REST by Example

From Resources to HATEOAS

CHRISTIAN CLAUSEN

Contents

Chapter 1

Introduction

Many tutorials (perhaps all) on REST start by defining what it is, and while I usually support this convention, this time I feel differently. To define REST one needs a lot of fancy words (eg. stateless, representational, protocol) and even more meaningless abbriviations (eg. HATEOAS, HTTP, URI, HAL, indeed REST). I think it is easier to understand the principles and rules of REST through exploration and concrete examples. Therefore I postpone the definition of REST to the end of the document, by this time hopefully most of the words and abbriviations can be put into context. I will provide the expansion when I introduce an abbriviation, but I think it is more useful to think of the abbriviation as a name, so you shouldn't put too much effort into learning the expansion.

1.1 Richardsons Maturity Model

This presentation of REST is based on *Richardsons Maturity Model*, as described by Fowler [2]. This model describes 3 major stages involved in migrating from a non-REST service (level 0), to a glorious, pure REST service (level 3). At each stage our *API* (Application Programming Interface) change in very specific ways. Therefore we start with a presentation of a tiny non-REST service. We then proceed with the three stages in-turn, one section for each. Finally as promissed a conclusion containing the definition of REST.

Chapter 2

Level 0; Case and starting point

We have a small service for keeping track of our todo-list. The service can be accessed through a POST-request (to the base URL of the service). The body of the POST-request contains a field '`Function`' which determine the action we want to perform. The entire API is summerized in figure 2.1.

As can be seen from the table our API supports adding and deleting todo-notes, getting an overview of all todo-notes, and viewing details about a specific todo-note.

2.1 The Code

We can build this service in Typescript using nodejs and express. We want to treat the in from our POST-request as regular JSON objects, we therefore also need body-parser.

```
2 import * as Express from 'express';
3 let app = Express();
4
5 import * as BodyParser from 'body-parser';
6 app.use(BodyParser.json());
7 app.use(BodyParser.urlencoded({ extended: true }));
```

Listing 2.1: `level0/index.ts`

`Function`	Arguments	Description
add	title, description	Adds a new todo-note with specified content.
list	-	Returns the list of all todo-notes.
show	id	Returns the todo-note with the specified id.
delete	id	Deletes the todo-note with the specified id.

Figure 2.1: Todo-list api, level 0

2.2 The Data

For simplicity we choose not to use any sort of database, or advanced datastructure to store the todo-notes. Instead we use a simple array. This means, of course, that all our data is in-memory, and will be reset each time we reset the server. We initialize our array with two dummy todo-notes, to have some test data. Now we are ready to start responding to some POST-requests.

```
18 let notes = [{
19   title: "Laundry",
20   description: "Do the laundry",
21 }, {
22   title: "Read homework",
23   description: "'REST by example'",
24 }];
25
26 app.post("/", (req, resp) => {
27   let result;
28   let func = req.body.function;
```

Listing 2.2: `level0/index.ts`

2.3 Add

Keeping with the API the first request we should be able to handle is `add`. This is straight forward; we fetch the arguments from the request, and push a new todo-note onto our list. We then report back to the caller that the todo-note was added.

```
31   if (func === "add") {
32     let title = req.body.title;
33     let description = req.body.description;
34     notes.push({ title, description });
35     result = "Added";
36   }
```

Listing 2.3: `level0/index.ts`

2.4 List and Show

The next function we should support is `list`, and since we already store our todo-notes in a list, this is completely trivial. Similarly for selecting a single element with `show`.

```
39   else if (func === "list") {
40     result = notes;
```

```
41   }
```

Listing 2.4: `level0/index.ts`

```
44   else if (func === "show") {
45     let id = req.body.id;
46     result = notes[id];
47   }
```

Listing 2.5: `level0/index.ts`

2.5 Delete

Finally we have `delete`. For this we use Javascripts `splice` method to remove the element from the list.

```
50   else if (func === "delete") {
51     let id = req.body.id;
52     notes.splice(id, 1);
53     result = "Deleted";
54   }
```

