



CREATE



READ



UPDATE



DELETE

C

R

U

D

Sumario

1. INTRODUCCIÓN AL EJERCICIO.....	3
2. COMENZANDO.....	3
3. MVC.....	4
4. PREPARÁNDONOS PARA MVC.....	6
5. CONEXIÓN.....	6
6. CREAR ARCHIVOS RESTANTES.....	7
7. CREANDO MENÚ PRINCIPAL.....	8
8. FORMULARIO DE INSERCIÓN.....	11
8.1. Formulario de inserción.....	11
8.2. Modelo, Preparar la Inserción de datos.....	12
8.3. Controlador. Seguimos Preparando la inserción de datos.....	14
8.4. Ahora sí insertamos.....	14
9. VER DATOS.....	15
9.1 Método read. Cambios en el modelo.....	15
9.2 Método ver. Cambios en el controlador.....	16
9.3 Modificar el index.php.....	16
9.4 Código show.php.....	17
10. LISTAR DATOS.....	17
10.1 Tabla estática.....	17
10.2 Tabla dinámica.....	18
10.3 Agregando enlaces.....	21
11. ELIMINAR DATOS.....	22
12. EDITAR, ACTUALIZAR O MODIFICAR DATOS.....	25
12.1 Editando Datos.....	25
13. BUSCAR DATOS.....	29

EJERCICIO PARA EXPLICAR PDO

1. INTRODUCCIÓN AL EJERCICIO.

Si bien podría utilizar los apuntes para explicar PDO, vamos a realizar un cambio de táctica, usaremos estos apuntes como tutorial para entender la instrucciones PDO.

Para ello realizaremos una pequeña aplicación CRUD con base de datos MYSQL.

Pero, ¿qué es CRUD? ¿Qué es PDO? Como ya ampliaremos esta información en los apuntes los conceptos, sólo a definir los conceptos.

PDO significa PHP Data Objects, Objetos de Datos de PHP, una extensión para acceder a bases de datos. PDO permite acceder a diferentes sistemas de bases de datos con un controlador específico (MySQL, SQLite, Oracle...) mediante el cual se conecta. Independientemente del sistema utilizado, se emplearán siempre los mismos métodos, lo que hace que cambiar de uno a otro resulte más sencillo.

CRUD: Acrónimo de "Crear, Leer, Actualizar y Borrar" (del original en inglés: Create, Read, Update and Delete), que se usa para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.

Vamos que vamos hacer una pequeña aplicación que:

- Cree: Insertar datos → INSERT
- Lea datos → SELECT
- Actualice Datos → UPDATE
- Borre Datos → DELETE

2. COMENZANDO

Para realizar un CRUD qué es lo que necesito.... Evidentemente una Base de datos y sus tablas. Aunque PDO soporta múltiples tipo de base de datos, vamos a realizar la prueba en MYSQL. Empecemos por ahí.

```
CREATE DATABASE IF NOT EXISTS pruebas;
USE pruebas;
CREATE TABLE personas(
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
    usuario VARCHAR(50) NOT NULL,
    password VARCHAR(50) not NULL,
    nombre VARCHAR(255) NOT NULL,
    apellidos VARCHAR(255) NOT NULL,
    genero CHAR(1) NOT NULL,
    unique (usuario),
    PRIMARY KEY(id)
);

INSERT INTO personas (usuario, PASSWORD, nombre, apellidos, genero) VALUES
('luis','1234','Luis','Cabrera','M');
INSERT INTO personas (usuario, password, nombre, apellidos, genero)
VALUES('rosa','1234','Rosa','Martin','F');
INSERT INTO personas (usuario, password, nombre, apellidos, genero) VALUES
('ramon','1234','Ramon','Lopez','M');
INSERT INTO personas (usuario, password, nombre, apellidos, genero)
VALUES('pedro','1234','Pedro','Perez','M');
```

Sencillo, ¿verdad? Este script nos crea un base de datos denominada pruebas y una tabla llamada personas.

Esto evidentemente lo tenemos que ejecutar en nuestro cliente mysql preferido, phpmyadmin,

workbench y como no, el gran HEIDISQL en Windows.

3. MVC

Como me parece poco introducir como concepto nuevo el CRUD, además vamos a introducir un segundo concepto: MVC (Modelo Vista Controlador)

La arquitectura MVC (modelo, vista, controlador) consiste en un patrón de diseño de software que se utiliza para separar en tres componentes los datos, la metodología y la interfaz gráfica de una aplicación. La gran ventaja que posee esta técnica de programación es que permite modificar cada uno de ellos sin necesidad de modificar los demás, lo que permite desarrollar aplicaciones modulares y escalables que se puedan actualizar fácilmente y añadir o eliminar nuevos módulos o funcionalidades de forma paquetizada, ya que cada “paquete” utiliza el mismo sistema con sus vistas, modelos y controladores.

Una de las aplicaciones de la arquitectura MVC se centra en el desarrollo de páginas web o en el desarrollo de Apps para móviles. Los tres componentes de un sistema basado en arquitectura MVC son:

Modelo

El Modelo se encarga de manipular, gestionar y actualizar los datos. Si se utiliza una base de datos aquí es donde se realizan las consultas, búsquedas, filtros y actualizaciones.

Vista

La Vista sirve para mostrarle al usuario final la interfaz gráfica (pantallas, ventanas, páginas, formularios...) como resultado de una solicitud enviada a través del controlador. Desde la perspectiva del programador este componente es el que se encarga del frontend; la programación de la interfaz de usuario si se trata de un aplicación de escritorio, o bien, la visualización de las páginas web (CSS, HTML, HTML5 y Javascript).

Controlador

El Controlador es el componente principal de la aplicación, donde se especifican los métodos y funcionalidades que una aplicación (o módulo de una aplicación) tienen que realizar. Se encarga de gestionar las instrucciones que se reciben, atenderlas y procesarlas. A través del controlador se realizan las consultas al modelo (una búsqueda por ejemplo), y una vez se hayan obtenido dichos datos, se envía a la vista las instrucciones necesarias para poder mostrarlos de una forma legible para el usuario.

Esta arquitectura es cada vez más utilizada por los programadores. En desarrollo de páginas web también se está convirtiendo en una metodología líder, por la flexibilidad que ofrece al desarrollar componentes o módulos para las aplicaciones web. Aunque WordPress aun no trabaja de forma nativa con arquitectura MVC, otros CMS como Joomla!, PrestaShop y Magento sí lo hacen, y es una técnica de programación cada vez más extendida.

Si estás pensando en realizar un proyecto web o una aplicación, intenta utilizar este tipo de arquitectura de programación para garantizar un desarrollo flexible, modular y escalable, básico para cualquier empresa que debido a su crecimiento necesite nuevas implementaciones o modificaciones en sus aplicaciones.

¿Qué ventajas nos proporciona un desarrollo MVC?

Como hemos visto en el apartado anterior, esta tecnología se centra en la escalabilidad, y permite a su vez dividir el trabajo entre un grupo de profesionales, al estar sus componentes separados entre sí.

Por ejemplo, si tienes que desarrollar un programa que haga cálculos complejos, tienes la interfaz ya diseñada y la conexión a la base de datos, pero compruebas los resultados con una calculadora de integrales y encuentras un error. Simplemente tendrás que modificar el controlador para corregir la falla, sin que el cambio afecte a todo lo demás.

Entre las principales ventajas que puede ofrecernos un desarrollo MVC podemos destacar las siguientes:

- Separación clara de dónde tiene que ir cada tipo de lógica, facilitando el mantenimiento y la escalabilidad de nuestra aplicación.
- Sencillez para crear distintas representaciones de los mismos datos.
- Facilidad para la realización de pruebas unitarias de los componentes, así como de aplicar desarrollo guiado por pruebas (Test Driven Development o TDD).
- Reutilización de los componentes.
- No existe ciclo de vida de las páginas. Con menos peso, menos complejidad.
- Motor de Routing asociando una URL concreta con su correspondiente controlador, permitiendo URL semánticas. Las URL semánticas se indexan mejor en los buscadores, siendo más adecuadas para el posicionamiento web.

