Breakfast

Coding

#breakfastcoding

A Session for Developers

Make sure you have:

- GET WiFi

- GET the code

  https://github.com/allanhojgaardjensen/breakfastcoding

- Download Code....

#breakfastcoding

A Session for Developers

Make sure you have:

- Java 8
- Maven 3.5
- Netbeans 8.2
- Git Bash

Installed on your machine.

Chrome plugin:
Postman will be handy during the session

#breakfastcoding

A Session for Developers

Make sure you have:

- SET “HTTP Proxy for HTTP and HTTPS“
- SET “Maven Main Repo as the sole repository“
- CREATE! a repository to share with your team
- CLONE! repository `git clone https://....`
- COPY! unzipped BreakfastCode into local repo
- BUILD! BreakfastCode `mvn verify`
- PUSH! BreakfastCode, commit and push....

#breakfastcoding

A Session for Developers

Make sure you get:

- a git repository <repo> to clone from

And in a command line run

- git clone https://.....<repo>
- mvn verify
- start Netbeans + open project

Breakfast Coding

Event

create your Repo

get your zip

start HACKING

#breakfastcoding

A Session for Developers

Make sure you have:

- VALUE “HTTP Proxy for HTTP and HTTPS“
- VALUE “Maven Main Repo as the sole repository“
- GOT “local repository with initial source“
- GOT Breakfast is ready for Coders....

A Session for Developers

Make sure you have:

● Java 8	`java -version`
● Maven 3.5	`mvn -v`
● Netbeans 8.2	`check top of window or help/about`
● Git Bash	`git config --list  / set git global settings`

Installed on your machine.

Chrome plugin:
Postman will be handy during the session

#breakfastcoding

You will learn...

To evolve a simple REST service

AND

the troubles that may cause despite all good intentions

AND

how to cope with that

Did you get your code

On Local Disc

Compiled
Packaged
Running

Netbeans

Did you get your code

```
git clone https://<repo>
```

On Local Disc

Compiled
Packaged
Running

Netbeans

Did you get your code

```
git clone https://<repo>
```

On Local Disc

```
mvn compile

mvn package

mvn exec:java
```

Compiled
Packaged
Running

Netbeans

Did you get your code

```
git clone https://<repo>
```

On Local Disc

```
mvn compile

mvn package

mvn verify  exec:java
```

Compiled
Packaged
Running

Netbeans

Did you get your code

`git clone https://<repo>`	On Local Disc
`mvn compile`	Compiled
`mvn package`	Packaged
`mvn verify  exec:java`	Running
`file - open project <repo>`	Netbeans

OK - Let us get started

Overview

The Starting Point

Resources and Resource Oriented API Design

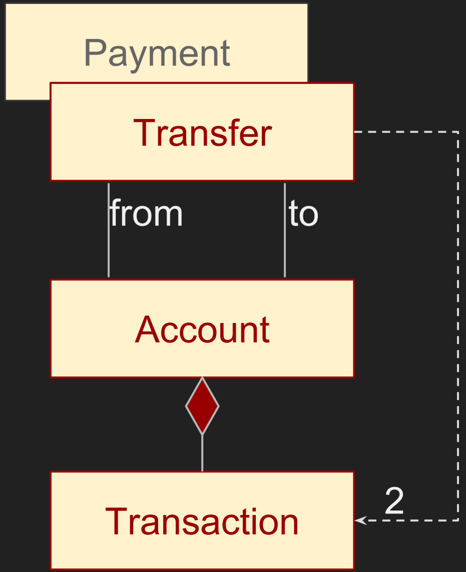
The difference between classical SOA and resource oriented design?

```
 https://account.services.example.com
  /accounts
           /{account}
                      /transactions
                                   /{transaction}
  /transfers
           /{transfer}

   /payments
...
```

vs.

```
https://services.example.com/ws/AccountService
```



The Starting Point

A very simple service with one resource:

