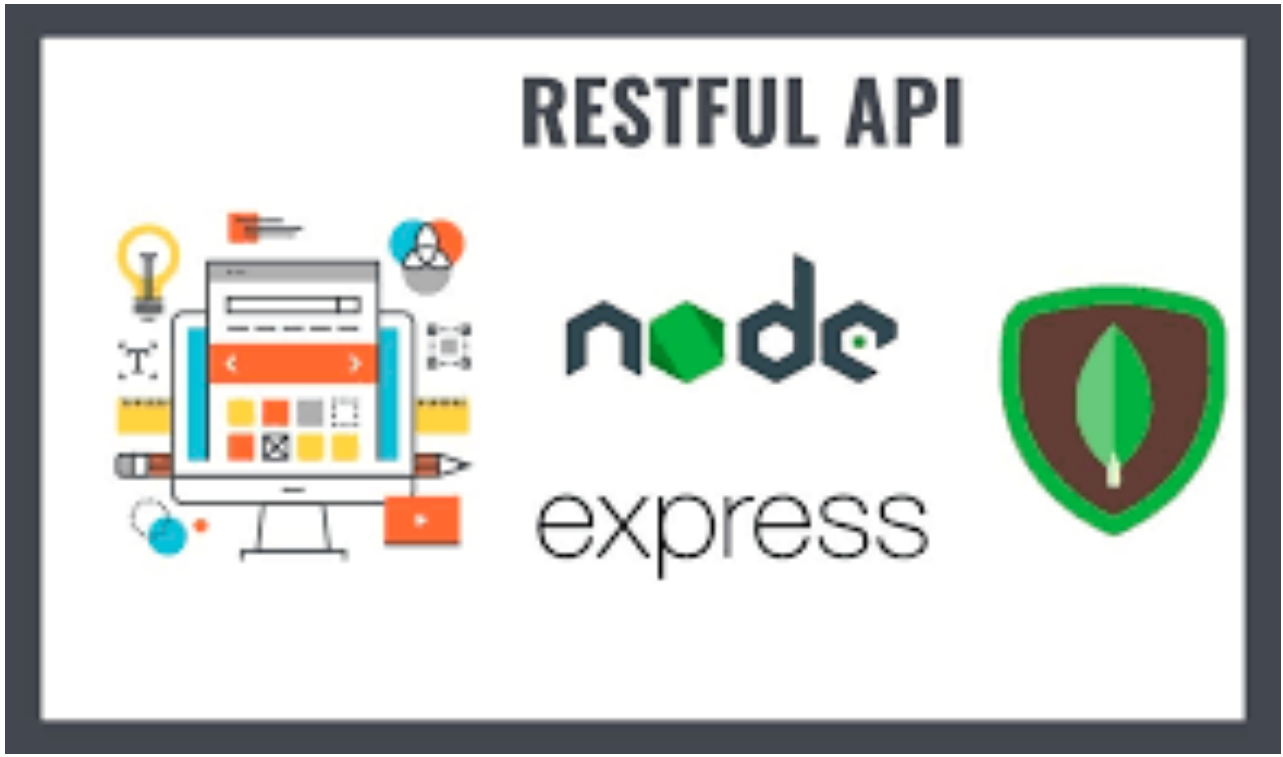
Node.js

Ch 4: RESTful API with Express

hogeschool vives



D. HOSTENS & M. DIMA

— We've already created an api with the [http] module

```
const http = require('http');
const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.write('Hello World');
        res.end();
    }
    if (req.url === '/api/courses') {
        //
    }
});
server.listen(3000);
```

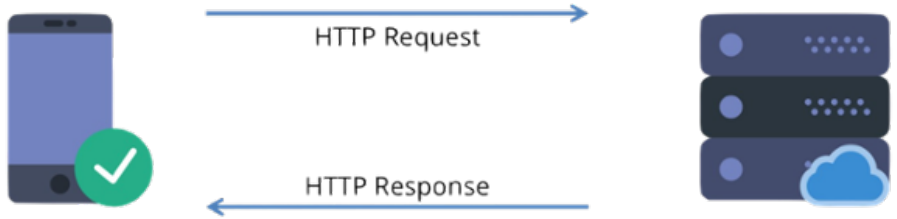
It works but it's not ideal for complex applications with lots of endpoints (too many if statements)

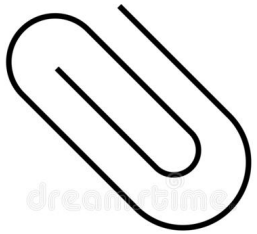
`http` request: `req` en `res` object

`req` object = request (client -> server)

`res` object = response (server -> client)

Client sends a request and receives a response from the server.





