



AARHUS UNIVERSITET

Software Architecture in Practice

REpresentational **S**tate **T**ransfer



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



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**
- **Not all architectures are suited for REST !**



Question

- How many of you work/code using REST on a daily basis?

Roy Fielding's work

- *Keep the scalable hypermedia properties of www*
- REST
 - 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)*
 - Interactions are *stateless*
 - Each request contains all the information necessary

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

- Data: Inger's blood pressure
- Representation of data:
 - { pid: "251248-1234", sys: 120.0, dia:70.0 } (json)
- Resource identifier
 - <http://telemed.org/251248-1234/ae23f1879ef6>

Location of server

ID of measurement

ID of patient



AARHUS UNIVERSITET

HTTP

The Protocol driving the WWW

The 'scalable hypermedia properties
that Fielding tells us to value



Message Format

Text format !

HTTP version

```
GET /contact.html HTTP/1.1
Host: www.baerbak.com
Accept: text/html
```

```
HTTP/1.1 200 OK
Date: Mon, 19 Jun 2017 09:58:25 GMT
Server: Apache/2.2.17 (FreeBSD) mod_ssl/2.2.17 OpenSSL/1.0.0c ...
Last-Modified: Mon, 13 Apr 2015 12:34:07 GMT
ETag: "b46bce-676-5139a547e2dc0"
Accept-Ranges: bytes
Content-Length: 1654
Vary: Accept-Encoding,User-Agent
Content-Type: text/html

<html>
  <head>
    <title>Flexible, Reliable Software</title>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    <link href="style.css" rel="stylesheet" type="text/css">
```

- Request
 - Verb resource
 - Header key-values
 - Empty line
 - (Message body)
- Reply line
 - Status line
 - HTTP codes
 - Header fields
 - Empty line
 - (Message body)



- HTTP Verbs
 - GET
 - request representation of a resource (URI)
 - POST
 - accept enclosed entity as new subordinate of resource (URI)
 - PUT
 - request enclosed entity to be stored under URI
 - DELETE
 - request deletion of resource (URI)

- The four verbs actually match standard database operations: **CRUD**
 - Create
 - = post
 - Read
 - = get
 - Update
 - = put
 - Delete
 - 😊

- HTTP Status codes
 - Well defined vocabulary of error conditions

- 200 = OK
- 201 = Created
- 400 = Bad request
- 404 = Not found

- etc

400 Bad Request

The server cannot or will not process the request due to an apparent deceptive request routing.^[33]

401 Unauthorized (RFC 7235^[34])

Similar to *403 Forbidden*, but specifically for use when authentication is required and has failed or not supplied. The WWW-Authenticate header field containing a challenge applicable to the requested resource. The semantic means "unauthenticated",^[35] i.e. the user does not have the appropriate credentials. Note: Some sites incorrectly issue HTTP 401 when an IP address is required.

402 Payment Required

Reserved for future use. The original intention was that this code would be used when a payment gateway or other financial transaction took place. It has not happened, and this code is not usually used. *Google Developers* / *Google Cloud* uses this code when a user does not have sufficient funds to start a call.^[38] *Shopify* uses this code when the user is not logged in.

403 Forbidden

The request was valid, but the server is refusing action. The user or resource does not have the right to perform the operation.

404 Not Found

The requested resource could not be found but may be available in the future.

405 Method Not Allowed

A request method is not supported for the requested resource; for example, a GET request on a form that only supports POST.

406 Not Acceptable

The requested resource is capable of generating only content not acceptable by the user agent.

