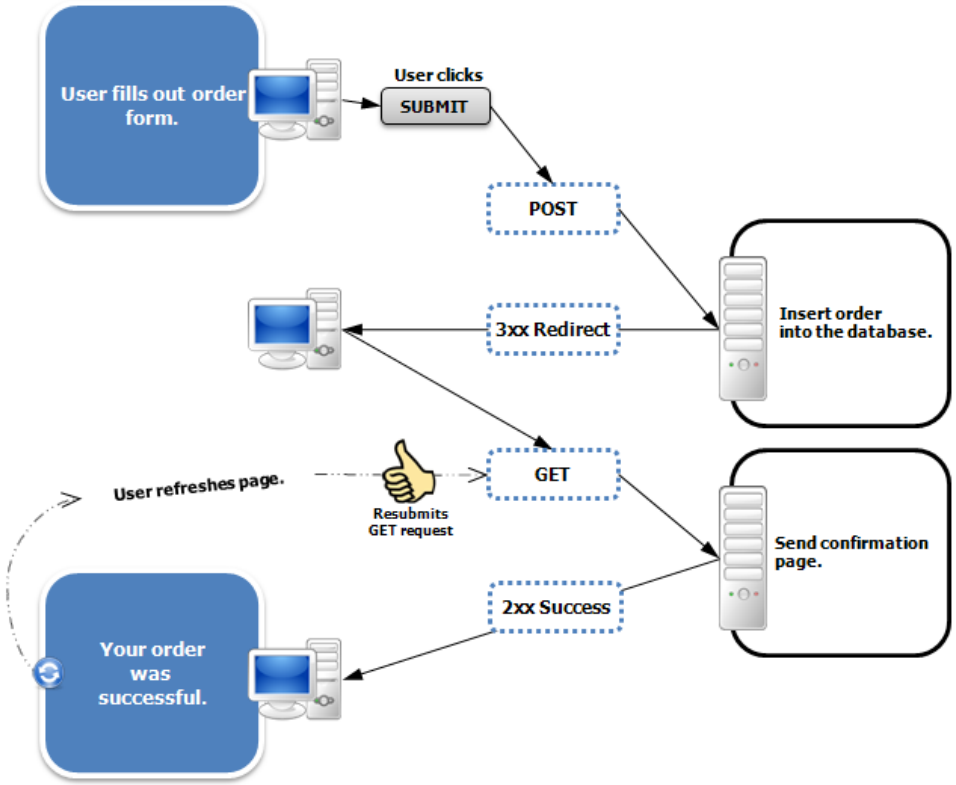
[4] https://www.php-fig.org/psr/psr-4/examples/#closure-example

- Una vez terminado el ejercicio, comprime la carpeta con nombre "ej3_<Apellido>" y añade un archivo TXT, con nombre "alumno.txt", que contenga tu nombre y apellidos. Este archivo comprimido debe tener como nombre **ej3_i<Apellido>.zip** y se debe subir a la entrega habilitada en el aula virtual antes del final que se indica en el buzón.
- Contenido de la carpeta "ej3_i<Apellido>":
  - Carpeta "Paso6": todos los ficheros y carpetas necesarios para el paso 6.
  - Carpeta "Paso7": todos los ficheros y carpetas necesarios para el paso 7.

Anexo 1: Implementación de atributos de petición

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **01-anexo.zip**.*

En el flujo de gestión de las peticiones de los formularios, en el caso de que la petición haya sido gestionada de manera exitosa, lo habitual es redirigir al usuario a otra página para evitar que al usuario le aparezca un mensaje del navegador similar a "Desea reenviar la petición al servidor". No obstante, con esta manera de proceder perdemos la posibilidad de mostrar algún mensaje de éxito para el usuario.



Fuente: Wikipedia

La única manera de resolver este problema sería dejando el mensaje dentro de la sesión del usuario (`$_SESSION`) que es el único mecanismo que tenemos para mantener información entre peticiones. Sin embargo, la sesión es un recurso que hay que intentar minimizar. Como solución de compromiso se puede almacenar en la sesión uno o varios atributos para que se puedan **consultar y borrar automáticamente** en la siguiente petición que gestione la aplicación para dicho usuario.

El objetivo de este apartado es:

1. Añade en `Aplicacion` un atributo privado `$atributosPeticion` y asegúrate de inicializarlo en el método `Aplicacion::init($bdDatosConexion)` con el valor que esté dentro de un attributo '`attsPeticion`' de la sesión, si existe, o con un array vacío si no existe. En el caso de que exista dicho atributo asegúrate de borrarlo de la sesión (`unset($_SESSION['attsPeticion'])`) para liberar el recurso.

