AARHUS UNIVERSITET

Software Engineering and Architecture

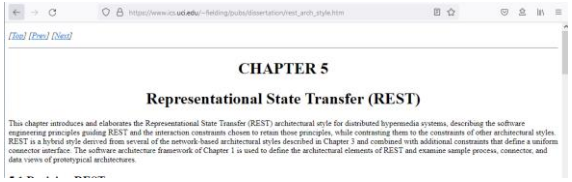
REpresentational State Transfer

AARHUS UNIVERSITET

What is REST

Representational state transfer (**REST**) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave. The REST architectural style emphasises the scalability of interactions between components, uniform interfaces, independent deployment of components, and the creation of a layered architecture to facilitate caching components to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]

REST has been employed throughout the software industry and is a widely accepted set of guidelines for creating stateless, reliable web APIs. A web API that obeys the REST constraints is informally described as **RESTful**. RESTful web APIs are typically loosely based on HTTP methods to access resources via URL-encoded parameters and the use of JSON or XML to transmit data.

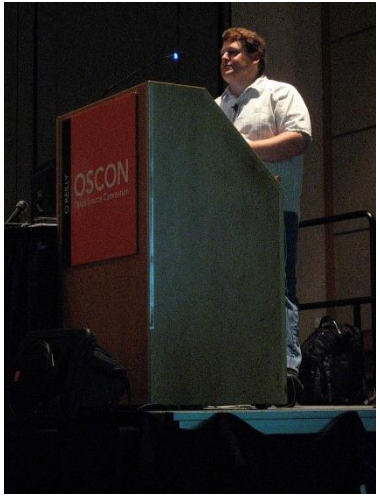


CHAPTER 5

Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

One of the rare cases, in which a PhD dissertation *actually* moved IT industry a lot!



AARHUS UNIVERSITET

What is REST

- As a software architect, I see it as an
  - **Architectural style / pattern**
- It is *another programming model*
  - Functional programming:
    - Computation is passing data through chains of functions
  - Object programming:
    - Computation is community of objects passing messages
  - RPC over Client-Server:
    - Computation is clients invoking procedures on remote servers
  - REST
    - Computation is clients manipulating resources using CRUD ops and moving through states using hypermedia links

AARHUS UNIVERSITET

Programming Model

- Broker pattern
  - Supports RPC/RMI between clients and servers
    - State changes through accessors and mutator methods
    - Any interface is possible

- REST
  - Supports only CRUD on remote resources (=Data objects)
  - Supports workflow through hypermedia links

- **Very different programming model required compared to Remote Method Invocation/Broker**
- **Not all systems are suited for REST !**

AARHUS UNIVERSITET

Roy Fielding's work

- *Goal: Keep the **scalable** hypermedia properties of WWW*
- REST = **RE**presentational **S**tate **T**ransfer
  - Transferring a *representation of data* in a format matching one of standard data types (media types)
  - *Resource*: any information that can be named
  - Identified by a *resource identifier*
    - *URI = Uniform Resource Identifier*
  - Interactions are *stateless*
    - Each request contains all the information necessary

Exercise: Why is everybody so keen on 'stateless'? What QA is involved?

AARHUS UNIVERSITET

Representing Resources

AARHUS UNIVERSITET

Example

- Resource: Inger's blood pressure measured on 29/6/2017
- Representation of data using standard media type:
  - { pid: "251248-1234", sys: 120.0, dia:70.0 } (json)
- Resource identifier
  - http://telemed.org/bp/251248-1234/made-29-06-2017-09-59-17
  - I.e. Inger's resource (her blood pressure measurement) is uniquely identified using this URI

AARHUS UNIVERSITET

Example: CRUD

- Inger makes the measurement CREATE
- POST /bp
  - Body: { pid: ”251248-1234”, sys: 120.0, dia:70.0 }
- Response
  - StatusCode: 201 CREATED
  - Location: /bp/251248-1234/made-29-06-2017-09-59-17
  - Body: { pid: ”251248-1234”, sys: 120.0, dia:70.0, status: ”new” }

- Meaning
  - The resources was created, has resource id
    - /bp/251248-1234/made-29-06-2017-09-59-17

AARHUS UNIVERSITET

Example: CRUD

- Inger reviews the measurement READ
- GET /bp/251248-1234/made-29-06-2017-09-59-17
  - Body: (none)
- Response
  - StatusCode: 200 OK
  - Body: { pid: ”251248-1234”, sys: 120.0, dia:70.0, status=”new” }

