

Documentación del Proyecto para aplicación de los transportistas

1. Introducción

Este documento describe las funcionalidades, estructura y requisitos del proyecto para una aplicación para interactuar con la base de datos.

Para realizar las consultas, se realizan a través de procedimientos almacenados en sql.

Tras iniciar sesión un usuario, el usuario tiene un panel principal donde dispondrá de un menú desplegable con cuatro funcionalidades y un botón salir. A mayores del menú, tiene un botón de inicio y un panel de control (body) con todas las funcionalidades (actualmente, una solo).

Actualmente, su única funcionalidad *lista inventario*, realiza una serie de consultas, la primera cuenta con dos bombos en los que se selecciona el tipo y el estado. Se mostrará y se podrá seleccionar un tipo y estado y mostrará una tabla tras realizar otra consulta mediante un procedimiento almacenado. Se podrá seleccionar una de las filas de la tabla (inventario), lo que realizará otro procedimiento almacenado y nos redirige a una nueva vista que muestra una tabla de ese inventario en la que se podrán modificar la fecha de caducidad o el número de productos reales.

2. Estructura

ApplicacionCofanoMVC/

| — index.php

| — styles.css

|

| — controller/

| | — LoginController.php

| | — LogoutController.php

|

| — model/

| | — connexion.php

| | — user.php

|

| — view/

| | — dashboardView.php

```
|   └─ loginView.php
|
└─ lista_inventario/
    └─ controller/
        ├── DetalleInventarioController.php
        └─ InventarioController.php
    └─ model/
        ├── DetalleInventarioModel.php
        └─ InventarioModel.php
    └─ view/
        ├── DetalleInventarioView.php
        └─ InventarioView.php
|
└─ templates/
    ├── header.php
    └─ header-login.php
|
└─ js/
    ├── paginacion_detalle_inventario.js
    └─ paginacion_inventario.js
|
└─ imagenes/
    └─ logocofano.png
```

3. Relación de los archivos

El proyecto sigue una arquitectura modelo-vista-controlador (MVC).

El punto de entrada para la aplicación es *index.php*. Y la hoja de estilos principal para todas las carpetas (menos para la funcionalidad lista inventario) es *styles.css*.

El flujo general:

1. El usuario accede a una vista (archivo */view/*).
2. La vista envía solicitudes a un controlador (archivo */controller/*).

3. El controlador procesa la solicitud y consulta/modifica datos a través del modelo (archivo /model/).
4. El modelo interactúa con la base de datos (conexion.php).
5. El controlador devuelve la respuesta a la vista, que la muestra al usuario.

Para la autenticación:

1. El usuario accede a loginView.php.
2. Envía sus credenciales a LoginController.php.
3. LoginController.php valida los datos con user.php (del model) en la base de datos.
4. Si es exitoso, redirige a dashboardView.php.

4. Modelo-Vista-Controlador

Model

Los modelos manejan la lógica y la conexión con la BBDD.

conexión.php

Manejo de conexión con la base de datos.

El código:

- Maneja la conexión con la base de datos mediante PDO y ODBC.
- Usa un patrón Singleton (self::\$pdo === null) para evitar múltiples conexiones.
- Si falla la conexión, se captura la excepción y muestra un error.

user.php

Gestiona los usuarios.

Código:

- Valida usuarios llamando al procedimiento almacenado PAVALIDAUSR.
- Utiliza parámetros de salida (bindParam con INPUT_OUTPUT) para obtener la respuesta.
- Devuelve un array con el resultado de autenticación y un mensaje.

Controller

El controlador recibe las solicitudes, gestiona la lógica y devuelve la respuesta.

LoginController.php

Maneja el inicio de sesión.

Código:

- Recibe el formulario de login y verifica si los campos están vacíos.
- Llama al modelo (validarUser) para verificar las credenciales.
- Si la autenticación es correcta, guarda el usuario en \$_SESSION y redirige al Dashboard.
- Si falla, muestra un mensaje de error.

LogoutController.php

Cierra la sesión.

Código:

- Elimina la sesión actual (\$_SESSION = array();).
- Elimina las cookies asociadas a la sesión.
- Redirige al login (loginView.php).