407 Proxy Authentication Required (RFC 7235^[34])



REST Level 0

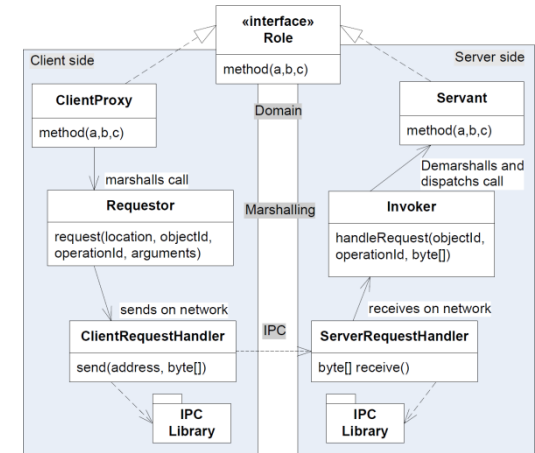
URI Tunneling

URI Tunneling

- *Use HTTP as raw IPC layer*
 - Advanced form of socket connection
 - Only uses POST on a fixed /path

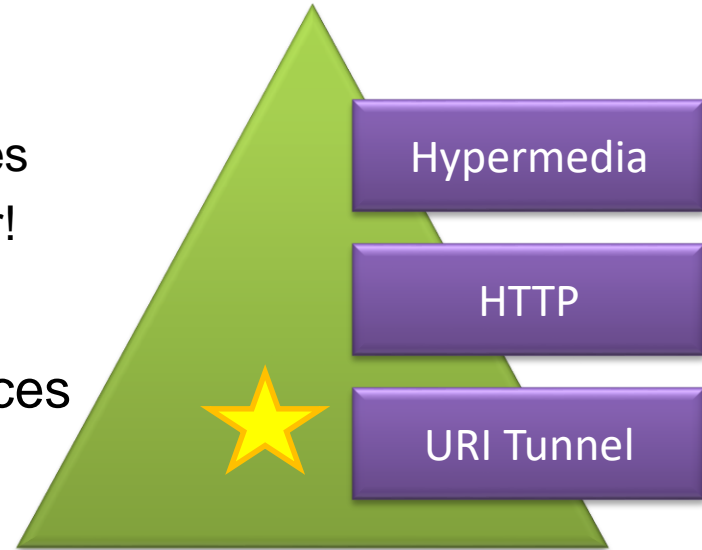
- Examples

- Our FRDS.Broker using the HTTP CRH and SRH
 - ‘gradle serverHttp’ and ‘gradle homeHttp’ uses URI tunnel
- SOAP + WSDL



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





REST Level 1

Proper HTTP Usage



- So – we have
 - **Resources** (= named information) with
 - an **identifier** (= URI)
 - represented in a **standard format** (JSON or XML or...)
 - that can be **CRUD'ed** (POST, GET, ...)
 - and we can see the outcome (response, status code)



Level 1: HTTP Verbs

- Thus Level 1 can model many simple *information systems*
 - Akin a database 😊
- Example: PasteBin
 - An internet clipboard for cut/paste of text strings...
 - Ala: <http://pastebin.com>

AP: pastebin

- REST is way more lightweight than SOAP
 - 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

AARHUS UNIVERSITET

CREATE fisk and hest
READ 100, 101, 102

```
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"
}
```

```
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
```

```
saip@SaipDev: ~/dev/saip-f16-lab/restbin
File Edit Tabs Help
saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i -X POST -d '{"contents":"Fisk"}' localhost:4567/bin
HTTP/1.1 201 Created
Date: Tue, 10 May 2016 06:34:22 GMT
Location: localhost:4567/bin/100
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.2.v20150730)

{"contents":"Fisk"}saip@SaipDev:~/dev/saip-f16-lab/restbin$
saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i -X POST -d '{"contents":"Hest"}' localhost:4567/bin
HTTP/1.1 201 Created
Date: Tue, 10 May 2016 06:35:11 GMT
Location: localhost:4567/bin/101
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.2.v20150730)

{"contents":"Hest"}saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i -X POST -d '{"contents":"Hest"}' localhost:
curl -i -X POST -d '{"contents":"Elefant"}' localhost:4567/bin
HTTP/1.1 201 Created
Date: Tue, 10 May 2016 06:35:34 GMT
Location: localhost:4567/bin/102
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.2.v20150730)

{"contents":"Elefant"}saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i -X POST -d '{"contents":"Elefant"}' loca
Fisk567/bin
saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i localhost:4567/bin/101
HTTP/1.1 200 OK
Date: Tue, 10 May 2016 06:35:58 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.2.v20150730)

{"contents":"Hest"}saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i localhost:4567/bin/117
HTTP/1.1 404 Not Found
Date: Tue, 10 May 2016 06:36:02 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.2.v20150730)

nullsaip@SaipDev:~/dev/saip-f16-lab/restbin$
```

