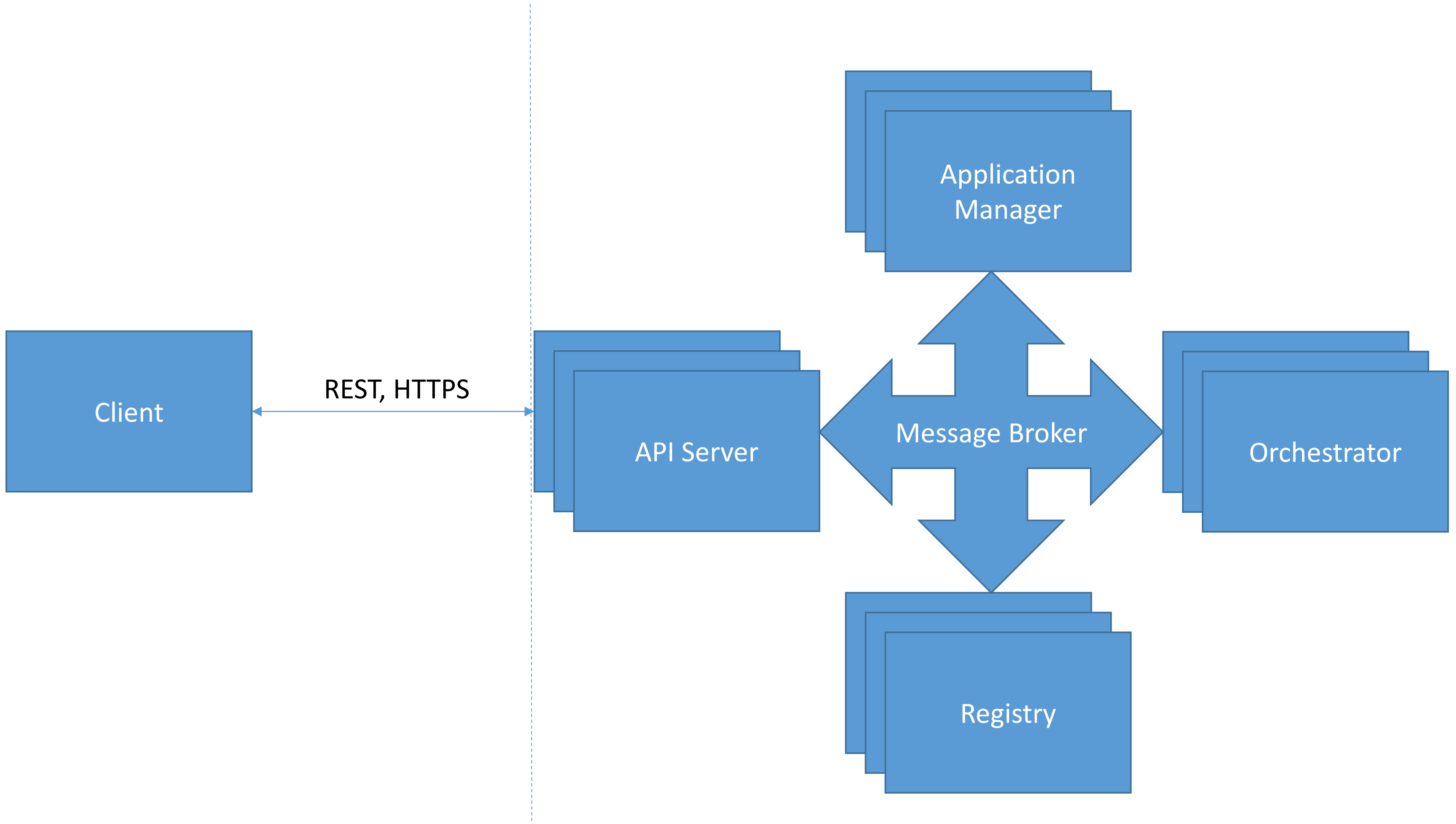
Representational State Transfer (REST)

Applications to Science Gateways

State in Distributed Systems

- Managing state in a distributed is a challenging problem
- Log-centric architectures are a popular way to manage state replication
  - Logs are records of state, not logging messages
  - Kafka, Zookeeper internals
  - Next lecture: the Raft protocol
- States in gateways:
  - Job status changes, file transfers, workflows
  - States (logs) should be a first-class design consideration
- Two types of state to consider
  - State within gateway components
  - State between a gateway and external clients

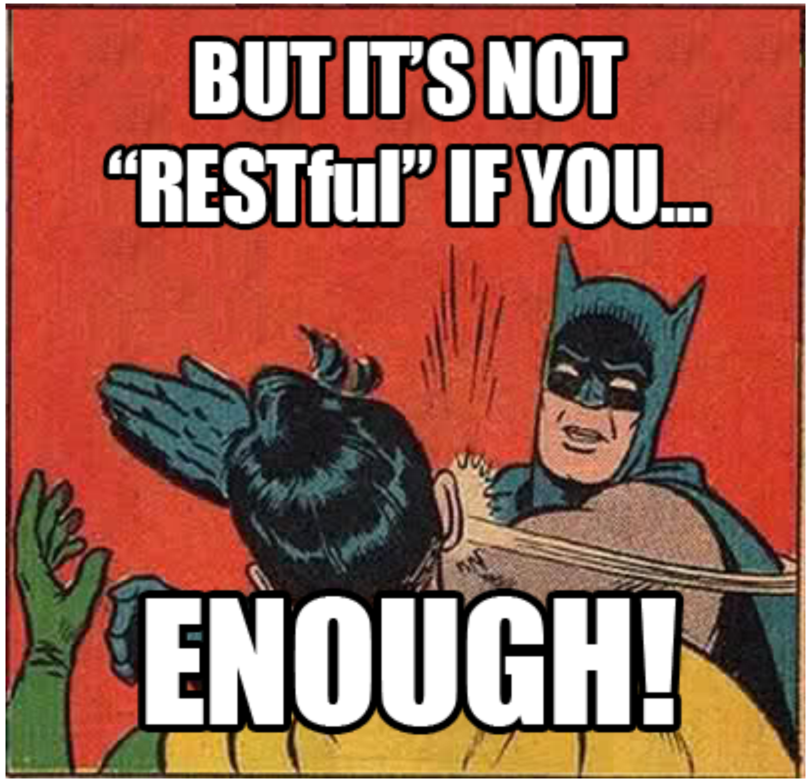

Client
REST, HTTPS
API Server
Application Manager
Message Broker
Orchestrator
Registry

State in Client-Server Systems

- Stateless servers:
  - Do no keep information on the state of the interaction with clients.
  - **Idempotent**: sending the same request multiple times gives the same result as just sending once.
  - Server states can change without communicating this to clients.
    - Someone else bought the last plane ticket
- Stateful servers
  - Maintain information on the clients
    - ”Conversation”
  - Failure recovery can be complicated
- HTTP and REST use stateless servers
- But state for the system has to be *somewhere*

