REST Workshop

Erlend Hamnaberg Trygve Laugstøl

October 2012 Edition

Chapter 1

Exercises

Before starting on an exercise please read through the entire text to get a better overview of the exercise. Also check out the cheat sheets for Node.js and curl, they contain valuable hints and tips.

Each exercise has two directories: `start/` is where you as a participant should start to solve each exercise. `solution/` contains a complete solution for the exercise. There is also a README file that describe the exercise in detail.

1.1 Background

You are working for Ads'r'us; a company that sell classified advertisement online. As the company is thriving and growing your job is to implement the Ads'r'us API.

1.2 Cheat sheets

- Node.js
- curl

1.3 Participant Requirements

Each participant needs to have these tools installed:

- node.js and npm. All the exercises use node.js.

- MongoDB
- git to clone this repository
- curl, a command line HTTP client. Useful for scripting and debugging http servers

1.3.1 Node.js and NPM

Linux

Linux users can install binaries through their distribution's package repository.

OS X and Windows

For OS X and Windows we recommend using the binaries that are available for download from nodejs.org.

1.3.2 MongoDB

Linux

Linux users can install binaries through their distribution's package repository.

OSX

For OS X, MongoDB is available through MacPorts, Homebrew and as binary downloads from mongodb.org. Choose the binary download unless you're already familiar with MacPorts or Homebrew.

The instructions describes how to get started in any case.

Windows

For Windows use the binaries available from http://mongodb.org and follow the instructions. You do not need to install it as a Windows Service.

1.3.3 Git

Linux

Linux users can install binaries through their distribution's package repository.

Windows

There are generic git binaries available from git-scm.com but we recommend using GitHub for Windows as it includes a GUI in addition to the normal command line tools.

GitHub for Windows also includes curl so you don't have to install curl.

OS X

There are generic git binaries available from git-scm.com but we recommend using GitHub for Mac as it includes a GUI in addition to the normal command line tools.

1.3.4 curl

Linux

Linux users can install binaries through their distribution's package repository.

OS X

Curl is shipped with OS X.

Windows

For Windows you can download curl with Cygwin if you're already using Cygwin. If not, download the binaries from curl.haxx.se. Search for "win32" or "win64" depending on your platform.

Chapter 2

Exercise: HTTP RPC

2.1 Objective

Create a classic "RPC over HTTP" application for creating ads with pictures.

The application will encode the result of each operation in the body of the request.

For now, you only need to implement the server side. Use curl for testing the responses from the server.

2.2 Steps

2.2.1 Create the ad endpoint

Endpoint URL: `http://localhost:3000/create-ad`

This is where clients should POST a json object to create a new ad. The JSON that's sent to the server should look like this:

```
1 {
2   "title": "Nice house for sale!",
3   "body": "Four rooms, huge bath."
4 }
```

To test the endpoint you can use the example `ad.json` file like this:

```
1 curl --data-binary @../ad.json
    ↪http://localhost:3000/create-ad
```

The server should return a document like this:

```
{
  "result": "ok",
  "data": {
    "id": "4fc494d10b6a9fcb09000001",
    "body": "Four rooms, huge bath.",
    "pictures": [],
    "title": "Nice house for sale!"
  },
}
```

The `id` field inside the `data` object is used later on when generating URLs.

Similar for any unknown error, `result` should be `error`:

```
{
  "result": "error"
}
```

2.2.2 View ad endpoint

Endpoint URL: `http://localhost:3000/ad?id=...`

Clients can GET this with an "id" query parameter. The server will respond with something like this:

```
{
  "result": "ok",
  "data": {
    "title": "Nice house for sale!",
    "body": "Four rooms, huge bath."
  }
}
```

If the ad does not exist, `result` should be `notFound`:

```
{
  "result": "notFound"
}
```

2.2.3 Add picture endpoint

Endpoint URL: `http://localhost:3000/add-picture?id=...`

There are some example pictures available under pictures/.