Listing 2.6: `level0/index.ts`

Finally we just need to return the result to the caller, and start our server listening for requests.

```
57   resp.json(result);
58 });
59
60 console.log("Server listening on port 3000");
61 app.listen(3000);
```

Listing 2.7: `level0/index.ts`

2.6 Client

With our server up and running we can make a tiny client for testing it. The client only has to be able to send POST-requests with the four arguments from our API.

```
10   <form action="http://localhost:3000/" method="POST">
11     Function:    <input type="text" name="function" /><br/>
12     Title:       <input type="text" name="title" /><br/>
13     Description: <input type="text" name="description" /><br/>
14     Id:          <input type="text" name="id" /><br/>
15     <input type="submit" value="Go!" />
16   </form>
```

Listing 2.8: `level0/client.html`

2.7 Conclusion

We now have a starting point for our exploration of REST. The service that we have just build exhibit basically none of the characteristics of a REST service. Therefore it makes for a desent starting point for our adventure. Our API does have some things going for it, however it also have quite a few caveats, so let's have a look at some of the most significant.

Pros

First the API works! It is correct, it performs exactly what it should. This is always the most important thing. Second our API is simple! Therefore it was really fast to implement. It is also well documented, through table 2.1.

Cons

On the other hand, it is not very flexible, it is one big service. With just one endpoint we have to be able to submit all of our arguments in all combinations, although they are only used in specific combinations. This creates quite a tight coupeling between developer-API and code-API.

Our API also uses custom (hard-coded) messages, such as "Added". This again hurts our flexibility, what happens next time we want to introduce a message? Do we just keep inventing new ones? In a realisticly sized project we would end up with hundreds of non-standatized messages. Depending on the number of developers keeping roughly the same format is also challenging.

As mentioned we are tightly coupled with the API thus maintaining it is crucial. This is time consuming, and over time this API will quickly grow so large that noone will retain an overview. This in turn means that if we add a new feature it will most likely go overlooked.

2.8 Exercises

Chapter 3

Level 1; Resources

A central idea to REST is that everything is a *resource*, and as such has its own identity. Essentially what this means is that everything should have its own address, or endpoint. So instead of just sending all our POST-requests to the same place as in level 0 we now split everything up such that everything has its own unique endpoint.

3.1 Add and List

Add is pretty straight forward to factor out, we create a new route (`/notes/add`), and paste the code from inside the `if`. We can now remove the function parameter from this request as only `add` requests will come to this endpoint.

```
24 app.post("/notes/add", (req, resp) => {
25   let title = req.body.title;
26   let description = req.body.description;
27   notes.push({ title, description });
28   let result = "Added";
29   resp.json(result);
30 });
```

Listing 3.1: `level1/index.ts`

Lists are completely similar, create a new route (`/notes`), and paste the code.

```
34 app.post("/notes", (req, resp) => {
35   let result = notes;
36   resp.json(result);
37 });
```

Listing 3.2: `level1/index.ts`

3.2 Show and Delete

Now however we get to something different. Consider "Everything should be a resource", this raises a subtle question: what is a thing? The subtle point here is that in fact each todo-note is also a resource, which means that they should have their own endpoint. We can achieve this by using a *template* in the route (eg. `:id`). A template means that whatever is in that position will be put in a variable accible through `req.params`. This way to access the first note we send a POST-request to `/notes/0`, the second note would be at `/notes/1`, and so on.

After this change – and pasting in the code – we now have two way to access the id, the route, and the parameter. This is redundant and so we can remove the parameter. Resulting in the following code.

```
41 app.post("/notes/:id", (req, resp) => {
42   let id = req.params.id;
43   let result = notes[id];
44   resp.json(result);
45 });
```

Listing 3.3: `level1/index.ts`

Delete goes through a similar process. Notice that we can add routes after templates, as they only scope until the next `/`.

```
49 app.post("/notes/:id/delete", (req, resp) => {
50   let id = req.params.id;
51   notes.splice(id, 1);
52   let result = "Deleted";
53   resp.json(result);
54 });
```

Listing 3.4: `level1/index.ts`

3.3 Client and Updated API

Before we can design our new client let's just look at how the new API looks after our changes. It is no small change, indeed we need to completely restructure how to present the API. Where we before had a '`Function`' column we now also have an address column. The updated API apear in figure 3.1.