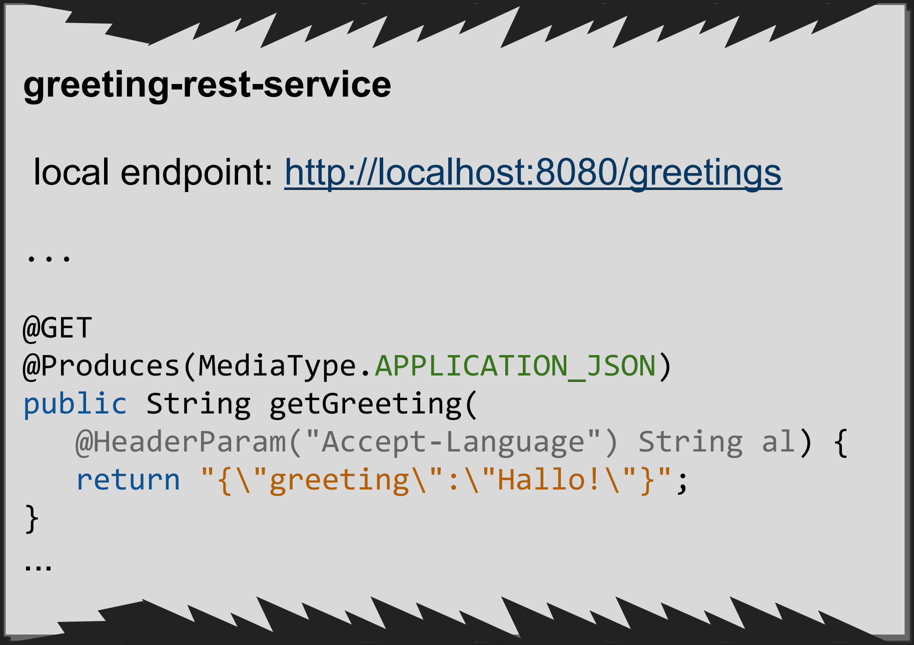
`/greetings`

supports one verb:

`GET`

it responds with Danish greeting:

```
{
    "greeting": "Hallo!"
}
```



greeting-rest-service

local endpoint: http://localhost:8080/greetings

```
...

@GET
@Produces(MediaType.APPLICATION_JSON)
public String getGreeting(
   @HeaderParam("Accept-Language") String al) {
   return "{\"greeting\":\"Hallo!\"}";
}
...
```

#breakfastcoding

Let us do the “code walk” v0.1

we are examining the initial feature a service that greets in Danish

Development of the “next” feature

In a real development setup, we would:

- been through a refinement of features on the backlog
- played the planning game and have prioritized the feature
- have a chosen set of features on a sprint backlog
- we would pick a feature from the sprint backlog
- create a short-lived feature branch*
- start development on that feature branch*

* there are currently two approaches:

1) using the short-lived team visible feature(s) branches

2) using only “feature” branching on the developer’s workstation/notebook

Suggested way of doing this session....

The goal is for you to do the coding of the service, if we are short on time I will go through the code and highlights for the feature.

Beware the initial features are very basic and primitive, the complexity and effort is exponentially increasing feature by feature and thus you may vote for some of them to “be walked through” if you cannot see yourself implementing that in a short time.

The focus is the evolutionary speed of the API and handling backwards compatibility whilst evolving the API and not so much the implementation.
The quality of the code etc. does have an effect on the evolutionary speed so it is covered, but i is not at the center of this session.

Sprint#1

Feature #1

Let’s make the first feature

That is having the service greet in a more widespread language than Danish.

The service must be able to greet in English.

The service endpoint has the “Accept-Language” header as a part of the initial implementation, which means that somebody (“me”) did start something in the initial implementation and did not finish that.....

hmm... that’s a code smell...

... let’s have the service greeting in English
in addition to the existing Danish greeting.

#1 Add English Greeting v0.2

Danish Greeting

```
{
    "greeting": "Hallo!"
}
```

English Greeting

```
{
    "greeting": "Hello!"
}
```

Findings

CORS - problems anyone? - perhaps you added a CORS filter

“Accept-Language” header implementation based on “NIHS/“me” or using the `Local.LanguageRange.parse()` or something else.

```
String[] languages = preferred.split(",");
String[] preferredLanguage = Arrays.stream(languages).filter(s ->!s.contains(";")).toArray(String[]::new);
return preferredLanguage[0];

----- OR -----

return Locale.LanguageRange.parse(preferred).stream().map(rang e -> new Locale(range.getRange()))
      .collect(Collectors.toList()).get(0).getLanguage();
```

