

REST Workshop

Trygve Laugstøl <trygve.laugstol@arktekk.no>
Erlend Hamnaberg <erlend.hamnaberg@arktekk.no>

Juli 2012

Contents

1	HTTP RPC	5
1.1	Objective	5
1.2	Steps	5
1.2.1	Create the ad endpoint: <code>http://localhost:3000/create-ad</code>	5
1.2.2	View ad endpoint: <code>http://localhost:3000/ad?id=...</code>	6
1.2.3	Add picture endpoint: <code>http://localhost:3000/add-picture?id=...</code>	6
1.3	Example <code>ad.json</code>	7
2	Basic HTTP	9
2.1	Objective	9
2.1.1	Common Requirements for Resources	9
2.1.2	Content Types	10
2.2	Steps	10
2.2.1	Create the ‘create ad endpoint’: <code>http://localhost:3000/create-ad</code>	10
2.2.2	View ad endpoint: <code>http://localhost:3000/ad?id=...</code>	10
2.2.3	Add picture endpoint: <code>http://localhost:3000/add-picture</code>	11
2.2.4	Create a client for creating ads	11
2.3	Bonus Tasks	11
2.3.1	Proper error handling in the client	11
2.3.2	Add a Last-Modified header to the responses	11
2.3.3	Client to view an ad	11
2.3.4	Caching Client	12
2.4	Example <code>ad.json</code>	12

<i>CONTENTS</i>	2
3 Media Types	13
3.1 Objective	13
3.2 Steps	13
3.3 Bonus	13
3.4 Example <code>ad.json</code>	14
4 Generic Media Types	15
4.1 Objective	15
4.2 Steps	15
A Ads'r'us API	17
A.1 General Architecture Rules	17
A.1.1 On Creating New Resources	17
A.1.2 On Links and Allowed Methods	18
A.2 Expected usage	18
A.3 Media Type: Single Ad / <code>application/vnd.ad+json</code>	18
A.3.1 Updating Ads	18
A.4 Media Type: Ad List / <code>application/vnd.ad-list+json</code>	19
B Ads'r'us API	21
B.1 General Architecture Rules	21
B.2 Expected usage	22
B.3 Ad Serialization Inside Collection+JSON	22
B.3.1 Data Fields	22
B.3.2 Links	22
B.3.3 Link Relations	22
B.3.4 Example	22
B.4 Single Ad vs List of Ad Resources	23

Chapter 1

HTTP RPC

1.1 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.

1.2 Steps

1.2.1 Create the ad endpoint: `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:

```
1 {
2   "result": "ok",
3   "data": {
4     "id": "4fc494d10b6a9fcb090000001",
5     "body": "Four rooms, huge bath.",
6     "pictures": [],
7     "title": "Nice house for sale!"
8   },
9 }
```

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

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

```
1 {
2   "result": "error"
3 }
```

1.2.2 View ad endpoint: `http://localhost:3000/ad?id=...`

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

```
1 {
2   "result": "ok",
3   "data": {
4     "title": "Nice house for sale!",
5     "body": "Four rooms, huge bath."
6   }
7 }
```

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

```
1 {
2   "result": "notFound"
3 }
```

1.2.3 Add picture endpoint: `http://localhost:3000/add-picture?id=...`

There are some example pictures available under `../pictures`.

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

1.3 Example ad.json

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

Chapter 2

Basic HTTP

2.1 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.

2.1.1 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.
- TODO: remove? Each resource should the appropriate caching information.
- 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.

2.1.2 Content Types

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

2.2 Steps

2.2.1 Create the ‘create ad endpoint’: `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.

2.2.2 View ad endpoint: `http://localhost:3000/ad?id=...`

Clients can GET this with an “id” query parameter. The server will respond with something 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 fetch an ad that doesn't exist and make sure you're getting a 404 error.

2.2.3 Add picture endpoint: `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.

2.2.4 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:

```
1 $ ./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.

2.3 Bonus Tasks

2.3.1 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.

2.3.2 Add a Last-Modified header to the responses

2.3.3 Client to view an ad

Create a client that displays an ad.

Suggested command line API:

```
1 $ ./view-ad.js <id>
2 Fetching ad ...
3 Last-Modified: <time>
4 Title: <title>
5 Body: <body>
```

2.3.4 Caching Client

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

2.4 Example `ad.json`

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

Chapter 3

Media Types

3.1 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.

3.2 Steps

Implement the API described in [Ads'r'us API](#). 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.

3.3 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.

3.4 Example ad.json

woot

Example ad.json

```
.json  "title":  "Nice house for sale!", "body":  "Four  
rooms, huge bath."
```

Chapter 4

Generic Media Types

4.1 Objective

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

4.2 Steps

Update the server and client to use the new variant of the [API](#).

This document is an example on how one can document an API by using the REST principles. This version of the API use a domain specific media type with JSON serialization.

Appendix A

Ads'r'us API

This document describes an architecture for how agents should interact with Ads'r'us. Interactions with Ads'r'us follow the general architecture and constraints described in the REST dissertation and add some domain specific constraints.

We only implement a HTTP interface to our resources which come with its own set of constraints. As the world progresses, new specifications, updates and enhancements arrive and we try to stay as much as possible in line. However, we will always remain backwards compatible with our own published specifications.

Make sure to check back regularly to get the latest version of this specification!

A.1 General Architecture Rules

These are general rules and principles used. They apply as long as a specific interaction/resource does not specify something else.

All resources should return an **Allow** header if the method is not allowed/supported.

It is expected that the domain and objects that the resources describe evolve over time and thus the media types will need evolve over time too. Clients must silently ignore unknown fields and should not expect optional fields to be present.

A.1.1 On Creating New Resources

When a new resource is created, the server **MUST** return the status code 201 and a **Location** header pointing to the newly created resource.

A.1.2 On Links and Allowed Methods

Many resources will include URIs to other resources and operations available for the client. The resources will include all links, even if the client may not be authorized to use them.

A.2 Expected usage

It is assumed that clients are pre-programmed with a link to an ad list when starting to interact with us.

A.3 Media Type: Single Ad / `application/vnd.ad+json`

