

TEMA 3.3

Validación y persistencia

Validando la información: Doble validación

1. El navegador realiza algunas validaciones en campos de formularios. Por ejemplo, si el input type es un "email", el navegador comprueba que el email esté correctamente escrito antes de hacer el submit. Ahora bien, no todos los navegadores son compatibles con estos atributos, por lo que no es una validación fiable.
2. Podemos realizar las validaciones de los campos de entrada en el cliente, mediante **JavaScript**.
3. En el servidor podemos hacer también validaciones mediante **PHP**.
4. Lo **recomendable** es realizar siempre una **doble validación**:
 - a. Una en el cliente, con Javascript
 - b. Otra adicional en el servidor, en nuestro caso en PHP

Librerías de validación en PHP

1. Aprenderemos a escribir nuestro propio código de validación con las funciones del lenguaje PHP. No obstante, es más fácil reutilizar código de validación ya probado.
2. Existen librerías de funciones que facilitan la validación de los formularios
3. A partir de **PHP 5**, puedes utilizar la función **filter_var** de PHP, que permite aplicar filtros diversos a una variable para comprobar si los cumple (comprobar que un email es válido, etc.).

<https://www.php.net/manual/en/function.filter-var.php>

4. Muchos programadores hacen sus propias funciones de validación utilizando expresiones regulares, y las comparten a través de stackoverflow, github, etc
5. Librerías de validación hay varias, por ejemplo, respect/validation o particle/validator
6. Los **frameworks para PHP** suelen incorporar librerías de validación de formularios. Muchas veces puedes instalarlas de forma independiente del propio framework:
 - Symphony forms: Librería de Symphony, instalable independientemente
 - El framework Laravel ofrece librerías de validación de formularios
 - Zen forms: Librería de Zen, instalable de forma independiente
 - El Framework Code Igniter incluye una librería de validación de forms
 - ...y mucho más

Validando con PHP: Función isset

1. Al hacer un request con parámetros, éstos se almacenan en el array `$_GET` o `$_POST`, según el método de request elegido.
2. **La primera vez** que hagamos request a un .php que contenga un formulario, no se ha realizado submit todavía, por lo que la request llega **sin parámetros**.
3. Si se hace **submit**, volverá a hacerse request del .php, esta vez con parámetros.
4. ¿**Cómo sabe el código** si es la primera vez o se pide con submit?: La función **isset** determina si una variable está definida, y si no es null (vacía)
5. Para comprobar en servidor si un parámetro ha sido recibido y tiene información, escribimos:
`isset($_GET["nombre_parametro"])`
6. El resultado será true si se recibió en el request un parámetro con ese nombre, y tiene valor.

```
isset.php
1  <?
2      if (isset($_GET["parametro"])) {
3          $parametro = $_GET["parametro"];
4          // Aquí valido la información
5      }
6  ?>
```

1. Con el **isset** compruebo que se ha recibido en el request un parámetro de nombre "parametro".
2. Una vez comprobado, asigno el valor del parámetro a una nueva variable `$parametro`
3. Finalmente hago las comprobaciones. Por ejemplo, ver si `$parametro` es tipo numérico (nos han colado un carácter), o si es un email y no lleva "@" (no es un email válido)

Otras funciones de validación

- ❑ La validación de los parámetros debe realizarse **antes** que cualquier otra acción que los implique.
- ❑ Para empezar, se verifica el formato de la información:
 - ❑ Ver si un string no está vacío con:
`strlen($string) > 0`
 - ❑ Comprobar que el valor es numérico con:
`is_numeric($valor)`
 - ❑ Ver que al menos hay una "@" en una dirección de email con:
`strpos($mail, '@') > 0`
 - ❑ Comprobar que un email es correcto con:
`filter_var($mail, FILTER_VALIDATE_EMAIL) !== false`
- ❑ Como ya comentamos, **filter_var** dispone de muchos filtros para validaciones típicas, tanto de emails, como de URLs, etc.
- ❑ La forma de comprobar que el **tipo de datos** es correcto es con las funciones de PHP `is_numeric`, `is_float`, `is_string`, etc.