Eso sí, también tiene sus desventajas:

- Para aplicaciones pequeñas puede aumentar su complejidad de desarrollo, pero generalmente se ve compensada por una reducción del tiempo en los mantenimientos de código, así que..
- La mayor desventaja del patrón MVC es utilizarla donde no debemos. Puede parecer obvio, pero se tiende a pensar que si mucha gente usa un patrón, este sirve para todo y no es así. Debes de tener en cuenta siempre que:
 - Debes recordar: MVC es un patrón relacionado con la IU (Interfaz de Usuario). Si estás creando una aplicación compleja, debería llevar todo, que no esté relacionado con la UI, de la MVC a otras clases, subsistemas o capas.
 - Así por ejemplo en aplicaciones de consola NO es necesario

Ejemplo de mal uso: En este ejemplo usar solo MVC puede ser una mala elección : intente diseñar un sistema de control de tráfico aéreo o una aplicación de procesamiento de préstamos e hipotecas para un banco grande. Solo MVC por sí solo sería una mala elección . Inevitablemente, tendrá colas de mensajes / colas de eventos junto con una arquitectura de múltiples capas con MVC dentro de capas individuales y posiblemente un diseño general de MVC / PAC para mantener la base del código mejor organizada.

Aún así, suele ser recomendable para el diseño de aplicaciones web compatibles con grandes equipos de desarrolladores y diseñadores web que necesitan gran control sobre el comportamiento de la aplicación. De hecho , múltiples de frameworks en Web utilizan este patrón. Por nombrar algunos:

- PHP
 - Symfony, Laravel, Zend Framework, CodeIgniter
- JS
 - React, Angular, Backbone, Meteor
- Java
 - Spring,
- Python
 - Django, Flask

Como puede ver los frameworks más conocidos en la web soportan MVC. Por cierto:

Framework: Conjunto de herramientas que no sólo nos proporciona un API con el que trabajar sino también una filosofía de desarrollo, una "forma de hacer las cosas" estructurada y modular. Por tanto, el primer beneficio de usar un framework es que estamos haciendo las cosas de una forma ya probada, la misma idea que constituye la base de los patrones de diseño software.

Vale, todo esto está muy bien, pero... ¿No es suficiente con aprender CRUD/PDO? Sí, la idea de introducir MVC en este tutorial, no es tanto en profundizar con el patrón MVC, si no "acostumbrarnos" a algunas de sus reglas/estilos de programación.

Eso sí, debes recordar sólo una regla en MVC:

Las vistas SOLO pueden comunicarse con el controlador, el controlador invoca al modelo y le solicita datos, el modelo SOLO puede comunicarse con el controlador, por tanto el modelo sólo le devuelve los datos al controlador, el cual se, si fuera necesario los devuelve a la vista.
En otras palabras todo pasa por el controlador y NUNCA NUNCA NUNCA, las vistas accederán directamente al modelo y el modelo NUNCA NUNCA NUNCA le dará datos directamente a las vistas.

4. PREPARÁNDONOS PARA MVC

Una vez creada la base de datos, vamos a generar una estructura de carpetas concreta dentro de nuestros proyectos.

```
assets/
    ....Aquí pondremos CSS, JS, imágenes... aquello "extra"

config/
    ....Archivos de configuracion
controllers/
    ....Aquí van los controladores.php
models/
    ....Aquí van los modelos.php
views/
    ....Aquí van las vistas.php
index.php
```

Sencillo y ordenado. Se entiende donde irá cada cosa.

5. CONEXIÓN

Una vez creada la base de datos, tendremos que "sacar" datos. Para interactuar con la base de datos necesitamos conectarnos a la misma. Para ello, generaré un archivo db.php, dentro de la carpeta **config**, y crearemos un clase db, con una método de conexión. Después simplemente lo incluimos en donde queramos usarlo. Voy a poner el código aquí, y lo explicaré más abajo.

config/db.php

```
<?php
class db
{
    const HOST = "localhost";
    const DBNAME = "pruebas";
    const USER = "root";
    const PASSWORD = ""; // Evidentemente adapta los valores

    static public function conexion()
    {
        $conexion = null;
        try {
            $opciones = array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION);
            $conexion = new PDO('mysql:host=localhost;
dbname=' . self::DBNAME, self::USER, self::PASSWORD, $opciones);
        } catch (Exception $e) {
            echo "Ocurrió algo con la base de datos: " . $e->getMessage();
        }
        return $conexion; //Es un objeto de conexion PDO
    }
}
```

Como podemos ver, lo que hacemos es crear un objeto PDO y le pasamos 3 parámetros:

- 1.Nombre del origen de datos
- 2.El usuario
- 3.La contraseña

PDO permite cambiar de gestor de base de dato en una sola línea, sólo tendríamos que cambiar el primer parámetro. Aquí dejo la lista de [gestores de bases de datos soportadas](#). Por otro lado, aquí dejo la [documentación oficial](#) de la clase.

Ojo: ¿Hay algo nuevo? El try/catch Lo ponemos en un try/catch porque en algunas ocasiones puede que nos equivoquemos de usuario, contraseña, base de datos, etcétera. Profundizaremos en ello en los apuntes.

Nota: recuerda que la base de datos debe existir, así como la tabla “personas”. Puedes cambiar el usuario, la contraseña o el nombre de la base de datos como tú desees.

6. CREAR ARCHIVOS RESTANTES.

Como hemos indicado el MVC me impone una serie de reglas. Algunas de ellas se corresponden a como vamos a organizar nuestras distintas carpetas/ficheros. Lo que por un lado puede parecer un imposición, una vez acostumbrado a ellos, es una forma de ordenar nuestras aplicaciones, lo cual, a la larga, mejora el mantenimiento y el desarrollo de las aplicaciones.

Carpeta assets:

- Crear Carpeta: **css**
- Crear Carpeta: **images**
- Crear Carpeta: **js**

Carpeta Controller:

- personasController.php

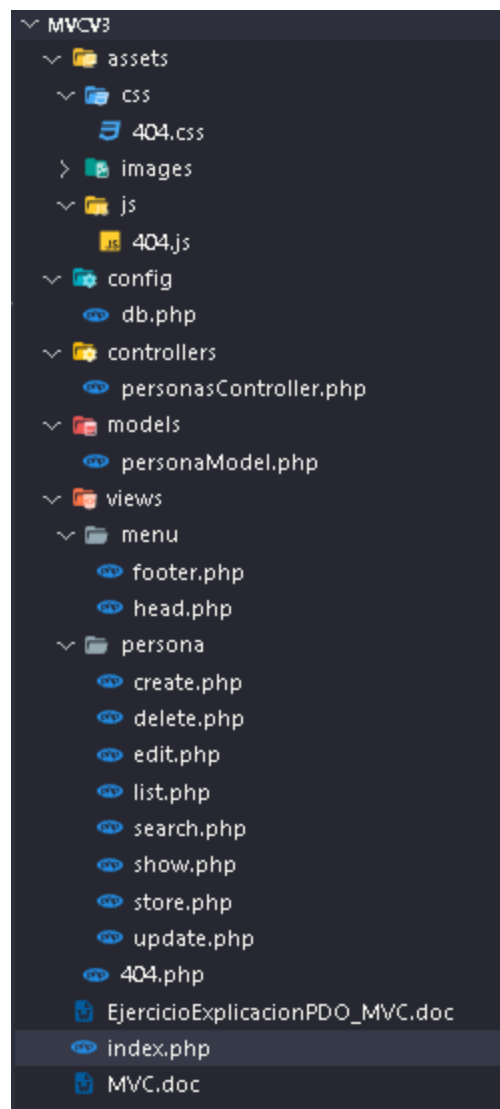
Carpeta Models:

- personasModel.php

Carpeta Views:

- Crear fichero: 404.php .
- Crea Carpeta: **menu**
 - Dentro de **menu**:
 - footer.php
 - head.php
- Crear Carpeta: **persona**
 - Dentro de **persona**:
 - create.php
 - delete.php
 - edit.php
 - list.php
 - list.php
 - show.php
 - store.php
 - update.php

Quedando algo parecido a:



7. CREANDO MENÚ PRINCIPAL

Editamos el index.php

```
<?php
require_once ("views/menu/head.php");

require_once ("views/menu/footer.php");
```

Me parece que la idea está clara, tendremos una cabecera y un pie común en toda nuestra aplicación.

Evidentemente, esto implica generar los ficheros tanto head.php como footer.