View

Las vistas muestran la información.

loginView.php

Formulario de inicio de sesión.

En caso de error en la sesión, los muestra y luego los elimina.

dashboardView.php

Es la página a la que se redirige una vez iniciado sesión.

5. Funcionalidad *lista_inventario*

La estructura de esta funcionalidad es:

```
├── lista_inventario/  
|   ├── controller/  
|   |   ├── DetalleInventarioController.php  
|   |   ├── InventarioController.php  
|   |   └── model/  
|   |       ├── DetalleInventarioModel.php  
|   |       └── InventarioModel.php  
|   └── view/  
|       ├── DetalleInventarioView.php  
|       └── InventarioView.php
```

Flujo de Funcionamiento

- El usuario accede al inventario → `InventarioController.php` obtiene la lista desde `InventarioModel.php` y la muestra en `InventarioView.php`.
- El usuario selecciona un inventario → `DetalleInventarioController.php` obtiene la información desde `DetalleInventarioModel.php` y la muestra en `DetalleInventarioView.php`.
- El usuario modifica la cantidad o la fecha de caducidad → Se envía una solicitud a `DetalleInventarioController.php`, que actualiza la base de datos con `DetalleInventarioModel.php`.

A mayores, se implementa una paginación para `InventarioView.php` y otra para `DetalleInventarioView.php` para filtrado y ordenación. Estas paginaciones, se encuentran en los códigos de java script (carpeta `/js/`).

Model

Los modelos son responsables de interactuar con la base de datos para obtener y actualizar información del inventario.

InventarioModel.php

- Este modelo gestiona el inventario, obteniendo datos sobre estados, tipos y listas de inventarios desde la base de datos.
- Incluye métodos:
 - `getEstadosTipos()`: Obtiene los estados y tipos de productos ejecutando `PABLOG.PASELTXINV(1)` y `PABLOG.PASELTXINV(3)`.
 - `getInventarios($estado, $tipo)`: Obtiene la lista de inventarios según filtros de estado y tipo, ejecutando `PABLOG.PASELLSTINV(?, ?)`.

DetalleInventarioModel.php

- Este modelo gestiona los detalles del inventario y la actualización de stock.
- Incluye métodos:
 - `getDetalleinventario($idinventario)`: Obtiene detalles de un inventario ejecutando `PABLOG.PASELDETINV(?)`.
 - `saveCambiosDetalle($idinventario, $ean, $uds, $caducidad, $usuario)`: Actualiza datos en la base de datos con `PABLOG.PAUPDINV(?, ?, ?, ?, ?)`.

Controller

Los controladores gestionan la lógica y conectan los modelos con las vistas.

InventarioController.php

- Funciones:
 - `obtenerEstadosTipos()`: Obtiene estados y tipos del inventario.

- obtenerInventarios(\$estado, \$tipo): Filtra inventarios según estado y tipo.

DetalleInventarioController.php

- Funciones posibles:
 - obtenerDetalleInventario(\$idInventario): Obtiene los detalles de un inventario.
 - guardarCambiosDetalle(\$idInventario, \$ean, \$uds, \$caducidad, \$usuario): Guarda los cambios en la base de datos.

View


InventarioView.php muestra el listado de inventarios con filtros y DetalleInventarioView.pphp muestra detalles de inventario con opción para modificar stock.

Respecto al **script** del semáforo de DetalleInventarioView.php:

El script realiza tres funciones principales:

Resumen del funcionamiento


Detectar cambios:

- Si el usuario edita una celda (input), el semáforo cambia a  y el botón "Guardar" se habilita.



Guardar un solo cambio:

- Cuando el usuario presiona el botón "Guardar", los datos se envían al servidor y el semáforo vuelve a .

Guardar todos los cambios a la vez:

- Al presionar "Procesar cambios", se recopilan todas las filas modificadas y se envían al servidor.
- Una vez guardado, los semáforos vuelven a rojo .

1. Detectar cambios en los campos editables

Cuando el usuario modifica un valor en las celdas editables (unidades o fecha de caducidad), el semáforo cambia de  (rojo) a  (verde) y el botón "Guardar" se habilita.