ADDITIONAL INFORMATION

`req` object has properties such as `body`, `params`, `query`, ...

Data can be passed trough those objects in both directions

— RESTful services - (RESTful APIs)

Most web and mobile applications are built conforming to a CLIENT - SERVER architecture.

The (for the end user) visible app is the CLIENT and gets its data from the SERVER. Both 'speak' HTTP protocol.

On the SERVER we publish a few 'services'.

The CLIENT sends HTTP requests to those services/endpoints

REST = **RE**presentational **S**tate **T**ransfer = **set of rules** to build HTTP services => CRUD (create, read, update, delete operations)

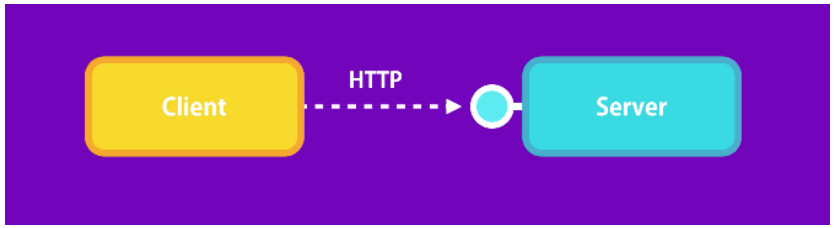


Fig 1 Description: Client- Server Architecture. Left a square labeled as Client. Right a square labeled as Server and with a circle on the left side Representing an endpoint. There is a dotted line with an arrow running from the Client to the endpoint of the Server. The line is labeled HTTP.

— Real case scenario: Vivesbib API (Lab)

We create a Library API to borrow books.

endpoint customers: https://bib.vivesapp.be/api/customers

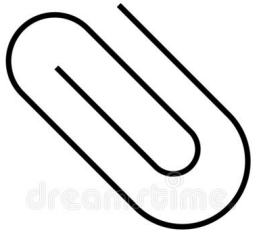
Different Clients can send HTTP requests to that endpoint

Breaking down the link:

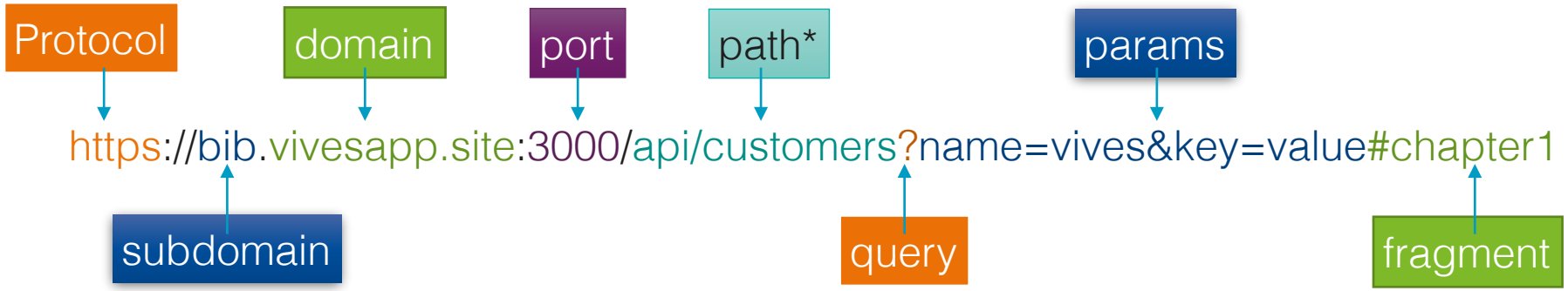
- Starts with **http** or **https** (data sent via a secure channel with HTTPS).
- **bib.vivesapp.be** → Our domain name.
- **/api** → Not mandatory but commonly used. Sometimes also seen as **api.bib.vivesapp.site**…
- **customers** → The resource. The name should be representative.

We send HTTP requests to this endpoint to create, modify, etc., customer data.