Hagamos las cosas bonitas, que tampoco cuestan tanto. Utilicemos Bootstrap, dentro de head.php <https://getbootstrap.com/docs/5.2/getting-started/introduction/>

Cojamos como partida el ejemplo 2 de la pagina, el cual ya viene introducidos los ficheros .css y .js de bootstrap. Eso sí, lógicamente lo dividimos en dos partes, la primera parte en head.php y la segunda en footer.php

head.php

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
ZenH87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeu0xjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
    </head>
    <body>
      <h1>Hello, world!</h1>
```

footer.php

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.mi
n.js"
integrity="sha384-0ERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJua0e923+mo//
f6V8Qbsw3" crossorigin="anonymous"></script>
</body>
</html>
```

Falta el menú, para usar un menú de bootstrap, primero busquemos **nav** en la barra de búsquedas. Copiemos la que está justo encima del texto **FORMS**. Qué casualidad, el más largo.

Lo introducimos en el body del head.php y probamos.

Anda, que bonito!!!!

Hay que hacer algunos cambios, añadir unas opciones, quitar otras... Os pongo el código después de los cambios, puesto que son cuestiones estéticas que en este momento no son especialmente relevantes.

Así que definitivamente quedan:

head.php

```
<!doctype html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeu0xjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css">
    <link href="assets/css/404.css" rel="stylesheet" >

  </head>
  <body>
    <div class="container-fluid bg-primary p-2 mb-3">
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary ">
    <div class="container-fluid">
      <a class="navbar-brand" href="index.php">Inicio</a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target="#navbarNavDropdown" aria-controls="navbarNavDropdown" aria-
expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarNavDropdown">
        <ul class="navbar-nav">
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="index.php" role="button"
data-bs-toggle="dropdown" aria-expanded="false">
              Persona
            </a>
            <ul class="dropdown-menu">
              <li><a class="dropdown-item" href="index.php">Personas</a></li>
              <li><hr></li>
              <li><a class="dropdown-item" href="index.php?
accion=crear">Añadir Nueva Persona</a></li>
              <li><a class="dropdown-item" href="index.php?
accion=listar">Listar Personas</a></li>
              <li><a class="dropdown-item" href="index.php?
accion=buscar">Buscar Personas</a></li>
            </ul>
          </li>
        </ul>
      </div>
    </div>
  </nav>
</div>
<div class="container-fluid">
```

Observa que aparecen algunas opciones, ya las iremos rellenando posteriormente.

index.php

```
<?php
require_once ("views/menu/head.php");

if (isset($_REQUEST["accion"])){
    switch ($_REQUEST["accion"]){
        case "crear":
            require_once ("views/persona/create.php");
            break;
        default:
            require_once ("views/404.php");
            break;
    }
}
else{
    ?>
    <a href="index.php?accion=crear" class="btn btn-primary">Agregar
    Usuario</a>
<?php
}
require_once ("views/menu/footer.php");
```

footer.php

```
</div>
</body>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min
.js" integrity="sha384-0ERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJua0e923+mo//
f6V8Qbsw3" crossorigin="anonymous"></script>
</body>
</html>
```

Con toda seguridad, lo más llamativo sean los enlaces, fíjate que los href quedan algo parecido a:

```
<a href="index.php?accion=crear"
```

Sin embargo los require_once ponen

```
require_once ("views/persona/create.php");
```

La idea es que, mediante la variable acción, le indiquemos cuál es la acción a realizar al controlador y este a su vez, llame al método/página deseada.

En otras palabras:

a) que quiero crear una persona:

- index.php?accion= crear
- necesito la vista de crear=> require_once("views/persona/create.php")

b) que quiero borrar una persona:

- index.php?accion= borrar
- necesito la vista de borrar=> require_once("views/persona/delete.php")

c) que quiero modificar una persona :

- index.php?accion= editar
- vista de modificar=> require_once("views/persona/edit.php")

Más o menos te haces una idea. En el caso anterior de momento sólo hemos añadido la acción, crear una nueva persona.

A esta técnica se le denomina sistema de ruteo y en nuestro caso es, créeme, MUY SIMPLE.

8. FORMULARIO DE INSERCIÓN

8.1. Formulario de inserción

Comencemos insertando datos. Para ello, necesitamos un simple formulario en HTML de bootstrap. Después de realizar las adaptaciones necesarias nos quedaría algo así.

create.php

```
<form action="index.php?accion=guardar&evento=crear" method="POST">
  <div class="form-group">
    <label for="usuario">Usuario </label>
    <input type="text" required class="form-control" id="usuario"
name="usuario" aria-describedby="usuario" placeholder="Introduce Usuario">
    <small id="usuario" class="form-text text-muted">Compartir tu usuario lo
hace menos seguro.</small>
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" required class="form-control" id="password"
name="password" placeholder="Password">
  </div>
  <div class="form-group">
    <label for="nombre">Nombre </label>
    <input type="text" class="form-control" id="nombre" name="nombre"
placeholder="Introduce tu Nombre">
  </div>
  <div class="form-group">
    <label for="apellidos">Apellidos </label>
    <input type="text" class="form-control" id="apellidos" name="apellidos"
placeholder="Introduce tu Apellido">
  </div>
  <div class="form-radio">
    <input type="radio" class="form-radio-input" id="genero1" name="genero"
value="M">
    <label class="form-radio-label" for="genero1">Masculino</label>
    <input type="radio" class="form-radio-input" id="genero2" name="genero"
value="F" checked>
    <label class="form-radio-label" for="genero2">Femenino</label>
  </div>
  <button type="submit" class="btn btn-primary">Guardar</button>
  <a class="btn btn-danger" href="index.php">Cancelar</a>
</form>
```

Recordamos: Lo único que quiero señalar es el atributo action, en la acción a realizar, en este caso guardar. Fijate que la acción se define en GET, pero la información se define por POST, por tanto, si quiero recoger la información usaremos REQUEST.

¿Y donde pondremos ese REQUEST? Fácil, a donde vamos. ¿Y donde vamos? ¿A store.php? Directamente NO. Si no que queremos ir a store.php pasando por index. Recuerda que al index le paso la acción. Veamos como quedaría entonces el **index.php**

index.php

```
<?php
require_once ("views/menu/head.php");

if (isset($_REQUEST["accion"])){
  switch ($_REQUEST["accion"]){
    case "crear":
      require_once ("views/persona/create.php");
      break;
    case "guardar":
```

```

        require_once ("views/persona/store.php");
        break;
        default:
            require_once ("views/404.php");
            break;
    }
}
else{
    ?>
    <a href="index.php?accion=crear" class="btn btn-primary">Agregar Usuario</
a>
<?php
}
require_once ("views/menu/footer.php");

```

En realidad sólo le hemos añadido al case la acción guardar, y cargamos store.php. Creo que ya lo vamos entendiendo.

8.2. Modelo, Preparar la Inserción de datos.

Podríamos entender que debemos ir a store.php donde realizaremos la inserción. Pero, hasta ahora, sólo hemos manejado la carpeta views (las vistas) y el index para “redireccionar”. Teniendo en cuenta que este patrón es MVC, me da que de Modelo y Controlador, de momento bien poco. Y de PHP tampoco mucho todo sea dicho de paso.

Llegó el momento.

Antes de nada vamos a realizar el modelo.

Pero, ¿qué es el modelo? El modelo es la capa de acceso a DATOS. Es la capa que accede a los datos, los modifica, los crea, los borra. En Base de Datos, sería la parte que hacemos los SQL, en archivos, pues la funciones de archivos...

Simplificando mucho, el modelo va a ser la clase que me permita manejar la tabla de datos, donde estarán las funciones de inserción, borrado...

Así que vayamos a la carpeta **models** y modifiquemos el **personaModel.php**.

```

<?php
require_once('config/db.php');
//require_once('class/persona.php');

class PersonaModel
{
    private $conexion;

    public function __construct()
    {
        $this->conexion = db::conexion();
    }
    //o Persona p1
    public function insert(array $persona):?int //devuelvo entero o null
    {
        $sql="INSERT INTO personas(usuario, password, nombre, apellidos,
genero) VALUES (?, ?, ?, ?, ?)";
        $sentencia = $this->conexion->prepare($sql);
        $arrayDatos=[
            $persona["usuario"],
            $persona["password"],
            $persona["nombre"],
            $persona["apellidos"],
            $persona["genero"],

```

```

    ];
    $resultado = $sentencia->execute($arrayDatos);

    /*Pasar en el mismo orden de los ? execute devuelve un booleano.
    True en caso de que todo vaya bien, falso en caso contrario.*/
    //Así podríamos evaluar
    return ($resultado==true)?$this->conexion->lastInsertId():null ;
}
}

```

Veamos cosas.