Findings

CORS - problems anyone? - perhaps you added a CORS filter

“Accept-Language” header implementation based on
“NIHS/“me” or using the `Local.LanguageRange.parse()` or something else.

```
String[] languages = preferred.split(",");
String[] preferredLanguage = Arrays.stream(languages).filter(s ->!s.contains(";")).toArray(String[]::new);
return preferredLanguage[0];

----- OR -----

return Locale.LanguageRange.parse(preferred).stream().map(rang e -> new Locale(range.getRange()))
      .collect(Collectors.toList()).get(0).getLanguage();
```

Did anyone break the backwards compatibility
- en in english and everything else in danish

Readability of the source...

Feature #2

#breakfastcoding

Let’s move on to the next feature

The greeting is detailed and made an explicit resource.

A concrete greeting must include details on country.

The service endpoint is extended with an explicit greeting, e.g. endpoints like

```
/greetings/hello
```

and

```
/greetings/hallo
```

The response must contain text for the UI in language specified in the Accept-Language header, however only supporting Danish and English currently.

GET greetings/hello

Accept-Language “en”

```
{
    "greeting": "Hello!",
    "country": "GB",
    "_links": {
        "href": "/greetings/hello",
        "title": "English Greeting Hello"
    }
}
```

Accept-Language “da”

```
{
    "greeting": "Hello!",
    "country": "GB",
    "_links": {
        "href": "/greetings/hello",
        "title": "Engelsk Hilsen Hello"
    }
}
```

GET greetings/hallo

Accept-Language “en”

```
{
    "greeting": "Hallo!",
    "country": "DK",
    "_links": {
        "href": "/greetings/hallo",
        "title": "Danish Greeting Hallo"
    }
}
```

Accept-Language “da”

```
{
    "greeting": "Hallo!",
    "country": "DK",
    "_links": {
        "href": "/greetings/hallo",
        "title": "Dansk Hilsen Hallo"
    }
}
```

GET /greetings

Accept-Language “en” or any other - response content-type is “application/hal+json

```
{ "greetings": {
      "info": "a list containing current greetings",
      "_links": {
          "self": {
              "href": "/greetings",
              "type": "application/hal+json;concept=greeetinglist;v=1",
              "title": "List of Greetings"
          },
          "greetings": [{
              "href": "/greetings/hallo",
              "title": "Danish Greeting - Hallo"
              },{
              "href": "/greetings/hello",
              "title": "English Greeting - Hello"
              }]
        }
    }
}
```

#2 Add Greeting Details v0.3

#breakfastcoding

Findings

CORS filter added

Build now includes uber-jar

Original getGreetings deprecated, but still working

Preparation for content based versioning for /greetings/{greeting}

Handling of unsupported media-types using getOrDefault(...)

Backwards compliance - producers as a last-resort control-mechanism (semantics?)

Your opinion?

Is it correct to use application/json for the resources in the current case?

/greetings and /greetings/{greeting}

would it be correct to use application/hateoas+json? - where would application/hal+json fit?

Feature #3

Let’s move on to the next feature

The greeting is detailed and made an explicit resource.

A greeting must have more details on language and country.

The feature targets the explicit greeting endpoints, e.g.

```
   /greetings/hello
```
and
```
   /greetings/hallo
```

The response must contain text for the UI in language specified in the Accept-Language header, and a native json object in the consumer’s preferred language - however only supporting Danish and English currently.

GET greetings/hello

Accept-Language “en”

```
{
    "greeting": "Hello!",
    "language": "English",
    "country": "England",
    "native": {
        "language": "English",
        "country": "England"
    },
    "_links": {
        "href": "/greetings/hello",
        "title": "English Greeting Hello"
    }
}
```

Accept-Language “da”

```
{
    "greeting": "Hello!",
    "language": "English",
    "country": "England",
    "native": {
        "language": "Engelsk",
        "country": "England"
    },
    "_links": {
        "href": "/greetings/hello",
        "title": "Engelsk Hilsen Hello"
    }
}
```

GET greetings/hallo

Accept-Language “en”

```
{
    "greeting": "Hallo!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
        "language": "Danish",
        "country": "Denmark"
    },
    "_links": {
        "href": "/greetings/hallo",
        "title": "Danish Greeting Hallo"
    }
}
```