— URL structure



ADDITIONAL INFORMATION



*api is not mandatory but recommended mostly mentioning a version too, e.g. /api/v2

— HTTP verbs

The type of HTTP request determines the operation.

Type = **verb** or **method**

Some standard HTTP methods:

GET = retrieve data

POST = create data

PUT = update data updaten (**PATCH** if partial update)

DELETE = remove data

— GET CUSTOMERS

Request = **GET** /api/customers

Response =

```
[
{id:1, name: ' x '},
{id:2, name: ' y '},
...
]
```

1 klant ophalen => Request = GET /api/customers/1

Response = `{id:1, name: ' x '}`

— UPDATE CUSTOMER

Request = **PUT** /api/customers/1 + also provide a customer object (JSON) containing data

```
{ name: 'z' }
```

Response = `{id:1, name: 'z'}`

With partial updates you should use PATCH instead of PUT

— DELETE CUSTOMER

Request = **DELETE** /api/customers/1
customer object does **not** need to be passed along with request

Response = `{id:1, name: 'z'}`

— CREATE CUSTOMER

Request = **POST** /api/customers + also provide a customer object (JSON) containing data

```
{ name: 'x' }
```

Response = `{id:1, name: 'x'}`

server should create the ID

— Onze API (documentation)

The available services:

GET /api/customers

GET /api/customers/1

PUT /api/customers/1

DELETE /api/customers/1

POST /api/customers

We will first build this with Express without a Database. We keep the objects in memory.

— Express

https://www.npmjs.com/package/express

in the terminal:

```
milan@nodevb~ mkdir les4
milan@nodevb~ cd les4
milan@les4~ npm init
...
milan@les4~ code .
```

we create a new file index.js (you can choose the name)

```
const express = require('express');
//returns a function
const app = express(); //we call the function and
// store the returned Express object in a new object 'app'
```

Returns an object of the type Express

`app` => standard practice

— app object

Has a few available methodes:

```
app.get();
app.post();
app.put();
app.delete();
```

Which correspond with the HTTP verbs

We will for the moment use `app.get()`

Has 2 arguments: endpoint and callback function

```
app.get('/', function(req,res){ res.send ('hello')}
```

Request object has lots of useful information also see: https://expressjs.com/en/4x/api.html#req

body, cookies, url, ip etc…

— Endpoint with Express

The first endpoint (the website root)

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
    res.send('Hello World');
});
//app.listen takes a port and a callback function
app.listen (3000, () => console.log('Listening on port 3000...'));
```

in the terminal

```
milan@les4〽 node index.js
Listening on port 3000...
```

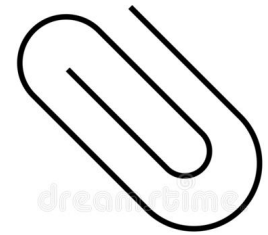
in the browser we surf to http://localhost:3000/

`res.write`: additional information!

Builtin method in the http module to print a response in the browser = `res.write()`

Several writes possible

```
response.write('<html>');
response.write('<body>');
response.write('<h1>Hello, World!</h1>');
response.write('</body>');
response.write('</html>');
response.end();
```



ADDITIONAL INFORMATION

Always close with `res.end()`!

Express provides an own method. No need for closing ~~`res.end()`~~

`res.send() = res.write() + res.end()`

— We create an additional endpoint: /courses

we use an array (no DB)

```
app.get('/api/courses', (req, res) => {
    res.send([1, 2, 3]);
});
```

in the terminal (stop server CTRL-C)

```
milan@les4〽 node index.js
Listening on port 3000...
```

in the browser we surf to http://localhost:3000/api/courses

browser output: [1,2,3]

With Express there is no more need for **if blocks**!

— Nodemon

With each change now we need to restart the server.

Improvement: we install **[nodemon]** (globally!)