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)

- 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’



```
saip@SaipDev: ~/dev/saip-f16-lab/restbin
File Edit Tabs Help
saip@SaipDev:~/dev/saip-f16-lab/restbin$ curl -i -X POST -d '{"contents":"Fisk"}' localhost:4567/bin
HTTP/1.1 201 Created
Date: Tue, 10 May 2016 06:34:22 GMT
Location: localhost:4567/bin/100
Content-type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.3.2.v20150730)

{"contents":"Fisk"}saip@SaipDev:~/dev/saip-f16-lab/restbin$
```

Server code

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

```
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");
});
}
```

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

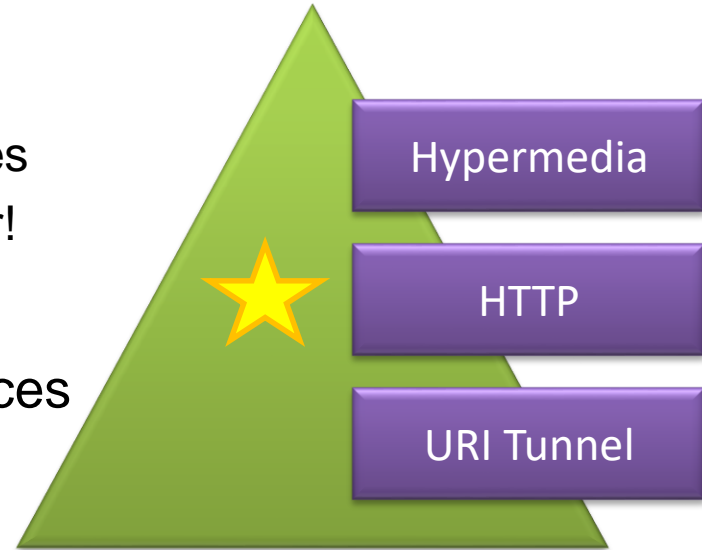
- We should be able to *update* a text in pastebin
 - PUT verb

- And delete an entry
 - DELETE verb

- 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)

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





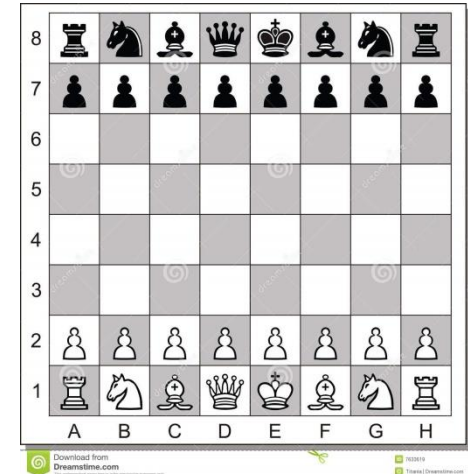
AARHUS UNIVERSITET

REST Level 2

Proper Hypermedia Usage

Level 2: Hypermedia

- However, many (most?) systems must handle more than just CRUD on isolated entities
 - 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)





- 'move()' using HTTP verbs
- Analysis A:
 - "No can do 😞"
 - Because 'move' is not a create, it is not a read, nor update, nor delete of a single resource (stateless)



- '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?