- Meaning
  - The resources was found, and the measurement returned

AARHUS UNIVERSITET

Example: CRUD

- Inger updates the measurement UPDATE
- PUT /bp/251248-1234/made-29-06-2017-09-59-17
  - Body: { pid: ”251248-1234”, sys: 126.0, dia:69.0 }
- Response
  - StatusCode: 201 CREATED
  - Body: { pid: ”251248-1234”, sys: 126.0, dia:69.0, status=”revised” }

- Meaning
  - The resources was found, and the measurement updated

AARHUS UNIVERSITET

Example: CRUD

- Inger deletes the measurement DELETE
- DELETE /bp/251248-1234/made-29-06-2017-09-59-17
  - Body: (none)
- Response
  - StatusCode: 204 No Content
  - Body: none

- Meaning
  - The resources was found, and the measurement deleted

AARHUS UNIVERSITET

Prototype: pastebin

- REST is pretty lightweight programming wise…
  - Goal: AP to demonstrate ”pastebin”
    - Online service for storing text messages = ‘post-its’
  - Total time: 1.5 hour (well – a bit cheating)

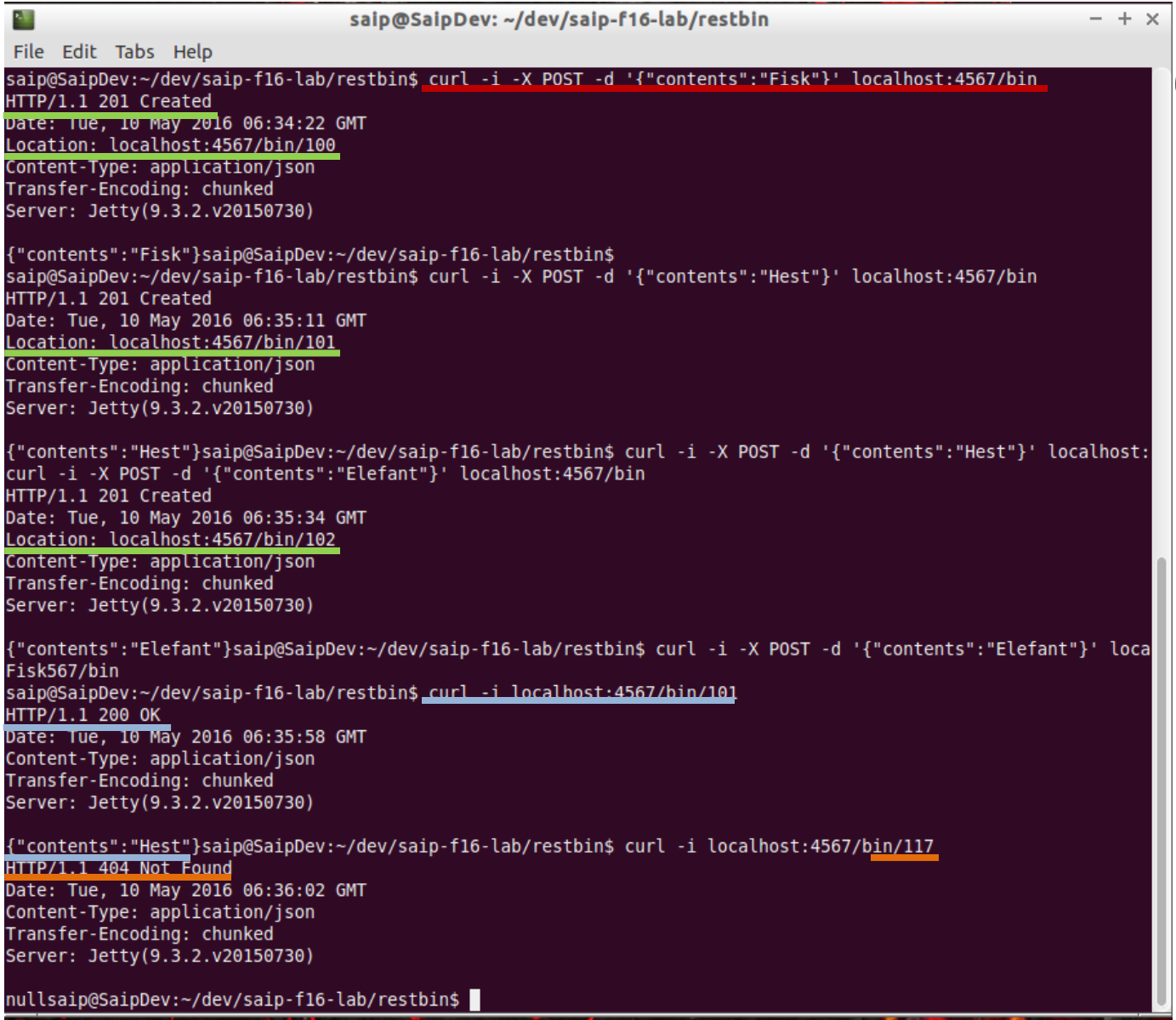
- Developed
  - Webserver, accepting POST and GET
    - Using Spark-java framework (IPC) and GSON (Marshaling)
  - Client: curl or httpie ☺