We are now in position to create our client, because we have 4 endpoints we also need four different forms. However, we actually have a dynamic number of endpoints corresponding to how many todo-notes we have. To remidy this we have to introduce an extra text field, and use a little Javascript to set the action dynamically.

```
10   <form action="http://localhost:3000/notes/add" method="POST">
11     Title:       <input type="text" name="title" /><br/>
```

Function	Address	Arguments	Description
add	`/notes/add`	title, description	...
list	`/notes`	-	...
show	`/notes/X`	-	...
delete	`/notes/X/delete`	-	...

Figure 3.1: Todo-list api, level 1

```
12      Description: <input type="text" name="description" /><br/>
13      <input type="submit" value="Add!" />
14    </form>
15    <form action="http://localhost:3000/notes" method="POST">
16      <input type="submit" value="List!" />
17    </form>
18    Id: <input type="text" id="id"/><br/>
19    <form action="http://localhost:3000/notes/"
20          method="POST"
21          onsubmit="this.action += document.getElementById('id').value">
22      <input type="submit" value="Show!" />
23    </form>
24    <form action="http://localhost:3000/notes/"
25          method="POST"
26          onsubmit="this.action += document.getElementById('id').value + '/delete'">
27      <input type="submit" value="Delete!" />
28    </form>
```

Listing 3.5: `level1/client.html`

3.4 Conclusion

At this stage we have introduced a concept of 'resources'. One of the effects of this is that we now have split our one endpoint into severel. In fact, the number of endpoints are now affected by the data in our system. To support this we have used templates in the routes, otherwise the effect on the code was mostly simple refactorings.

Pros

Again, most importantly, the API works! The API is simple. We implemented it with only very little extra effort compared to level 0. It is also well documented. It is more flexible than before, as we could now move different parts of the code around, changing the endpoint for one resource for example only requires changing the code that accesses this resource, and everything else stays untouched. Our client is more intuitive and easy to use, as it has no unnessasary fields.

Cons

We need Javascript to get the proper function of our client. We have the same issue that our API will get cluttered, and new functionality may go overlooked.

3.5 Exercises

Chapter 4

Level 2; HTTP (and CORS)

In the next step we are actually going to start utilizing that we are using *HTTP* (HyperText Transfer Protocol). For starters HTTP supports multiple messages other than POST. In fact, POST is meant to function as create. In figure 4.1 we present a table of the *HTTP verbs* and what their intended meaning is. As we can see we already have many of these intentions in our API, but use only one message type.

Another aspect of HTTP is that the response messages are also fairly standardized, in form of *status codes*. The most well-known being '404 Not found'. A comprehensive list of status codes is out of the scope of this document, but can be found in [1]. For us suffice it to say that numbers are categorized, these are summerized in figure 4.2. Status codes are especially useful when we don't return anything.

Message	Intention
POST	Create
GET	Read
PUT	Update
DELETE	Delete
OPTIONS	Capabilities
...	...

Figure 4.1: HTTP messages

Status code	Intention	Example
1xx	Informational	102 Processing
2xx	Success	201 Created
3xx	Redirection	301 Moved Permanently
4xx	Client Error	403 Forbidden
5xx	Server Error	501 Not Implemented

Figure 4.2: HTTP status categories

To work easily with these response codes we add another dependency to our project: http-status-codes.

```
5 import * as HttpStatus from 'http-status-codes';
```

Listing 4.1: `level2/index.ts`

To summerize, at this level we need to incorreporate two seperate things. First we need to use the HTTP verbs, and second we need to return appropriate HTTP status codes.