This Is Not a REST Tutorial

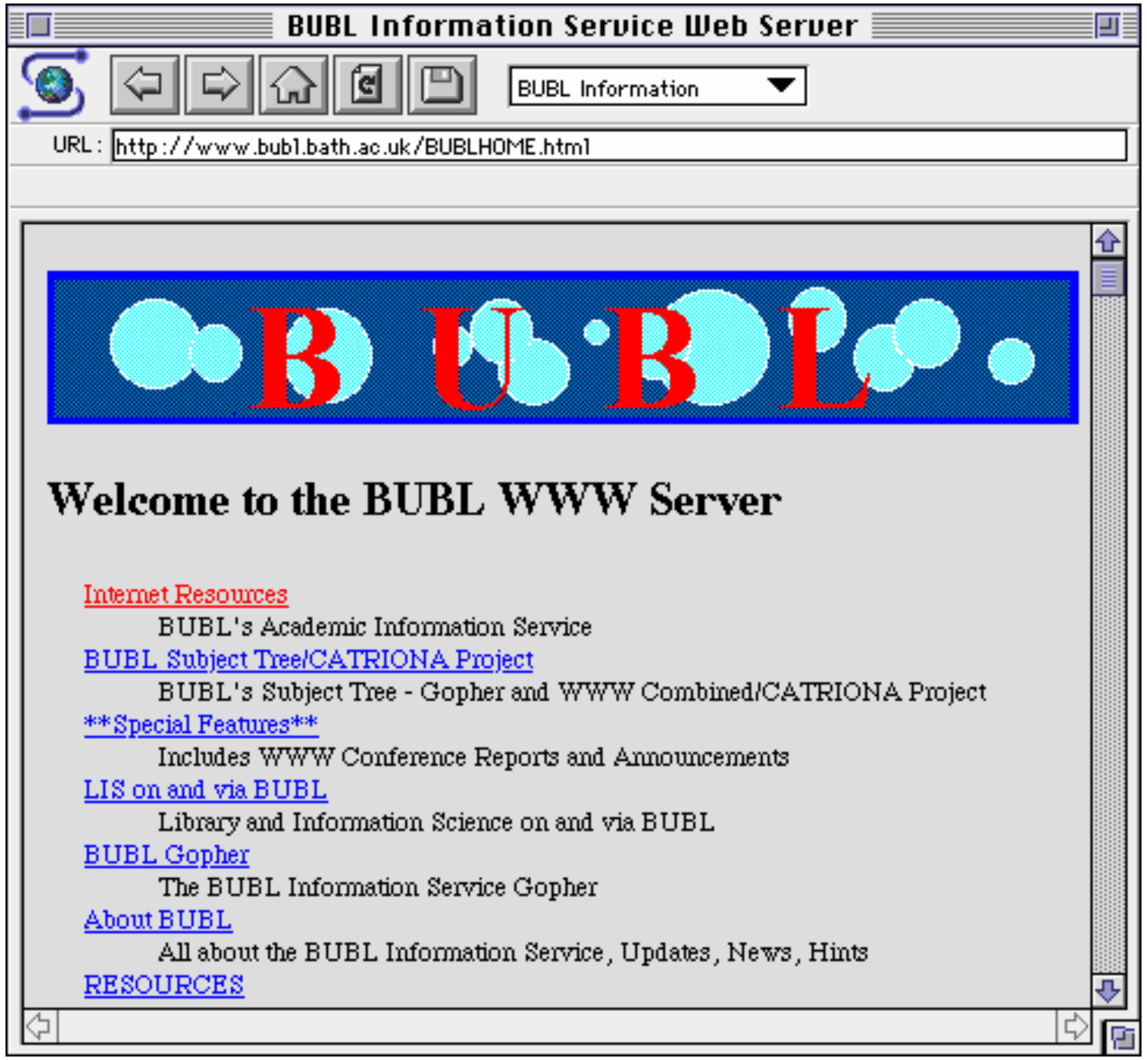
- You can find lots of tutorial by searching.
- REST is an architectural style, not a protocol, so you may find conflicting discussions on some fine (and not so fine) points.
- There are good ways and bad ways to implement REST architecture.
- REST has some shortcomings, so make informed choices.



What Is REST? Consider Web browsers

- Web browsers have the state.
- The user picks what she/he wants to do next based on links.
- The HTML in the browser is the state of the dialog between the user and the web server.
- The server doesn't maintain any state about this dialog.

REST extends these ideas to machine-to-machine communication


BUBL Information Service Web Server
BUBL Information
URL : http://www.bubl.bath.ac.uk/BUBLHOME.html
BUBL

Welcome to the BUBL WWW Server

Internet Resources
BUBL's Academic Information Service
BUBL Subject Tree/CATRIONA Project
BUBL's Subject Tree - Gopher and WWW Combined/CATRIONA Project
Special Features
Includes WWW Conference Reports and Announcements
LIS on and via BUBL
Library and Information Science on and via BUBL
BUBL Gopher
The BUBL Information Service Gopher
About BUBL
All about the BUBL Information Service, Updates, News, Hints
RESOURCES