```
milan@les4〽 npm i -g nodemon
```

No instead of starting our server with `node` we will use `nodemon`

```
milan@les4〽 nodemon index.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Listening on port 3000...
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Listening on port 3000...
```

— Environment variables

For a production app we usually do **not** hard code a port but check first if $PORT exists

```
app.listen (3000, () => console.log('Listening on port 3000...'));
```

We poll the **env** property of the **process** global object

```
const port = process.env.PORT || 3000;
app.listen (port, () => console.log(`Listening on port $
{port}...`));
```

How to set the environment variable $PORT

Unix (Linux/Mac): `export PORT=5000`

Windows: `set PORT=5000` (cmd) `$Env:PORT = 5000`

```
milan@les4〽 export PORT=5000
milan@les4〽 nodemon index.js
...
Listening on port 5000...
```

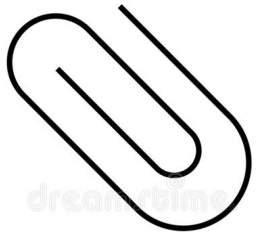
Environment variables: additional information!

Environment variables are bound to a shell process!

A variable created inside a shell session is not available in another shell instance!

Best practice: create the variables inside the VS Code builtin terminal

Persistent env vars are best to be created in the control panel in windows or .bashrc, .zshrc,... in Linux/MacOS



ADDITIONAL INFORMATION

— Endpoint: get one course

Provide id e.g. /api/course/1 (becomes a param :id)

The value of id can be accessed from `req.params.id`

```
app.get('/api/courses/:id', (req, res) => {
    res.send(req.params.id)
})
```

We test it inside the browser => output = 1

You can have several params in one route

e.g.: `/api/posts/:year/:month`

Express also supports query string parameters (optional)

e.g.: http://localhost:5000/api/posts/2021/25/?sortBy=name

Can be accessed with `req.query`

Endpoint: get one course (cont.)

We first store the courses in a global array

```
const courses = [
{id: 1, name: 'course1'},
{id: 2, name: 'course2'},
{id: 3, name: 'course3'},
];
```

We then edit our courses endpoint

```
app.get('/api/courses', (req, res) => {
    res.send(courses);
});
```

Now we need the logic to search Innside an array

The .find() method is available for arrays in JS

```
const course = courses.find(c => c.id === 
parseInt(req.params.id));
```

— Endpoint: get one course (cont. 2)

If the course is not found we need to issue a 404 error = convention with APIs

```
if(!course) return res.status(404).send('the course with the given id was not found');
```

If we do find it we need to send the right course

```
res.send(course);
```

Let's test this in the browser. We surf to:
http://localhost:5000/api/courses/1

Browser output: {"id":1,"name":"course1"}

we surf to: http://localhost:5000/api/courses/10

Browser output: the course with the given id was not found. (Also see respons in headers: Network)

— POST request. Create a course

```
app.post('api/courses', (req, res) => {
    const course = {
        id: courses.length + 1,
        name: req.body.name
    };
});
```

we generate a new id (nr. of courses + 1)

in order to be able to read the `req.body.name` we need to enable json parsing

```
app.use(express.json());
```

we push the new course to the array

```
    courses.push(course);
    res.send(course); //convention for id
```

— Full POST endpoint

at the top of the file together with the other declarations we add:

```
app.use(express.json());
```

and then between the endpoints we add:

```
app.post('/api/courses', (req,res) => {
    const course = {
        id: courses.length + 1,
        name: req.body.name
    };
    courses.push(course);
    res.send(course);
});
```

How can we test this endpoint? => Postman! or REST Client Plugin in VS Code

— Postman

We create a request in Postman

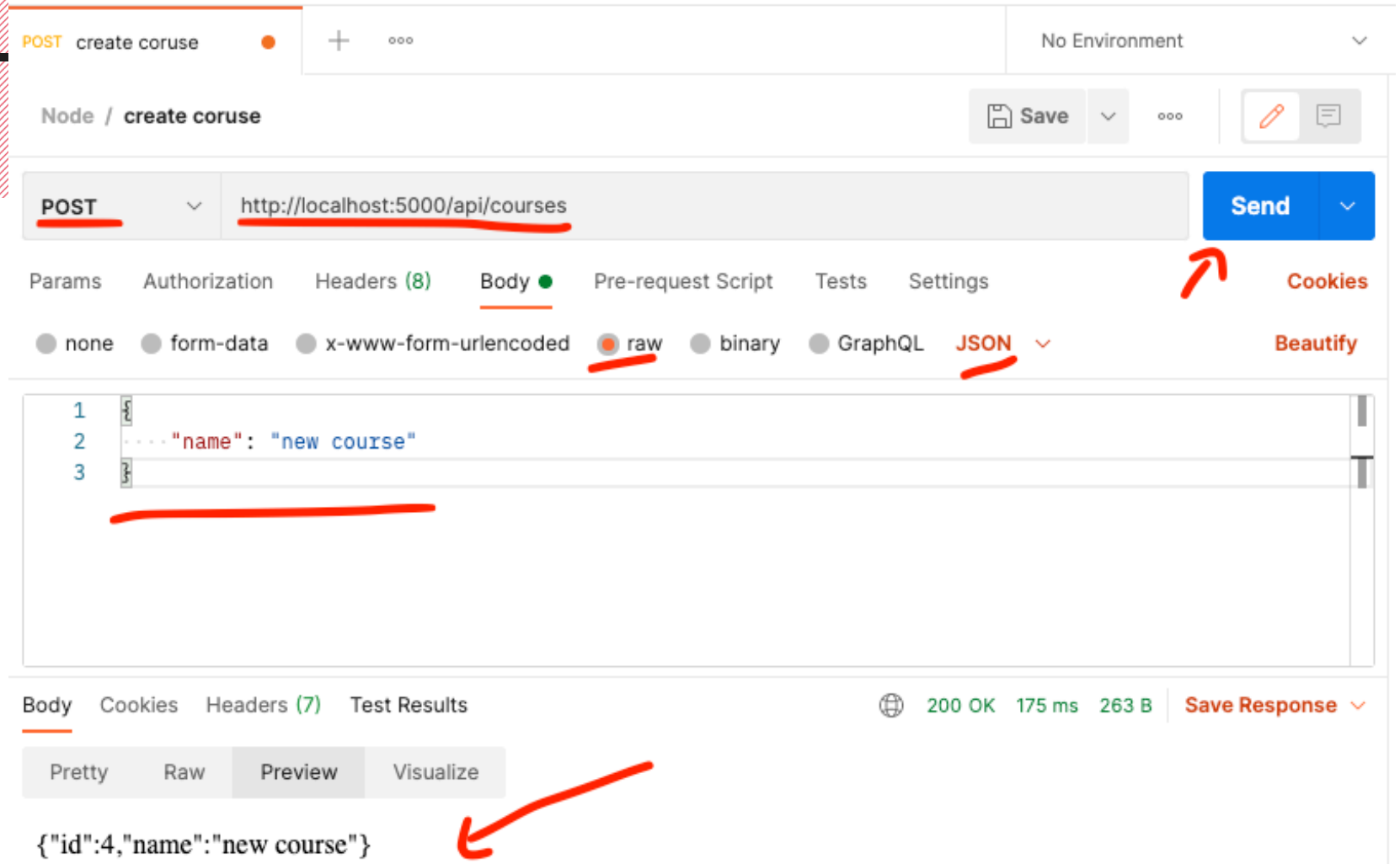
POST *url:* http://localhost:5000/api/courses

Body => RAW => JSON

```
{
    "name": "new course"
}
```

Click send and check the response

```
{
    "id": 4,
    "name": "new course"
}
```



Figuur 2 Beschrijving: Screenshot uit postman mer markeringen bij POST, URL, Body? raw, JSON, Send, inhoud Body en Inhoud response

REST Client {}

rest.http > ...

Send Request

```
POST http://localhost:3000/api/courses
content-type: application/json

{
    "name": "course7"
}

###

Send Request
GET http://localhost:3000/api/courses
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 25
ETag: W/"19-BgTV6IIBHSJIzFuvG/JjswU2PXA"
Date: Sun, 10 Mar 2024 18:29:52 GMT
Connection: close

