DigitalHouse >
Coding School

EXPRESS
VALIDATOR

“

Cada vez que le solicitemos al usuario que nos **envíe** información, es importante **validar** esos **datos** tanto desde el *front-end* como desde el *back-end*.

express-validator

Es una librería que nos permite validar campos de una manera sencilla y eficiente.

Para empezar a trabajar con él hace falta instalarlo a través de npm.

```
>_ npm install express-validator --save
```

Al momento de requerir el módulo, nos devolverá un objeto con muchas propiedades. De todas ellas nos interesan sólo tres: check, validationResult y body. Usando destructuring podemos requerirlas para implementarlas en nuestro proyecto.

```
{} const { check, validationResult, body } = require('express-validator');
```

1.

VALIDAR DATOS

PRIMEROS PASOS

En la ruta que va a procesar la información que nos llegue del formulario, deberemos configurar un middleware. *¿Por qué?. Porque estas validaciones tienen que ejecutarse* ***después*** *de que el usuario envió la información, pero* ***antes*** *de que se envíe la respuesta.*

Como todo middleware, deberemos enviarlo **entre** el *request* y el *response*, sólo que en este caso lo enviaremos como un **array**.

{}

```
router.post('/login', [], userController.login);
```

check()

Es una función que nos va a permitir validar campo por campo. La misma recibe como parámetro un string, que será el nombre del campo que queremos validar.

{}

```
router.post('/login', [
     check('email'),
     ...
], userController.login);
```

Recordar que el **nombre del campo** que le pasemos a *check()* deberá ser el mismo que hayamos configurado para ese **campo** en el atributo **name** del formulario.

AGREGANDO VALIDACIONES

Esta librería incluye **métodos** que podemos implementar directamente sobre cada uno de los **checks** que hagamos, para validar diferentes cosas. Algunos de los más usados son:

isLength()

Valida la longitud del campo. Recibe un objeto literal con la propiedad min y max, para configurar la cantidad de caracteres mínimos y máximos que debería tener ese dato.

{}
```
check('campo').isLength( {min:6} )
```

isEmail()

Valida que el dato de ese campo tenga un formato de email correcto.

{}
```
check('campo').isEmail()
```

AGREGANDO VALIDACIONES

isint()

Valida que el dato de ese campo sea un integer. Adicionalmente, puede recibir un objeto literal con la propiedad min y max, para configurar el número mínimo y máximo que puede ser ese dato.

```
check('campo').isInt({min:18, max:99})
```

isEmpty()

Valida que el campo no esté vacío.

```
check('campo').isEmpty()
```

Podés encontrar todos los métodos que incluye la librería en [este link](#).

Las validaciones con **check()** las haremos dentro del **array** que le pasamos como *parámetro* a la **ruta** que esté **procesando** el formulario.

2.

CONFIGURAR
ERRORES

validationResult()

Es una **función** que nos va a permitir capturar los datos que no hayan pasado las validaciones que hicimos con `check()`. Recibe el objeto request, y **retorna** un objeto con los errores del formulario.

En el método del controlador que esté manejando la petición, crearemos la variable `errors` para almacenar en ella lo que devuelva el método `validationResult()`.

{}

```
const userController = {
  login: (req, res) => {
    let errors = validationResult(req);
  }
}
```

ENVIAR LOS ERRORES

El objeto que devuelve `validationResult()` almacena los errores en la propiedad `errors`, que es un **array**. De modo que, si hubo errores, eso es lo que queremos enviar a la vista.

{}

```
const userController = {
  login: (req, res) => {
    let errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.render('login', {errors: errors.errors} );
    }
  }
}
```

MOSTRAR LOS ERRORES

En la vista que retornamos, tendremos que **recorrer** ese array de errores que nos llegó. Cada error será un objeto con propiedades, en donde, en este escenario, nos interesa la propiedad `msg`, que almacena el *mensaje de error* que queremos mostrarle al usuario.

ejs

```ejs
<ul>
  <% for(var i = 0; i < errors.length; i++) { %>
    <li> <%= errors[i].msg %> </li>
  <% } %>
</ul>
```

Cuando el usuario ingresa por **primera vez** al formulario, la variable *errores* que enviamos a la vista **no existe**, ya que la petición aún no pasó por el **controlador** que procesa esos datos.

MOSTRAR LOS ERRORES

Para evitar fallas, por fuera del for que recorre el array, tendremos que crear un condicional que verifique que la variable `errors` no esté indefinida antes de iniciar el bucle.

ejs

```
<% if(typeof errors != 'undefined') { %>
  <ul>
    <% for(var i = 0; i < errors.length; i++) { %>
      <li> <%= errors[i].msg %> </li>
    <% } %>
  </ul>
<% } %>
```

3.

EDITAR
MENSAJES

withMessage()

Es una función que nos va a permitir configurar el mensaje de error del campo a validar Recibe un **string**, que será el que le mostraremos al usuario en la vista en caso de que exista un error en ese campo.

{}

```
router.post('/login', [
      check('email').withMessage('El formato ingresado no es válido'),
      ...
], userController.login);
```

Si bien **express-validator** ya trae resueltas muchas validaciones, puede haber ocasiones en las que necesitemos validar datos de una forma más **personalizada**.

4.

VALIDACIONES PROPIAS

CREANDO VALIDACIONES

Para validar un campo de manera personalizada, haremos uso de dos funciones que nos da la librería:

body()

Recibe un string como parámetro, el nombre del campo que queremos validar.

{}

```
body('email')
```

custom()

Recibe un callback con un **valor**. Dentro desarrollaremos la lógica necesaria para validar ese valor. *La ejecutamos sobre el campo que queremos validar.*

{}

```
body('email').custom(function(value){
     // lógica a implementar
})
```

Al igual que *check()*, las validaciones con **body()** las haremos dentro del **array** que le pasamos como *parámetro* a la **ruta** que esté **procesando** el formulario.