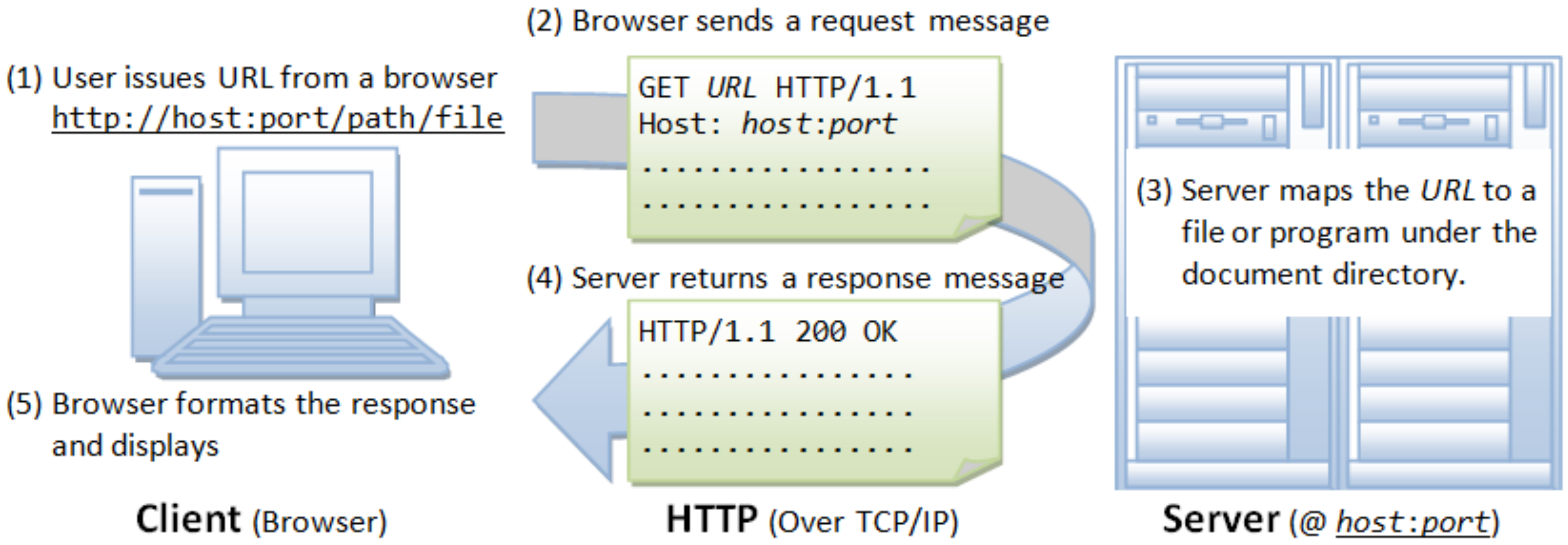
From the Source: Roy Fielding

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a **virtual state-machine**), where the user progresses through an application **by selecting links (state transitions),** resulting in the next page (**representing the next state of the application**) being transferred to the user and rendered for their use."

Fielding, Roy Thomas. "Architectural styles and the design of network-based software architectures." PhD diss., University of California, Irvine, 2000.

In Other Words…

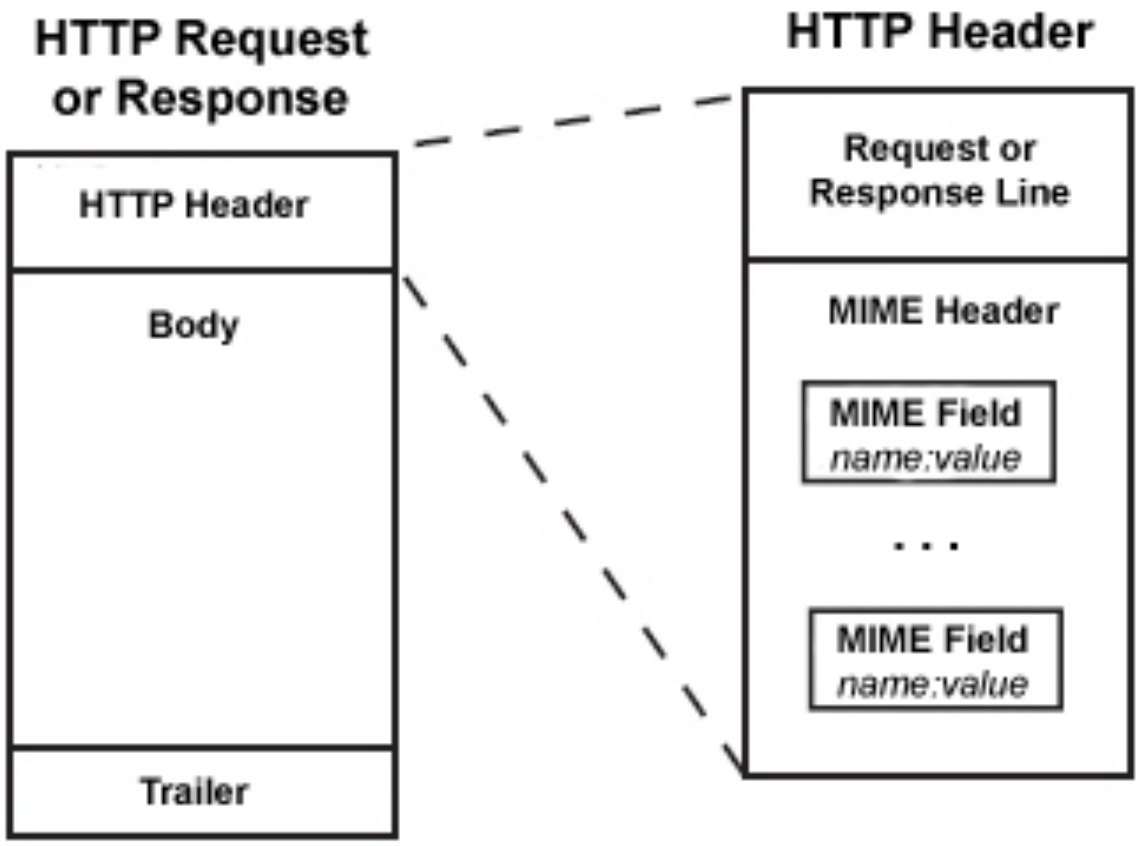
- REST is a generalization of the way the Web works

