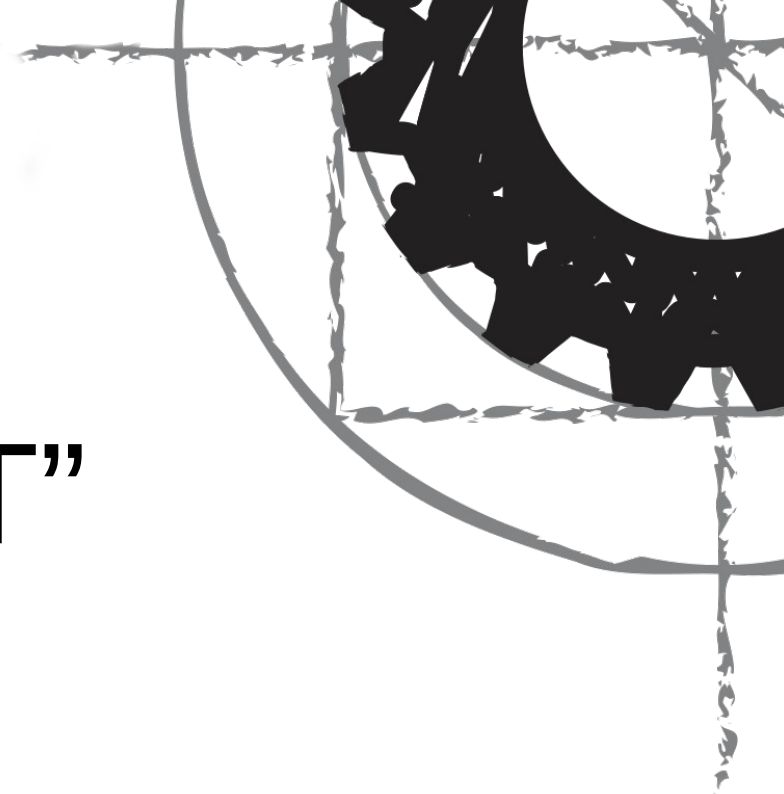


There is REST and  
then there is “REST”



Radovan Semančík  
November 2017

# Who Am I?

**Ing. Radovan Semančík, PhD.**

Software Architect at **Evolveum**

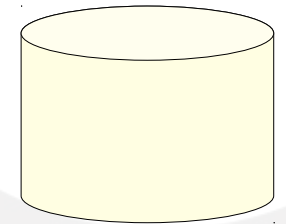
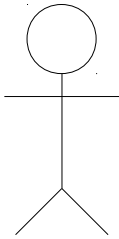
Architect of Evolveum **midPoint**

Apache Foundation committer

Contributor to **ConnId** and **Apache Directory API**

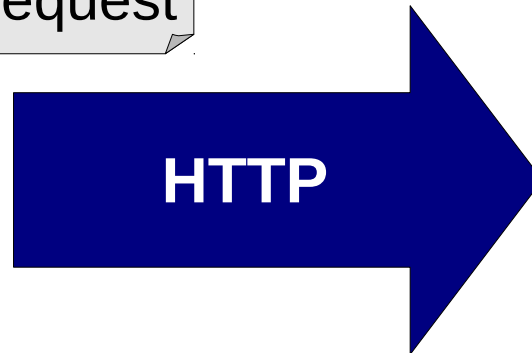


# What is REST?



**Client**

JSON  
request



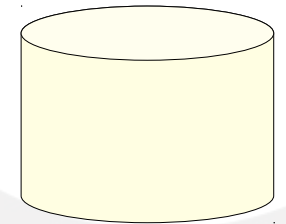
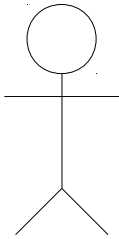
**Server**

JSON  
response

JavaScript in browser  
Mobile application  
Enterprise application  
CLI tool  
Desktop client  
...

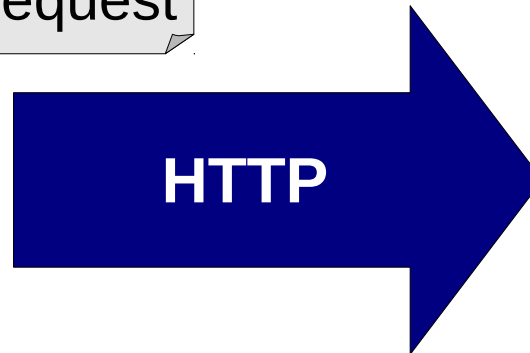
Web application  
Enterprise application  
"Cloud" application  
*Whatever* as a Service  
"Serverless" server  
...