Demo

```
csdev@m51:~$ http POST localhost:4567/bin contents=fisk
HTTP/1.1 201 Created
Content-Type: application/json
Date: Wed, 20 Nov 2019 13:20:05 GMT
Location: localhost:4567/bin/100
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "contents": "fisk"
}

csdev@m51:~$ http POST localhost:4567/bin contents=hest
HTTP/1.1 201 Created
Content-Type: application/json
Date: Wed, 20 Nov 2019 13:20:20 GMT
Location: localhost:4567/bin/101
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "contents": "hest"
}
```

CREATE fisk and hest
READ 100, 101, 102

```
csdev@m51:~$ http localhost:4567/bin/101
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 20 Nov 2019 13:21:14 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "contents": "hest"
}

csdev@m51:~$ http localhost:4567/bin/100
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 20 Nov 2019 13:21:17 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "contents": "fisk"
}

csdev@m51:~$ http localhost:4567/bin/102
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Wed, 20 Nov 2019 13:21:19 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

null
```



Or Curl…

- POST ‘Fisk’, ‘Hest’ and ‘Elefant’ in bins
- Assigned bin 100, 101, 102

- GET bin 101
- Which is ‘Hest’

- GET bin 117
- Which is not found (404)

Note

- POST of course needs to tell client the *resource identifier* of the newly created object!
  - Response contains a ‘Location’ field
    - Standard way for POST communicate ‘resource id’

```
public Server() {
  /**
   * POST /bin. Create a new bin, if success, receive a Location header
   * specifying the bin's resource identifier.
   *
   * Parameter: req.body must be JSON such as {"contents":
   * "Suzy's telephone no is 1234"}
   */
  post("/bin", (req, res) -> {
    // Convert from JSON into object format
    Bin q = gson.fromJson(req.body(), Bin.class);

    // Create a new resource ID
    String idAsString = ""+id++;

    // Store bin in the database
    db.put(idAsString, q);

    // 201 Created
    res.status(HttpServletResponse.SC_CREATED);

    // Location = URL of created resource
    res.header("Location", req.host()+"/bin/"+idAsString);

    // Return the constructed bin
    return q;
  }, json());

  /**
   * GET /bin/<id>. Get the bin with the given id
   */
  get("/bin/:id", (req, res) -> {
    // Extract the bin id from the request
    String id = req.params(":id");

    // Lookup, and return if found
    Bin bin = db.get(id);
    if (bin != null) { return bin; }

    // Otherwise, return error
    res.status(HttpServletResponse.SC_NOT_FOUND);

    return null;
  }, json() );

  // Set all response types to JSON
  after((req, res) -> {
    res.type("application/json");
  });
}
```

Server code

- A PasteBin server in 50 lines of Java
  - OK, Spark-java helps quite a bit!

Is in the ‘FRDS.Broker’ codebase, as an isolated project. (You have to change to the pastebin folder to make it work)

Left as an Exercise

- We should be able to *update* a text in pastebin
  - PUT verb
- And delete an entry
  - DELETE verb