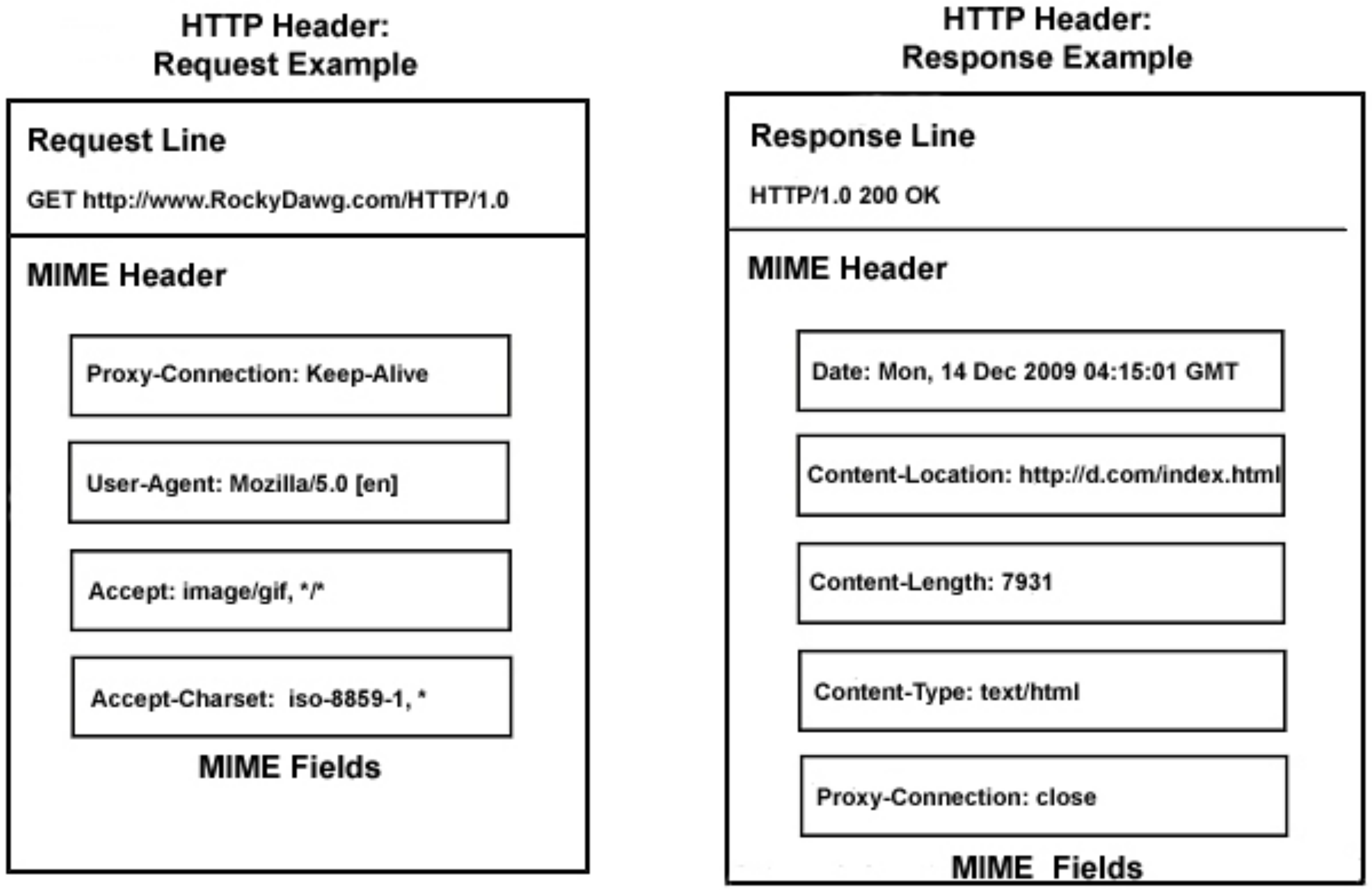

(2) Browser sends a request message
(1) User issues URL from a browser
http://host:port/path/file
GET *URL* HTTP/1.1
Host: *host*:*port*
....
....
(3) Server maps the *URL* to a file or program under the document directory.
(4) Server returns a response message
HTTP/1.1 200 OK
....
....
....
(5) Browser formats the response and displays
Client (Browser)
HTTP (Over TCP/IP)
Server (@ *host*:*port*)

Generalize this for machine-to-machine.

Features of the HTTP Protocol in REST

- HTTP official specifications
  - https://tools.ietf.org/html/rfc2616
- Request-Response
- Uses URLs to identify and address resources.
- Stateless (but extendable)
- Limited set of operations
  - GET, PUT, POST, DELETE, HEAD, ...
- Transfers hypermedia in the body
  - HTML, XML, JSON, RSS, Atom, etc.
- Extendable by modifying its header
  - Security, etc.
- Point to point security
  - TLS: transport level
- Well defined error codes





```
GET http://www.RockyDawg.com/HTTP/1.0
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 [en]
Accept: image/gif, */*
Accept-Charset:  iso-8859-1, *
```

```
HTTP/1.0 200 OK
Date: Mon, 14 Dec 2009 04:15:01 GMT
Content-Location: http://d.com/index.html
Content-Length: 7931
Content-Type: text/html
Proxy-Connection: close
```

https://trafficserver.readthedocs.org/en/stable/_images/http_headers.jpg

REST and HTTP

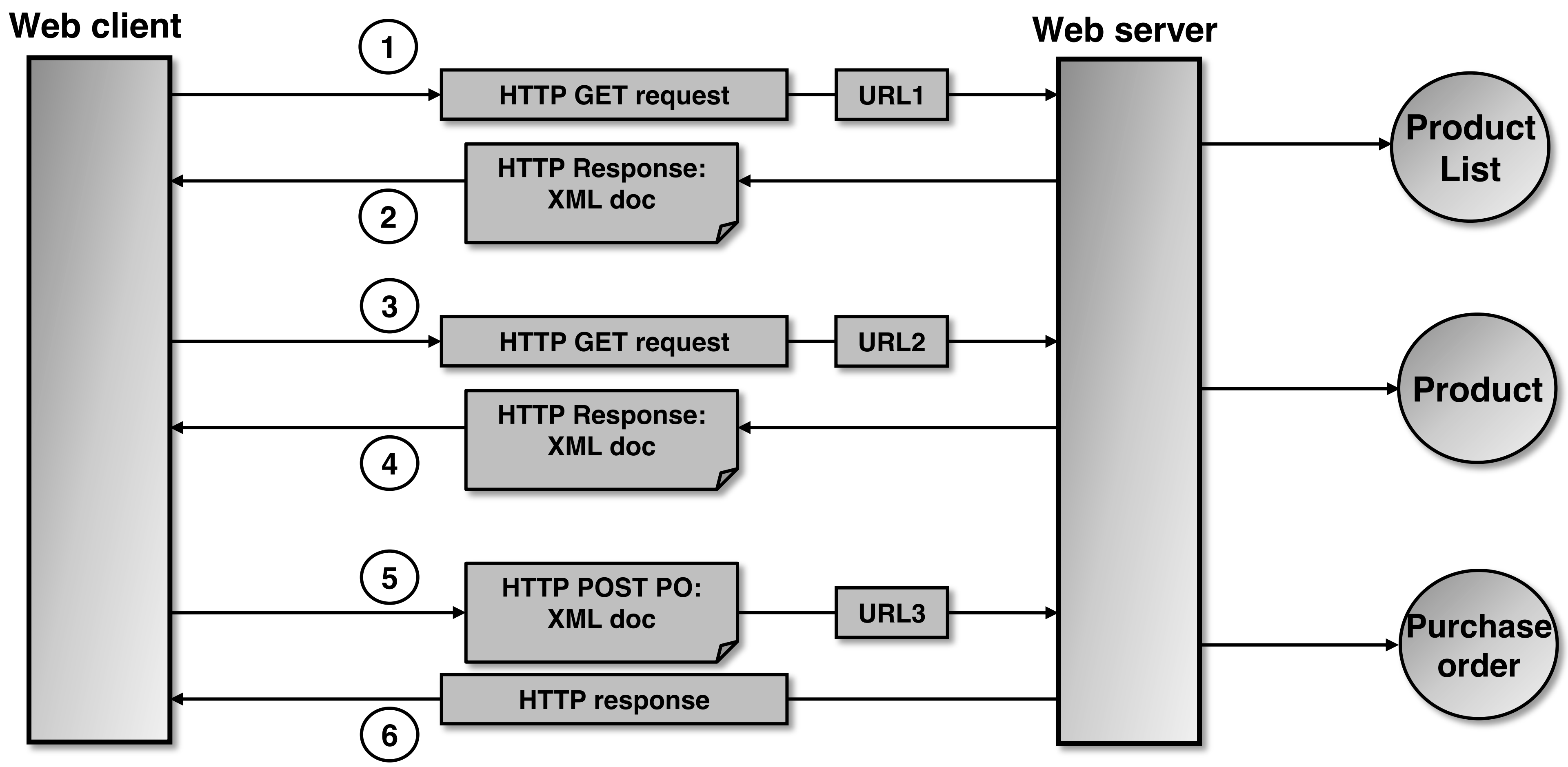
- HTTP is a formally specified protocol for network interactions.
- REST is an architectural style that uses HTTP.
  - Because the Web scales
- “Architectural style”
  - Art + technical skill
- The art of REST is to define APIs using HTTP that are beautiful and elegant as well as useful.