Ejercicio 1:

Prueba de isset



← → ↻ ⓘ localhost/marta/transp-33-1.php

Mi formulario

Texto no recibido en el servidor

Texto

Vamos a experimentar con la función **isset**

1. Crea un programa php que tenga un formulario HTML con un único campo "texto" y un submit.
2. Añade, antes del formulario, un bloque PHP donde se compruebe, con la función **isset**, si el servidor ha recibido el parámetro "texto".
3. Si no se ha recibido, muestra el mensaje "Texto no recibido en el servidor"
4. Si se ha recibido::
 1. Si el contenido es numérico, escribe: "El texto *texto_enviado* es numérico"
 2. Si el contenido es string, escribe : "El texto *texto_enviado* es un string"
5. Prueba el programa, pidiéndoselo a tu localhost desde un navegador:
 1. La primera vez no añadas parámetros en la URL
 2. Luego haz diversos submit:
 1. Con un número en el campo "texto"
 2. Con tu nombre en el campo "texto"
6. Comprueba qué pasa si haces submit con el texto en blanco.
7. Detecta la situación en PHP y saca el mensaje "El texto está en blanco".

PISTA: Puedes utilizar la función `empty` de php para solucionar el punto 7

Persistencia: Cómo no perder los datos del formulario

- Como has comprobado en el ejercicio anterior, cuando hacemos submit del formulario, vuelve a cargarse la página, con lo que **perdemos los datos** introducidos en el formulario.
- Si estoy validando el formulario en el servidor, puedo enviar los errores en el Response (hago echos). Ahora bien, como el formulario HTML programado está vacío, llegará vacío al cliente.
- Se llama **sticky form** (formulario pegajoso), a aquel que no pierde los datos una vez se han validado en servidor.
- Para conseguir que el form no pierda los datos, tenemos que escribir el código necesario que se encargue de volver a poner en el formulario los datos que escribió el cliente.
 1. Los datos los tengo porque han venido en el \$_GET o en el \$_POST
 2. Sabemos que el atributo que escribe datos por defecto en los input es **value**.
 3. Por tanto, tendremos que programar un **contenido variable para el value**, de forma que esté en blanco la primera vez que nos pidan el programa (un request inicial), y con datos cuando nos lo pidan mediante submit.

Ejemplo de persistencia

```
<?php
// Persistencia
$texto_anterior = isset($_GET['prueba']) ? $_GET['prueba'] : "";
?>

<!-- FORMULARIO -->
<h2>Juego de adivinar un número</h2>
<form method="get">
  <label for="prueba">Prueba un número</label>
  <input type="text" name = "prueba"
  | | | | value="<?= $texto_anterior ?>" id="prueba"/>
  <input type="submit">
</form>
```

1. La primera vez que me piden este PHP, `$_GET` está vacío, por tanto `$texto_anterior` estará en blanco.
2. Cuando me piden el programa con un submit, `$_GET` tendrá el parámetro "prueba", cuyo valor será el escrito por el cliente en el navegador
3. En todos los casos, pongo como contenido del `value` la variable `$texto_anterior`, con lo que se asegura la persistencia

Ejercicio 2: Persistencia



A screenshot of a web browser window. The address bar shows 'localhost/marta/transp-33-1.php'. The page title is 'Mi formulario'. Below the title, there is a message 'Texto no recibido en el servidor'. At the bottom, there is a light blue box containing a text input field labeled 'Texto' and a button labeled 'Enviar'.