2.3 Hints

2.3.1 Starting mongodb

```
cd /path/to/exercises/ && ./mongo.sh
```

2.3.2 Converting JSON string to object

```
var object = JSON.parse(data);
```

2.3.3 Converting javascript object to string

```
var str = JSON.stringify({hello: "world"});
```

2.3.4 Curl

POST image file

```
curl -X POST -T <path-to-file> <url>
```

2.3.5 MongoDB / Mongoose

Insert a new object

```
var ad = new Db.Ad();
ad.title = payload.title;
ad.body = payload.body;
ad.save();
```

When an object has been saved, the id of the object is available as the `_id` attribute.

Find a object

```
Db.Ad.findOne({_id: <my id>}, function(err, doc) {
    ....
});
```

`err` will be set if there was an error while talking to the DB. `doc` will null if not found, or the object if found.

Update an existing object:

```
var cmd = {$push: {arr: item}}
Db.Ad.update({_id: <my id>}, cmd, {}, function(err,
    ↪numAffected) {
  ....
});
```

2.4 Retrospective

2.4.1 Why is this bad?

We are tunneling application semantics through an application level protocol. We should rather delegate to HTTP to do the error handling, and delivery of correct status code. We are by every definition not a good Web citizen.

When implementing a real application, you would delegate routing to a framework, for instance Express, or whatever you use in your programming language.

2.4.2 Hard-coding of URIs

All URIs are hard-coded in the client, we have a lot of coupling between our service, data and client.

If you only have one client and server, this might be a diserable property, but once you have more than one client, this may be come unmanageable. You have to upgrade all clients at the same time as you upgrade your clients.

2.5 Example `ad.json`

```
1 {
2   "title": "Nice house for sale!",
3   "body": "Four rooms, huge bath."
4 }
```

Chapter 3

Exercise: Basic HTTP

3.1 Objective

The background and requirements are similar to the "RPC over HTTP" application, but now it's time to become a better and more "correct" HTTP component.

We will improve the application by moving generic application semantics one level down to the HTTP layer. We'll do this with proper use of status codes and headers.

We're expanding on the previous server, but this time we're also adding a client.

3.1.1 Common Requirements for Resources

The application shall follow these general requirements:

- All data/non-binary content types should be 'application/json'. This applies both for both requests and responses.

Status codes:

- 200: Everything is ok!
- 404: Unknown resource.
- 406: The client posted an unprocessable type.
- 500: Unknown server-side fault.

Methods:

- GET: Used to fetch data. Does not change the object
- POST: Used to create new objects.

3.1.2 Content Types

Use `application/json` for all normal messages, and `text/plain` for error messages.

3.2 Steps

3.2.1 Create the 'create ad endpoint'

Endpoint URL: `http://localhost:3000/create-ad`

This is where clients should POST a json object to create a new ad. The JSON that's sent to the server should look like this:

```
1 {
2   "title": "Nice house for sale!",
3   "body": "Four rooms, huge bath."
4 }
```

The server should return `200 Ok` and a document like this:

```
1 {
2   "type": "ad",
3   "id": "4fc494d10b6a9fcb09000001",
4   "body": "Four rooms, huge bath.",
5   "pictures": [],
6   "title": "Nice house for sale!"
7 }
```

Try to send an invalid object (for example an empty document like `/dev/null` on unix or `nul` on windows) and make sure you're getting a proper 500 error code back.

3.2.2 View ad endpoint

Endpoint URL: `http://localhost:3000/ad?id=...`

Clients can GET this with an "id" query parameter. The server will respond with something like this:

```
{
  "type": "ad",
  "id": "4fc494d10b6a9fcb09000001",
  "body": "Four rooms, huge bath.",
  "pictures": [],
  "title": "Nice house for sale!"
}
```

Try to fetch an ad that doesn't exist and make sure you're getting a 404 error.

3.2.3 Add picture endpoint

Endpoint URL: `http://localhost:3000/add-picture`

There are some example pictures available under pictures/.

See the node cheat sheet on how to add pictures to a Mongo object.