REST and HTTP

- In REST, HTTP operations are VERBS.
  - There are only ~4 verbs.
- URLs are NOUNS
  - Don’t have API methods like “/getUserID”, “/updateUserID”.
  - Why not?
- VERBS act on NOUNS and may change resource state.
- Client states are contained in the response message.
- Resource states are maintained by the server

4. REST ‘protocol’ (3/5)

Example of a REST-ful access (1/3):



URL Patterns for REST Services

- Designing good URLs for REST services is an art.
    - Bad design will still work
    - Leads to insidious technical debt: you need your API to be understandable and consistent
- Best practice is to use structured, not flat URLs.
    - Remember to make them nouns.
- Think of your resources as collections
    - Experiments and their subcomponents, computing resources, applications,
- More importantly, have the right abstraction API
    - Define your resources first
    - Specific URLs may change.
    - HATEOAS
- Are any gateway operations not covered by GET, PUT, POST, DELETE?

Status Codes and Errors

- REST services return HTTP status codes.
- Remember that APIs are used by external developers
- So return the right codes.
    - 200’s: everything is OK
    - 400’s: client errors: malformed request, security errors, wrong URLs
    - 500’s: server errors: processing errors, proxy errors, etc
- Error codes are machine parse-able.
- HTTP doesn’t have application specific errors for your service.
- Include helpful information on why the error occurred.
    - Challenge to make this machine parse-able.
    - Compare with exceptions and try-catch blocks in RPC-ish methods

Some REST Advantages

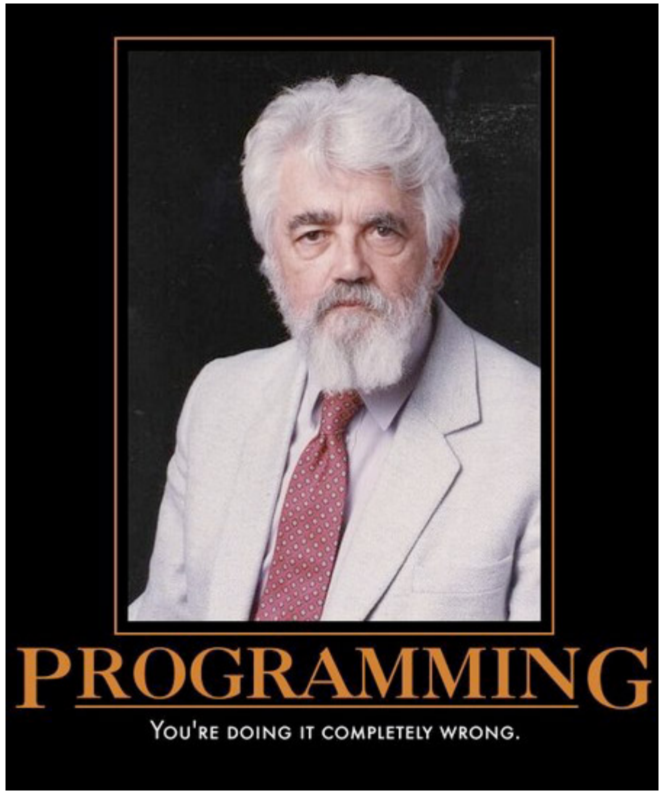
- Leverage 25 years of HTTP investments
  - Security, extensibility, popularity
- Low entry barrier to get people to try your service
  - Use curl command to try things out
- Message format independent
  - Like JSON? Use JSON
  - Like XML? Use XML
  - Like CSV?

REST Challenges

- Data models for your messages are application specific
  - Need to pick a language (JSON, etc)
  - Types are language-dependent
- Changing message formats will break services.
- Message formats are validated by the service implementation, not at the message transport level
- Versioning is informal
  - Every REST implementation comes up with its own strategy.
  - Backward-forward compatibility of different versions is a challenge
- Programmatic error catching is hard
  - No equivalent to Java exceptions
  - Everything is an HTTP error code plus some conventional text

Compare this to Apache Thrift

Hypermedia as the Engine of Application State



HATEOAS

http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

You Are Doing It Completely Wrong

- REST APIs evolve.
  - You add new features.
  - You change the message formats
  - You change the URL patterns
- This breaks RPC-ish clients.
  - Maintaining backward compatibility for legacy clients gets harder over time
- HATEOAS is a “design pattern” to prevent this problem.
  - Keep your clients and server loosely coupled
  - Part of Fielding’s original REST conception that is frequently overlooked.

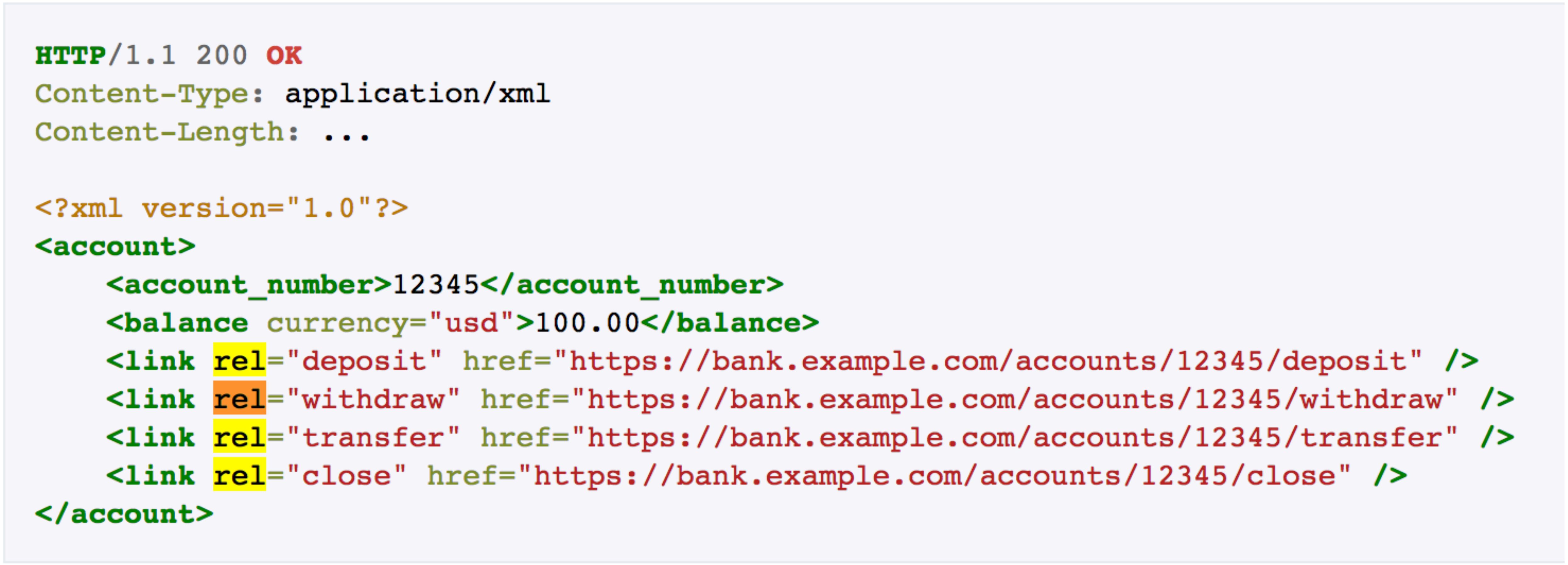
H is for Hypermedia

- Main idea of HATEOAS: **REST services return *hypermedia* responses.**
- Hypermedia is just a document with links to other documents.
- In ”proper” REST, hypermedia documents contain links to what the client can do.
- Semantics of the API need to be understood and defined up front.
- Specific details (links that enable specific actions) can change
- Change can occur over different time scales
  - Resource state changes (think: buying an airplane ticket)
  - Service version changes