# What is REST?



**Client**

JSON request



**Server**

JSON response

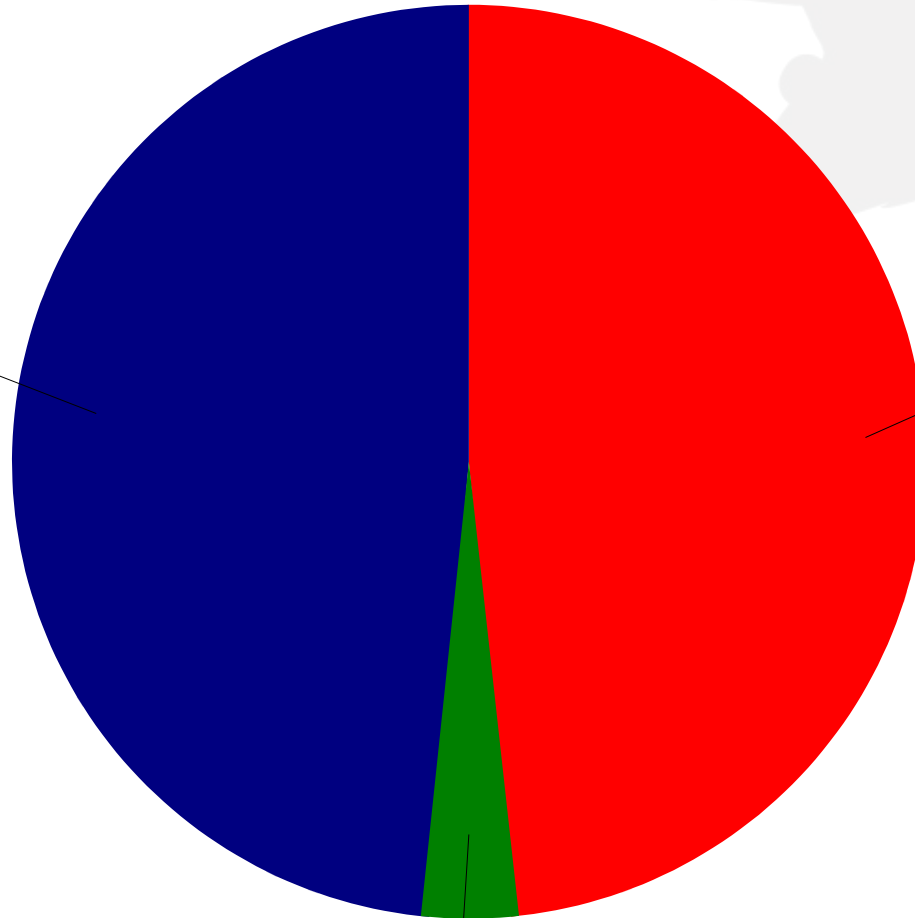
JavaScript in browser  
Mobile application  
Enterprise application  
CLI tool  
Desktop client  
...

Web application  
Enterprise application  
"Cloud" application  
*Whatever* as a Service  
"Serverless" server  
...

This is simple, easy to understand ... and **wrong**

# Who Are You?

I know  
everything  
about  
REST  
already



Whatever.  
I do not need  
to know REST

This may be interesting

# Back to 1990s ...

- 1990: WWW invented
  - Tim Berners-Lee
- 1996: HTTP 1.0
- 1997: HTTP 1.1
- 2000: Representational State Transfer
  - Roy Fielding
- 2002: WWW Architecture

# **RE**presentational **S**tate **T**ransfer

# REST (according to Fielding)

- Architectural style
  - Not “architecture”, not “protocol”, not “API”
- Retro-architecture (HTTP 1.1)
- Transfer of resource representations
  - HTML page is a representation, HTTP is transfer protocol
- Architectural constraints
  - Client-server, Stateless, Cache, Uniform interface, Layered system, Code on demand