3.2.4 Create a client for creating ads

The client shall first create the ad and then attach all the pictures to the ad.

Suggested command line API:

```
$ ./add-ad.js "Nice house for sale" "Four rooms, huge
  ↪bath." car.jpg dinosaur.jpg
```

The client must assert the status codes and content types returned. Proper error handling is not important now so just throw an exception on any failure.

3.3 Bonus Tasks

3.3.1 Proper error handling in the client

If the server returns an unexpected result or unprocessable content, do not crash but rather try to show the error message to the user. If the server returns `text/plain`, the user can be shown the text directly.

3.3.2 Add a `Last-Modified` header to the responses

3.3.3 Client to view an ad

Create a client that displays an ad.

Suggested command line API:

```
$ ./view-ad.js <id>
Fetching ad ...
Last-Modified: <time>
Title: <title>
Body: <body>
```

3.3.4 Caching Client

Improve the client and store the ad as `ad-<id>.js`. If executed twice it should do a conditional GET.

3.4 Hints

3.4.1 Converting JSON string to object

```
var object = JSON.parse(data);
```

3.4.2 Converting javascript object to string

```
var str = JSON.stringify({hello: "world"});
```

NOTE: Headers in the node.js http request object are ALWAYS in lower-case.

Setting status codes from node:

response.writeHead(200, headers);

3.4.3 Curl

Change HTTP method

```
-X GET|PUT|POST|DELETE|OPTIONS
```

POST image file

```
curl -X POST -H 'Content-Type: image/jpeg' -T
   ↪pictures/car.jpg <url>
```

Add `If-None-Match`

This header is mostly useful on conditional GET requests:

```
-H 'If-None-Match: <value-of-etag-header>'
```

Add `If-Match`

This header is mostly useful on conditional PUT|POST requests:

```
1 -H 'If-Match: <value-of-etag-header>'
```

Add `If-Modified-Since`

This header is mostly useful on conditional GET requests:

```
1 -H 'If-Modified-Since:
    ↪<value-of-last-modified-header>'
```

Add `If-Unmodified-Since`

This header is mostly useful on conditional PUT|POST requests:

```
1 -H 'If-Unmodified-Since:
    ↪<value-of-last-modified-header>'
```

3.4.4 MongoDB / Mongoose

Insert a new object

```
1 var ad = new Db.Ad();
2 ad.title = payload.title;
3 ad.body = payload.body;
4 ad.save();
```

When an object has been saved, the id of the object is available as the `_id` attribute.

Find a object

```
1 Db.Ad.findOne({_id: <my id>}, function(err, doc) {
2     ....
3 });
```

`err` will be set if there was an error while talking to the DB. `doc` will null if not found, or the object if found.

3.4.5 Update an existing object

```
1 var cmd = {$push: {arr: item}}
2 Db.Ad.update({_id: <my id>}, cmd, {}, function(err,
    ↪numAffected) {
3  ....
4 });
```

3.5 Retrospective

3.5.1 Why is this better than the rpc solution?

We have not utilized HTTP as an application protocol, mening we are delivering the application semantics through the protocol instead of layering on top.

We have started using HTTP the way it was intended to be used.

Delivering status codes, with the correct content-type headers allows us to become better Web citizens.

When implementing a real application, you would delegate routing and content-negotiation to a framework, for instance Express, or whatever you use in your programming language.

3.5.2 What are the benefits of caching?

Caching allows us to utiltize the scaling properties of the Web.

HTTP is optimized for GET and polling. Meaning that if we don't cache, we put a lot of strain on the origin server. Adding caching allows us to potentially never go to the origin server, except for verification of the representation of the resource.

3.5.3 Invalidation

This is a difficult problem to solve, because of the distributed nature of HTTP's caching model.

There are a few methods:

- Using a different method than GET or HEAD (invalidates the current resource)
- Using PURGE on a cache server (Does not invalidate local caches)
- Cache-Channels
- Edge Side Includes (ESI)
- Linked Cache Invalidation

3.5.4 Coupling?

There are still a lot of coupling between the client and the server. We are still hard-coding the URIs which we use.