Accept-Language “da”

```
{
    "greeting": "Hallo!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
        "language": "Dansk",
        "country": "Danmark"
    },
    "_links": {
        "href": "/greetings/hallo",
        "title": "Dansk Hilsen Hallo"
    }
}
```

#3 Add Language Details v0.4

#breakfastcoding

Findings

Structure improved

Versioning /greetings/{greeting} supported in version 1 and 2

Version 2 is default

We are supporting individual consumer controlled fall-back using Accept header

Original getGreetings still deprecated, whereas the G1V1 is not (Yet)

Findings

The change is within Open/Close but it is potentially breaking if we did not maintain version 1 as well, some consumers may not have the room for more than 2 letters from the country code.

UPS - did somebody notice that even if we stayed within the contract we managed to break /greetings in the situation, where the consumer did not send an Accept header, that used to return the simple greeting and now it returns the list. It can be fixed, but it is easy to make that sort of mistake and backwards compatibility is the victim.

Note

Still using “application/hal+json” for
    /greetings and /greetings/{greeting}
where only valid for the former,
the latter should be “application/json” or “application/hateoas+json”

Feature #4

Let’s move on to the next feature

The greeting service is only visible to people knowing it and know the endpoint to get the WADL eg. GET `http://localhost:8080/application.wadl?detail=true`

Greeting service must have standard API documentation.

The feature targets the having an easy to read documentation of the API.

The API documentation is preferred to be automatically generated and thus following the development of the code. OpenAPI version 2 seems to be the way things are moving for most services.

#4 Add API Documentation v0.5

Findings

Annotations like e.g. ApiOperation etc. has been added to the code.
A service API specification was generated as a part of the build.

The documentation can now be viewed in the swagger editor and apiary, they both support the OpenAPI.

The API will have a version that follows the code and could be deployed as a part of an service catalogue having a service catalogue client that can view the code, deliver endpoints for “try it”, and links to subscriptions, contracts etc. That application is promoted using every normal means for promoting applications such as SEO to make your service rank well.

Findings

If you have done the implementation using swagger annotations directly in the code, you may have used quite some space in the Greeting resource and that may clutter the development experience.

The development speed of the service is important and the complexity should not be within the documentation of the service, that should be with the services and the ability to allow for individual consumers to catch up on service content versions in their own pace.

Thus we want to automate that part further.

Feature #5

Let’s move on to the next feature

The greeting service now have some basic documentation, that can be viewed in tools understanding OpenAPI specification version 2. There are some parts missing though. There is no expectations defined in the API that states a consumer must be able to handle a bad request, non-supported content-type, permanently moved etc. Specifying that using annotations would further clutter the code.

Service must have elaborated API documentation.

The feature targets the having an easy to read documentation of the API and prepare consumers for changes to the API, resources may be moved, operations may be deferred, content may not be authoritative from a given endpoint.

#5 Automate v0.6

Elaborated API Doc

```
mvn clean verify exec:java@api-docs exec:java@start-server
```

Findings

Annotations like e.g. ApiOperation is still in the code, main changes here is in the build file. An elaborated service API specification was generated as a part of the build.

The documentation can now be viewed in the swagger editor and apiary, they both support the OpenAPI and looking at the API specification we can see that the endpoints are enriched with a number of response codes and headers as well as some request headers.

There needs to be a balance and everyone has to figure out what their needs are in relation to responses and headers. The important part is that it is possible to signal to consumers what they can expect and what responses they need to be able to react to.

Feature #6

Let’s move on to the next feature

The greeting service now have elaborated API documentation, that can be viewed in tools understanding OpenAPI specification version 2. We do not really utilize the headers and the response headers yet, we want to optimize the bandwidth and support some of the response types like 304 not modified. The code has some checks, but more could be relevant for better code quality(development speed).

Service must be bandwidth efficient and trackable.

The feature targets the having an implementation that lets the consumer track a request and correlate that to own correlation ids and support for not re-sending responses to consumers having the right version already.