# Representational State Transfer

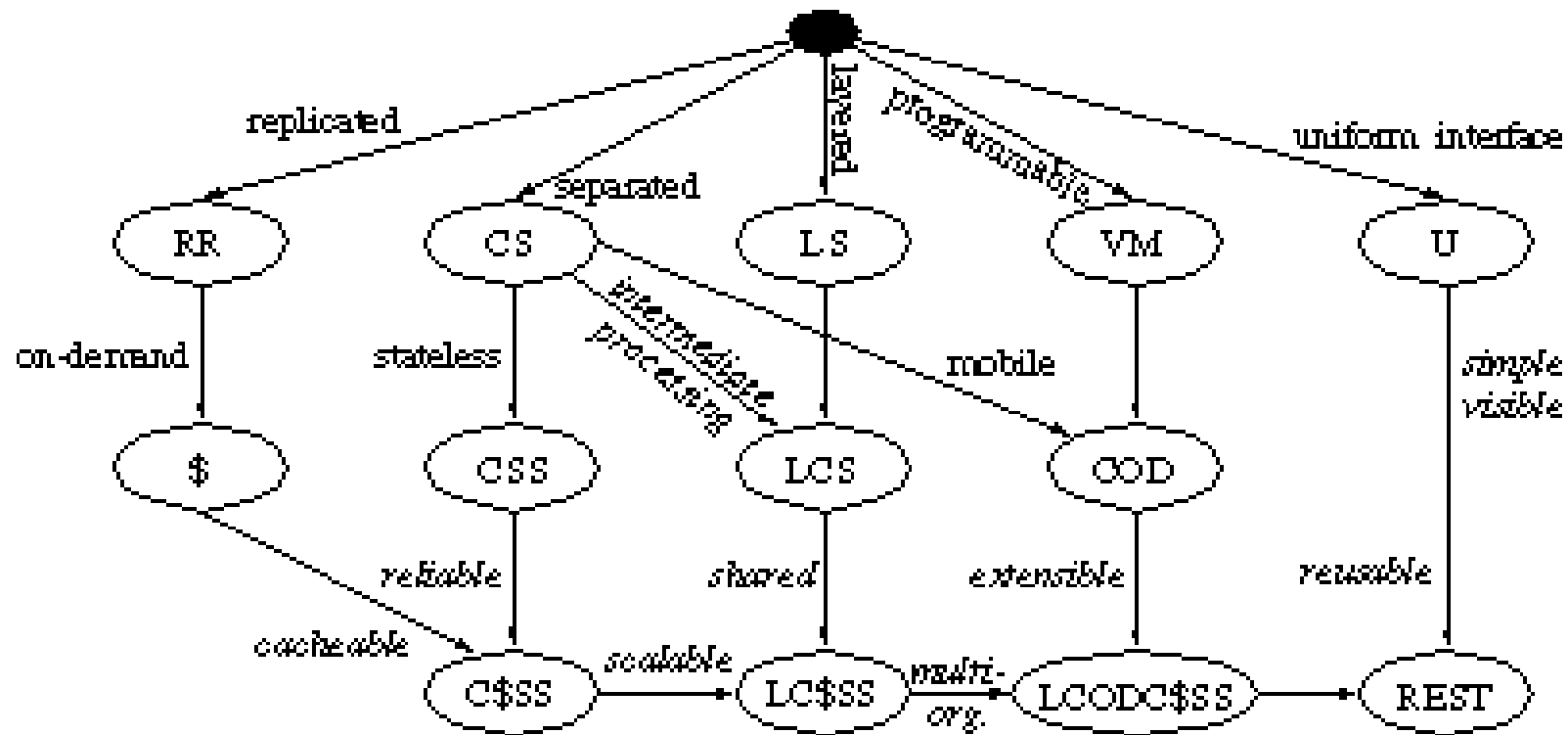
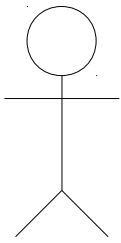


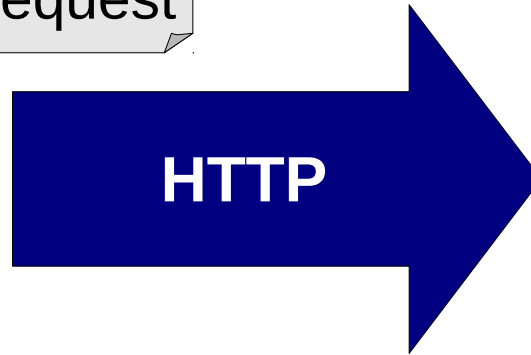
Figure 5-9. REST Derivation by Style Constraints