3.6 Example `ad.json`

```
1 {
2   "title": "Nice house for sale!",
3   "body": "Four rooms, huge bath."
4 }
```

Chapter 4

Exercise: Media Types

4.1 Objective

In this exercise it's time to move from `application/json` to use Ads'r'us' own media type. We will also remove all URL generation from the clients. The server will decide which interactions that are allowed.

4.2 Steps

Implement the API described in Ads'r'us API (with domain-specific media types). Continue with the implementation you completed in the previous exercise.

Replace the usage of `application/json` with `application/vnd.ad+json` and `application/vnd.ad-list+json`. Update the `add-ad.js` client to start from the "ad list" URL and find the `add-ad` URL from the list document. In the end, the client should not hard code or generate any URLs.

4.3 Bonus

Work with another person/group and try your client against the other groups server. Add extra fields and headers to the responses to try to get the other client to fail. Make sure that you stay within the specification when adding fields and headers. It is also possible to change the URL that's served to the client.

4.4 Retrospective

4.4.1 What have we gained by moving URI creation out of the client?

We can now change the servers URI scheme, except from the initial URI, to whatever we want. We have thereby reduced the coupling of the client to the server.

We can now introduce new network components into the stack without having to change the client and how it works. For instance adding servers, and changing the links to point to another if the current server is overloaded.

The connection points are:

- The initial URI
- Link relations used
- the data formats used.

4.5 Example `ad.json`

```
1 {
2   "title": "Nice house for sale!",
3   "body": "Four rooms, huge bath."
4 }
```

Chapter 5

Exercise: Existing Media Types

5.1 Objective

Now we're moving from a domain specific media type to encode the same concepts by using an existing media type.

5.2 Steps

Update the server and client to use the new variant of the Ads'r'us API (with existing media types).

5.3 Retrospective

5.3.1 What is the gain by re-using existing media-types?

First of all, reusing existing parsers, is a good idea, as most bugs have been ironed out when you use them. Existing mindshare is also a desirable property.

5.3.2 Can hypermedia formats be too generic?

Yes, they can also be too specific, making them hard to change. HAL is an example of a too generic media type.

Chapter 6

Exercise: Hypermedia Design

6.1 Goal

Create a hypermedia type that represents an Ad. You should learn how to create a document which is extensible, and allows change.

This exercise should be done as a group.

6.2 Background

You are building Ads'r'us' new RESTful webservice and need to build a hypermedia type which expresses a classified advertisement with metadata.

6.2.1 Requirements

You may choose either XML or JSON as your base type.

The following fields MUST be modeled in your format:

published The published date of the ad

created The created date of the ad

last-modified The last-modified date of the ad

title The title of the ad

body The body of the ad.

creator A representation of the creator of the ad. This may be a person or an organization.

images A list of attached images.

There are also other types of fields in an ad:

subtitle A subtitle, optional.

lead Lead text, optional.

The named listed above are concepts, which your format must contain, the encoding of concepts to elements/attributes/properties are up to you.

More fields MAY appear at a different date. You must choose a strategy for modeling optional fields.

There must also be some hypermedia controls in the format which identifies resources connected to the ad-representation.

There might be advantages to choosing a generic format for representing links, but you are free to choose which solution you want.

The images which are connected to the ad, must be displayed. Depending on which hypermedia factor you choose, it should be up to the user agent, on when to download and display the images.

Choosing for instance LE, Link Embed, then the user agent can know ahead of time, that the image is important to the ad, and should thereby be preloaded, this is for instance how <img src=“”> works in HTML.

There MUST be a way of identifying which ad we are looking at. The id MUST NOT be a simple identifier.

6.3 Steps

6.3.1 Step 1

Choose base format (XML or JSON) and add required fields.

6.3.2 Step 2

Find a way of adding new fields without breaking backward compability

6.3.3 Step 3

Discuss the format you have created with the group.