- 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 updated
 - And can be successively GET to see new board state

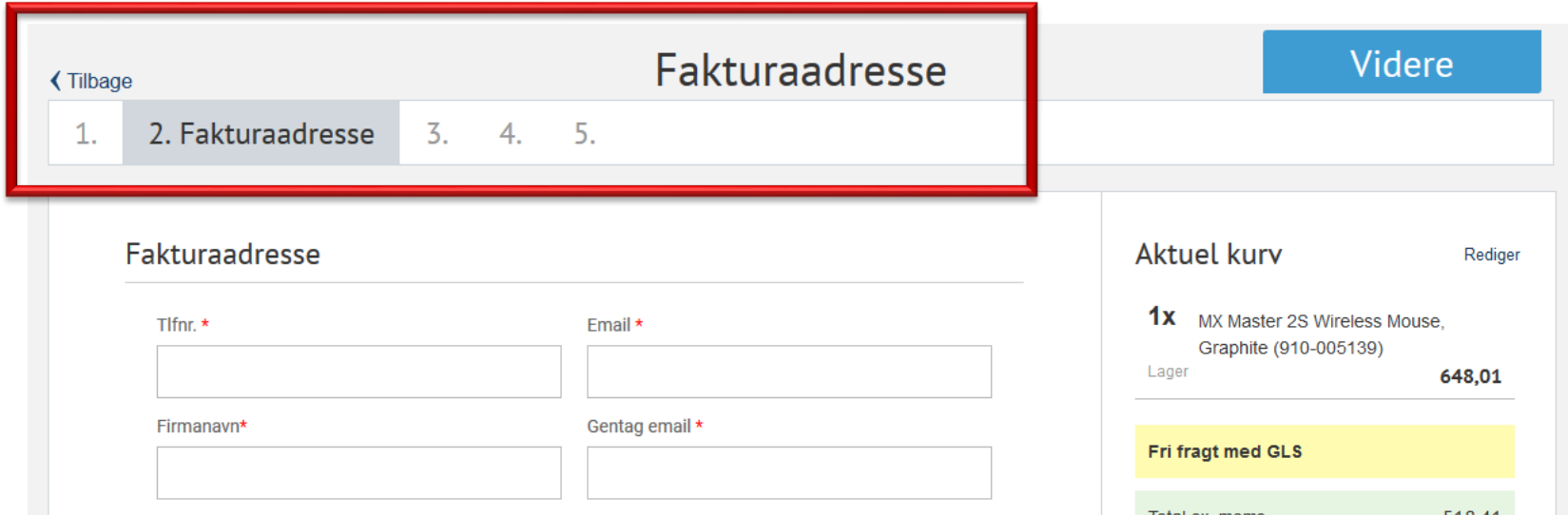
- But how do we return **two** resources from the game create POST message?
 - We can not, but we can use the WWW way – provide hypermedia links!!!

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

- 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.