# So, what exactly is wrong?



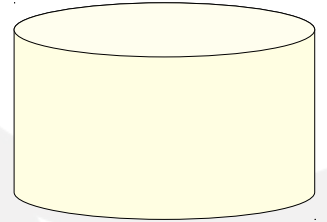
**Client**

JSON  
request

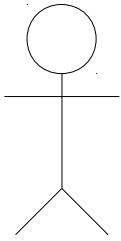


**Server**

JSON  
response

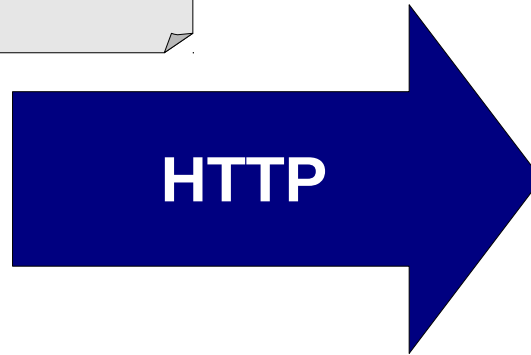


# JSON? Not really ...



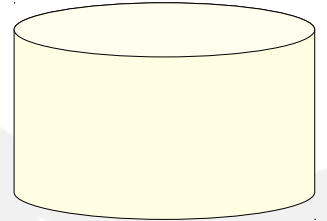
**Client**

request

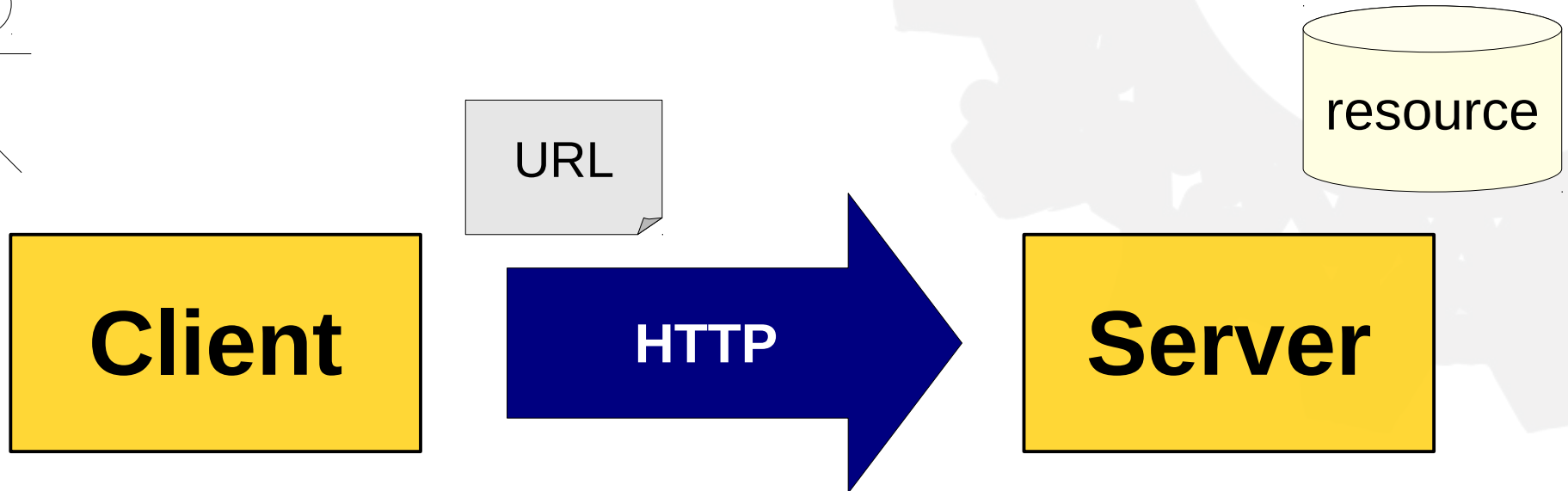
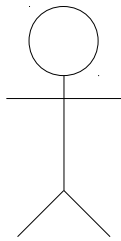


**Server**

response



# Request-response ... kind of ...



Operations (verbs):

- GET
- PUT
- DELETE
- POST
- ...

# REST is not RPC

- Objects (resources) addressed by URL
- Fixed operation set (verbs)
  - GET, HEAD, PUT, DELETE, POST, OPTIONS, TRACE, CONNECT
- Stateless
  - Sessions are evil
- Hypertext
  - Thou shalt not construct your URLs

# Theory and Practice

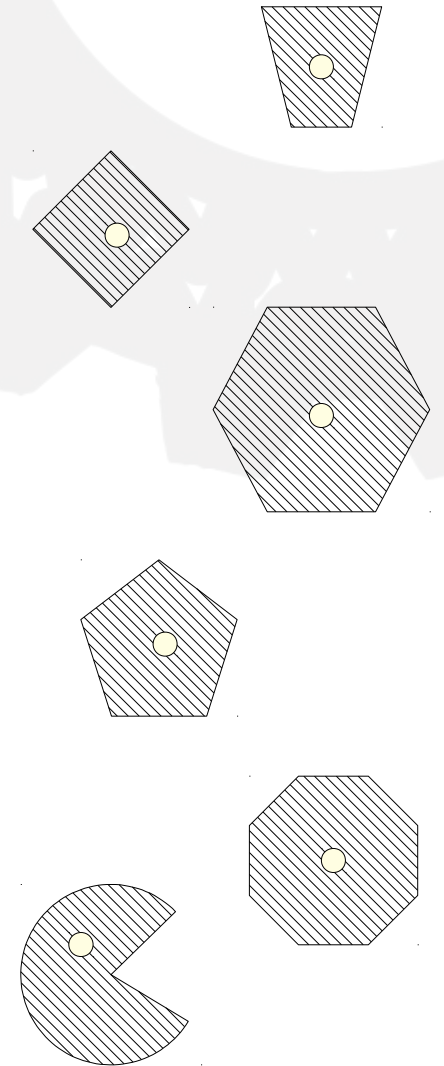
- **Theory:** REST is architectural style for **hypertext** applications
- **Practice:** we really need **RCP**
- REST is not good for RPC  
... but REST is popular, we want it ...
- **Result:** RESTful APIs

