Learning Objectives: Extending the Server

Learners will be able to…

- **process the DELETE method**
- **parse the body of POST, PUT (or PATCH) requests.**
- **distinguish between PUT, POST and PATCH**
- **interpret the request headers**

info

Make Sure You Know

This assignment assumes a basic knowledge of JavaScript, HTML and CSS and the material covered in the previous assignments.

Limitations

We will not explain simple programming concepts, it is assumed that the user is familiar with these concepts.

REST verbs

REST uses several “verbs” that are mapped to the HTTP request methods. These “verbs” are the methods you can use to take the following actions on a resource.

Verb	Action
GET	Fetches either a list of resources or a particular resource. It should not be used to alter the state of the resource.
POST	Mostly used to create a new resource and add it to the list of resources.
PUT	Used to replace existing resources
DELETE	Deletes an existing resource. It can also delete several members of a list of resources by using a filter.
PATCH	Used to partially modify a resource without replacing it entirely
OPTIONS	Tells which of the methods above are supported by the endpoint.
HEAD	Similar to GET but does not return the body of the response, only the metadata: the status and the headers (including the correct `Content-Length`)

DELETE

DELETE is one of the simplest of the methods to handle because the request does not have a body.

For example, to delete an order from the store database:

There are several possible options for the return status:
* `204 No Content` means that the deletion was successful and there is no additional returned status.
* `200 OK` should be sent when there is some additional data returned to the client.
* `202 Accepted` may be used when the deletion is waiting to be processed.

Sometimes API developers use `404 Not Found` when the resource was not found due to already having been deleted. The client may choose to treat the `404` status as a successful deletion since `DELETE` is idempotent.

▼ What does **idempotent** mean?

In the context of a REST API **idempotent** means that a call can be made multiple times and the outcome will be the same as the first time. Subsequent calls will **not** modify the resource. The POST and PATCH requests are not idempotent. GET, DELETE, HEAD, OPTIONS, PUT and TRACE are idempotent.

info

Installing the node modules

The first time you run something in each of the assignments in this course you need to make sure that you have all the proper node modules installed. To do that run the commands below in the terminal.

```
cd examples
npm i
```

Starting the App

To start the app enter the following in the terminal, you must be in the examples directory. To stop the app use Ctrl-C in the terminal.

```
nodejs 01-delete.js
```

Try it out

- First list out all the pets `curl -s http://localhost:3000/pets`
- Delete the pet with identifier 5 `curl -X DELETE http://localhost:3000/pets/5`
- Use the button above to see that pet 5 is now missing `curl -s http://localhost:3000/pets`

Request body

middleware

express.js uses `middleware` functions to pre-process the request and/or the response. The name **middleware** refers to the fact that the software usually sits between the request made by the client and the resource that sits on the server.

Example middleware tasks include compressing HTTP responses, error handling and serving static files. View the supported Express middleware.

Parsing request bodies

`POST`, `PUT` and `PATCH` requests typically contain a request body and **express.js** has built-in middleware to support requests that contain text, JSON or form data.

The following code snippet shows how to enable support of JSON request bodies for the whole Express app:
`tex -hide-clipboard app.use(express.json());`

The `express.json()` function will parse all the request bodies that have the `Content-Type: application/json` header as an object. Without using this middleware the `request.body` will be `undefined`.

```tex -hide-clipboard
app.use(express.json());

app.post('/orders', (request, response) => {
console.log(request.body);

response.sendStatus(201);
});
```

~~~
### Content Types
The content type, also referred to as [MIME type]
(https://developer.mozilla.org/en-
US/docs/Web/HTTP/Basics_of_HTTP/MIME_types), or media type, is a
string that indicates the type of the file. The string usually
consists of two parts (type/subtype) separated by a slash with
no spaces. The type is the general category of the data, for
example, image or text. The subtype is the specific type of
data, in the case of image, some examples of subtype are jpeg
`image/jpeg` or png `image/png`.


Sometimes the body contains JSON but the `Content-Type` is not
`application/json`. This can be the case when the JSON has
particular standardized properties. For example, GeoJSON:
`application/geo+json`. By default `express.json()` middleware
won't parse requests with this type. To fix this you can specify
the Content-Type in the parameter:

```tex -hide-clipboard
app.use(express.json({
  type: 'application/geo+json'
}));
~~~

You can use a wildcard with the type name for example, `type: application/*+json`. To support both the default and a subformat of JSON you can pass an array of types:
`tex -hide-clipboard app.use(express.json({   type: [     'application/json',     'application/*+json'   ] }));`

Another frequently used format is `application/x-www-form-urlencoded` which is sent with HTML forms.
`tex -hide-clipboard app.use(express.urlencoded());`

The previously mentioned data types can all be parsed using the default Express configuration. If you want to parse data with an XML format you can load a package called body-parser-xml.

Difference between POST, PUT and PATCH

POST is usually used to create new instances of a resource. The request body can contain the partial sufficient representation of the resource. Upon successful creation, the returned status is `201 Created`.

In REST it is recommended to put the URI of the newly created resource in the `Location:` header of the response.
Usually the server creates an identifier (ID) of the new resource and returns it to the client.

In cases when the action does not create a new resource either `200 OK` or `204 No Content` are appropriate response statuses.

Making the same request twice should try to create two identical resources.

PUT
usec
exist
reso
requ
shou
cont
repr
of th
reso
Usua
shou
who

Mak
sam
twic
have
resu

If th
was
befo
a PU
ther
optio
depe
the c

the a
1. Re
erro
404
Foun
1. Cr
reso
with
succ
statu

PATCH is usually used for the partial modification of a resource.

For example, in case of the JSON representation its body may contain only those properties that need to be updated. These properties should be merged by the server with the stored object.

As with **PUT** it should be idempotent and making the same request twice should have the same result without creating any additional resources. There are circumstances where a **PATCH** command is not idempotent.

info

Starting the App

To start the app enter the following in the terminal. To stop the app use Ctrl-C in the terminal.

```
cd examples
nodejs 02-post-put-patch.js
```

Try it out

- First list out all the pets `curl -s http://localhost:3000/pets`
- Add a rabbit `curl -X POST -H "Content-Type: application/json" -d '{"kind":"Rabbit","name":"Bugs"}' http://localhost:3000/pets`
- Click the **List Pets** button above to see the change.
- Switch Turtle to Iguana `curl -X PUT -H "Content-Type: application/json" -d '{"kind":"Iguana","name":"Iggy"}' http://localhost:3000/pets/4`
- Click the **List Pets** button above to see the change.

Headers

HTTP headers are used to pass additional information along with a request or a response. An HTTP header consists of a case-insensitive name followed by a colon followed by the header value. The **request header** can be used to provide context for the request. For example, the preferred format of the response. The **response header** also provides context, for the response. For example the `Age` header will tell how long a response has been sitting in a proxy cache.

Access the values of the request headers

```
app.get('/', (req, res) => {
  console.log(req.get('x-custom-header'));
  // ...
});
```

This is case-insensitive and works the same for `X-Custom-Header:` or `x-custom-header:`.

Set the value of the response header

```
app.get('/', (req, res) => {
  res.set({
    'x-custom-header': 'my header value'
  });
  // ... send the response
});
```

There are convenience functions that let you set the type or location information in the header. The following will set the header to `Content-Type: image/png`.

```
res.type('png')
```

This will set `Location: /orders/534`:

```
res.location('/orders/534')
```

OPTIONS method and Cross-Origin Resource Sharing (CORS)

The OPTIONS method is used to determine which HTTP methods are supported on an endpoint. There is another important role for the OPTIONS method, it is related the browser security. Sometimes the client and the server are loaded from the different hosts. E.g. `client.example.com` and `server.example.com`. By default the browser won't make a request from such a client.

To get around this, before making a request it makes a preflight request with an OPTIONS method. If the server sends `Access-Control-Allow-Origin: https://client.example.com` in the headers, the server tells the browser that it trusts the host name of the client and the browser will allow it to make the subsequent request.

There is a flexible library available to use with express.js to configure Cross-Origin Resource Sharing. You can install it with the following command:
`npm i cors`

This is an example of how you might use it:

```
import express from 'express';
import cors from 'cors';

const app = express();

app.use(cors({
  origin: 'client.example.com'
}));
```