#6 Improve v0.7

Bandwidth
Traceability
Code Doc/Quality
Artifacts
....

```
mvn clean verify exec:java@api-docs exec:java@start-server
```

Findings

We see that a response is only returned if the consumer has an outdated version. Outdated means either content has changed or the the timestamp for the object is newer that what the consumer has indicated as last time for the object in question.

A word of caution here is:

The detection of the last modified and the entity-tag calculation should be very easy to get, if you have to load an object (possible via an internal model - persistency etc.) it may be better just to return the object if small, otherwise keep a tuple allowing you to see the key::resource and its (entity-tag, last modified) in a map in mem lazy-loaded and tracking a feed on updates to resources.

Findings

We see that a logtoken (aka correlation) can be specified from the client side making it easier to communicate with the consumer and if a log/error self-service is something we offer they consumer developers can easily correlate with what they have. If unspecified from the consumer side a token is drawn and may be used onwards or a new is drawn every request.

We see that reports have been added on source, source-doc, coverage, bug and code smells as well as at the “project level”

Findings

The versions are preserved as the existing consumers are capable to continue, on each their version or move to the newest version.

All consumers using “`application/hal+json`” for their “Accept” header value are moved along with the newest edition of the content for that endpoint in the service.

Whereas they can individually fall-back to previous version using the “`application/hal+json;concept=greeting;v=<version>`” in their `Accept` header.

Lessons learned

An initial mess always complicates things, having an unclear semantics and improper use of semantics whilst having consumers using services will slow us down going forward.

This is a very simple service with an unreal simple implementation, we did not really invest a lot in semantics, but the last couple of features brought us the ability to move the /greetings/{greeting} resource faster forward, whilst being somewhat more stuck with the /greetings resource as the first feature was a groce mistake and work needs to be done to get rid of that asap.

Other points?

Sprint #2

Feature #7

Let’s move on to the next feature

The greeting service is a bit boring as it only has two greetings and supports two languages English and Danish. In order for our number of potential consumers to increase, we desperately need to be able to create new greetings in other languages as well.

It must be possible to create new greetings.

This means that resources are to be:

```
/greetings
          /hallo
          /hello
          /{greeting}
```

mvn clean verify exec:java@api-docs site exec:java@start-server

POST greetings

```
Accept-Language “en”
  {
    "greeting": "Ohøj!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "Danish",
      "country": "Denmark"
    },
    "_links": {
      "self": {
        "href": "greetings/ohoj",
        "title": "Danish Greeting Ohoj"
      }
    }
  }
```

```
Accept-Language “da”
  {
    "greeting": "Ohøj!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "Dansk",
      "country": "Danmark"
    },
    "_links": {
      "self": {
        "href": "greetings/ohoj",
        "title": "Dansk Hilsen Ohøj"
      }
    }
  }
```

#7 Ability to Create Greeting(s) v0.8

```
mvn clean verify exec:java@api-docs site exec:java@start-server
```

Findings

We see that a response from a POST new greeting returns a 201 created in the event of a successful create and it includes a Location header informing the consumer about the whereabouts of the newly created greeting.

Furthermore we see that a 400 Bad Request is returned if the request was malformed.

Recreation of the same object is allowed and thus there is no way for the consumer to see if the initial create went well or not, unless the consumer does a GET to a location the consumer guesses is the place for the newly created greeting and reacts to a not found. This would be a good thing to address in a later feature.

Findings

We see that a successfully created greeting is now part of the greetings list.

If stop for a moment and looks at the semantics here, is it ok to have greetings posing as separate objects, where language is the differentiator or not.

The code looks different, the objects are no longer Strings they are real objects in the implementation, although very simple ones. The objects capable of creating resulting json are postfixed Representation. They are not the internal model of a service, they are merely representing a given view or projection, that is signalled by the content parameter `concept=<view/projection>` and currently that is version 3 of the content `;v=3` in the endpoint `/greetings/{greeting}`

Findings

We see that the implementation of the initial implementation of
/greetings/{greeting}
for version 1 has become deprecated. There is no standardized way to tell that to the consumer in real time, we use “`X-Status`” response header is used for that.

We se that we are running “2 version overlap” and that means we have 3 versions when we change and deprecate the oldest. You can run any overlapping #.