Oftentimes visible in UI

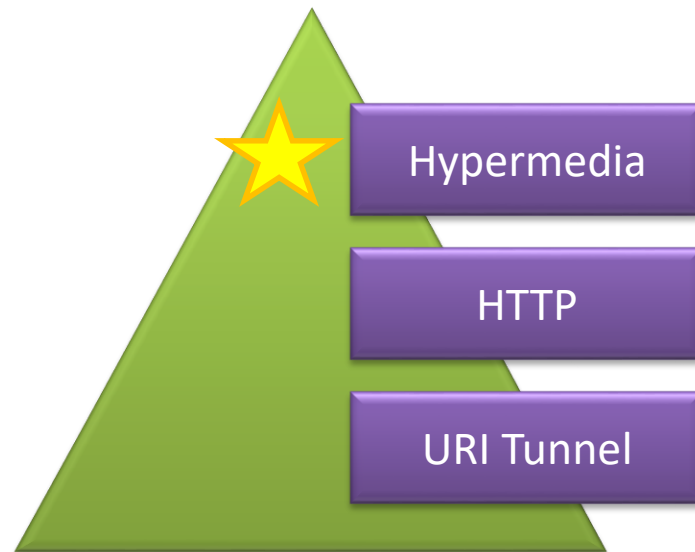
- The state changes of the *order*



The screenshot shows a checkout page with a progress bar at the top. The progress bar has five steps: 1. (selected), 2. Fakturaadresse, 3., 4., and 5. The 'Fakturaadresse' step is highlighted with a red box. To the right of the progress bar is a blue button labeled 'Videre'. Below the progress bar is a form titled 'Fakturaadresse' with four input fields: 'Tlfnr. *', 'Email *', 'Firmanavn*', and 'Gentag email *'. To the right of the form is a section titled 'Aktuel kurv' with a 'Rediger' link. The 'Aktuel kurv' section shows '1x MX Master 2S Wireless Mouse, Graphite (910-005139)' with a price of '648,01'. Below this is a yellow box with the text 'Fri fragt med GLS' and a green box with the text 'Totalt pris' and a price of '519,44'.

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.

- 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.

- 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

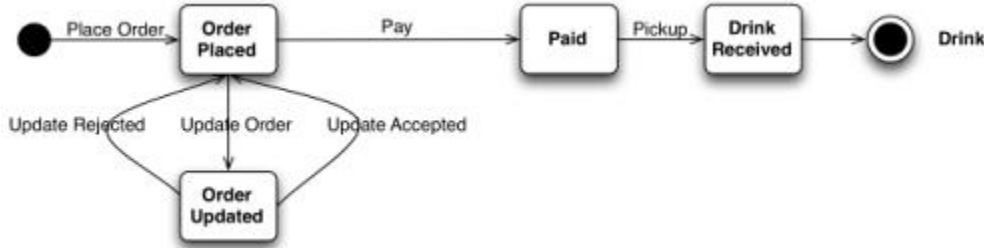
```
csdev@m1:~/proj/frsproject/coffeeshop$ http GET localhost:4567/order/100
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 25 Nov 2020 07:45:11 GMT
Server: Jetty(9.4.6.v20170531)
Transfer-Encoding: chunked

{
  "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)

- 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



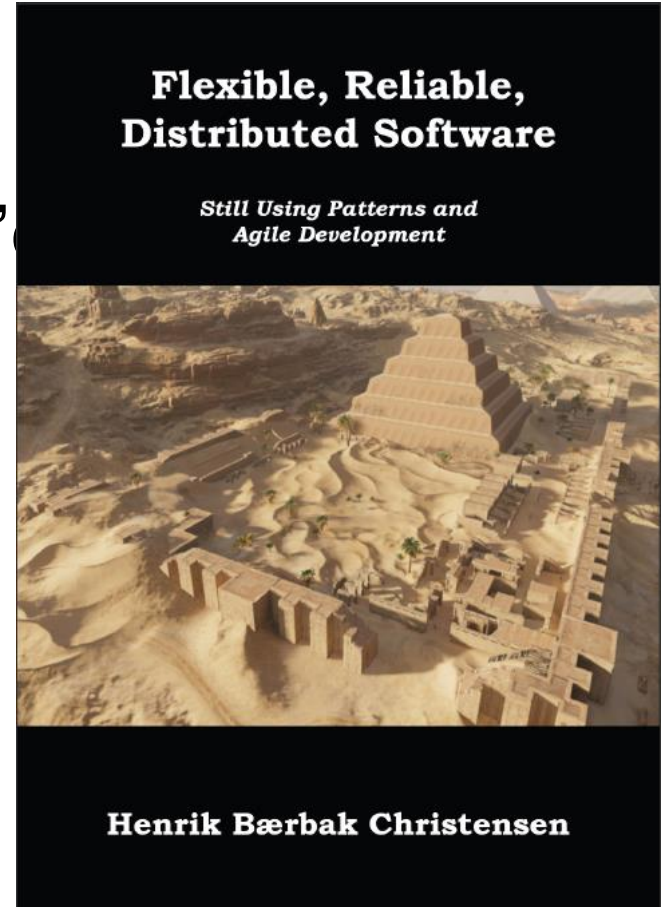
AARHUS UNIVERSITET

Example 2

- Joining a Game result in a game resource to be CRUD'