HATEOAS in Brief

- Responses return documents consisting of **links**.
- Use links that contain “rel”,“href”, and “type” or equivalent.
- The specific links in a specific message depend on the current state of the dialog between client and server.
  - Not every message contains all of your rels.

Attribute	Description
Rel	This is an API method. You should have persistent, consistent “rels” for all your API methods.
Href	This is the URL that points to the “rel” noun in a specific interaction.
Type	This is the format used in the communications with the href. Many standard types (”text/html”). Custom types should follow standard conventions for naming

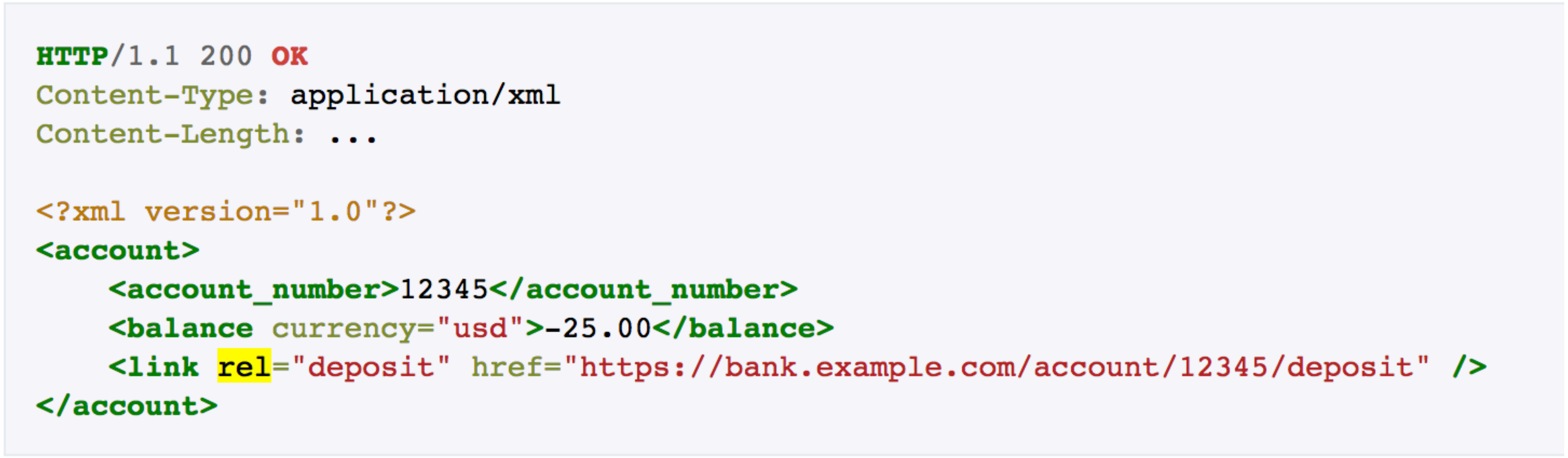


```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
    <link rel="deposit" href="https://bank.example.com/accounts/12345/deposit" />
    <link rel="withdraw" href="https://bank.example.com/accounts/12345/withdraw" />
    <link rel="transfer" href="https://bank.example.com/accounts/12345/transfer" />
    <link rel="close" href="https://bank.example.com/accounts/12345/close" />
</account>
```

The RELs define the general API methods available to you for your next operation, and the URLs provide specific resources that you can operate on. Note the HTTP verbs are not specified, and you are not told what format to send your message. Also, it would be better to use URIs for your rels instead of English verbs.

https://en.wikipedia.org/wiki/HATEOAS



```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">-25.00</balance>
    <link rel="deposit" href="https://bank.example.com/account/12345/deposit" />
</account>
```

HATEOS in action: the same request can give a different response with different possible future actions if you have overdrawn your account.

~ API should tell us what to do ~

```
GET .../item/180881974947
{
 “name” : “Monty Python and the Holy Grail white rabbit big pointy teeth”,
 “id” : “180881974947”,
 “start-price” : “6.50”,
 “currency” : “GBP”,
 ...
 “links” : [
   { “type: “application/vnd.ebay.item”,
     “rel”: “Add item to watchlist”,
     “href”: “https://.../user/12345678/watchlist/180881974947”},
   {
     // and a whole lot of other operations
   ]
}
```

http://www.slideshare.net/josdirksen/rest-from-get-to-hateoas

JSON, XML, HTML, and HATEOAS

- What’s the best language for HATEOAS messages?
- JSON: you’ll need to define “link” because JSON doesn’t have it.
- XML:
  - Extensions like XLINK, RSS and Atom are also have ways of expressing the “link” concepts directly.
  - Time concepts built into RSS and Atom also: use to express state machine evolution.
- HTML: REST is based on observations of how the Web works, so HTML obviously has what you need.

The OpenAPI Specification and Swagger

Using REST to describe REST services

REST Description Languages

- General problem to solve: REST services need to be discoverable and understandable by both humans and machines.
  - “Self Describing”
  - API developers and users are decoupled.
- There are a lot of attempts: https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages

Real problem #1: humans choose APIs, but then the APIs evolve, endpoints change, etc.

Examples of Real Problem #1

- You add a new API method
- You change the way an old API method works.
- You change the inputs and outputs
- You want to add some error handling hints associated with the API
- You change API end points.

HATEOAS may help with some of this.

Real problem #2: Data models are out of scope for REST

More about Real Problem #2

- Science gateway data model examples
  - Computing and data resources, applications, user experiments
- Data models can be complicated to code up so every client has a local library to do this.
- Data models evolve and break clients.
- HATEOAS types in data models depend on data model language (JSON, XML, etc).

Usual solution is to create an SDK wrapper around the API.

Helps users use the API correctly, validate data against data models, etc.