Primero importamos la clase db, la cual tiene la función de conexión de base de datos. Eso está claro.

Posteriormente creamos un método insertar el cual recibe en un array los datos de la nueva persona y da de alta. La idea está clara.

Pero ¿qué hace el siguiente código?

```

$sql="INSERT INTO personas(usuario, password, nombre, apellidos, genero)
VALUES (?, ?, ?, ?, ?);";
$sentencia = $this->conexion->prepare($sql);
$arrayDatos=[
    $persona["usuario"],
    $persona["password"],
    $persona["nombre"],
    $persona["apellidos"],
    $persona["genero"],
];
$resultado = $sentencia->execute($arrayDatos);

```

Lo que vemos en el ejemplo anterior es una consulta preparada. Una consulta preparada es la forma que tiene PDO de ejecutar cualquier instrucción SQL. Lo cierto es que existen otras maneras, de hecho, más adelante, veremos un ejemplo utilizando el método query. Las consultas preparadas frente a por ejemplo el método query aporta grandes ventajas con respecto a la seguridad, pero esta información la ampliaremos en los apuntes.

Analicemos el código con calma.

\$sql: Es una cadena de texto en la cual introducimos el código SQL. Como llamativo tienes unos ?

\$conexion: Es el objeto de manejo de conexión a la base de datos. Con prepare, le indicamos el **formato** de la sentencia que viene a continuación.

Dentro del INSERT nos encontramos varios símbolos de interrogación ?. Cada uno de estos símbolos será sustituido por un valor (un dato real).

\$sentencia → Almacena la sentencia preparada para ejecutarse posteriormente.

Mientras no haga el EXECUTE no hace nada, sólo prepara

Pasamos a ejecutar

```

$arrayDatos=[
    $persona["usuario"],
    $persona["password"],
    $persona["nombre"],
    $persona["apellidos"],
    $persona["genero"],
];
$resultado = $sentencia->execute($arrayDatos);

```

Ahora, ejecuta la sentencia (el insert anterior) y devuelve true si se ejecuta correctamente y false si no. Esto lo almacenamos en resultado.

Pero date cuenta que dentro del execute hay un array [], con los valores usuario, password, nombre, apellidos y genero.

La sentencia preparada enlaza las interrogaciones del prepare anterior con los datos de execute. ¿Cómo? Cada interrogación es un dato, de manera que lo enlaza cada interrogante con cada variable del array de manera posicional.

Resumiendo:

- Primer ? -> Enlaza con \$persona["usuario"],
- Segundo ? -> Enlaza con \$persona["password"],
- Tercero ? -> Enlaza con \$persona["nombre"],
- Cuarto ? -> Enlaza con \$persona["apellidos"],
- Quinto ? -> Enlaza con \$persona["genero"],

Evidentemente el array \$persona es un array asociativo con los datos de la persona

Código del final

```
return ($resultado==true)?$this->conexion->lastInsertId():null ;
```

Como el id es autonuméricos, devolveremos el id de la ultima inserción o null si algo ha fallado.

8.3.Controlador: Seguimos Preparando la inserción de datos.

Como hemos visto el modelo es el encargado de realizar las acciones con respecto a la base de datos. Pero como indicamos al principio, la vista (create.php) no puede llamar directamente al modelo, si no que tendrá que interactuar con el controlador. Vamos con el.

personasController.php

```
<?php
require_once "models/personaModel.php";
//nombre de los controladores suele ir en plural
class PersonasController {
    private $model;

    public function __construct(){
        $this->model = new PersonaModel();
    }

    public function crear (array $arrayPersona):void {
        $id=$this->model->insert ($arrayPersona);
        ($id==null)?header("location:index.php?
accion=crear&error=true&id={$id}"): header("location:index.php?
accion=ver&id=".$id);
    }
}
```

En esencia la única función, en este caso del controlador es llamar al modelo, que en realidad es el que inserta y redirige en función si lo ha hecho bien o mal.

8.4. Ahora sí insertamos

Una vez definido el controlador y el modelo, pasamos a insertar.

store.php

```
<?php
require_once "controllers/personasController.php";
//recoger datos
if (!isset ($_REQUEST["usuario"])) header('Location:index.php?accion=crear' );

$id= ($_REQUEST["id"])??""; //el id me servirá en editar
$arrayPersona=[
    "id"=>$id,
    "usuario"=>$_REQUEST["usuario"],
```

```

        "password"=>$_REQUEST["password"],
        "nombre"=>$_REQUEST["nombre"],
        "apellidos"=>$_REQUEST["apellidos"],
        "genero"=>$_REQUEST["genero"]
    ];
//pagina invisible
$controlador= new PersonasController();
if ($_REQUEST["evento"]=="crear"){
    $controlador->crear ($arrayPersona);
}

```

Por fin un código que se entiende.

El proceso de inserción queda de la siguiente manera:

1. Desde create.php, genero el formulario.
2. Envío los datos a store.php, utilizando la acción guardar.
3. En store.php
 1. Recojo Datos.
 2. Creo el controlador.
 3. Le paso los datos al controlador.
4. Este invoca al modelo e inserta.
5. El modelo devuelve null si no inserta o un id entero si ha insertado correctamente.
6. El controlador recoge el resultado. Si es null, vuelve a la pagina anterior indicando que es un error. Si todo correcto, pasaremos a la acción ver, la cual me debe mostrar dicho dato insertado.

Facilísimo XD

Ahora para acabar esta parte me faltaría añadir al inicio de **create.php**, antes del form el manejo de mensajes de error.

```

<?php
    $cadena=(isset($_REQUEST["error"]))?"Error, ha fallado la inserción":"";
    $visibilidad=(isset($_REQUEST["error"]))?"visible":"invisible";
?>
<div class="alert alert-danger <?=$visibilidad?>" ><?=$cadena?></div>

```

9. VER DATOS.

Acabamos de insertar datos, pero que sentido tiene insertar si no puedo verlos. De hecho si te fijas, el enlace nos redirige a acción ver. Por ejemplo: **index.php?accion=ver&id=24**. Así que empecemos a desarrollar la acción ver.

Para ello lo que haremos es:

1. Agregar en el modelo el método **read**.
2. Agregar en el controlador el método **ver**.
3. Añadir el código de la vista **ver.php**
4. Modificar el **index.php** añadiendo dicha acción

9.1 Método read. Cambios en el modelo

personaModel.php

```

public function read(int $id):?stdClass
{
    $sentencia = $this->conexion->prepare("SELECT * FROM personas WHERE
id=:id");
    $arrayDatos=[":id"=>$id ];
    $resultado = $sentencia->execute($arrayDatos);
    // ojo devuelve true si la consulta se ejecuta correctamente
    // eso no quiere decir que hayan resultados
    if (!$resultado) return null;
}

```

```
//como sólo va a devolver un resultado uso fetch
// DE Paso probamos el FETCH_OBJ
$persona = $sentencia->fetch(PDO::FETCH_OBJ);
//fetch devuelve el objeto estándar o false si no hay persona
return ($persona==false) ? null:$persona;
}
```

Como podemos observar hay bastantes conceptos nuevos.