```
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

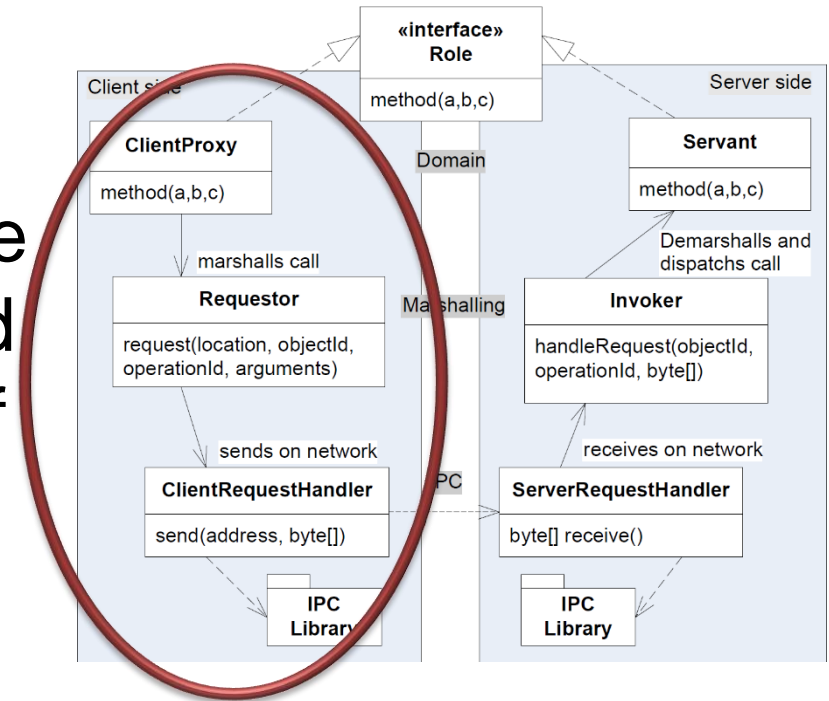
Pros And Cons

Rest or Broker

- REST is the better choice because
 - Today 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, just ‘conventions’
 - Rest often ‘overfetch’ or ‘underfetch’ data
 - 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

```
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>
```

- Domain object

- An Order

- UI/State object

- Links that are URL encoded mixed into the 'ball of mud'

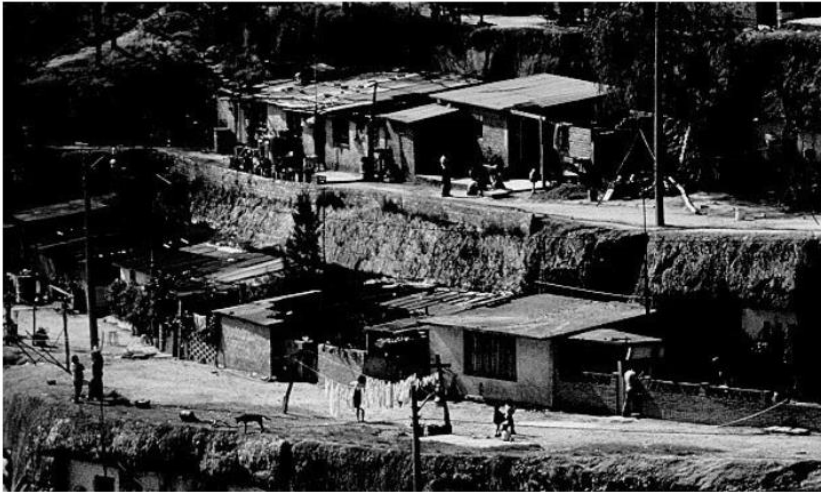
- How, then, separate domain code, application code, UI code, when the 'basic model objects' mix it up badly?

Both pull the same way ☹️

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

BIG BALL OF MUD

alias
SHANTYTOWN
SPAGHETTI CODE





Ex: TeleMedRESTProxy

AARHUS UNIVERSITET

```
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



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
 - Avoid ‘chatty’ interfaces for ‘chunky’ ones
 - ... which we will come back to in the next course



- 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

- 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...**