{
  "id": 7,
  "name": "course7"
}
```

Fig 3 Description: Screenshot VSCode, file rest.http where a new Course object is created

— Input validation

What if the client forgets to send an object with name-property along with the post request?

What is the name property is nog valid?

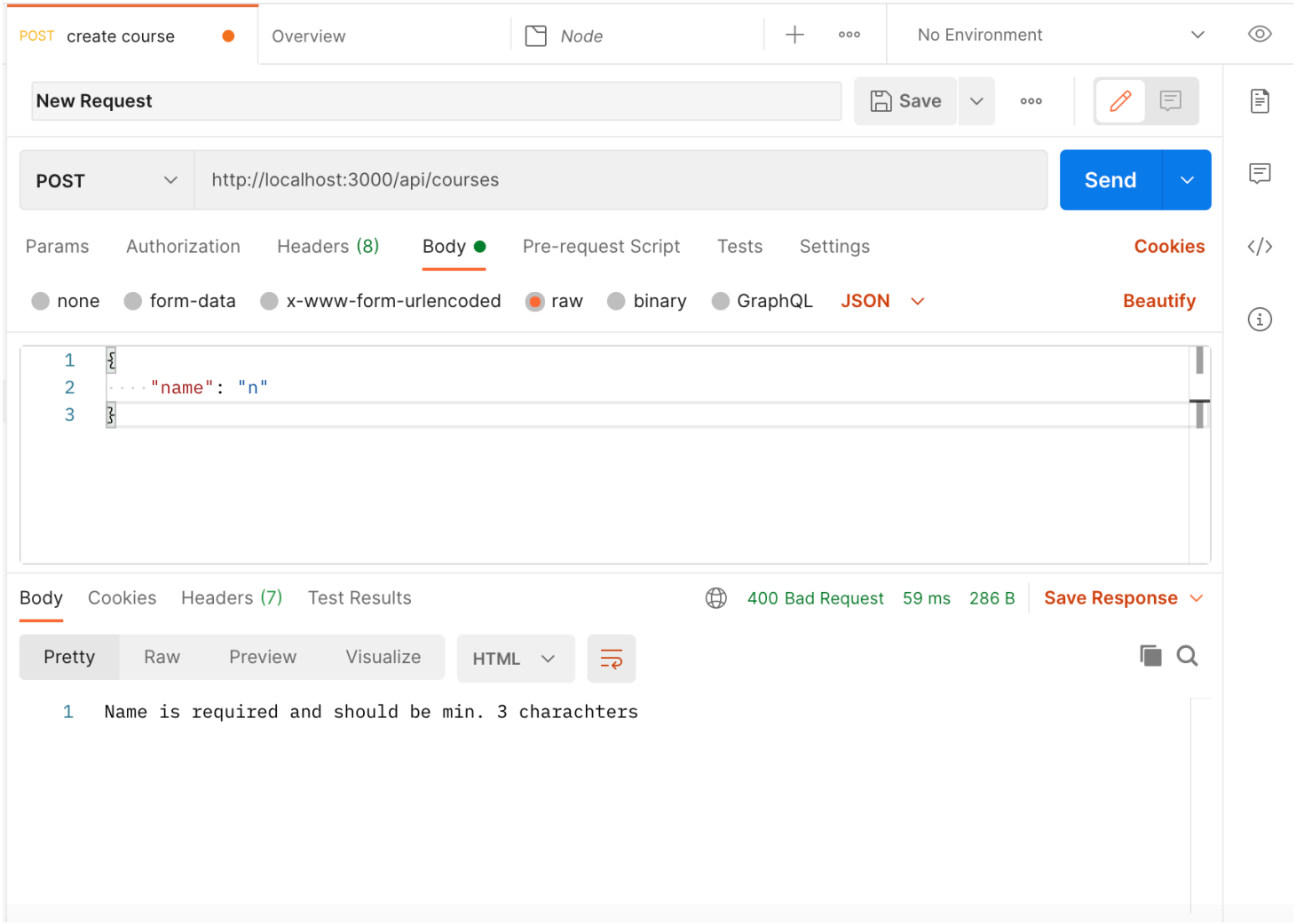
Never trust user input! Always validate!

Validation basic example:

```
    if(!req.body.name || req.body.name.length <3){
        //400 Bad Request
        res.status(400).send('Name is required and should be
min. 3 characters');
        return; //zodat de rest niet uitgevoerd wordt
    }
```

In a real world app objects can get very complex!

Validation works!



— Input validation with complex objects

npm Joi package https://www.npmjs.com/package/joi

we install Joi: `npm i joi`

```
const Joi = require('joi');
```

We no need to create a Schema (object-'shape')

In our app.post route we add the schema schema:

```
    const schema = Joi.object({
        name: Joi.string().min(3).required()
    });
    const result = schema.validate(req.body);
        console.log(result);
```

We try to create a *course* with as name: 'n'

Also see the result-object in the console!

Note: with older versions of Joi the syntax is: Joi.validate(obj,schema) instead of schema.validate(obj)

— Joi-validation

We again try to create an object with 'n' as a name