Swagger -> OpenAPI Initiatve, or OAI

- OAI helps automate SDK creation for REST services
- Swagger was a specification for describing REST services
- Swagger is tools for implementing the specification
- OpenAPI Initiative spins off the specification part
- OAI is openly governed, part of the Linux Foundation, available from GitHub
  - https://github.com/OAI/OpenAPI-Specification

OAI Goals

- Define a standard, language-agnostic interface to REST APIs
- Allow both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.
- When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.
- Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service.

http://swagger.io/introducing-the-open-api-initiative/

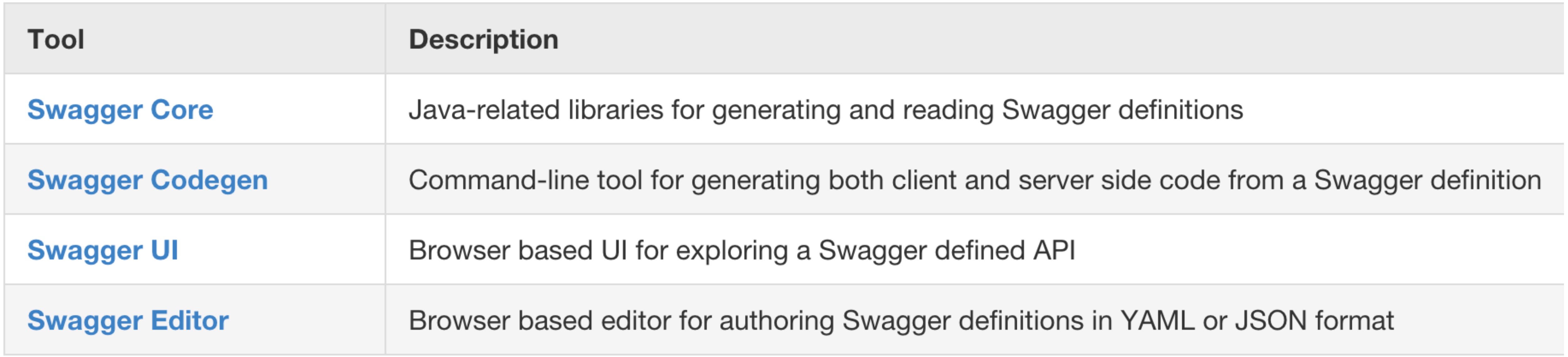
”Hello World!” in OAI

More examples: https://github.com/OAI/OpenAPI-Specification/tree/master/examples/v2.0/json

```
swagger: "2.0"
info:
  version: "1.0"
  title: "Hello World API"
paths:
  /hello/{user}:
    get:
      description: Returns a greeting to the user!
      parameters:
        - name: user
          in: path
          type: string
          required: true
          description: The name of the user to greet.
      responses:
        200:
          description: Returns the greeting.
          schema:
            type: string
        400:
          description: Invalid characters in "user" were provided.
```

http://swagger.io/getting-started-with-swagger-i-what-is-swagger/

Swagger Tools



Tool	Description
Swagger Core	Java-related libraries for generating and reading Swagger definitions
Swagger Codegen	Command-line tool for generating both client and server side code from a Swagger definition
Swagger UI	Browser based UI for exploring a Swagger defined API
Swagger Editor	Browser based editor for authoring Swagger definitions in YAML or JSON format

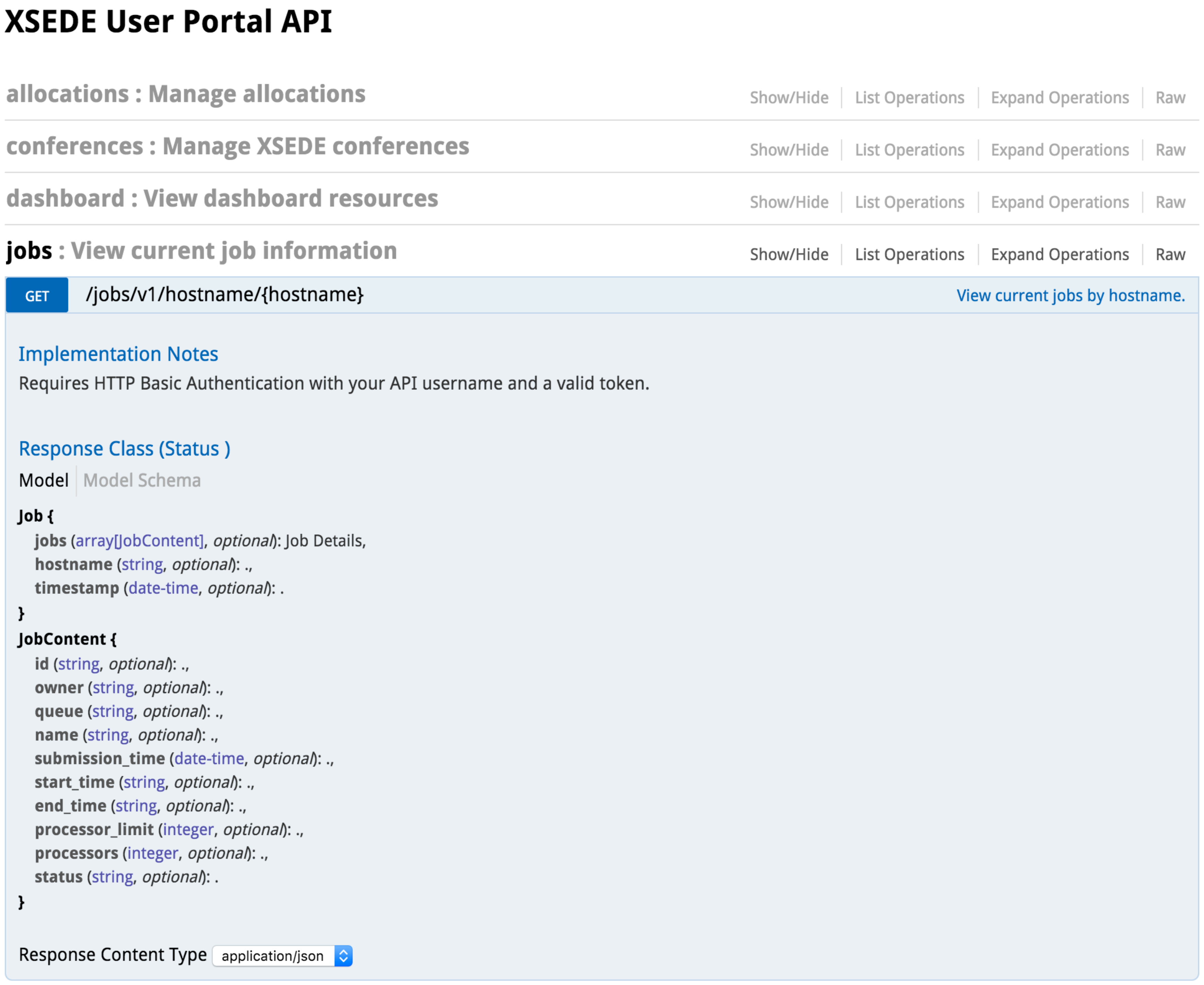
http://swagger.io/tools/

Creating OAI Definitions

- Top Down: You Don’t Have an API
  - Use the **Swagger Editor** to create your Swagger definition
  - Use the integrated **Swagger Codegen** tools to generate server implementation.
- Bottom Up: You Already Have an API
  - Create the definition manually using the same Swagger Editor, OR
  - Automatically generate the Swagger definition from your API
    - Supported frameworks: JAX-RS, node.js, etc
- My advice: be careful with automatically generated code.

