El siguiente documento es a modo de resumen para consultar la documentación ingresar en: https://express-validator.github.io/docs/

express-validator

1 - Instalación del paquete desde la terminal

```
npm install --save express-validator
```

2 - Una vez instalado es necesario requerirlo en los archivos donde se vayan a usar.
=> En archivos de rutas, es donde se realiza la validación de los campos, a través de un middleware.
=> En el controlador, si es que se quiere mostrar los errores en la vista.

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

3 - Como mencionamos, la validación se realiza a través de un middleware. Esto se puede realizar de dos formas:
=> Como middleware directamente en el archivo de rutas.

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

=> Con un middleware externo, si es que dicha validación se repite para otras rutas, se puede reutilizar a través de un archivo middleware. Este archivo a su vez puede contener una única validación, o varias que se utilizan como métodos

```
var {check} = require ('express-validator')

module.exports = [check('email').isEmail().withMessage('No es un email válido'),
        check('password').isLength({
            min: 8
        }).withMessage('La contraseña debe tener 8 caracteres como minimo.'),
    ]
```

Middleware con una sola validación

```
const validationMiddleware = require('../middlewares/validationMiddleware')

router.post('/register', validationMiddleware, userController.register)
```

Archivo de rutas

```
var {check} = require ('express-validator')

module.exports = {
    register: [check('email').isEmail().withMessage('No es un email válido'),
    check('password').isLength({
        min: 8
    }).withMessage('La contraseña debe tener 8 caracteres como minimo.'),
    ],
    login: [check('email').isEmail().withMessage('No es un email válido'),
    check('password').isLength({
        min: 8
    }).withMessage('La contraseña debe tener 8 caracteres como minimo.'),
    ]
}
```

Middleware con varias validaciones

```
const validationMiddleware = require('../middlewares/validationMiddleware')

router.post('/register', validationMiddleware.register, userController.register)
```

Archivo de rutas

4 - Desde el lado del controlador se reciben los errores y se debe hacer un procesamiento. En el controlador necesitaremos requerir validationResult de express-validator. Dentro de validationResult(req) viajan los errores. Si no hay errores, es vacío.
Por esta razón se realiza un condicional, si valitadionResult(req) está vacío, el controlador debe operar la lógica de que no hubo errores, y por ejemplo guardar el usuario en la db.
Por el contrario, si hay errores de validación se suele renderizar nuevamente la vista del formulario mostrando los errores.

```
register: (req, res) => {
    let errors = validationResult(req)
    if (errors.isEmpty()) {

        /* Lógica del controlador si no hay errores */

    } else {
        return res.render('./users/register', {
            errors: errors.mapped()
        })
    }
}
```

5 - Como se mencionó en el punto anterior, si hay errores se debe volver a la vista y mostrarlos, a través de los tag de ejs. Pero es importante saber que la primera vez que se carga un formulario no habrá errores. Por esta razón, se utiliza una lógica en la cuál se pregunta si el *typeof* de la variable *errors* es distinto de *'undefined'* muestre los errores, en caso contrario no muestre nada. De esta manera solo se verán los errores si los mismos viajan desde el controlador.

```
<label for="email">E-mail</label>
<div class="campo-datos">
    <i class="fas fa-envelope"></i>
    <input type="email" name="email" id="email" class="campo-datos-type">
</div>
<div class="register-errors">
    <p>
        <%=typeof errors.email != 'undefined' ?  errors.email.msg : ''%>
    </p>
</div>
```

Podemos ver en esta imagen lo mencionado previamente, realizado para el campo email. Por esta razón es que accedemos haciendo *errors.email* ya que la variable errors contiene los errores para cada campo del formulario que no validó correctamente. A su vez, cada uno de los campos tiene un mensaje asociado que se accede con *errors.campo.msg*. Con la sintáxis mostrada arriba se puede mostrar cada error al lado de cada uno de los campos, para facilitar la comprensión del usuario.

6 - Por último, veamos la sintáxis propia de la validación. El módulo de express-validator tiene muchos componentes, pero únicamente requerimos 3: { check, validationResult, body}.
=> validationResult ya vimos que guarda los errores de validación
=> check se utiliza para realizar validaciones predefinidas en el paquete express-validator
=> body se utiliza para definir validaciones propias que no existan en el paquete.

VALIDACIÓN PREDEFINIDA EN EXPRESS-VALIDATOR

```
check('nombre_del_campo')
.isLength({min:1}).withMessage('Mensaje que queremos que guarde validationResult para la longitud')
.isEmail().withMessage('Mensaje que queremos que guarde validationResult para el email')
```

Como se ve en la imagen, la sintáxis es simple. Dentro del check se pone el nombre del campo a validar y luego se utilizan los métodos predefinidos (en esta página se encuentran todos los métodos predefinidos https://github.com/validatorjs/validator.js#validators), y luego se puede editar el mensaje de error con *.withMessage('mensaje editado')*.
Se pueden realizar varias validaciones a un campo seguidas como se muestra. Para realizar validaciones a otros campos es importante recordar que las validaciones se escriben como un array, donde cada posición del array es la validación de un campo diferente.

Algunos ejemplos de métodos predefinidos son: isLength( ), isEmail( ), isInt( ), isEmpty( ), equals ( ), isNumeric( ).

VALIDACIÓN CUSTOMIZADA

En el caso de las validaciones customizadas, la sintaxis es similar. La diferencia es que se utiliza *body* y no *check*. Dentro de body también se pone el nombre del campo a validar. Luego sigue el método *custom* donde se realiza la lógica de la validación.
En este ejemplo, dentro de *custom* se está recibiendo el valor del campo en *value*, y luego se compara con el contenido del campo de la contraseña. Luego con un condicional se muestra un error si los campos no coinciden. Y se retorna el valor del campo en caso de coincidir. Esta lógica se puede utilizar para validar los campos de *contraseña* y *repetir la contraseña* en un formulario de registro.

```
body('re_password')
    .custom((value,{req, loc, path}) => {
        if (value !== req.body.password) {
            // trow error if passwords do not match
            throw new Error("Las contraseñas deben coincidir");
        } else {
            return value;
        }
    })
```

Existe un punto intermedio en el cuál a un campo se le quiere realizar validaciones predefinidas, pero también validaciones customizadas. Para ello se realiza la siguiente sintaxis, utilizando *check*, los métodos predefinidos, y también *custom*.

```
check('re_password').isLength({min:1}).withMessage('Este campo es obligatorio')
    .custom((value,{req, loc, path}) => {
        if (value !== req.body.password) {
            // trow error if passwords do not match
            throw new Error("Las contraseñas deben coincidir");
        } else {
            return value;
        }
    })
```

BONUS
La siguiente imagen muestra la validación para que una contraseña tenga una longitud mínima de 8 caracteres, y al menos una mínuscula, una mayúscula y un número.

```
check('password')
.isLength( {min:8})
.withMessage('La contraseña debe tener 8 caracteres como mínimo y al menos una minúscula, una mayúscula y un número')
.matches(/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])[0-9a-zA-Z]{8,}$/, "i")
.withMessage('La contraseña debe tener 8 caracteres como mínimo y al menos una minúscula, una mayúscula y un número')
```