1. Cambios en la consulta preparada: Por tocar la narices, he cambiado un poco la consulta preparada, he utilizado otra notación (sin `?`, por cierto, esta me gusta más, ahí lo dejo). Observa que a diferencia de antes, no pongo una `?`, sino una marca creada por mi `:id`, precedida de `:`. Lo que estamos haciendo es sustituir la marca `?` por `:id`. Evidentemente esto se lo tengo que indicar al método `execute` en el array, indicando que la marca `:id` se enlaza con `$id`. ¿Ventajas de esto? Básicamente dos: No importa el orden en el array y me permite hacer un código más legible ya que pondré marcas que aportan más información.
2. **\$resultado**: Ahora sigue devolviendo true y false, pero el resultado SOLO NOS DICE si se ha ejecutado correctamente, no si devuelve datos.
3. **\$sentencia**: Sentencia, ha creado un enlace al inicio de un resource. En castellano, es parecido al inicio de un fichero, sólo que en vez de usar `fgets` para recorrer ese recurso, usamos el método `fetch`.
4. **\$sentencia->fetch (PDO::FETCH_OBJ)**: Una vez ejecutado utilizamos el método **fetch**: El método `fetch` me devuelve un elemento de una consulta o false si no quedan elementos. En este caso nos devuelve la primera fila del resultado de la consulta. Si quisiera devolver todos los elementos, podríamos poner el `$sentencia->fetch` dentro de `while`. Sólo necesitamos la primera fila en este ejemplo puesto que la consulta devolverá como máximo un registro.
5. **PDO::FETCH_OBJ**: En este parámetro le indico en que formato quiero que devuelva el registro. En este caso lo quiero en notación `$objeto->valor`. Por ejemplo: `$persona->id`, `$persona->nombre`....
6. `public function read(int $id):?stdClass`: Aquí utilizamos la interrogación `?` Para indicar que puede devolver **null** y a continuación el tipo que devuelve. Como `PDO::FETCH_OBJ`, crea objetos estándar devolvemos `stdClass`. Como puedes observar si todo va bien devolvemos un objeto estándar y sino **null**.

9.2 Método ver. Cambios en el controlador

Añadimos el método `ver` en el controlador.

controladorPersona.php

```
public function ver (int $id):?stdClass {
    return $this->model->read ($id);
}
```

Nada que decir, sólo llamamos al modelo y ya

9.3 Modificar el index.php

Añadimos la acción `ver` en `switch`

index.php

```
case "ver":
    require_once ("views/persona/show.php");
    break;
```


9.4 Código show.php

show.php

```
<?php
require_once "controllers/personasController.php";
if (!isset($_REQUEST['id'])){
    header ("location:index.php");
}
$id=$_REQUEST['id'];
$controlador= new PersonasController();
$persona= $controlador->ver ($id);
?>

<div class="card" style="width: 18rem;">
    <div class="card-body">
        <h5 class="card-title">Persona: ID: <?=$persona->id?> USUARIO: <?
=$persona->usuario?></h5>
        <p class="card-text">
            Nombre: <?=$persona->nombre?> <?=$persona->apellidos?>
            Genero: <?=( $persona->genero=="M"? "Masculino": "Femenino")?>
        </p>
        <a href="index.php" class="btn btn-primary">Volver a Inicio</a>
    </div>
</div>
```

Cargo la persona y en un elemento card, mostramos los datos.

10. LISTAR DATOS.

Qué sentido tiene insertar datos si no puedo verlos. En un CRUD siempre, siempre, siempre podré leer y mostrar los datos (R Read)

Para ello primero realizaremos una aproximación a lo que queremos.

10.1 Tabla estática

Si lo que queremos es mostrar datos fijos, haríamos algo parecido a esto

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Tabla de ejemplo</title>
    <style>
        table, th, td {
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Nombre</th>
                <th>Apellidos</th>
                <th>Género</th>
            </tr>
        </thead>
        <tbody>
```

```

        <tr>
            <td>1</td>
            <td>John</td>
            <td>Doe</td>
            <td>M</td>
        </tr>
        <tr>
            <td>2</td>
            <td>María José</td>
            <td>Ejemplo</td>
            <td>F</td>
        </tr>
        <tr>
            <td>3</td>
            <td>Pedro</td>
            <td>Ramírez</td>
            <td>M</td>
        </tr>
    </tbody>
</table>
</body>
</html>

```

Si observas bien, puedes ver que la estructura se conserva. Es decir, se repite la parte del código en donde dice <tr><td>...

La tabla, al mostrarla, se ve así:

ID	Nombre	Apellidos	Género
1	John	Doe	M
2	María José	Ejemplo	F
3	Pedro	Ramírez	M

Pero todo esto ya lo sabes... la idea es que genere la tabla automáticamente en función del contenido de la base de datos.

10.2 Tabla dinámica

Antes de nada, vamos a poner el código del index.php para no tocarlo más.

Lo que podemos intuir, es que para cada acción: crear, guardar, listar, borrar, ver.... Tendremos un require asociado.

index.php (final)

```

<?php
require_once ("views/menu/head.php");

if (isset($_REQUEST["accion"])){
    switch ($_REQUEST["accion"]){
        case "crear":
            require_once ("views/persona/create.php");
            break;
        case "guardar":
            require_once ("views/persona/store.php");
            break;
        case "ver":
            require_once ("views/persona/show.php");
            break;
    }
}

```

```

        case "listar":
            require_once ("views/persona/list.php");
            break;
        case "buscar":
            require_once ("views/persona/search.php");
            break;
        case "borrar":
            require_once ("views/persona/delete.php");
            break;
        case "editar":
            require_once ("views/persona/edit.php");
            break;
        default:
            require_once ("views/404.php");
            break;
    }
}
else{// Vista Por defecto
    ?>
    <a href="index.php?accion=crear" class="btn btn-primary">Agregar Usuario</
a>
<?php
}
require_once ("views/menu/footer.php");

```

Si lo que queremos es mostrar todos los datos, lo que tenemos que hacer es una consulta a la base de datos y extraer sus datos. Claro, el problema es que esto implica tocar tanto modelo como controlador, antes de imprimir los datos. Vamos a ello:

Añadimos en `personaModel.php`

```

public function readAll():array
{
    $sentencia = $this->conexion->query("SELECT * FROM personas;");
    //usamos método query
    $personas = $sentencia->fetchAll(PDO::FETCH_ASSOC);
    return $personas;
}

```

Vayamos con el código.

```
$sentencia = $conexion->query("SELECT * FROM personas;");
```

En este caso no utilizamos una consulta preparada, sino usamos el método `query`. Por supuesto, podría ser una consulta preparada.

NOTA: ¿Por qué usamos `query`? El único motivo es ver formas diferentes de ejecutar consulta, pero yo soy **MUY DE CONSULTA PREPARADAS**, por tanto vosotros también

A diferencia del caso de la inserción, donde la ejecución de la instrucción SQL devuelve `true/false`, en el caso de consultas de tipo **SELECT** se almacena en `$sentencia` el resultado de la consulta. **Ojo**, `$sentencia`, es un objeto, con los datos recibidos de la consulta, por tanto, si quiero acceder a los datos que contiene el objeto `$sentencia`, utilizaré los métodos de acceso a datos (`fetch`)

```
$personas = $sentencia->fetchAll(PDO::FETCH_ASSOC);
```

En este caso, para recoger los datos usamos el método **`fetchAll`**. Este método devuelve todo el conjunto de datos resultantes de la consulta. Vamos que me devuelve en un array asociativo con todos los datos de la consulta.

El parámetro que pasamos a `fetchAll` es una constante estática pública de la clase PDO, (`PDO::FETCH_ASSOC`) y permite acceder a las filas de la tabla en formato array asociativo. Es decir, podemos acceder al nombre de la persona usando `$persona["nombre"]` en lugar de (es la opción que está por defecto) `$persona[1]`. El `fetchAll` tiene más opciones, por tanto existen más constantes, por ejemplo `PDO::FETCH_OBJ`, que como hemos visto antes, crea objetos genéricos, lo que me permite acceder con la notación:

`$persona->nombre`

Existen más tipos de acceso, pero eso lo dejamos para los apuntes.

En este caso lo que tengo en `$persona` es un array de objetos.

Añadimos en `personasController.php`

```
public function listar () {
    return $this->model->readAll ();
}
```

Este código no puede ser más simple. Devolvemos el array de personas a la página principal.

Por último nos queda definir la vista

list.php

```
<?php
require_once "controllers/personasController.php";