1. Copia el código del ejercicio 1
2. Modifícalo para conseguir la persistencia del campo texto

Peligro: HTML Injection (inyección de HTML)

- El ejemplo anterior es **el peor de los códigos** posibles. Hemos conseguido la persistencia de forma fácil e intuitiva... atentando **contra nuestra seguridad**.
- Al sustituir directamente el valor de la variable \$texto_anterior por lo que viene en el parámetro, permitimos que un cliente con conocimientos de HTML escriba, en vez de un dato, más código HTML, o javascript.
- El cliente **"inyecta"** HTML en el campo, y yo, en el servidor, lo añado directamente en el formulario, sirviéndolo en bandeja en el response

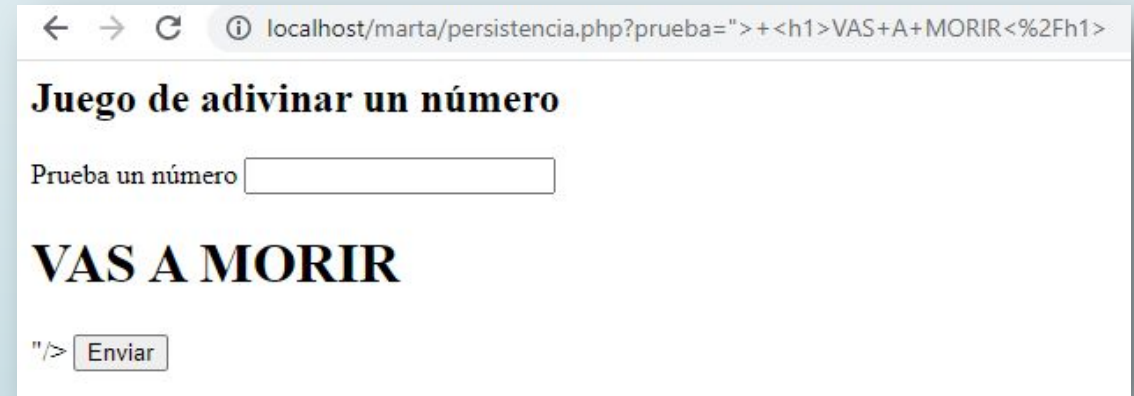


localhost/marta/persistencia.php

Juego de adivinar un número

Prueba un número

Enviar



localhost/marta/persistencia.php?prueba=">+<h1>VAS+A+MORIR<%2Fh1>

Juego de adivinar un número

Prueba un número

VAS A MORIR

> />

Solución al HTML injection: Función htmlentities

- ❑ Para resolver el problema de la inyección de html, se utiliza la función htmlentities
- ❑ Esta función sustituye todos los caracteres aplicables como código HTML en entidades
- ❑ Así, si el cliente escribe ">", htmlentities lo convertirá en ">"
- ❑ El navegador mostrará ">", pero internamente el código es el otro, por lo que no ejerce acción ninguna en HTML



3 - Navegador: Ver código fuente

```
<!-- FORMULARIO -->
<h2>Juego de adivinar un número</h2>
<form method="get">
  <label for="prueba">Prueba un número</label>
  <input type="text" name = "prueba" id="prueba"
    value="&quot;&gt; &lt;h1&gt;VAS A MORIR&lt;/h1&gt;"/>
  <input type="submit">
</form>
```

`$texto_antiguo = "> <h1>VAS A MORIR</h1>"`

`htmlentities($texto_antiguo) = ""> <h1>VAS A MORIR</h1>"`

Peligro: Cross site scripting (scripts maliciosos)

- ❑ En este ejemplo, utilizaremos el mismo código del ejercicio 2, sin la salvaguarda del `htmlentities`
- ❑ Aquí inyectaremos otro botón. A través de botones podríamos incluir scripts maliciosos.




← → ↻ ⓘ localhost/marta/persistencia.php

Juego de adivinar un número

Prueba un número

A blue arrow points from the 'Enviar' button to the next screenshot.