# RESTful APIs

- Not really REST
  - Not hypertext
  - Mostly resource-based, but still some RPC
- Not really API
  - Application Programming **Interface**
  - Problematic interface definition (Swagger/OpenAPI?)
  - Security? Reliability? Transactions?
- Read: “HTTP-based service”

# Reinvented Wheel ... again

- 1976: RFC 707
- 1981: Xerox Courier
- 1991: CORBA
- 1993: DCE/RPC → DCOM
- 1995: SunRPC
- 1998: SOAP
- 200x: “RESTful” API





# Pure REST

- It *is* possible to implement pure REST service
  - ... but it is going to be real pain (very likely)
    - Careful design and implementation
    - Unnecessary complexity
    - Too many network round-trips
    - Expect poor performance
- Hypertext is not suitable for everything

# What do we do?

- Keep the parts that fit
  - Resources, URLs, representations
  - GET, DELETE, POST, ...
- Ditch the parts that do not fit
  - Hypertext
- Compromises
  - Statelessness, caching, security, consistency, ...
- Interface definition
  - Swagger/OpenAPI

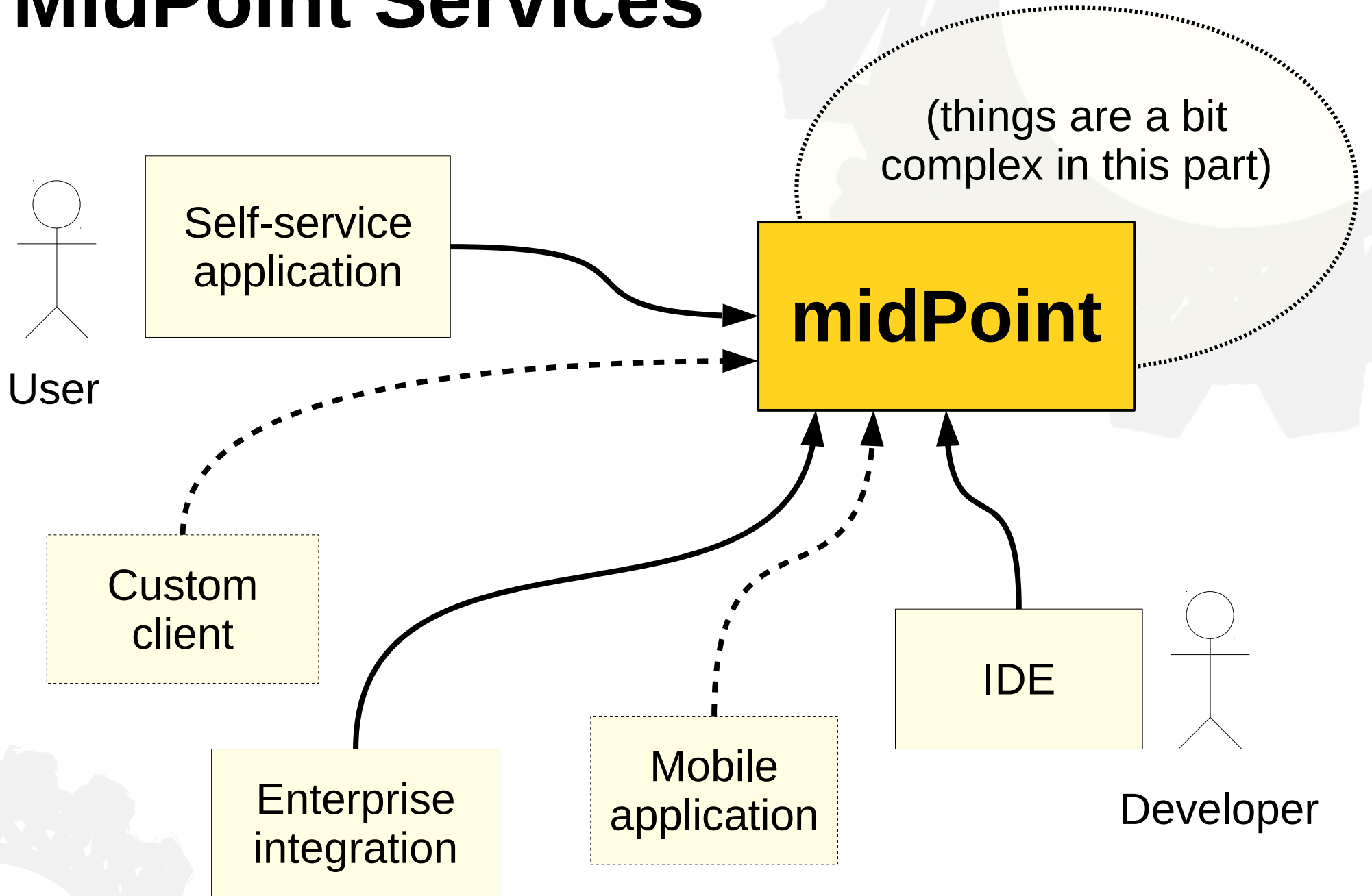
# Practical “REST”