$controlador= new PersonasController();
$personas= $controlador->listar ();
?>

<table class="table table-light table-hover">
<thead class="table-dark">
    <tr>
        <th scope="col">ID</th>
        <th scope="col">Usuario</th>
        <th scope="col">Nombre</th>
        <th scope="col">Apellidos</th>
        <th scope="col">Genero</th>
    </tr>
</thead>
<tbody>
<?php
    foreach($personas as $persona): ?>
        <tr>
            <th scope="row"><?=$persona["id"]?></th>
            <td><?=$persona["usuario"]?></td>
            <td><?=$persona["nombre"]?></td>
            <td><?=$persona["apellidos"]?></td>
            <td><?=( $persona["genero"]=="M")?"Masculino":"Femenino" ?></td>
        </tr>
<?php
    endforeach;
    ?>
</tbody>
</table>
```

Como tengo un array de personas, lo recorro usando un `foreach`. En cada iteración almaceno en el objeto `$persona` los datos que leo de la base de datos.

Sólo una cosa a comentar del código:

```
<?php
    foreach($personas as $persona): ?>
```

```

        <tr>
            <th scope="row"><?=$persona["id"]?></th>
            <td><?=$persona["usuario"]?></td>
            <td><?=$persona["nombre"]?></td>
            <td><?=$persona["apellidos"]?></td>
            <td><?=( $persona["genero"]=="M")?"Masculino":"Femenino" ?></td>
        </tr>
    <?php
        endforeach;
    ?>

```

¿¿¿Foreach sin llaves???? WTF???

A ver esa notación no es la común, pero cuando se está en una vista, donde mezclamos php y html, muchas veces las llaves lo único que hacen es complicar la lectura, así que existe una notación alternativa, con la intención de facilitar su lectura.

foreach (\$personas as \$persona):

.....

endforeach;

Por lo demás es montar un tabla y poner algo de bootstrap.

10.3 Agregando enlaces

Para eliminar o actualizar vamos a necesitar el id de la persona. Si no, ¿a qué persona eliminaremos o editaremos?.

Combinando HTML y PHP podemos crear un link que dirija a cierto archivo, pasándole el id. Y luego leer ese id. (GET implícito).

Por el momento sólo hay que crear los enlaces, no hace falta explicar el funcionamiento. Así que la versión 3 del código queda así:

Antes de nada añade al head.php

```

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">

```

Este css es necesario para que se carguen los iconos que le añadiremos a los botones.

list.php

```

<?php
require_once "controllers/personasController.php";

$controlador= new PersonasController();
$personas= $controlador->listar ();
?>

<table class="table table-light table-hover">
<thead class="table-dark">
    <tr>
        <th scope="col">ID</th>
        <th scope="col">Usuario</th>
        <th scope="col">Nombre</th>
        <th scope="col">Apellidos</th>
        <th scope="col">Genero</th>
        <th scope="col">Eliminar</th>
        <th scope="col">Editar</th>
    </tr>
</thead>
<tbody>
<?php foreach($personas as $persona):

```









```

        $id=$persona["id"];
    ?>
    <tr>
        <th scope="row"><?=$persona["id"]?></th>
        <td><?=$persona["usuario"]?></td>
        <td><?=$persona["nombre"]?></td>
        <td><?=$persona["apellidos"]?></td>
        <td><?=( $persona["genero"]=="M")?"Masculino":"Femenino" ?></td>
        <td><a class="btn btn-danger" href="index.php?accion=borrar&id=<?=$id?
>"><i class="fa fa-trash"></i> Borrar</a></td>
        <td><a class="btn btn-success" href="index.php?accion=editar&id=<?
=$id?"><i class="fa fa-pencil"></i> Editar</a></td>
    </tr>
</tr>
<?php
    endforeach;

    ?>
</tbody>
</table>

```

Al ejecutarse veríamos algo similar a esto:

Inicio Persona ▾						
ID	Usuario	Nombre	Apellidos	Genero	Eliminar	Editar
1	luis	Luis	Cabrera	Masculino	 Borrar	 Editar
2	rosa	Rosa	Martin	Femenino	 Borrar	 Editar
3	ramon	Ramon	Lopez	Masculino	 Borrar	 Editar
4	pedro	Pedro	Perez	Masculino	 Borrar	 Editar

Queda bastante guapo.

(Si no se ve exactamente igual ten en cuenta que he añadido css propios, pero más o menos así quedará).

Si acercas el cursor en donde dice “Borrar” verás que te va a dirigir a un enlace como:

index.php?accion=borrar&id=1

Pero ese id va cambiando en cada fila. Si le das click no te mostrará nada, pues si bien tenemos creadas las acciones en el index (accion: borrar y acción: editar) todavía están vacías las vistas correspondientes..

Eso lo veremos más adelante. Recuerda ya hemos visto la C (Create: insertar datos INSERT), acabamos de ver la R (Read. Leer datos SELECT) y nos falta por ver las U y la D (Update: Editar datos y Delete: Eliminar Datos).

11. ELIMINAR DATOS

Aunque ahora debería venir la U, me adelanto a la D (borrado) pues es bastante más simple. Borraremos los datos a partir de la página de listar datos, esto lo hacemos por qué nos permite “ver” que dato queremos borrar y enviarlo a una página para eliminarlo.

En este momento me atrevo a decir que ya sabéis lo que toca.

- Modificar el Modelo añadiéndole el método delete
- Modificar el Controlador añadiéndole el método/accion borrar.
- Rellenar la vista.

Añadir a personaModel.php

```
public function delete (int $id):bool
{
    $sql="DELETE FROM personas WHERE id =:id";
    try {
        $sentencia = $this->conexion->prepare($sql);
        //devuelve true si se borra correctamente
        //false si falla el borrado
        // Pero, si el id existe el borrado es correcto
        // Pero no borra
        $resultado= $sentencia->execute(["id" => $id]);
        // Si no ha borrado nada considero borrado error
        return ($sentencia->rowCount()<=0)?false:true;
    } catch (Exception $e) {
        echo 'Excepción capturada: ', $e->getMessage(), "<br>";
        return false;
    }
}
```

Creo que el código es bastante claro. Eso sí, por tocar la narices, he cambiado un poco la consulta preparada, he utilizado otra notación (sin ?, por cierto, esta me gusta más, ahí lo dejo).

Observa que a diferencia de antes, no pongo una ?, sino una marca creada por mi **:id**, precedida de **:**. Lo que estamos haciendo es sustituir la marca ? Por **:id**. Evidentemente esto se lo tengo que indicar al método execute en el array, indicando que la marca **:id** se enlaza con **\$id**.

¿Ventajas de esto? Básicamente dos: No importa el orden en el array y me permite hacer un código más legible ya que pondré marcas que me den más información.

Además le añadimos un control de tipo try catch.

Como último comentario, el \$resultado será true siempre que la consulta se ejecute correctamente. Una consulta que no borra se considera correcta.

\$sentencia->rowCount()<=0

rowCount(): devuelve la cantidad de registros afectados por la última consulta, por tanto si devuelve 0, es que aunque la consulta se ha ejecutado correctamente, no ha borrado nada, por tanto consideramos error.

Pequeña gran nota: repito que esto es una introducción con un ejercicio a lo que haremos con base de datos. Así que ten en cuenta que no es buena idea hacer que se elimine usando un link, y mucho menos sin confirmación. ¿Imaginas que Facebook tuviera esa opción para, por ejemplo, eliminar mensajes con alguna persona? supongamos que el link fuera algo como:

`eliminar_mensajes.php?idPersona=111`

Ahora yo te digo que visites ese link, pero antes lo pongo en un acortador para que no haya sospechas. Una vez que hagas click en él, adiós a esa conversación. Obviamente no perdemos nada con eliminar una conversación (al menos que incrimine a alguien, tenga valor sentimental, etcétera) pero basta con esto para darnos una idea.

En este ejemplo eliminamos con el valor pasado por un link y no hay vuelta atrás, pero igual no perdemos nada. Para casos reales, es mejor hacerlo usando **POST**, pues ahí los datos no se ven en la URL. Además, podemos mandar una confirmación, comprobar si el usuario tiene permisos, etcétera.

Por cierto, siempre podríamos poner un confirm con javascript, o algo así.

Añadir a personasController.php

```

public function borrar(int $id):void {
    $borrado= $this->model->delete ($id);
    $redireccion="location:index.php?accion=listar&evento=borrar&id={$id}";
    $redireccion.=( $borrado==false)?"&error=true":"";
    header($redireccion);
}

```

Solamente se llama al método borrar.

delete.php

```

<?php
require_once "controllers/personasController.php";
//pagina invisible
if (!isset ($_REQUEST["id"])) header('Location:index.php' );
//recoger datos
$id=$_REQUEST["id"];

```

```

$controlador= new PersonasController();
$controlador->borrar ($id);

```

Como puedes observar esta página es invisible. Carga el controlador y se realiza el borrado. Al final del código redireccionamos a **listar**, eso sí enviamos que evento ha llamado, en este caso borrar, e indicamos si se ha producido algún error o no.

En el caso de que se produzca un error quiero mostrarlo.¿Dónde? Pues si redirigimos a listar, pues en la vista de listar (**list.php**).

Sustituimos el principio de **list.php**, hasta que empieza el <table> por el siguiente código.