AARHUS UNIVERSITET

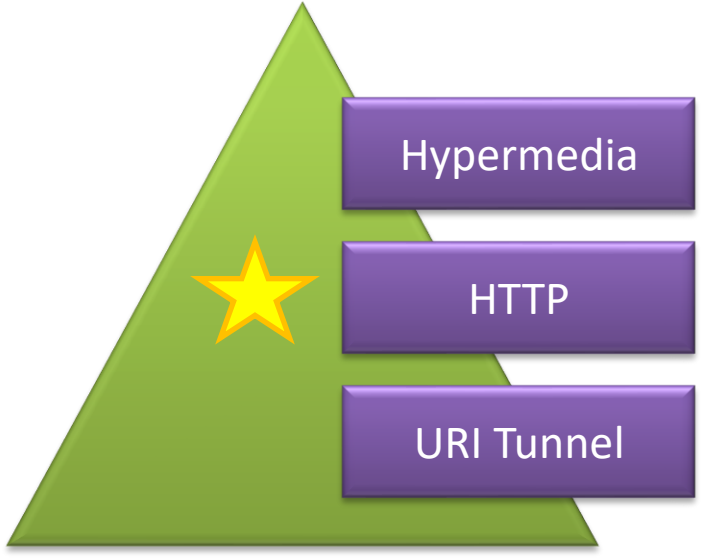
Discussion

- REST uses the **HTTP as designed**
  - CRUD verbs and Status Codes (methods, return type)
    - Virtually allows all *Information Systems* operations !
  - URLs as resource identifiers (location+object)
    - Always identify the *same* resource, and representation of state is *always communicated*
  - Well defined *data representations* (media types)
    - JSON has become favorite (readable + small footprint)

AARHUS UNIVERSITET

Richardson's Maturity model

- From low maturity to high maturity
  - URI Tunnel
    - Just use HTTP as IPC layer
      - SOAP, WSDL, WebServices
      - And our URI Tunnel Broker!
  - HTTP
    - Use CRUD Verbs on resources
  - Hypermedia
    - Use links to define workflows



AARHUS UNIVERSITET

Level 2 REST

AARHUS UNIVERSITET

Workflow

- Business systems can often be modelled as workflows
  - CS term: State machines / state graphs ☺

- Ex: Book a flight
  - I *search* for flights available — get list of links
  - I pick one particular flight — get ‘book’ link
  - I *book* the flight — enter personal details
  - I *pay* for the flight — enter credit card details
  - I *get* a) e-ticket b) receipt — get two links

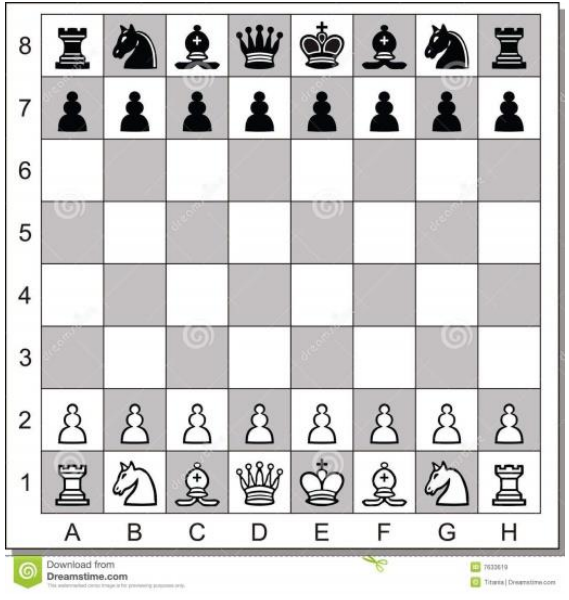
Exercise

- I *search* for flights
  - What HTTP verb is that? What resources are involved?
- I *book* the flight
  - What HTTP verb is that? What resources are involved?
- I *pay* for the flight
  - What HTTP verb is that? What resources are involved?
- I *get* my e-ticket
  - What HTTP verb is that? What resources are involved?

AARHUS UNIVERSITET

Level 2: Hypermedia

- Workflows are not just ‘CRUD a resource’, rather more complex
  - Transactions: Multiple entities atomically updated
  - State transitions: *Mutator* methods that updates several entities and/or updates state
  - Ex: A game’s move(f,t) method
    - Validate move (may return ‘not valid’)
    - Update board state (transaction, e.g. king castling)



Analysis

- ‘move()’ using HTTP verbs ???
  - Is it possible at all?

- Analysis A:
  - ”No, we cannot do that”
    - Because ‘move’ is not a create, it is not a read, nor update, nor delete of a *single* resource (stateless)

AARHUS UNIVERSITET

Analysis

- ‘move()’ using HTTP verbs

- Analysis B: *Maybe it is an update of game*
  - PUT /game/47
  - Body: Full board state with the move executed
    - But – then the server has to *infer* the move from the *delta between state ‘before’ and state ‘after’* which is weird!
      - And it is definitely not **stateless** – right?