- Try to stick to *resources* and *representations*
  - URL represents object (resource)
- GET as safe operation
- Use POST for RPC when needed
  - But do not abuse
- Define the interface
  - URL formats and meaning, data types, inputs/outputs, ...

# Practical case study: midPoint

- **MidPoint**: comprehensive identity management and governance system
  - Identity management, provisioning, role-based access control, audit, workflow, entitlement management, password management, policy management, segregation of duties, ...
- 100% open source
- Started in 2011
- 700k lines of (mostly) Java code

# MidPoint Services



# MidPoint “REST” service

- Development started in 2013
- Existing data model and operations
  - Object-based: User, Role, Organizational unit, Account, Task, Security policy, ...
  - CRUD ... but there are exceptions
- “RESTful” part and RPC part
- Kept as close to REST ideas as was practical

# RESTful Operations: object

prefix

type

object identifier (OID)

<http://.../rest/users/02c15378-c48b-11e7-b010-1ff8606bae23>

Verbs:

- GET: read
- POST: modify
- DELETE: delete
- PUT: create (not recommended)

# How does object look like?

```
<user oid="02c15378-c48b-11e7-b010-1ff8606bae23">  
  <name>jack</name>  
  <description>Where's the rum?</description>  
  <fullName>Jack Sparrow</fullName>  
  <givenName>Jack</givenName>  
  <familyName>Sparrow</familyName>  
  <emailAddress>jack.sparrow@evolveum.com</emailAddress>  
  <locality>Caribbean</locality>  
  <activation>  
    <administrativeStatus>enabled</administrativeStatus>  
  </activation>  
</user>
```

XML

```
{  
  "name" : "jack",  
  "fullName" : "Jack Sparrow",  
  "givenName" : "Jack",  
  "familyName" : "Sparrow",  
  ...  
}
```

JSON



# RESTful Operations: collection

prefix type  
<http://.../rest/users>

Verbs:

- GET: list objects, search (theoretically)
- POST: create new object
- DELETE: not supported
- PUT: not supported

# Search

<http://.../rest/users/search>

- Query language

```
<q:equal>
```

```
  <q:path>name</q:path>
```

```
  <q:value>jack</q:value>
```

```
</q:equal>
```

- Search with POST: convenience

- Deviation from REST

- Dedicated “search” resource for POST instead of GET

- Should return list of URLs, but returns objects (to save round trips)

# HTTP verbs

- GET is safe
- POST used for many things
- DELETE deletes
- PUT is not very useful
- PATCH for modification
  - But POST works as well
- Typical “REST” API

# Error Codes

- 1xx: Information. Stay tuned.
- 2xx: Success. All done.
- 3xx: Redirection or “in progress” (we will get to that)
- 4xx: Client error (e.g. bad request)
- 5xx: Server error (e.g. bug in server code)