```

<?php
require_once "controllers/personasController.php";

$mensaje = "";
$clase = "alert alert-success";
$visibilidad = "hidden";

$controlador = new PersonasController();
$personas = $controlador->listar();

if (isset($_REQUEST["evento"]) && $_REQUEST["evento"] == "borrar") {
    $visibilidad = "visibility";
    $clase = "alert alert-success";
    //Mejorar y poner el nombre/usuario
    $mensaje = "El usuario con id: {$_REQUEST['id']} Borrado correctamente";
    if (isset($_REQUEST["error"])) {
        $clase = "alert alert-danger ";
        $mensaje = "ERROR!!! No se ha podido borrar el usuario con id: {$_REQUEST['id']}";
    }
}
?>
<div class="<?= $clase?>" <?=$visibilidad?> role="alert">
    <?= $mensaje?>
</div>
//esto ya no se copia
<table class="table table-light table-hover">

```


Lo que hacemos aquí es comprobar si venimos de borrar (**delete.php**) y si es así, en el caso de que se haya producido un error, lo mostramos en rojo y en el caso que todo ok, mostramos un mensaje. Para ello jugamos con los atributos de visibilidad, clase de html y bootstrap.

Ejercicio 1: Que no sólo muestre el ID si no también el usuario, el nombre y los apellidos del elemento borrado. Además añade a la inserción un control vía try/catch.

12. EDITAR, ACTUALIZAR O MODIFICAR DATOS

La actualización sin duda es la operación más compleja de todas. Esto es debido a que en primer lugar debemos conocer el id, luego tenemos que recuperar los datos para mostrarle al usuario lo que ya existe. Finalmente realizar la consulta UPDATE en la base de datos.

Además, como veremos más adelante, tiene todos los problemas del **INSERT** más los problemas del borrado. Pero de momento no nos preocuparemos demasiado en esto.

¿Cómo realizar una actualización?

1. Actualizaremos los datos a partir de la página de listar datos, esto lo hacemos por qué nos permite “ver” que dato queremos modificar y enviarlo a una página para editarlo.
2. Página de edición de datos. En ella cargaremos los datos del registro correspondiente y modificaremos aquellos datos que así lo requieran
3. Página que realmente actualiza datos.

Podríamos realizar el paso 2 y 3 en una misma página pero para esta guía lo haremos en dos.

12.1 Editando Datos

Como en el caso anterior, para llegar aquí debemos pasar por la acción de listar, la cual nos proporcionará el ID del registro a modificar. (recuerda GET implícito).

En esencia edit.php es una página que posee todos los elementos de create.php, por tanto copia el contenido de create.php y realicemos los cambios correspondientes.

Pero antes de ponernos con edit.php, ya sabemos que toca. Modelo y controlador,

personaModel.php

```
public function edit (int $idAntiguo, array $arrayPersona):bool{
    try {
        $sql="UPDATE personas SET nombre = :nombre, apellidos = :apellidos,
genero = :genero, ";
        $sql.= "usuario = :usuario, password= :password ";
        $sql.= " WHERE id = :id;";
        $arrayDatos=[
            ":id"=>$idAntiguo,
            ":usuario"=>$arrayPersona["usuario"],
            ":password"=>$arrayPersona["password"],
            ":nombre"=>$arrayPersona["nombre"],
            ":apellidos"=>$arrayPersona["apellidos"],
            ":genero"=>$arrayPersona["genero"],
        ];
        $sentencia = $this->conexion->prepare($sql);
        return $sentencia->execute($arrayDatos);
    } catch (Exception $e) {
        echo 'Excepción capturada: ', $e->getMessage(), "<br>";
        return false;
    }
}
```

```
}
```

Simplemente generamos una consulta de tipo UPDATE y devolvemos true o false si se ha modificado los datos. Observa que le pasamos el \$id anterior. Si bien no lo modificamos podríamos modificarlo.

Añadir a personasController.php

```
public function editar (int $id, array $arrayPersona):void {
    $editadoCorrectamente=$this->model->edit ($id, $arrayPersona);
    $redireccion="location:index.php?accion=editar&evento=guardar&id={$id}";
    $redireccion.=( $editadoCorrectamente==false)?"&error=true":"";
    //vuelvo a la pagina donde estaba
    header ($redireccion);
}
```

Modifico y vuelvo a la página donde estábamos. Para distinguir entre el caso de que acabo de llegar, o he modificado utilizamos la variable evento=guardar, para indicar a editar que vengo de guardar información. Esto me facilitará los mensajes.

edit.php

```
<?php
require_once "controllers/personasController.php";
//recoger datos
if (!isset ($_REQUEST["id"])) header('location:index.php?accion=listar');
$id=$_REQUEST["id"];
$controlador= new PersonasController();
$persona= $controlador->ver ($id);

$visibilidad="hidden";
$mensaje="";
$clase= "alert alert-success";
$mostrarForm=true;
if($persona==null) {
    $visibilidad="visbility";
    $mensaje="La persona con id: {$id} no existe. Por favor vuelva a la pagina anterior";
    $clase= "alert alert-danger";
    $mostrarForm=false;
}

if (isset ($_REQUEST["evento"]) && $_REQUEST["evento"]=="guardar"){
    $visibilidad="vibility";
    $mensaje="Persona con id {$id} Modificado con éxito";
    if (isset($_REQUEST["error"])){
        $mensaje="No se ha podido modificar el id {$id}";
        $clase="alert alert-danger";
    }
}
?>
<div class="<?=$clase?>" <?=$visibilidad?>> <?=$mensaje?> </div>
<?php
if ($mostrarForm) {
?>
<form action="index.php?accion=guardar&evento=editar" method="POST">
    <input type="hidden" id="id" name="id" value="<?=$persona->id?>">
    <div class="form-group">
        <label for="usuario" >Usuario </label>
        <input type="text" required class="form-control" id="usuario"
name="usuario" aria-describedby="usuario" value="<?=$persona->usuario?>">
        <small id="usuario" class="form-text text-muted">Compartir tu usuario lo
```

```

hace menos seguro.</small>
</div>
<div class="form-group">
  <label for="password" >Password</label>
  <input type="password" required class="form-control" id="password"
name="password" value="<?=$persona->password?>">
</div>
<div class="form-group">
  <label for="nombre">Nombre </label>
  <input type="text" class="form-control" id="nombre" name="nombre"
value="<?=$persona->nombre?>">
</div>
<div class="form-group">
  <label for="apellidos">Apellidos </label>
  <input type="text" class="form-control" id="apellidos" name="apellidos"
value="<?=$persona->apellidos?>">
</div>
<div class="form-radio">
  <input type="radio" class="form-radio-input" id="genero1" name="genero"
value="M" <?=( $persona->genero=='M')?'checked':'?'>>
  <label class="form-radio-label" for="genero1">Masculino</label>
  <input type="radio" class="form-radio-input" id="genero2" name="genero"
value="F" <?=( $persona->genero=='F')?'checked':'?'>>
  <label class="form-radio-label" for="genero2">Femenino</label>
</div>
<button type="submit" class="btn btn-primary">Guardar</button>
<a class="btn btn-danger" href="index.php?accion=listar">Cancelar</a>
</form>
<?php
}
else{
  ?>
  <a href="index.php" class="btn btn-primary">Volver a Inicio</a>
  <?php
}

```

Este código es mucho más recargado, pero en realidad no es tan complejo.

La parte inicial:

```

<?php
require_once "controllers/personasController.php";
//recoger datos
if (!isset ($_REQUEST["id"])) header('location:index.php?accion=listar');
$id=$_REQUEST["id"];
$controlador= new PersonasController();
$persona= $controlador->ver ($id);

$visibilidad="hidden";
$mensaje="";
$clase= "alert alert-success";
$mostrarForm=true;
if($persona==null) {
  $visibilidad="visbility";
  $mensaje="La persona con id: {$id} no existe. Por favor vuelva a la pagina
anterior";
  $clase= "alert alert-danger";
  $mostrarForm=false;
}

```

Todo esto sólo son controles pensados en el caso de que alguien intente acceder sin pasar por listar.

