Version: 7.2.0

Getting Started

One of the best ways to learn something is by example! So let's roll the sleeves up and get some coding happening.

Set up

The first thing that one needs is a express.js application running. Let's implement one that says hi to someone; for this, create a new file using your favorite language and add the following code:

JavaScript TypeScript

index.js

```
const express = require('express');
const app = express();

app.use(express.json());
app.get('/hello', (req, res) => {
  res.send(`Hello, ${req.query.person}!`);
});

app.listen(3000);
```

Now run this file by executing `node index.js` or `ts-node index.ts` on your terminal.
The HTTP server should be running, and you can open http://localhost:3000/hello?person=John to salute John!

TIP

You can use node-dev or ts-node-dev to run your application instead. These automatically restart the application whenever a file is changed, so you don't have to do this yourself!

Adding a validator

So the application is working, but there are problems with it. Most notably, you don't want to say hey when the person's name is not set.
For example, going to http://localhost:3000/hello will print "Hello, undefined!".

That's where express-validator comes in handy: let's add a **validator** that checks that the `person` query string cannot be empty, with the intuitively named validator `notEmpty`:

JavaScript TypeScript

index.js

```
const express = require('express');
const { query } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty(), (req, res) => {
  res.send(`Hello, ${req.query.person}!`);
});

app.listen(3000);
```

Now, restart your application, and go to http://localhost:3000/hello again. Hmm, it still prints "Hello, undefined!"... why?

Handling validation errors

express-validator validators do not report validation errors to users automatically.
The reason for this is simple: as you add more validators, or for more fields, how do you want to collect the errors? Do you want a list of all errors, only one per field, only one overall...?

So the next obvious step is to change the above code again, this time verifying the validation result with the `validationResult` function:

JavaScript TypeScript

index.js

```js
const express = require('express');
const { query, validationResult } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty(), (req, res) => {
  const result = validationResult(req);
  if (result.isEmpty()) {
    return res.send(`Hello, ${req.query.person}!`);
  }

  res.send({ errors: result.array() });
});

app.listen(3000);
```

Now if you access http://localhost:3000/hello again, what you'll see is the following JSON content:

```
{
  "errors": [
    {
      "location": "query",
      "msg": "Invalid value",
      "path": "person",
      "type": "field"
    }
  ]
}
```

Now, what this is telling us is that

- there's been exactly one error in this request;
- the error is in a field (`type: "field"`);
- this field is called `person`;
- it's located in the query string (`location: "query"`);
- the error message that was given was `Invalid value`.

This is a better scenario, but it can still be improved. Let's continue.

Sanitizing inputs

While the user can no longer send empty person names, it can still inject HTML into your page! This is known as the Cross-Site Scripting vulnerability (XSS).

Let's see how it works. Go to http://localhost:3000/hello?person=<b>John</b>, and you should see "Hello, **John**!".

While this example is fine, an attacker could change the `person` query string to a `<script>` tag which loads its own JavaScript that could be harmful.

In this scenario, one way to mitigate the issue with express-validator is to use a **sanitizer**, most specifically `escape`, which transforms special HTML characters with others that can be represented as text.

JavaScript TypeScript

index.js

```
const express = require('express');
const { query, validationResult } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty().escape(), (req, res) => {
  const result = validationResult(req);
  if (result.isEmpty()) {
    return res.send(`Hello, ${req.query.person}!`);
  }

  res.send({ errors: result.array() });
});

app.listen(3000);
```

Now, if you restart the server and refresh the page, what you'll see is "Hello, <b>John</b>!". Our example page is no longer vulnerable to XSS!

Accessing validated data

This application is pretty simple, but as you start growing it, it might become quite repetitive to type `req.body.fieldName1`, `req.body.fieldName2`, and so on.

To help with this, you can use `matchedData()`, which automatically collects all data that express-validator has validated and/or sanitized:

JavaScript TypeScript

index.js

```js
const express = require('express');
const { query, matchedData, validationResult } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty().escape(), (req, res) => {
  const result = validationResult(req);
  if (result.isEmpty()) {
    const data = matchedData(req);
    return res.send(`Hello, ${data.person}!`);
  }

  res.send({ errors: result.array() });
});

app.listen(3000);
```

What's next?

These steps conclude the basic guide on getting started with express-validator.
You might want to continue reading about the other available features:

- Learn about the validation chain
- Master the field selection
- Deeply customize express-validator

Edit this page