Código:

```
$(".editable").on("input", function(){
    let row = $(this).closest("tr");
```

```

    row.find(".semaforo").removeClass("rojo").addClass("verde").text("●");
    row.find(".guardar-btn").prop("disabled", false);
  });

```

Explicación:

- Se detecta un cambio en los campos editables (.editable).
- Se busca la fila (<tr>) más cercana a la celda modificada usando closest("tr").
- Se cambia la clase del semaforo (.semaforo) de rojo a verde (removeClass("rojo").addClass("verde")).
- Se actualiza el texto del semáforo a "●" (text("●")).
- Se habilita el botón "Guardar" (.guardar-btn.prop("disabled", false)).

2. Guardar cambios individualmente

Cuando el usuario hace clic en el botón "Guardar" de una fila, se envían los datos al servidor para actualizar la base de datos y luego el semáforo vuelve a rojo (●).

Código:

```

$(".guardar-btn").click(function(){
    let row = $(this).closest("tr");
    let idInventario = <?= $idInventario; ?>;
    let usuario = "<?= $usuario; ?>";
    let idEan = row.find(".guardar-btn").data("id");
    let uds = row.find(".uds").val();
    let caducidad = row.find(".caducidad").val();

    $.post("../controller/DetalleInventarioController.php", {
        action: "guardar",
        id_inventario: idInventario,
        ean: idEan,
        uds: uds,
        caducidad: caducidad,
        usuario: usuario
    }, function(response){
        alert(response);
    });

```

```

row.find(".semaforo").removeClass("verde").addClass("rojo").text("●");
    row.find(".guardar-btn").prop("disabled", true);
});
});

```

Explicación:

1. Captura la fila (<tr>) donde se hizo clic en "Guardar".
2. Recoge los valores del inventario:
 - idInventario: ID del inventario.
 - idEan: Código EAN del producto.
 - uds: Nuevas unidades ingresadas.
 - caducidad: Nueva fecha de caducidad.
 - usuario: Usuario que realizó el cambio.
3. Envía los datos al servidor usando AJAX (\$.post).
4. Después de guardar:
 - Muestra un mensaje con alert(response).
 - Cambia el semáforo a rojo ● (removeClass("verde").addClass("rojo")).
 - Deshabilita el botón "Guardar" (.prop("disabled", true)).

3. Guardar todos los cambios a la vez

Cuando el usuario presiona el botón "Procesar cambios", el script busca todas las filas modificadas (verde ●) y las envía al servidor en un solo lote.

Código:

```

$("#procesar-todos").click(function(){
    let cambios = [];

    $("tr").each(function() {
        let row = $(this);

        if (row.find(".semaforo").hasClass("verde")) { // Se procesan y
guardan las filas modificadas

            let idInventario = <?= $idInventario; ?>;
            let idEan = row.find(".guardar-btn").data("id");
            let uds = row.find(".uds").val();
            let caducidad = row.find(".caducidad").val();

            cambios.push({

```



```

        id_inventario: idInventario,
        ean: idEan,
        uds: uds,
        caducidad: caducidad,
        usuario: "<?= $usuario; ?>"
    });
}
});

if (cambios.length > 0) {
    $.post("../controller/DetalleInventarioController.php", {
        action: "guardar_todos",
        cambios: JSON.stringify(cambios)
    }, function(response) {
        alert(response);
        $("tr").each(function() {
            let row = $(this);

            row.find(".semaforo").removeClass("verde").addClass("rojo").text("●");
            row.find(".guardar-btn").prop("disabled", true);
        });
    });
} else {
    alert("No hay cambios para procesar.");
}
});

```

Explicación:

1. Se crea un array cambios para almacenar todas las filas modificadas.
2. Recorre todas las filas (<tr>) de la tabla:
 - Si la fila tiene el semáforo en verde (.hasClass("verde")), significa que ha sido modificada.
 - Se capturan los valores idInventario, idEan, uds, caducidad y usuario.

- Se añaden estos datos al array cambios[].
- 3. Si hay cambios, los envía al servidor con \$.post() en formato JSON.
- 4. Después de la respuesta del servidor:
 - Muestra una alerta con el resultado.
 - Vuelve todos los semáforos a rojo ●.
 - Deshabilita los botones de guardar.