AARHUS UNIVERSITET

Analysis

- Analysis C: *A 'state transition resource'*
  - *Creating a game, is creation of **two** resources*
    - The game resource /game/47/
    - The **move** resource /game/47/move or /game/move/47
  - PUT /game/47/move
  - Body: { from: e2, to: e4, player:white}
- This will
  - Try to UPDATE the state => 200 OK or 401 Invalid
  - If 200 OK, then the game resource is *also* updated
    - And can be successively GET to see new board state

Challenge

- But how do we return **two** resources from the game create POST message?
  - We can not, but we can use the WWW way – provide hypermedia links!!!

```
{
    playerOne: Pedersen,
    playerTwo: Findus,
    boardState: [ ... ],
    playerInTurn: Pedersen
    next: /lobby/game/move/{game-id}
}
```

Aka

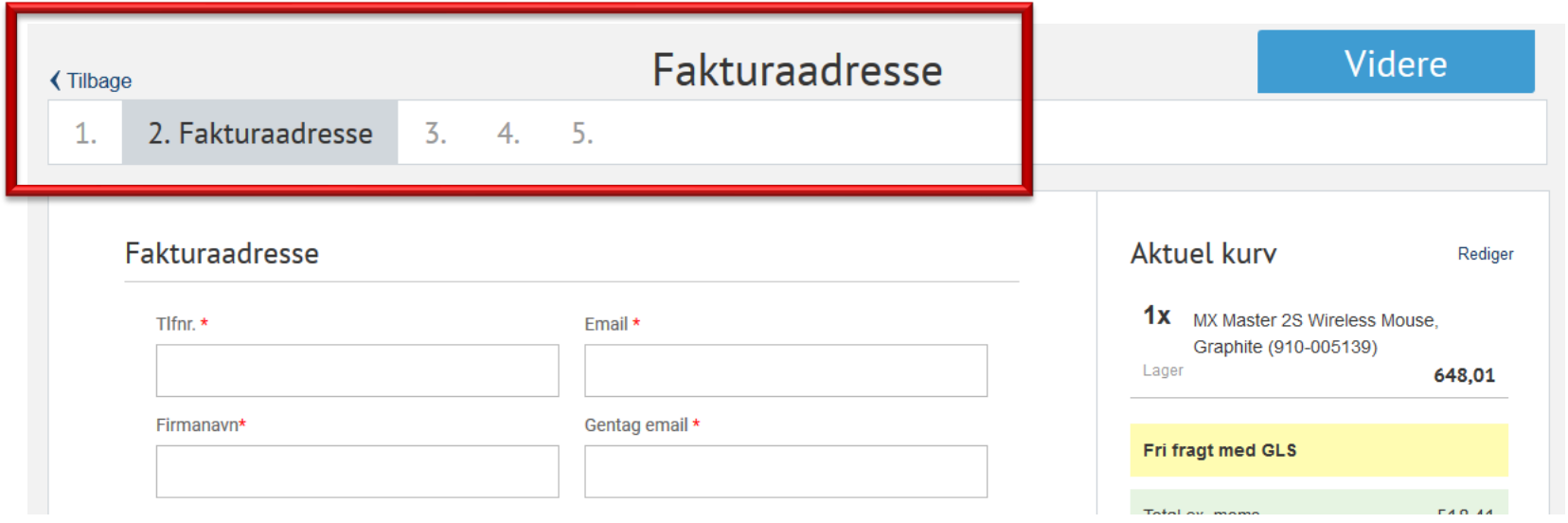
- HATEOAS:
  - *Hypermedia As The Engine Of Application State.*

- Application state changes are modelled as hypermedia links, each to a resource that objectify the change itself, not the old/new state of underlying objects
  - A ‘move’ resource, a ‘payment’ resource, a ‘send items to address’ resource, etc.

AARHUS UNIVERSITET

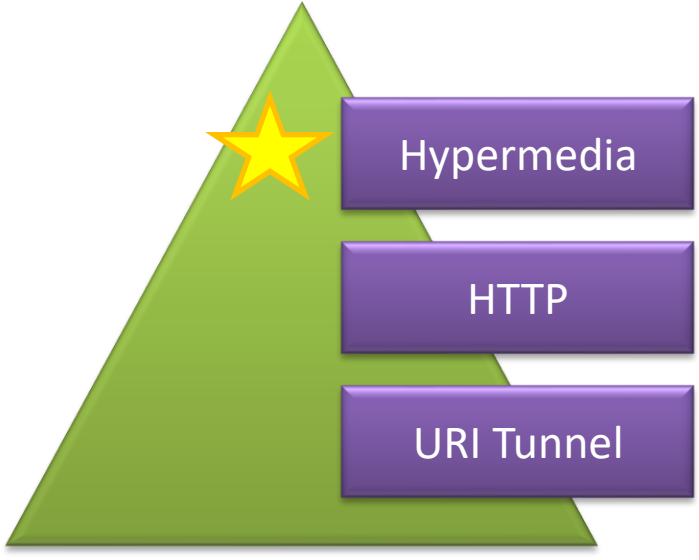
Often visible in UI

- The state changes of the *order*



Level 2: Hypermedia

- So – REST is a radically different architectural pattern/style, different from OO and interface-based paradigms
- POST to create a resource
  - May return several hypermedia links that define valid state transitions for the resource
    - Which are then manipulated through the HTTP verbs
  - Makes potential state transitions *discoverable*
    - Just like any new web page presents links that I may follow



Example 1

Strong inspiration from:

”How to GET a cup of Coffee”

By Webber et al.