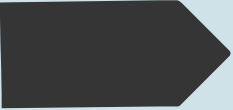
← → ↻ ⓘ localhost/marta/persistencia.php?prueba=" "><hr><input+type%3D"submit"+value%3D"MONSTRUO"%2F><hr>

Juego de adivinar un número

Prueba un número

Ojo: Cualquier persistencia

- Cualquier parámetro recibido que vayamos a insertar en la respuesta al cliente, tiene que pasar por **htmlentities** antes de ser reutilizado en el response.
- Si hago un echo, o pongo la información del parámetro en un <p> o cualquier otro elemento HTML, tengo que "**desinfectarlo**" primero con la función htmlentities.
- Los formularios son la situación más habitual de persistencia, por eso controlamos los value, no obstante hay otras situaciones en que quiero mantener visible la información que introdujo el usuario



Ejercicio 3: Inyección de HTML

1. Copia el código del ejercicio 2
2. Envía el formulario con el texto:
`">VAS A MORIR`
3. Observa qué sucede, no hagas nada
4. Ahora prueba a enviar el texto:
`"><hr><input type="submit" value="MONSTRUO"/><hr>`
5. Ahora prueba a enviar el texto:
`"><hr><button type="button" onclick="alert('Vas-a-morir!!')">MONSTRUO</button><hr>`
6. ¿Por qué ha conseguido el cliente inyectar HTML nocivo?
7. Soluciona el problema

Validación de la entrada

```
<?php
    $titulo = "Adivina";
    include_once("encabezado.php")
?>
<h1>Juego de Adivinar un número</h1>
<?php
    // VALIDACIÓN DE INFORMACIÓN: Ejemplos
    $color = "red";
    $mensaje = "";
    if ( ! isset( $_GET['prueba'])) {
        $mensaje = "Falta el parámetro prueba";
    } else if ( strlen($_GET['prueba']) == 0 ) {
        $mensaje = "Tu número es muy corto!!";
    } else if ( ! is_numeric($_GET['prueba'])) {
        $mensaje = "Tu prueba no es un número";
    } else if ( $_GET['prueba'] > 36 ) {
        $mensaje = "Demasiado alto";
    } else if ( $_GET['prueba'] < 36 ) {
        $mensaje = "Demasiado bajo";
    } else {
        $mensaje = "Felicidades !! Has acertado, es el número 36";
        $color = "green";
    }

    // PERSISTENCIA
    $prueba_anterior = (isset($_GET["prueba"])) ? $_GET["prueba"] : "";
?>
```

- ❑ La validación de los datos se realiza antes de cualquier acción sobre ellos.
- ❑ Las validaciones deben ser exclusivas, es decir, no se sigue comprobando una vez se ha identificado una incidencia. Por eso utilizamos if anidados.
- ❑ En el ejemplo de la izquierda, se prepara un mensaje para mostrar en el navegador, pero NO se muestra la información.
- ❑ La idea es separar el control de la parte de visualización
- ❑ En la página siguiente puedes ver el resto del código

Parte inicial del Código: Controlador

Resto del código del juego de adivinar un número

```
<p>Adivina un número del 1 al 50</p>
<form method="get">
  <p>
    <label for="prueba">Prueba</label>
    <input type="text" name="prueba" id="prueba"
      value="<?=htmlentities($prueba_anterior)?>" />
  </p>
  <input type="submit" />
</form>

<h3 style="color: <?=$color?> "><?=$mensaje?></h3>

<?php include_once("pie.html")?>
```

Parte final del código: "Vista"