Swagger and the XSEDE User Portal

https://portal.xsede.org/



XSEDE User Portal API

allocations : Manage allocations — Show/Hide | List Operations | Expand Operations | Raw

conferences : Manage XSEDE conferences — Show/Hide | List Operations | Expand Operations | Raw

dashboard : View dashboard resources — Show/Hide | List Operations | Expand Operations | Raw

jobs : View current job information — Show/Hide | List Operations | Expand Operations | Raw

GET /jobs/v1/hostname/{hostname} — View current jobs by hostname.

Implementation Notes

Requires HTTP Basic Authentication with your API username and a valid token.

Response Class (Status)

Model | Model Schema

```
Job {
  jobs (array[JobContent], optional): Job Details,
  hostname (string, optional): .,
  timestamp (date-time, optional): .
}
JobContent {
  id (string, optional): .,
  owner (string, optional): .,
  queue (string, optional): .,
  name (string, optional): .,
  submission_time (date-time, optional): .,
  start_time (string, optional): .,
  end_time (string, optional): .,
  processor_limit (integer, optional): .,
  processors (integer, optional): .,
  status (string, optional): .
}
```

Response Content Type: application/json

https://api.xsede.org/swagger/

REST and Science Gateways

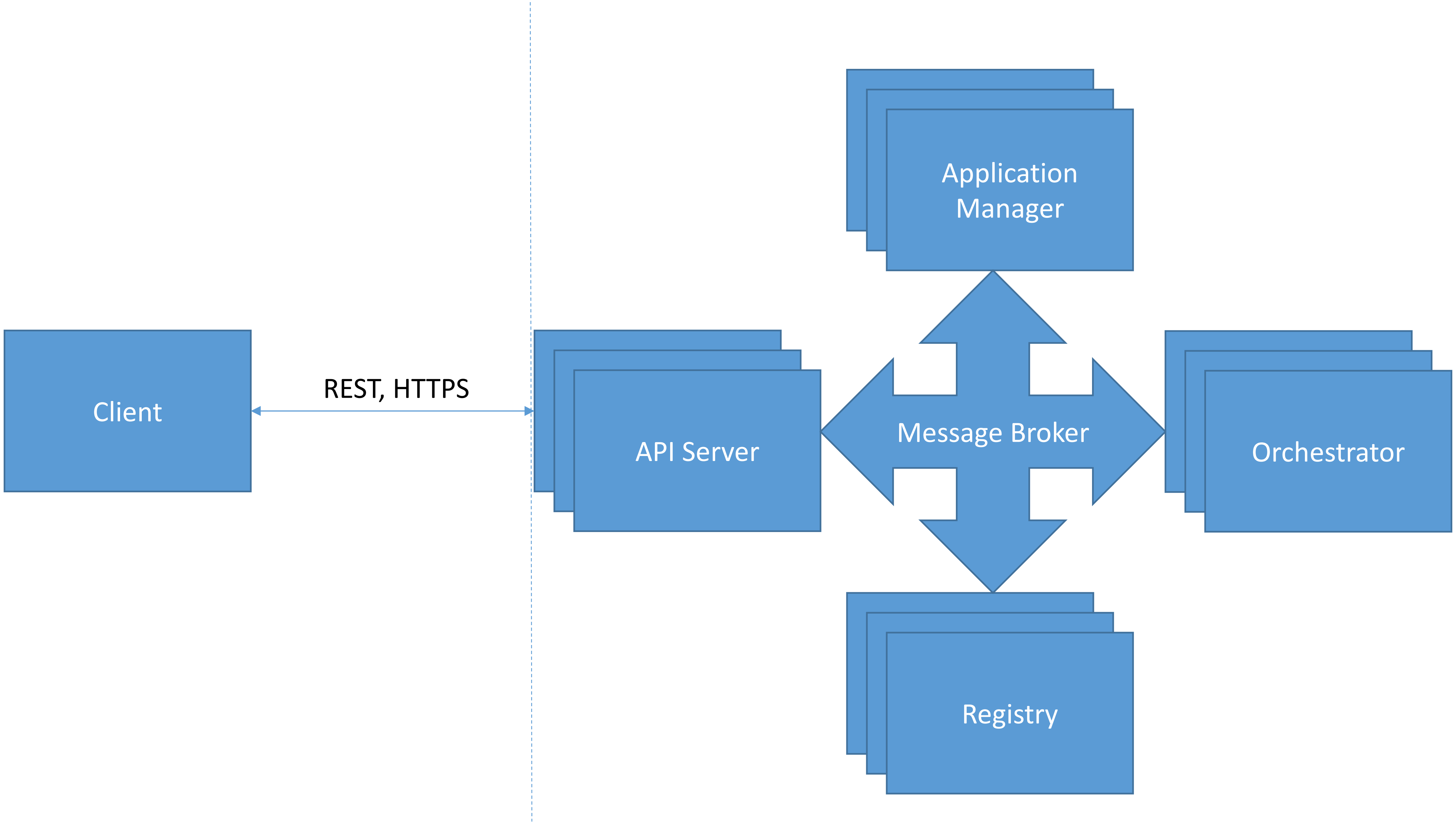
Applying to Science Gateways

REST and Science Gateways

- Your actions are already defined: GET, PUT, POST, DELETE
- Define your nouns and noun collections: you need to get this right
  - Computing resources: static information and states
  - Applications: global information about a specific scientific application
  - Application interfaces: resource specific information about an application
  - Users
  - User experiments: static information and states
- Define data models for your nouns
  - You will get this wrong, but don’t worry
- Define the operation patterns on your nouns
  - Composed of request-response atomic interactions
- You need to specify your HATEOAS hypermedia formats
  - Your operation patterns map to these.

Is REST appropriate for microservice systems?

Assumptions: you have a lot of microservices, each service may maintain redundant versions for load balancing, and services come and go.


Application Manager
Client
REST, HTTPS
API Server
Message Broker
Orchestrator
Registry

Comparison of REST and Microservices

REST over HTTP	Gateway Microservices
• Request-response • Synchronous • URLs must be resolvable • Server determines possible future states • Client picks the next state it wants • Internal server state changes are hidden • Fault recovery, elasticity, etc are not inherent considerations	• Work well with push messaging, queues, many-to-many messaging • Asynchronous • State manager (orchestrator) decides what should be done next • Orchestrator needs to be notified by external state changes • Fault recovery, elasticity, etc are first class considerations