```
1 {
2   "title": "Nice house for sale!",
3   "body": "Four rooms, huge bath.",
4   "pictures": [
5     URI, ...
6   ]
7   "self": URI,
8   "add-picture": URI,
9   "publish": URI,
10  ...
11 }
```

Fields:

- **title**: the title and body of the ad. Available: summary
- **body**: the title and body of the ad.
- **self**: the canonical URI for this ad. Can be used to send a link to other agents or when editing the ad.
- **pictures**: A JSON list of URLs that point to image files.
- **add-picture**: Send pictures here to add them to an ad.

A.3.1 Updating Ads

When updating the ad, only these fields needs to be sent:

- **title**

- body

There is no described way to update only a subset of the fields in an ad.

A.4 Media Type: Ad List / `application/vnd.ad-list+json`

```
1 {
2   "count": 10,
3   "add-ad": URI,
4   "ads": [
5     {
6       // See specification for application/vnd.ad+json
7     }
8   ],
9   "next-page": URI,
10  "prev-page": URI,
11  ...
12 }
```

To insert a new ad into the list, send the data to the `add-ad` URI.

This document is an example on how one can document an API by using the REST principles. This version of the API uses the generic `application/vnd.collection+json` format instead of a specific media type.

Appendix B

Ads'r'us API

This document describes an architecture for how agents should interact with Ads'r'us. Interactions with Ads'r'us follow the general architecture and constraints described in the REST dissertation and add some domain specific constraints.

We only implement a HTTP interface to our resources which come with its own set of constraints. Most of our data resources (non-binary) uses the existing `application/vnd.collection+json` media type, commonly known as “Collection+JSON”:

Collection+JSON is a JSON-based read/write hypermedia-type designed to support management and querying of simple collections. It is similar to the The Atom Syndication Format (RFC4287) and the The Atom Publishing Protocol (RFC5023) . However, Collection+JSON defines both the format and the semantics in a single media type. It also includes support for Query Templates and expanded write support through the use of a Write Template.

As the world progresses, new specifications, updates and enhancements arrive and we try to stay as much as possible in line. However, we will always remain backwards compatible with our own published specifications.

Make sure to check back regularly to get the latest version of this specification!

B.1 General Architecture Rules

These are general rules and principles used. They apply as long as a specific interaction/resource does not specify something else.

It is expected that the domain and objects that the resources describe evolve over time and thus new fields and link relations might be introduced later on. Clients must silently ignore unknown fields and should not expect optional fields to be present.

B.2 Expected usage

It is assumed that clients are pre-programmed with a link to an ad list when starting to interact with us.

B.3 Ad Serialization Inside Collection+JSON

B.3.1 Data Fields

When serializing an ad inside a Collection+JSON, these fields are available:

- **title**: The one-line title of the ad. MUST be present.
- **body**: The body part of the ad. MAY span multiple lines. MUST be present.
- **lastModified**: The timestamp of the ad. MUST use Atom's date format. See [3.3: Date Constructs](#). SHOULD be present when served from a server. MAY be present when a client sends one to the server.

B.3.2 Links

- **add-picture**: Send pictures here to add them to an ad.

B.3.3 Link Relations

In addition to the link relations specified by Collection+JSON, we use the **picture** relation to reference pictures that are related to the current resource.

B.3.4 Example

```
1 { "collection" :  
2   {  
3     "version": "1.0",  
4     "href": "http://localhost:3000/ads",
```

```

5
6     "links": [
7         {"rel": "self", "href":
8             ↪ "http://localhost:3000/ads"}
9         {"rel": "collection", "href":
10            ↪ "http://localhost:3000/ads"}
11         {"rel": "feed", "href":
12            ↪ "http://localhost:3000/ads.atom"}
13     ],
14     "items": [
15         {
16             "href" : "...",
17             "data" : [
18                 {"name" : "title", "value" : "Nice house
19                 ↪ for sale!", "prompt" : "Title"},
20                 {"name" : "body", "value" : "Four rooms,
21                 ↪ huge bath.", "prompt" : "Ad Body"}
22             ],
23             "links" : [
24                 {"rel" : "add-picture", "href" : "..."}
25                 {"rel" : "picture", "href" : "...",
26                 ↪ "render" : "image"}
27                 {"rel" : "picture", "href" : "...",
28                 ↪ "render" : "image"}
29             ]
30         },
31         ...
32     ]
33 }
34 }

```

B.4 Single Ad vs List of Ad Resources

As Collection+JSON does not have a way to send a single item only the context of the request specifies if the list will include one or more items.