```

if (!isset ($_REQUEST["id"])) header('location:index.php?accion=listar');

```

Si tengo el id, creo un objeto standard con la información de persona.

```
$id=$_REQUEST["id"];
$controlador= new PersonasController();
$persona= $controlador->ver ($id);
```

En el caso de de que exista el id, pero no existe dicha persona, es decir \$persona==null, no mostremos el formulario de edición (como edito una persona que no existe)

```
$visibilidad="hidden";
$mensaje="";
$clase= "alert alert-success";
$mostrarForm=true;
if($persona==null) {
    $visibilidad="visbility";
    $mensaje="La persona con id: {$id} no existe. Por favor vuelva a la pagina anterior";
    $clase= "alert alert-danger";
    $mostrarForm=false;
}
```

Pero los casos anteriores son si es la primera vez, que pasa si ya he actualizado. Pues realizo los controles correspondientes para imprimir un mensaje adecuado.

```
if (isset ($_REQUEST["evento"]) && $_REQUEST["evento"]=="guardar"){
    $visibilidad="vibility";
    $mensaje="Persona con id {$id} Modificado con éxito";
    if (isset($_REQUEST["error"])){
        $mensaje="No se ha podido modificar el id {$id}";
        $clase="alert alert-danger";
    }
}
```

Este es la zona de impresión de mensajes.

```
<div class="<?=$clase?>" <?=$visibilidad?>> <?=$mensaje?> </div>
```

Por último distinguimos entre mostrar Formulario o no. Evidentemente si no existe la persona, no muestro el formulario, si existe la muestro.

Aparentemente he acabado. Pero no, queda una parte muy importante:

```
<form action="index.php?accion=guardar&evento=editar" method="POST">
```

Observemos el action

```
action="index.php?accion=guardar&evento=editar"
```

En realidad, ¿a quién llamo para guardar? A **store.php**. Por tanto debo realizar los cambios correspondientes en store.php

Añadir al final store.php

```
if ($_REQUEST["evento"]=="editar"){
    //devuelve true si edita false si falla
    $controlador->editar ($id, $arrayPersona);
}
?>
```

Por fin le vemos la utilidad a la variable evento. La usamos para distinguir entre si vamos almacenar un registro nuevo o almacenar la modificación de uno antiguo

13. BUSCAR DATOS

En esencia, buscar es un listar ampliado. Así que podríamos copiar el **list.php** en **search.php**

Vayamos al código directamente

Añadir en **personaModel.php**

```
public function search (string $usuario):array{
    $sentencia = $this->conexion->prepare("SELECT * FROM personas WHERE
usuario LIKE :usuario");
    //ojo el si ponemos % siempre en comillas dobles "
    $arrayDatos=(":usuario"=>"%$usuario%");
    $resultado = $sentencia->execute($arrayDatos);
    if (!$resultado) return [];
    $personas = $sentencia->fetchAll(PDO::FETCH_ASSOC);
    return $personas;
}
```

Añadir en **personasController.php**

```
public function buscar (string $usuario):array {
    return $this->model->search ($usuario);
}
```

search.php

```
<?php
require_once "controllers/personasController.php";

$mensaje = "";
$clase = "alert alert-success";
$visibilidad = "hidden";
$mostrarDatos=false;
$controlador = new PersonasController();
$usuario="";

if (isset($_REQUEST["evento"])){
    $mostrarDatos=true;
    switch ($_REQUEST["evento"]){
        case "todos":
            $personas = $controlador->listar();
            $mostrarDatos=true;
            break;
        case "filtrar":
            $usuario=(($_REQUEST["busqueda"]))?" ";
            $personas = $controlador->buscar($usuario);
            break;
        case "borrar":
            $visibilidad = "visibility";
            $mostrarDatos=true;
            $clase = "alert alert-success";
            //Mejorar y poner el nombre/usuario
            $mensaje = "El usuario con id: {$_REQUEST['id']} Borrado
correctamente";
            if (isset($_REQUEST["error"])) {
                $clase = "alert alert-danger ";
                $mensaje = "ERROR!!! No se ha podido borrar el usuario con id:
{$_REQUEST['id']}";
            }
    }
}
```

```

        break;
    }
}

?>
<div class="<?= $clase ?>" <?= $visibilidad ?> role="alert">
    <?= $mensaje ?>
</div>

<form action="index.php?accion=buscar&evento=filtrar" method="POST">
<div class="form-group">
    <label for="usuario">Buscar Usuario</label>
    <input type="text" required class="form-control" id="busqueda"
name="busqueda"
    value="<?=$usuario?>" placeholder="Buscar por Usuario">
</div>
    <button type="submit" class="btn btn-success"
name="Filtrar">Buscar</button>
</form>
    <!-- Este formulario es para ver todos los datos -->
<form action="index.php?accion=buscar&evento=todos" method="POST">
    <button type="submit" class="btn btn-info" name="Todos">Ver
todos</button>
</form>

<?php
if ($mostrarDatos){
    ?>
<table class="table table-light table-hover">
    <thead class="table-dark">
        <tr>
            <th scope="col">ID</th>
            <th scope="col">Usuario</th>
            <th scope="col">Nombre</th>
            <th scope="col">Apellidos</th>
            <th scope="col">Genero</th>
            <th scope="col">Eliminar</th>
            <th scope="col">Editar</th>
        </tr>
    </thead>
    <tbody>
        <?php foreach ($personas as $persona) :
            $id = $persona["id"];
            ?>
            <tr>
                <th scope="row"><?= $persona["id"] ?></th>
                <td><?= $persona["usuario"] ?></td>
                <td><?= $persona["nombre"] ?></td>
                <td><?= $persona["apellidos"] ?></td>
                <td><?= ($persona["genero"] == "M") ? "Masculino" : "Femenino"
?></td>
                <td><a class="btn btn-danger" href="index.php?accion=borrar&id=<?=
$id ?>"><i class="fa fa-trash"></i> Borrar</a></td>
                <td><a class="btn btn-success" href="index.php?accion=editar&id=<?=
$id ?>"><i class="fa fa-pencil"></i> Editar</a></td>
            </tr>
        <?php
        endforeach;

        ?>
    </tbody>

```

```
</table>
<?php
}
?>
```

Aunque el código sea largo, básicamente definimos 3 eventos en el switch:

1. todos: Ver todos los elementos, de hecho llamamos a listar. Esto se producirá al pulsar el botón todos.
2. Filtrar: Nos llamamos a nosotros mismos e invocamos el método buscar, el cual, con el dato de búsqueda que le haya pasado devolverá el array de personas que cumple los requisitos de la búsqueda. Observa que realizamos en el modelo una búsqueda de tipo: **campo LIKE %valor%**.
3. Borrar: Este está copiado de listar.php. Como está copiando de listar.php, cuando borro me redirecciona a listar, esto puede ser discutible, pero es lo que se ha hecho.

Además le añadimos controles y una variable para decidir si hay que mostrar datos o no.

Y para llamar a todos o filtrar generamos dos formularios muy básicos que son los que crean los distintos eventos.

Resumiendo: Este código lo que nos hace es una búsqueda utilizando el campo usuario de cualquier usuario que contenga la cadena que le hemos pasado.

¿Qué tendríamos que hacer para que la búsqueda fuese por otros campos?

¿Y que además de elegir campo pudiese elegir tipos de búsqueda? Por ejemplo: Empieza por, Contiene, Es igual a, Acaba por...

Bien ya tenemos ejercicio.

Ejercicio 2. Modifica el código anterior para que pueda realizar la búsqueda por los campos: nombre, apellidos y genero. Además me permita buscar por el inicio, por el final, si contiene dicha palabra o es igual a la misma.

Ejercicio 3. Una vez finalizado los anteriores, realiza el CRUD de Digimon.