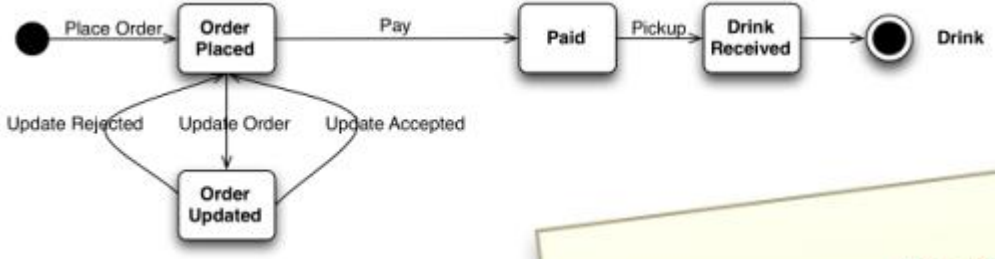
Context

- Webber et al.'s paper outline the full Hypermedia approach for building REST based systems

- We will take an alternative/simpler route
  - We will keep using JSON, instead of XML
  - We will encode the statemachine *in the code base* instead of coding it like links in the XML ('next' in webber's paper)
    - They need to code logic to interpret 'next' tag anyway so our binding is not that much harder than what Webber presents.

Coffee Shop

- A web shop for ordering coffee – and paying…



Story 1:

As a customer, I want to order a coffee so that Starbucks can prepare my drink

Example: Story 1 (Coffee order)

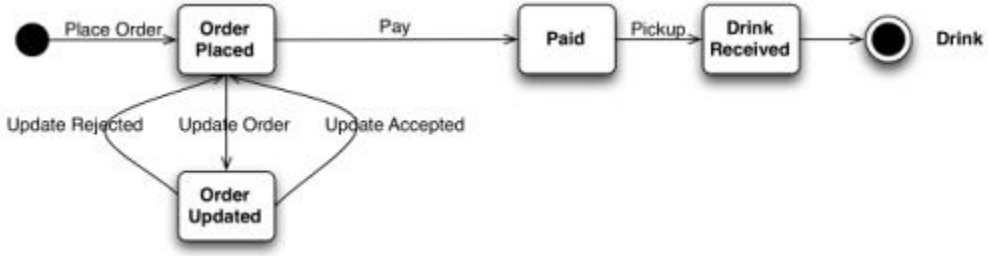
```
csdev@m1:~/proj/frsproject/coffeeshop$ http POST localhost:4567/order drink=latte
HTTP/1.1 201 Created
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:43:32 GMT
Location: localhost:4567/order/100
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "cost": "3.00",
    "drink": "latte",
    "paid": false
}
```

POST on /order

```
csdev@m1:~/proj/frsproject/coffeeshop$ http GET localhost:4567/order/100
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:45:11 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "cost": "3.00",
    "drink": "latte",
    "paid": false
}
```

GET on /order/{id}

Coffee Shop




```
csdev@m1:~/proj/frsproject/coffeeshop$ http GET
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:45:11 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "cost": "3.00",
    "drink": "latte",
    "paid": false
}
```

Story 3:

As a customer, I want to be able to pay my bill to receive my drink

Example: Story 3 (Coffee Payment)

AARHUS UNIVERSITET

- Another object /payment/order/{id} is **also** created
- Payment becomes **updating this object!**

```
csdev@m1:~/proj/frsproject/coffeeshop$ http PUT localhost:4567/payment/order/100 cardno=1234 amount=3.00
HTTP/1.1 201 Created
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:49:18 GMT
Location: localhost:4567/payment/order/100
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "amount": "3.00",
    "cardno": "1234"
}
```

PUT on /payment/order/{id}

- And a new get shows the *state change* of the order

```
csdev@m1:~/proj/frsproject/coffeeshop$ http GET localhost:4567/order/100
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:50:36 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "cost": "3.00",
    "drink": "latte",
    "paid": true
}
```

Example: Story 3 (Coffee Payment)

- In Webber et al.'s paper, the XML will provide the payment resource id as 'next' tags
  - The hypermedia approach:
    - Provide the client with multiple options to move to new info/actions through providing links