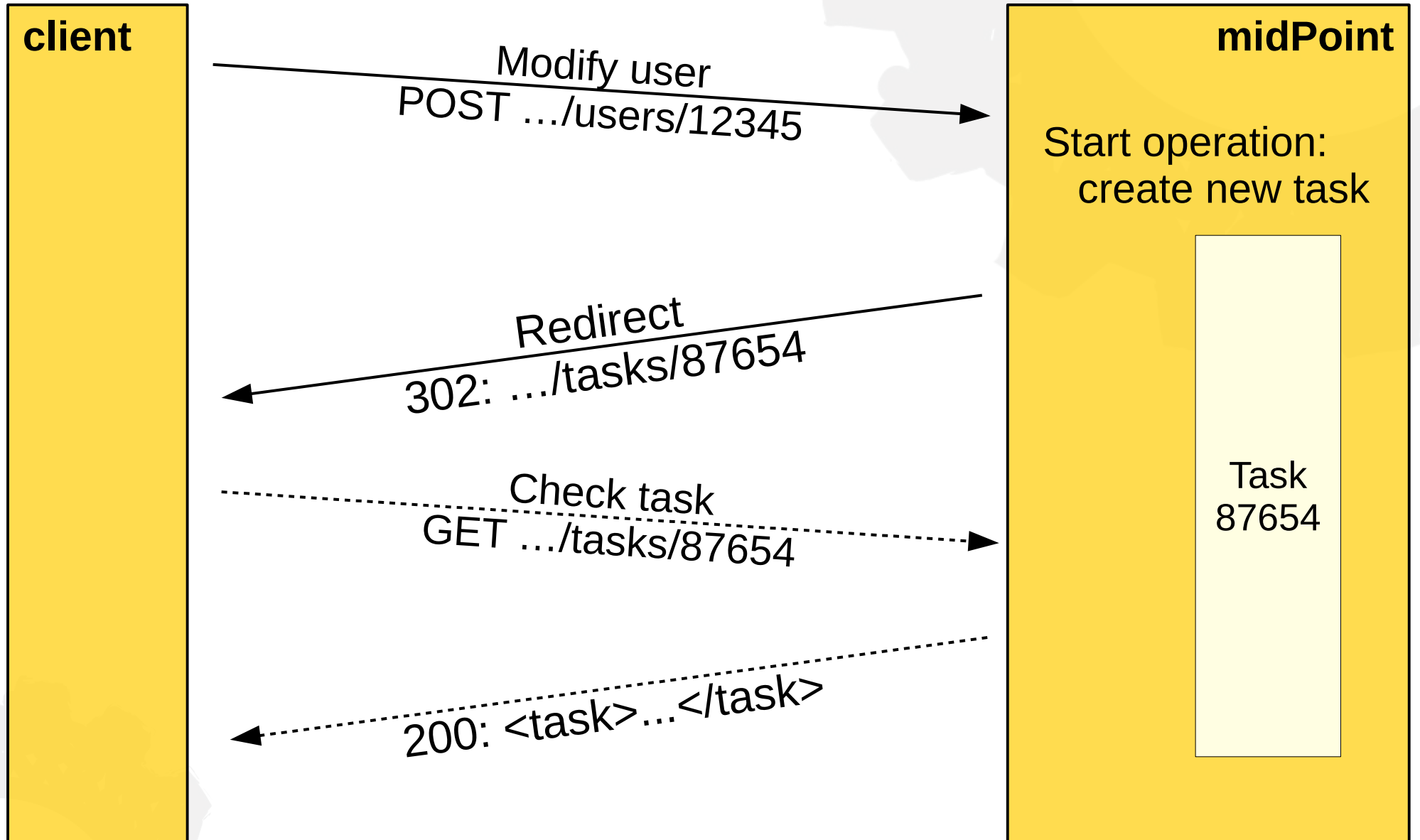


**Trivial, isn't it? Now let's have a look at the interesting stuff.**

# Asynchronous Operations

- Operations that take a loooooong time (days)
- Cannot return success (2xx)
- Solution: redirects (3xx)

# Asynchronous Operations



# RPC Operations: object-related

prefix

type

object identifier (OID)

op

<http://.../rest/tasks/c68d7770-c493-11e7-bce6-9bec1fc3b57c/suspend>

## Verbs:

- GET: not applicable
- POST: execute the operation
- DELETE: not applicable
- PUT: not applicable



# Object-related RPC operations

- Deviation from pure REST
  - Should be modeled as resource changes or separate resources
    - ... but that is a pain
- Error handling

# RPC Operations: global

prefix      operation  
<http://.../rest/notifyChange>

## Verbs:

- GET: not applicable
- POST: execute the operation
- DELETE: not applicable
- PUT: not applicable

# Global RPC operations

- Complex input and output data structures
- Huge deviation from pure REST
- It was necessary to keep the interface simple and efficient