If you have very important consumers you may want to keep some of the older versions and have them available still, this is somewhat similar to continuing to run old releases. You may run them in the same service implementation for a while, with a goal for them to be separated or the consumers upgraded. The X-Service-Generation header can be used for that purpose and used for segmentation.

Findings

Very simple “just enough” implementations of HAL has been made and the application/hal+json is finally appropriate to be used, there are libraries out there that can be used to map Representations between Objects and HAL, see HAL information. HAL in the form of application/hal+json is an informational standard.

Caution:

Normally you would use the libraries, the “just enough” implementation her is just to show the mapping as simple as possible. There are limitations to the implementation done here such as not support for array or object, this only supports object.

Findings

Did you have issues handling the “native” object as native is a reserved keyword in java and thus it needs some form of mapping.

This was chosen as a way to illustrate that sometimes you will run into thing that are platform implementation specifics and there will usually be a way round that. .

When you look at the code at this stage you will properly have seen, it is rapidly on its way to become a mess. This is entirely my fault, due to a number of bad decisions and lack of accuracy in requiring headers being set and correctness in using HAL etc. I created a situation, where the code inside the service get cluttered, despite the efforts made to exclude a big portion of the annotations.

So we will do a clean up - however in a real life setup - that would take long.

Feature #8

Let’s move on to the next feature

The greeting service is a little less boring now, where it is possible to create greetings. We need to be able to replace greetings and replace content in the existing greetings.

It must be possible to edit/replace greetings.

Targeted resources are:

```
/greetings
          /{greeting}
```

mvn clean verify exec:java@api-docs site exec:java@start-server

PUT greetings/{greeting}

direct create

```
Accept-Language “en”
  {
    "greeting": "Møjn!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "Danish",
      "country": "Denmark"
    },
    "_links": {
      "self": {
        "href": "greetings/mojn",
        "title": "Danish Greeting Mojn"
      }
    }
  }
```

```
Accept-Language “da”
  {
    "greeting": "Møjn!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "Dansk",
      "country": "Danmark"
    },
    "_links": {
      "self": {
        "href": "greetings/mojn",
        "title": "Dansk Hilsen Møjn"
      }
    }
  }
```

should return 201 Created with Location Header set to /greetings/{greeting}

PUT greetings/{greeting}

direct complete replace

```
Accept-Language “en”
  {
    "greeting": "Møjn!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "SouthJutlandish",
      "country": "Denmark"
    },
    "_links": {
      "self": {
        "href": "greetings/mojn",
        "title": "South Danish Greeting Mojn"
      }
    }
  }
```

```
Accept-Language “da”
  {
    "greeting": "Møjn!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "Sønderjysk",
      "country": "Danmark"
    },
    "_links": {
      "self": {
        "href": "greetings/mojn",
        "title": "Sønderjysk Hilsen"
      }
    }
  }
```

should return 200 OK

POST greetings

indirect create

```
Accept-Language “en”
  {
    "greeting": "Møjn!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "SouthJutlandish",
      "country": "Denmark"
    },
    "_links": {
      "self": {
        "href": "greetings/mojn",
        "title": "South Danish Greeting Mojn"
      }
    }
  }
```

```
Accept-Language “da”
  {
    "greeting": "Møjn!",
    "language": "Dansk",
    "country": "Danmark",
    "native": {
      "language": "Sønderjysk",
      "country": "Danmark"
    },
    "_links": {
      "self": {
        "href": "greetings/mojn",
        "title": "Sønderjysk Hilsen"
      }
    }
  }
```

should return 201 on Create and 409 Conflict on re-create

#8 Ability to Replace Greeting(s) v0.9

Findings

We see that a response from a PUT greeting returns a 201 created in the event of a successful create and it includes a Location header informing the consumer about the whereabouts of the newly created greeting.

Furthermore we see that a 201 Created is responded in the event of a POST, whereas now the 409 Conflict is returned if we try to re-create a greeting.

Recreation of the “same” object is no longer allowed using the POST verb. Creation and replace is allowed using the verb PUT and the creation is responded to using a 201 Created and a 200 OK in the event of a replace.

Findings

We see that `/greetings/{greeting}` the initial version (generation 1 version 1) has been removed and thus the application/hal+json;concept=greeting;v=1 is no longer available for consumers anymore, whereas the version 2 and 3 are still supported.
This initial greeting mix of a simple Danish and English greeting is also gone and the list is all that is left at the `/greetings` resource.