4.1 CORS

TODO

4.2 Add

The intention for add is to create a note in the system, thus the method we should use is POST, which we already was using. Regarding the return status code HTTP has one called "created", which is perfect for this case. The resulting code has only added the status:

```
28 app.post("/notes/add", (req, resp) => {
29   let title = req.body.title;
30   let description = req.body.description;
31   notes.push({ title, description });
32   resp.status(HttpStatus.CREATED).send();
33 });
```

Listing 4.2: `level2/index.ts`

4.3 List and Show

As both list and show simply retrives data, the appropriate verb for this is GET. For the HTTP status it makes sence to use the common "OK". "200 OK" is the default status, thus if we annotate nothing this is the returned code. Therefore in the resulting code only the verb was changed:

```
37 app.get("/notes", (req, resp) => {
38   let result = notes;
39   resp.json(result);
40 });
```

Listing 4.3: `level2/index.ts`

```
44 app.get("/notes/:id", (req, resp) => {
45   let id = req.params.id;
46   let result = notes[id];
```

```
47   resp.json(result);
48 });
```

Listing 4.4: `level2/index.ts`

4.4 Delete

Finally, delete is different as we need to change the verb to DELETE and because it doesn't return anything we also need a status code. Investigating the status codes we realize that there is no "deleted" analogous to the "created" code we used earlier. The common solution is to use "204 No content". This tells the reciever that the request was succesful, yet there is no data to be returned.

```
52 app.delete("/notes/:id/delete", (req, resp) => {
53   let id = req.params.id;
54   notes.splice(id, 1);
55   resp.status(HttpStatus.NO_CONTENT).send();
56 });
```

Listing 4.5: `level2/index.ts`

4.5 Client

At this point our client would need to support the HTTP verbs we use. Unfortunately by default HTML only supports POST (forms) and GET requests (forms or links).Therefore if we were to make a web client we need to implement it using AJAX. This is out of the scope of this note.

However, the client would look exactly like the one from last section, except that it would use delete request.

4.6 Conclusion

Pros

Cons

4.7 Exercises

Chapter 5

Level 3; Hypermedia

Kelly [3]

```
5  import { Resource } from 'hal';
```

Listing 5.1: level3/index.ts

```
1  ,,
21   resp.header("Content-Type", "application/hal+json");
```

Listing 5.2: level3/index.ts

5.1 Entry

```
1  ,,
31 app.get("/", (req, resp) => {
32   let res = new Resource({}, "/");
33   res.link("getNotes", "/getNotes?count=5&offset=0");
34   res.link("addNote", "/notes/add");
35   resp.json(res);
36 });
```

Listing 5.3: level3/index.ts

5.2 Add

```
1  ,,
40 app.post("/notes/add", (req, resp) => {
41   let title = req.body.title;
42   let description = req.body.description;
43   notes.push({ title, description });
44   resp.status(HttpStatus.CREATED).send();
45 });
```

Listing 5.4: level3/index.ts

5.3 List

```
1  ,,
49 app.get("/getNotes", (req, resp) => {
50   let res = new Resource({}, "/getNotes");
51   for (let i = 0; i < notes.length; i++) {
52     let title = notes[i].title;
53     let note = new Resource({ title }, `/notes/${i}`);
54     res.embed("notes", note);
55   };
56   resp.json(res);
57 });
```

Listing 5.5: `level3/index.ts`

5.4 Show

```
1  ,,
61 app.get("/notes/:id", (req, resp) => {
62   let id = req.params.id;
63   let res = new Resource(notes[id], `/notes/${id}`);
64   res.link("delete", `/notes/${id}/delete`);
65   resp.json(res);
66 });
```

Listing 5.6: `level3/index.ts`

5.5 Delete

```
1  ,,
76 app.get("/notes/:id/delete", (req, resp) => {
77   let id = req.params.id;
78   notes.splice(id, 1);
79   resp.status(HttpStatus.NO_CONTENT).send();
80 });
```

Listing 5.7: `level3/index.ts`

Client

HAL client.

5.6 Conclusion

Pros

Cons

5.7 Exercises

Chapter 6

Level 3: Documentation

Chapter 7

Definition of REST

Bibliography

[1] Http status codes. https://httpstatuses.com/.

[2] Martin Fowler. Richardson maturity model. https://martinfowler.com/articles/richardsonMaturityModel.html, 2010.

[3] Mike Kelly. Hal - hypertext application language. http://stateless.co/hal_specification.html, 2013.

Index