Nota: Observa que se ha utilizado htmlentities para la persistencia

localhost/marta/validacion.php?prueba=treinta+y+seis

Juego de Adivinar un número

Adivina un número del 1 al 50

Prueba

Tu prueba no es un número

localhost/marta/validacion.php?prueba=36

Juego de Adivinar un número

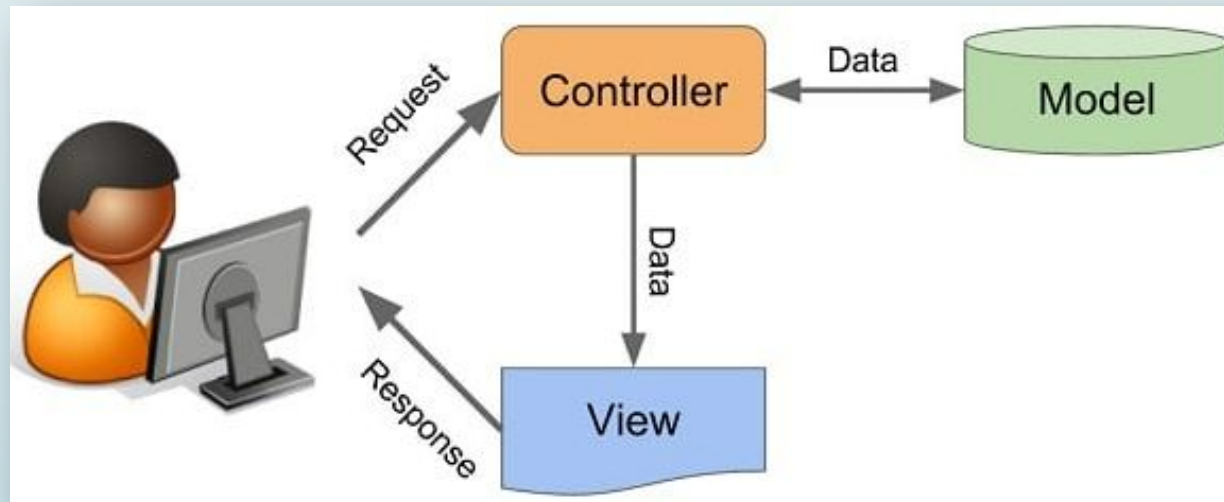
Adivina un número del 1 al 50

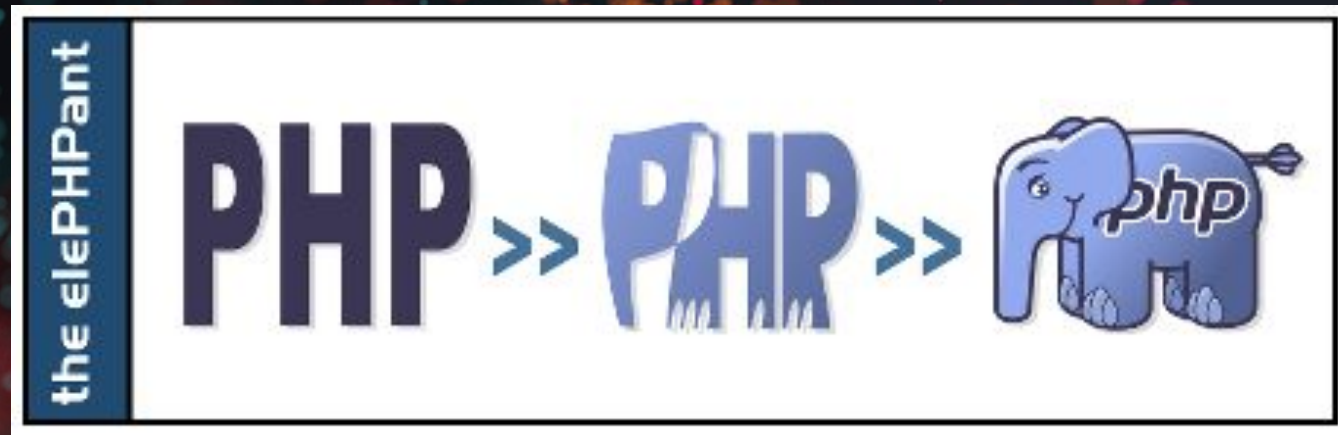
Prueba

Felicidades !! Has acertado, es el número 36

Modelo - Vista - Controlador

- Se ha realizado el ejercicio anterior intentando separar la parte de **control** y acciones del programa, de la parte de **visualización**. Por eso utilizamos la variable \$mensaje en vez de echos
- Es sólo una aproximación, con la idea de que, siempre que sea posible, nos acostumbremos a no mezclar comprobaciones o acciones con visualización.
- Mediante un **framework**, la separación entre vista y modelo es natural, y resulta más sencillo implementarla. Ejemplos: Angular, Symfony, Laravel, etc
- En el ejemplo faltaría la parte del **modelo**, que es la correspondiente a acceso a base de datos. Siempre es silencioso, no emite datos al response (echos, etc)





<https://www.php.net/docs.php>