- I just 'agreed' on the resource path in the code base…

```
csdev@m1:~/proj/frsproject/coffeeshop$ http PUT localhost:4567/payment/order/100 cardno=1234 amount=3.00
HTTP/1.1 201 Created
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:49:18 GMT
Location: localhost:4567/payment/order/100
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
    "amount": "3.00",
    "cardno": "1234"
}
```

PUT on /payment/order

Example 2

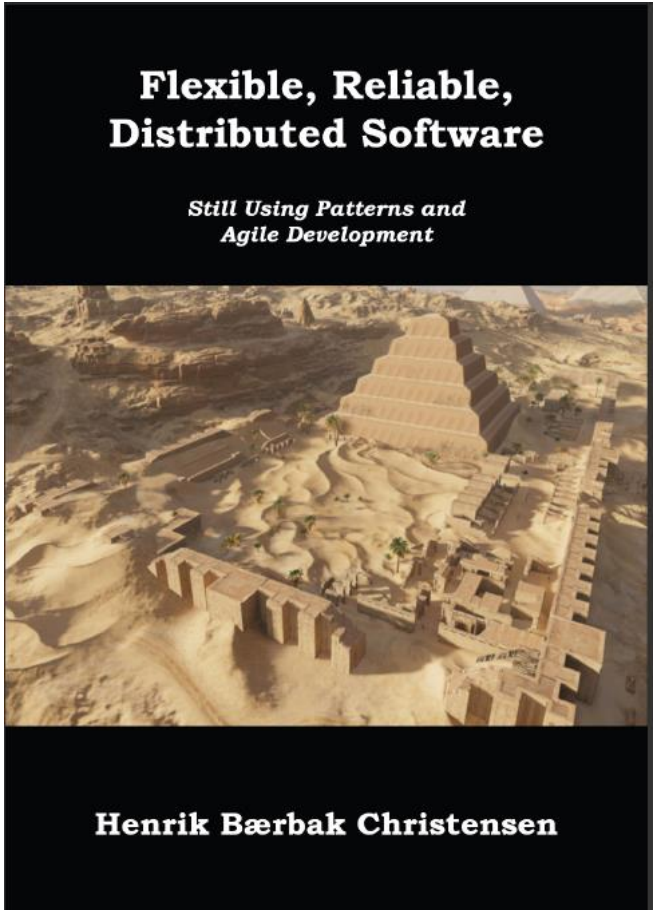
AARHUS UNIVERSITET

GameLobby

- Joining a Game result in a game ressource to be CRUD'ed

```
1  Join A Game
2  -----------
3
4  PUT /lobby/{future-game-id}
5
6  {
7    playerTwo: Findus
8  }
9
10 Response
11   Status: 200 OK
12
13   {
14       playerOne: Pedersen,
15       playerTwo: Findus,
16       level: 0,
17       available: true,
18       next: /lobby/game/{game-id}
19   }
20
21   Status: 404 Not Found
22   (none)
```