```
Listening on port 3000...
{
  value: { name: 'n' },
  error: [Error [ValidationError]: "name" length must be at
least 3 characters long] {
    _original: { name: 'n' },
    details: [ [Object] ]
  }
}
```

Without name:

```
...
error: [Error [ValidationError]: "name" is not allowed to be
empty] {
...
```

— Full Validation

```
app.post('/api/courses', (req,res) => {
    const schema = Joi.object({
        name: Joi.string().min(3).required()
    });

    const result = schema.validate(req.body);
    if(result.error){
        res.status(400).send(result.error);
        return;
    }

    const course = {
        id: courses.length + 1,
        name: req.body.name
    };
    courses.push(course);
    res.send(course);
});
```

— Error answer

In Postman:


```
{
  "_original": {
    "name": ""
  },
  "details": [
    {
      "message": "\"name\" is not allowed to be empty",
      "path": [
        "name"
      ],
      "type": "string.empty",
      "context": {
        "label": "name",
        "value": "",
        "key": "name"
      }
    }
  ]
}
```


Too complex to send back to the client!

— Error-message user friendliness

instead of

```
res.status(400).send(result.error);
```

we can do:

```
res.status(400).send(result.error.details[0].message);
```

Postman output now:

400Bad Request
122 ms
270 B
"name" is not allowed to be empty

we a name shorter than 3 chars:

400Bad Request
381 ms
285 B
"name" length must be at least 3 characters long

— PUT - Update *Course* (PATCH)

Steps:

1. Look up *Course*
2. 404 *error* if it doesn't exist

   we already have those two steps see app.get('/api/courses/:id'..)

3. Validate Input
4. 400 *error* if faulty input

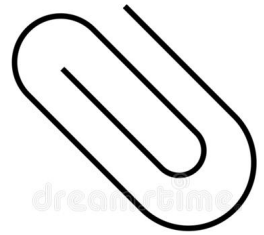
   we already have the validation see app.post('/api/courses'..) but in order to avoid duplicate code we will create a separate validateCourse() function.

5. Update *Course*
6. Return updated *course*

PUT vs PATCH verbs

PUT:
It is used to Update record and each and every details of that particular record will get updated.

PATCH:
It is also used to update record and here important thing is whatever details will provide in payload , only those details will get updated and rest all will remain untouched.



ADDITIONAL INFORMATION

— Full Update course route

```
app.put('/api/courses/:id', (req, res) => {
    const course = courses.find(c => c.id === parseInt(req.params.id));
    if(!course) return res.status(404).send('the course with the given id was
not found');
    const result = validateCourse(req.body);
    if(result.error){
        res.status(400).send(result.error.details[0].message);
        return;
    }
    course.name = req.body.name;
    res.send(course);
});

function validateCourse(course){
    const schema = Joi.object({
        name: Joi.string().min(3).required()
    });
    return schema.validate(course);
}
```

— We test it in postman

Methode: PUT

URL: http://localhost:3000/api/courses/1

Body: raw, JSON

Body inhoud:

```
{
   "name": "new course"
}
```

Send

Response:

200OK
55 ms
263 B

```
{
   "id": 1,
   "name": "new course"
}
```

— DELETE requests

Steps:

1. Look up *Course*
2. 404 *error* if it doesn't exist

   we already have those two steps see app.get('/api/courses/:id'..) and app.put('/api/courses/:id'..)

3. Remove *Course*
4. Return deleted *course*

— Full DELETE route

```
app.delete('/api/courses/:id', (req, res) => {
    const course = courses.find(c => c.id === 
parseInt(req.params.id));
        if(!course) return.      res.status(404).send('the 
course with the given id was not found');

    const index = courses.indexOf(course);
    courses.splice(index, 1);
    res.send(course);
});
```

— We test our DELETE route in Postman

Method: DELETE

URL: http://localhost:3000/api/courses/3

Send

200OK
26 ms
260 B


```
{
   "id": 3,
   "name": "course3"
}
```


URL2: http://localhost:3000/api/courses/10

Send

404Not Found
28 ms
277 B

the course with the given id was not found

— Lab: we start our Vivesbib API

we create an endpoint to return all book genres

https://bib.vivesapp.be/api/genres

see Github Classroom lecture 4 + slide 11

Link: see Toledo