**Next: really hard problems ...**

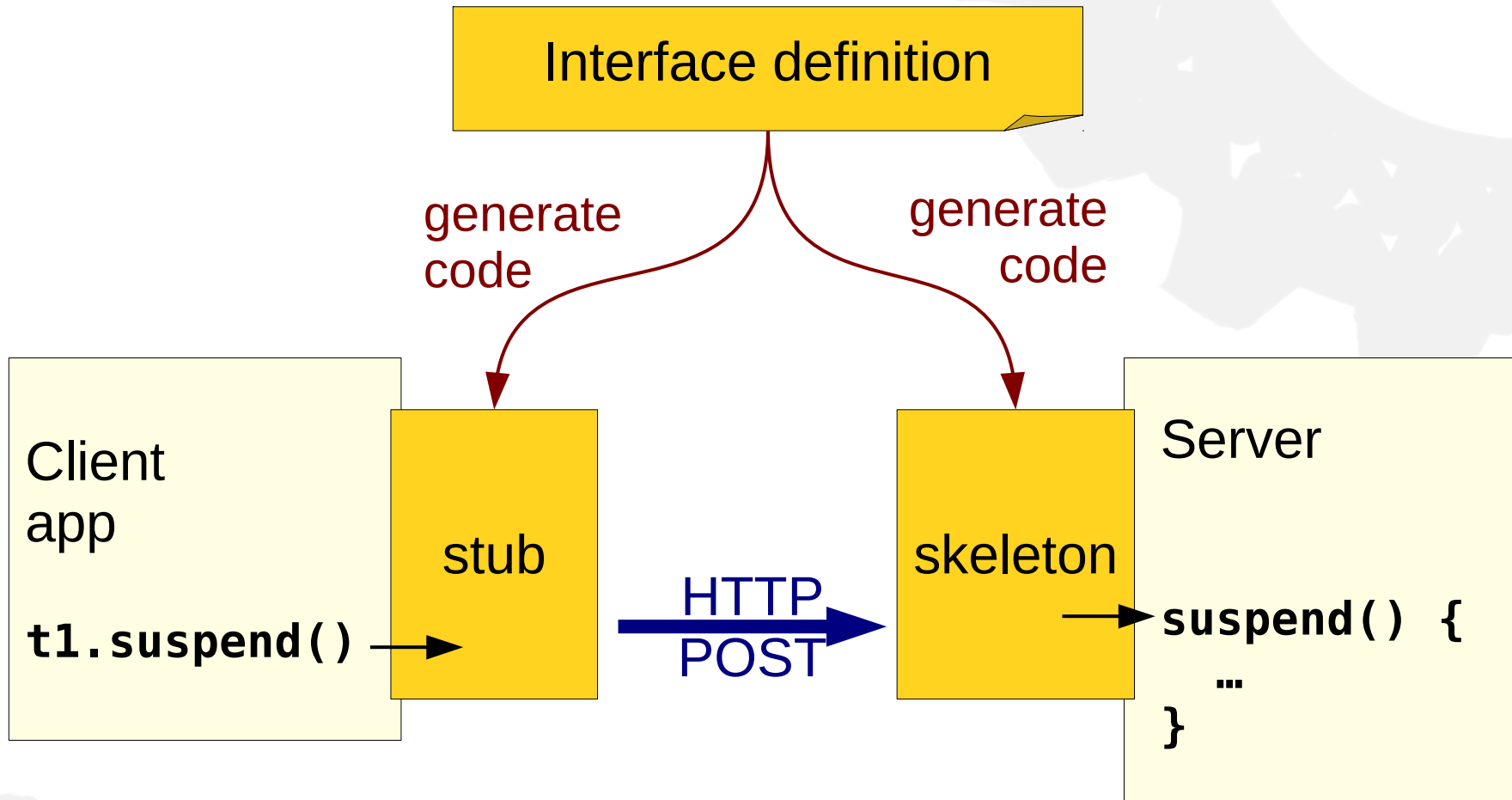
# Security

- REST is stateless
  - Cookie-based sessions are out
- HTTP Basic authentication
  - Oh, really?
- OAuth2 / OpenID Connect
  - “best” practice for REST APIs
- JSON Web Tokens (JWT)
  - SAML tokens reinvented

# Interface Definition

- **Swagger / OpenAPI**
  - WSDL reinvented (CORBA IDL reinvented)
  - JSON-oriented
  - Still quite limited, but at least something
- **REST purists hate it**
  - Because this goes directly against hypertext principles

# Stub and Skeleton



Does not really works in “RESTful” APIs (yet)

# Transactions and Consistency



- Transactions (ACID)
  - Forget it
- Consistency
  - Cannot forget it, but it is tricky
  - Optimistic locking / MVCC is best bet
  - ETag (RFC7232) is a standard mechanism
  - But a lot of proprietary mechanisms is used instead



# Conclusion

- REST is no good for RPC  
... but we are going to use it anyway
- And it is practical  
... with some tweaks and compromises
- Future?

# Questions and Answers



# Thank You

## Radovan Semančík

[www.evolveum.com](http://www.evolveum.com)