Read the book ☺



AARHUS UNIVERSITET

REST versus Broker

Pros And Cons

AARHUS UNIVERSITET

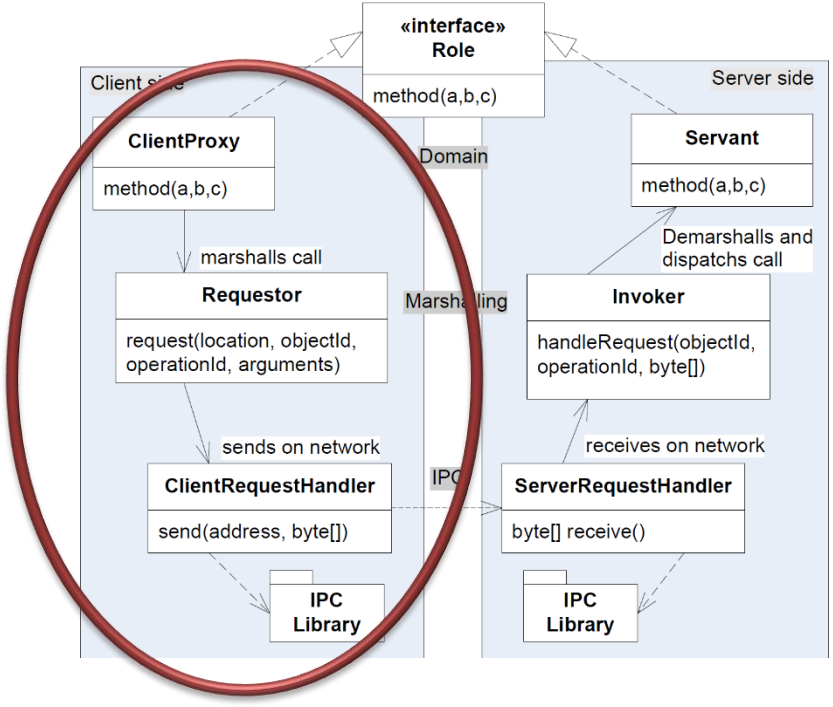
Rest or Broker

- REST is the better choice because
  - In 2020 it is much more widely used than Broker architectures
  - REST promises scalability, performance, reliability
  - Lighter and direct programming model (contrast SOAP/WSDL)
  - Direct interaction (manual test) via ‘http’ or ‘curl’
- REST is the lesser choice because
  - Programming model is at low abstraction level
  - All responsibilities are mixed together = Blob antipattern
  - HATEAOS even mixes UI responsibilities into domain ☹ ☹ ☹
    - ‘links’ are part of the domain object
      - Not just a coffee order but coffee order + URL links to state changes

Mixed Responsibilities

- Broker separate distinct responsibilities
  - Domain layer, marshalling layer, IPC layer

- REST actually addresses responsibilities on both the Marshalling, Location, and IPC level in one big ball of mud
  - *Low cohesion* ☹



Mixing UI State and Domain

- From Webber's CoffeeShop
- Domain object
  - An Order
- UI/State object
  - Links that are URL encoded mixed into the 'ball of mud'
- Demarshalling becomes tedious

```
200 OK
Location: http://starbucks.example.com/order/1234
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <additions>shot</additions>
  <cost>4.00</cost>
  <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

Both pull the same way ☹

- Spaghetti code // Big ball of mud
  - Look for Brian Foote and Joseph Yoder's paper

Ex: TeleMedRESTProxy

```
public class TeleMedRESTProxy implements TeleMed {

  private String baseURL;
  private Gson gson;

  public TeleMedRESTProxy(String hostname, int port) {
    baseURL = "http://"+hostname+":"+port+"/";
    gson = new Gson();
  }

  @Override
  public String processAndStore(TeleObservation teleObs) {
    String payload = gson.toJson(teleObs);
    HttpResponse<JsonNode> jsonResponse = null;

    String path = Constants.BLOODPRESSURE_PATH;
    try {
      jsonResponse = Unirest.post(baseURL+path).
          header("accept", Constants.APPLICATION_JSON).
          header("Content-type", Constants.APPLICATION_JSON).
          body(payload).asJson();
    } catch (UnirestException e) {
      throw new IPCException("UniRest POST failed for 'processAndStore'", e);
    }

    // TODO: Verify returned status code
    int statusCode = jsonResponse.getStatus();

    // String body = jsonResponse.getBody().toString();

    // Extract the id of the measurement from the Location header
    String location = jsonResponse.getHeaders().getFirst("Location");
    // Format: URI ending in /bp/{id}, thus let us split on '/'
    // and pick the last entry
    String parts[] = location.split("/");
    String teleObsID = parts[parts.length-1];

    return teleObsID;
  }
```

- Using a good REST library (here: uni-rest), the code is small …
- This Proxy plays all roles
  - Proxy
  - Requestor (GSON)
  - ClientReqH. (UniRest)
- Location field read to retrieve ID of resource

AARHUS UNIVERSITET

Restating my Claim

- Broker Pattern is as strong as REST iff
  - You simply obey the same fundamental architectural constraints as REST impose
    - Only pass-by-value
      - Use the ‘pass objectId’ technique for server -> client ‘pass by reference’
    - Pure client-server - no server calling methods on clients
  - Design your Remote Roles with distribution in mind
    - Not ‘game.getUnitAt(p)’ Chatty interface
    - But ‘game.getFullGameState()’ Chunky interface

AARHUS UNIVERSITET

Summary

- UR Tunnelling
  - Just uses HTTP and web technology/frameworks as the IPC layer in the Broker
    - That is : transport network packages to/from client and server

- REST
  - Architectural Pattern what deeply exploits HTTPs advantages
  - Lightweight with less tool support
  - Focus is on performance and scalability because
    - True Client-server No callback/observer pattern
    - Value passing of information

Summary

- Broker pattern and REST?
  - Only if the OO interfaces/roles are designed so they adhere to the REST way of architecture
    - CRUD on ‘objects’ = resources
    - State transitions modelled as ‘transition resources’
      - Bit similar to Command pattern objects…
  - … and generally you do not design OO that way…

- **REST and OO are two different architectural styles…**