6.3.4 Step 4

Implement server and client which understands the new format.

6.4 Hints

Useful link relations:

- `self`
- `alternate`
- `contact`
- `organization`
- `person`
- `image`

More link relations may be found at IANA's link-relations page.

For date-times you SHOULD use RFC3339. Example: `2012-06-05T12:00:02.52Z`.

Embedding images is not a good idea, so we need to link to them.

Embedding other formats, like HTML, within a format is a common trick for instance adding LE, Link Embed, for displaying images/video etc.

Use the hypermedia factors

6.4.1 XML

Use attributes where it makes sense instead of elements.

Links

In Atom, a hypermedia format, there's defined a link type which looks like this:

```
<atom:link href="http://example.com/ad/1" rel="self">
```

Domain specific

```
<ad href="http://example.com/ad/1"/>
```

Experiment, and try to find the best solution for your format.

6.4.2 JSON

It is very easy to introduce incompatible changes in JSON, thereby it's very important that one is diligent when designing the format.

Links

Like XML, JSON does not have any hypermedia controls built in. There are a couple of different ways of modeling links in JSON:

```
{
  "id": "http://example.com/ad/1",
  "property-1": "property-value",
  "property-2": "property-value",
  "property-3": "property-value"
}
```

Collection+JSON-style:

```
{
  "href" : "http://example.com/ad/1"
  "links": [
    {
      "href": "http://example.com/ad/1",
      "rel": "self",
      "title": "The current Ad"
    }
  ]
}
```

HAL-style:

```
{
  "property-1": "property-value",
  "links": {
```

```
4       "self": "http://example.com/ad/1"
5     }
6   }
```

6.5 Bonus

- Add a list-version of the current format.
- Add more links to other resources.
- Test against other servers.

6.6 Retrospective

6.6.1 How can you add new fields without introducing a breaking change?

This depends on your design. Adding a MUST-IGNORE property to your design allows you to add new fields without having to rewrite parsers to allow new field.

6.6.2 Why do you gain from hypermedia?

Hypermedia is a way of adding runtime components to your formats, it allows you to utilize the late-binding constraint which SHOULD be part of your design.

Adding hypermedia controls allows you to discover new services as they become available, enabling you to add features to your service without having to upgrade your clients on every server change.

6.6.3 Parsers

What requirements should parsers have?

The parsers need to conform to spec, meaning that if the design allows for extensions, the intermediary model should also allow for that.

Appendix A

Curl Cheat Sheet

A.1 POST a file

```
curl --data-binary @/path/to/file <url>
```

By adding `--data-binary`, curl will change the method to `POST` and set the `Content-Type` header to `application/x-www-form-urlencoded`.

A.2 POST image file

```
curl -X POST -H 'Content-Type: image/jpeg' -T
   ↪pictures/car.jpg <url>
```

A.3 Add Request Header

```
-H '<Header-Name>: <Header-Value>'
```

A.4 Change HTTP method

```
-X GET|PUT|POST|DELETE|OPTIONS
```

A.5 Verbose mode

See what curl is actually doing:

```
curl -v
```

A.6 Dump response header data to stdout

```
curl -D -
```

A.7 Do a GET, but ignoring the data

On unix:

```
curl -o /dev/null <other-options> <url>
```

On Windows:

```
curl -o nul <other-options> <url>
```

This is useful if combined with `-D` to give you the headers only:

```
curl -D - -o /dev/null <url>
```

A.8 Add `If-None-Match`

This header is mostly useful on conditional GET requests:

```
-H 'If-None-Match: <value-of-etag-header>'
```

A.9 Add `If-Match`

This header is mostly useful on conditional PUT|POST requests:

```
-H 'If-Match: <value-of-etag-header>'
```

A.10 Add `If-Modified-Since`

This header is mostly useful on conditional GET requests:

```
-H 'If-Modified-Since:
   ↪<value-of-last-modified-header>'
```