2. Define en la clase `Aplicacion` un par de métodos `putAtributoPeticion($clave,$valor)` y `getAtributoPeticion($clave)` que se encarguen de almacenar o consultar un atributo dentro del array asociativo que debes de tener asociado a `$_SESSION['attsPeticion']`.
3. Modifica `FormularioRegistro::procesaFormulario()` para que, en caso de registrar exitosamente al usuario, añadir un par de mensajes:
    - "Se ha registrado exitosamente"
    - "Bienvenido NOMBRE_USUARIO"
4. Modifica `plantilla.php` para mostrar los mensajes que se hayan dejado en la petición anterior, si existen.

Anexo 2: Gestión de errores en la aplicación

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **02-anexo.zip**.*

Tradicionalmente en PHP tanto en las funciones / clases del lenguaje o utilizadas habitualmente, así como en el código que generan los programadores, la gestión de errores es un tanto limitada.

Por ejemplo, al revisar las funciones / clases de acceso a BD Mysqli de PHP podemos apreciar en la documentación que la mayoría de los métodos / funciones devuelven un valor `false` si ha habido un error al realizar la operación.

Además, en los ejemplos de gestión de conexiones que habitualmente utilizamos se realiza una gestión de errores bastante limitada, simplemente se muestra un mensaje simple y se termina la ejecución del script. En este caso, la experiencia de usuario de la aplicación es bastante mala y poco profesional.

Por otro lado, si en nuestro código adoptamos la convención de devolver `false` en caso de error o un valor diferente si la operación es exitosa, tenemos un problema si uno de nuestros métodos o funciones puede tener / generar varios tipos de error. Por ejemplo, en el caso de Mysqli es necesario consultar `mysqli::errno` o `mysqli::sqlstate` para averiguar un código de error y `mysqli::error` para obtener el mensaje de error. Nótese que PDO también sufre este mismo problema.

No obstante, PHP 7 ya soporta un mecanismo más razonable de gestión de errores como son las Excepciones incluso para la gestión de errores internos de PHP. Internamente extensiones como Mysqli o PDO pueden configurarse para cambiar el comportamiento en la gestión de errores y lanzar excepciones en caso de error. Para activar este comportamiento con Mysqli, debes incluir el siguiente fragmento de código antes de crear una conexión a la BD:

```
$driver = new \mysqli_driver();
$driver->report_mode = MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT;
```

Revisa la documentación de Mysqli para `mysqli_driver::report_mode` para obtener más información. Para PDO existe un mecanismo similar configurando la opción `PDO::ATTR_ERRMODE`.

Por otro lado, en la gestión de errores tenemos **errores que son recuperables y errores no recuperables**. Un error no recuperable es, por ejemplo, si falla el código asociado a la gestión de conexiones con la BD. Este error está habitualmente relacionado con un problema de mala configuración de los parámetros de conexión a la BD o la falta de permisos o de configuración en el servidor de base de datos. En este tipo de errores, no se puede hacer más que mostrar una página de error y registrar el problema para que algún administrador de la aplicación lo arregle. Por otro lado, tenemos errores recuperables, esto es, son errores que ya sea o bien desde el código reintentando la operación (algunas operaciones de BD) y o bien informando al usuario y repitiendo la operación, el error puede ser solventado.

Esta distinción en tipos de error junto a las excepciones nos permite organizar la gestión de errores a varios niveles dentro de la aplicación:

- Podemos tener un mecanismo de gestión de excepciones por defecto, que se encargará de gestionar las excepciones / errores no recuperables. En PHP este mecanismo se configura con la función `set_exception_handler`.
- Podemos lanzar y tratar excepciones dentro de nuestro propio código de manera que no tenemos que depender del valor de retorno de los métodos y funciones de nuestro código para la gestión de errores.

Un ejemplo de gestión global de excepciones en este ejercicio podría ser el siguiente:

```
function gestorExcepciones(Throwable $exception) {
    error_log($exception->getMessage());
    http_response_code(500);
    $tituloPagina = 'Error';

    $contenidoPrincipal = <<<EOS
    <h1>Oops</h1>
    <p> Parece que ha habido un fallo. </p>
    EOS;

    require __DIR__.'/plantillas/plantilla.php';
}

set_exception_handler('gestorExcepciones');
```

El objetivo de este apartado es el siguiente:

1. Añade el código de gestión global de excepciones dentro del fichero `config.php`.
2. Modifica la clase `Aplicacion` para asegurarte que se configura la gestión de errores de Mysqli basada en excepciones utilizando el código que se ha visto previamente en este apartado.
3. Modifica las clases `Aplicacion` y `Usuario` para lanzar excepciones (puedes utilizar la clase `Exception`) en caso de problemas relacionados con la BD.
4. Define una clase `UsuarioYaExisteException` que extienda la clase `Exception`. Esta excepción representará un error gestionable en la clase Usuario a la hora de crear un usuario y que se lanzará si un usuario con el mismo nombre de usuario ya existe.
5. Modifica tu tabla asociada a tus usuarios de la BD para añadir una restricción **UNIQUE** en la columna asociada al nombre de usuario (e.g. `ALTER TABLE` `` `Usuarios` ADD UNIQUE(`nombreUsuario`); ``).
6. Modifica el código de `Usuario::crea()` para que intente insertar en la tabla correspondiente los datos del nuevo usuario sin haber consultado previamente con una sentencia SELECT si un usuario ya existía previamente con el mismo nombre de usuario. En este caso, si un usuario ya existía con el mismo nombre de usuario se violará la restricción que has añadido y mysqli lanzará una excepción `mysqli_sql_exception` en el método `mysqli::query()` que deberás de tratar. En caso de excepción será necesario que consultes el valor de la propiedad mysqli::sqlstate para averiguar el error que ha provocado la excepción, por ejemplo, el error 23000 es el que corresponde el haber violado una restricción UNIQUE. Si el error es el 23000 deberás de construir y lanzar una instancia de `UsuarioYaExisteException` o relanzar la excepción en caso

contrario. Un pseudocódigo asociado la gestión de este error sería:

```
private static function inserta($usuario) {
...
$app = Aplicacion::getInstance();
$conn = $app->conexionBd();
$query= ...
try {
  $conn->query($query);
  $usuario->id = $conn->insert_id;
} catch(\mysqli_sql_exception $e) {
  if ($conn->sqlstate == 23000) {
    throw new UsuarioYaExisteException("Ya existe el usuario {$usuario->nombreUsuario}");
  }
  throw $e;
}
return $usuario;
```

7. Modifica `FormularioRegistro::procesaFormulario()` para tratar convenientemente la excepción `UsuarioYaExisteException` y muestre el mensaje de error correspondiente al usuario como parte de la gestión del formulario de registro.

Anexo 3: Enrutado centralizado / patrón Frontend Controller

*NOTA PREVIA: Puedes seguir con tu propio proyecto, o si tienes problemas descarga el archivo **03-anexo.zip**.*

Cuando una aplicación adquiere cierta envergadura es habitual querer tener controlados los puntos de acceso a la aplicación (URLs) e incluso aplicar mecanismos de control de acceso homogéneos a la aplicación

Esta filosofía es la que ha llevado a numerosos frameworks de desarrollo web a facilitar / adoptar el patrón frontend controller con el objetivo de centralizar las rutas (URLs + verbo HTTP) que admite una aplicación web.

Al aplicar este patrón en el desarrollo web se centra en tener un único punto de entrada en la aplicación que se encarga de recibir todas las peticiones, aunque realmente la gestión de la petición en sí (e.g. la acción que desea realizar el usuario) se delega a otras funciones / clases.

En PHP es sencillo implementar este patrón, requiriendo de:

1. Soporte por parte de servidor (e.g. Apache, Nginx, etc.) para poder reescribir URLs. Aunque este requisito podría eliminarse, es deseable y recomendable

utilizarlo. En Apache podríamos añadir la siguiente configuración a un fichero .htaccess:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . index.php [L]
```

que reenviaría cualquier petición que no corresponda a un fichero o directorio que exista en el servidor; la reenvía al script index.php.

2. Modifica la carga automática de clases para acomodar clases que no tienen un namespace común:
    a. Crea la carpeta `includes/clases/es/ucm/fdi/aw`.
    b. Mueve todas las clases que tengas a la carpeta `includes/clases/es/ucm/fdi/aw`.
    c. Modifica las `$prefix` y `$base_dir` del código que has utilizado para la carga automática de clases en el apartado 7.
        - `$prefix = '';`
        - `$base_dir = __DIR__ . '/clases/';`
3. Crear / utilizar un enrutador que se encargue de procesar la petición y delegar la gestión a otras clases / funciones de tu código. Un ejemplo de enrutador existente y a la vez sencillo y potente es el proyecto github.com/bramus/router. Aunque hay unas instrucciones disponibles en el repositorio, presuponen que estáis utilizando una herramienta de gestión de dependencias para PHP llamada phpcomposer. No obstante, no es necesario utilizarla para utilizar este enrutador, puedes simplemente:
    a. Crear la carpeta `includes/clases/Bramus/Router`.
    b. Descargar el fichero github.com/bramus/router/src/Bramus/Router/Router.php y dejarlo en la carpeta creada en el paso previo.
    c. Instanciar el enrutador en el `index.php` del siguiente modo:

```
require_once __DIR__.'/includes/config.php';

$router = new \Bramus\Router\Router();

// Añadir rutas

$router->run();
```

y ya puedes seguir las instrucciones que aparecen en el README.md del repositorio.

Tendrás que reorganizar nuevamente el código, en especial modificar la clase `Formulario` y los formularios específicos. Si revisas Router.php podrás ver que esta clase procesa la variable `$_SERVER['REQUEST_URI']`, en vez de `$_SERVER['PHP_SELF']` para saber la URL que está procesando.