Finally we are on our way towards a semantically reasonable API. But wait a minute, how does the client know what versions are supported. They don´t - in the current implementation.
Usually I use a content producer application/hal+json or application/json having the content-type parameter concept=metadata which is an endpoint that runtime can inform the consumers about the endpoint capability. Annotations like the ApiOperation attribute “produces” can be used to include that in the service specification.

Feature #9

Let’s move on to the next feature

The greeting service is on it way to become a place where greetings live. They are created, remembered as long as the service is running. They may change but they cannot be removed yet.

It must be possible to delete greetings.

Targeted resources are:

```
/greetings
          /{greeting}
```

mvn clean verify exec:java@api-docs site exec:java@start-server

DELETE greetings/{greeting}

direct delete

```
Accept-Language “en”
```

```
Accept-Language “da”
```

should return 204 No Content and following that 404 Not Found

#9 Be able to Delete Greetings v0.10

#breakfastcoding

Findings

We see that a response from a DELETE greeting returns a 204 no content in the event of a successful delete and if you tru to delete again it returns a 404 not found.

This is probably the best way to see idempotency in action, PUT, DELETE are idempotent verbs. POST is not and may have side effects, whereas GET should never have side effects.

If you look at code right now it seems like the “infrastructural parts” are occupying a significant amount of the lines of code in the service and this does not have to be that way. There are ways to remove that into libraries and depend on these and in that way reduce the footprint inside the individual service.

It is included here to have “everything” within reach in this simple example service.

Lessons learned

A service with better semantics seems to emerge. It it still doubtful whether the semantics of having “multiple language dependent instances” under the same resource, this starts to get more confusing and that needs to be addressed going forward. Probably country needs to be a separate resource and the greetings may be explicitly made having an e.g. English greeting link to one or more countries and languages where the greeting gradually becomes a simpler object and deals with the meaning of the greeting and not with the whole on a superficial level as now. On the other hand is this too much to do for a simple service like the greeting service?

Other points?

Sprint#3

Feature #10

Let’s move on to the next feature

The greeting service is on it way to become a place where greetings live. They are created, remembered as long as the service is running, they can be removed. However they are created, deleted, replaced, but not able to change partially.

It must be possible to update parts of a greeting.

Targeted resources are:

```
/greetings
        /{greeting}
```

```
mvn clean verify exec:java@api-docs site exec:java@start-server
```

PATCH greetings/{greeting}

direct partial update

```
Accept-Language “en”
  {
   "op":"replace",
   "path":"links/self/title",
   "value":"This is the new title"
  }
```

```
Accept-Language “da”
  {
   "op":"replace",
   "path":"language",
   "value":"Land"
  }
```

maybe simplified edition of RFC5789/6902 -should return 200 OK if success, or 400, 409,404 - (422)?

#10 Partially Update of Greeting(s) v0.11

#breakfastcoding

Findings

We see that it is possible to partially update an object and get a 200 ok response as well as the 400 and 409 from a PATCH verb.

The current implementation is a very simple edition of parts of the RFC6902 only supporting replace and doing that for non-array objects. PATCH is usually only used when objects are humongous and not for objects of this size. There is still quite a lot of discussion around the actual path PATCH implementations should take, I have used the content-type ”`application/patch+json`” and not ”`application/json-patch+json`” as this is not strictly following or complete the RFC.

PATCH is not idempotent and save as we saw for e.g. PUT.

It is included here to have “everything” within reach in this simple example service.

Sprint#4

Feature #11

No Downloadable Code for this feature

you are the hero of your own service from now on

#11 Separate Country Information

#breakfastcoding

Feature #12

#breakfastcoding

No Downloadable Code for this feature

you are still the main hero of your own service

#11 Externalize Country Service

WrapuP

#breakfastcoding

What we learned from....

simple decisions

can have a significant n effect on

+ API evolution speed

+ code complexity

HATEOAS

HATEOAS can be done without HAL
+ specific
+ tools

Automation

Build and Documentation

+ annotation vs. magic

+ code complexity

+ quality

Want to learn more....

FIN

#breakfastcoding