A.11 Add `If-Unmodified-Since`

This header is mostly useful on conditional PUT|POST requests:

```
-H 'If-Unmodified-Since:
   ↪<value-of-last-modified-header>'
```

Appendix B

Node Cheat Sheet

References:

- Node home page
- Full documentation

B.1 Minimal http server

```
1 var http = require("http");
2
3 http.createServer(function(req, res)) {
4     res.end("Hello World"); //Must be called to
        ↪"close" the response.
5 });
```

B.2 Running a node application

```
1 $ node server.js
```

B.3 Getting the request body as a string

```
1 http.createServer(function(req, res) {
2   var s = "";
3   req.on('data', function(chunk) {
4     s += chunk;
5   });
```

```
6   req.on('end', function() {
7     // 's' is now a string with all the data that the
        ↪client sent
8   });
9 });
```

B.4 Base64-encode binary data on a request

```
1  var data = [];
2  var datalength = 0;
3  req.on('data', function(chunk) {
4    data.push(chunk);
5    datalength += chunk.length;
6  });
7
8  req.on('end', function() {
9    var buf = new Buffer(datalength);
10   data.forEach(function(d) { d.copy(buf); });
11   var base64 = buf.toString("base64");
12   //do something with the data.
13 });
```

NOTE: Headers in the node.js http request object are ALWAYS in lower-case.

B.5 Converting JSON string to object

```
1  var object = JSON.parse(data);
```

B.6 Converting javascript object to string

```
1  var str = JSON.stringify({hello: "world"});
```

B.7 MongoDB / Mongoose

Import mongoose:

```
1  var mongoose = require('mongoose');
```

Declare your types and your database:

```
1  var Ad = new mongoose.Schema({
2      title    : String
3    , body     : String
4    , pictures : [String]
```

```
5 });
6
7 var Db = {
8   Ad: mongoose.model('Ad', Ad)
9 };
```

Wrap your entire application with an Mongo connection:

```
1 mongoose.connect('mongodb://localhost/exercise-1',
    ↪function() {
2   // Add http.createServer() etc here
3 });
```

Insert a new object:

```
1 var ad = new Db.Ad();
2 ad.title = payload.title;
3 ad.body = payload.body;
4 ad.save();
5 res.write(JSON.stringify("id=" + ad._id));
```

When an object has been saved, the id of the object is available as the `_id` attribute:

Find a object:

```
1 Db.Ad.findOne({_id: <my id>}, function(err, doc) {
2     ....
3 });
```

`err` will be set if there was an error while talking to the DB. `doc` will null if not found, or the object if found.

Update an existing object:

```
1 var cmd = {$push: {arr: item}}
2 Db.Ad.update({_id: <my id>}, cmd, {}, function(err,
    ↪numAffected) {
3     ....
4 });
```

Store an image inside of an object (this is not something you should do normally):

```
1 // Collect all the data buffers into an array
2 var data = [];
3 var datalength = 0;
4 req.on('data', function(chunk) {
5     data.push(chunk);
6     datalength += chunk.length;
7 });
```

```
8  req.on('end', function() {
9      // Create a new, huge buffer.
10     var buf = new Buffer(datalength);
11     // Copy all the buffers into the huge buffer.
12     data.forEach(function(d) { d.copy(buf); });
13     // base-64 encode the data.
14     var cmd = {$push: {pictures:
           ↪buf.toString("base64")}}
15     // Store the picture on the object.
16     Db.Ad.update({_id: id}, cmd, {}, function(err,
           ↪numAffected) {
17         ...
18     });
19 });
```

B.8 Command line arguments

Node.js has the command line arguments available in the global object `process`. They can be accessed using:

```
1 // Removes node + "name of your javascript file"
2 var args = process.argv.slice(2)
```

B.9 HTTP client

Use “request” module:

```
1 var request = require('request');
```

Upload image to URI:

```
1 function uploadImage(uri, file) {
2   fs.createReadStream(file).pipe(request.post(uri));
3 }
```