SGCI

Assignment 1: Project Brainstorming and Napkin Drawings

How are things going? Any questions? Ready to present this afternoon?

Schedule for July 16-17

- July 16
  - Review available COVID-19 data
  - Brainstorm ideas
  - Come up with a team name for your GitHub repository
- July 17
  - Morning, Part 1: Technology overview for implementing gateways
  - Morning, Part 2: Prepare napkin drawing presentations
  - Afternoon: Each team gives their napkin drawing presentation “pitches”

Communication and Collaboration

What you need to work together

Communication

I’ve created a Slack channel, **#2020-summer-coding-institute,** that you can use to contact each other or your mentors.

Each team has its own dedicated slack channel

Feel free to email me also: marpierc@iu.edu

Collaboration

Create	If you don’t have a GitHub account, please create one.
Send	Send me your GitHub username
Add	I will add you to the SGCI-2020-Coding-Institute GitHub repo
Get	Each team gets its own repo

Choosing a Technology Stack for Your Projects

You Have a Few Choices

- You can use one of the gateway technologies from this week's tutorials.
- You can use a simple one that I've outlined here
  - https://github.com/SGCI-2020-Coding-Institute/sgci-summer-school
- But this needs to be your team's decision

No Matter What You Choose…

Use GitHub

Your Top Priority: Build a Working Prototype

Concentrate on implementing your napkin drawing idea.

Use GitHub to share code.

Everyone should contribute

Roles: Architect, Developer, UX Designer, Quality Assurance Tester, Operator

Project Challenge Levels

Use these if you are using the project template at https://github.com/SGCI-2020-Coding-Institute/sgci-summer-school

Optional Project Challenge Levels

Challenge 1	Put your code in a docker container
Challenge 2	Deploy your container into Kubernetes
Challenge 3	Deploy your code on Jetstream
Challenge 4	Automatically update your Jetstream deployment whenever you push a commit to your GitHub repository

Representational State Transfer (REST)

Applications to Science Gateways

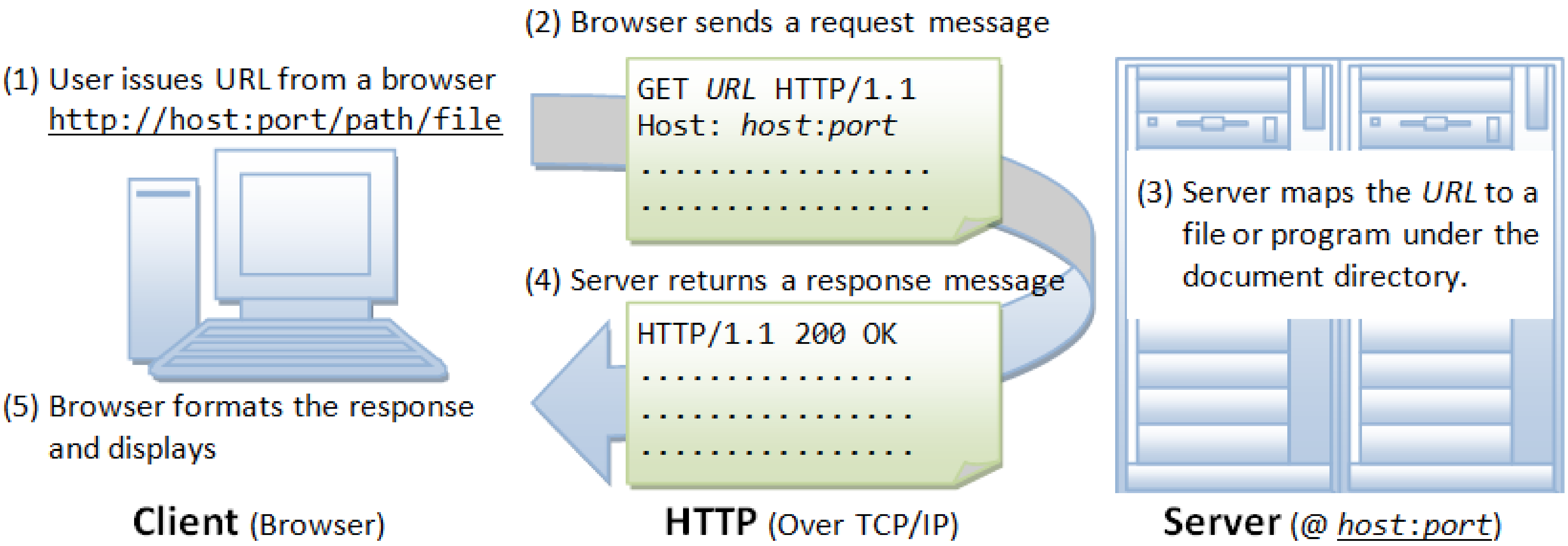
From the Source: Roy Fielding

"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)

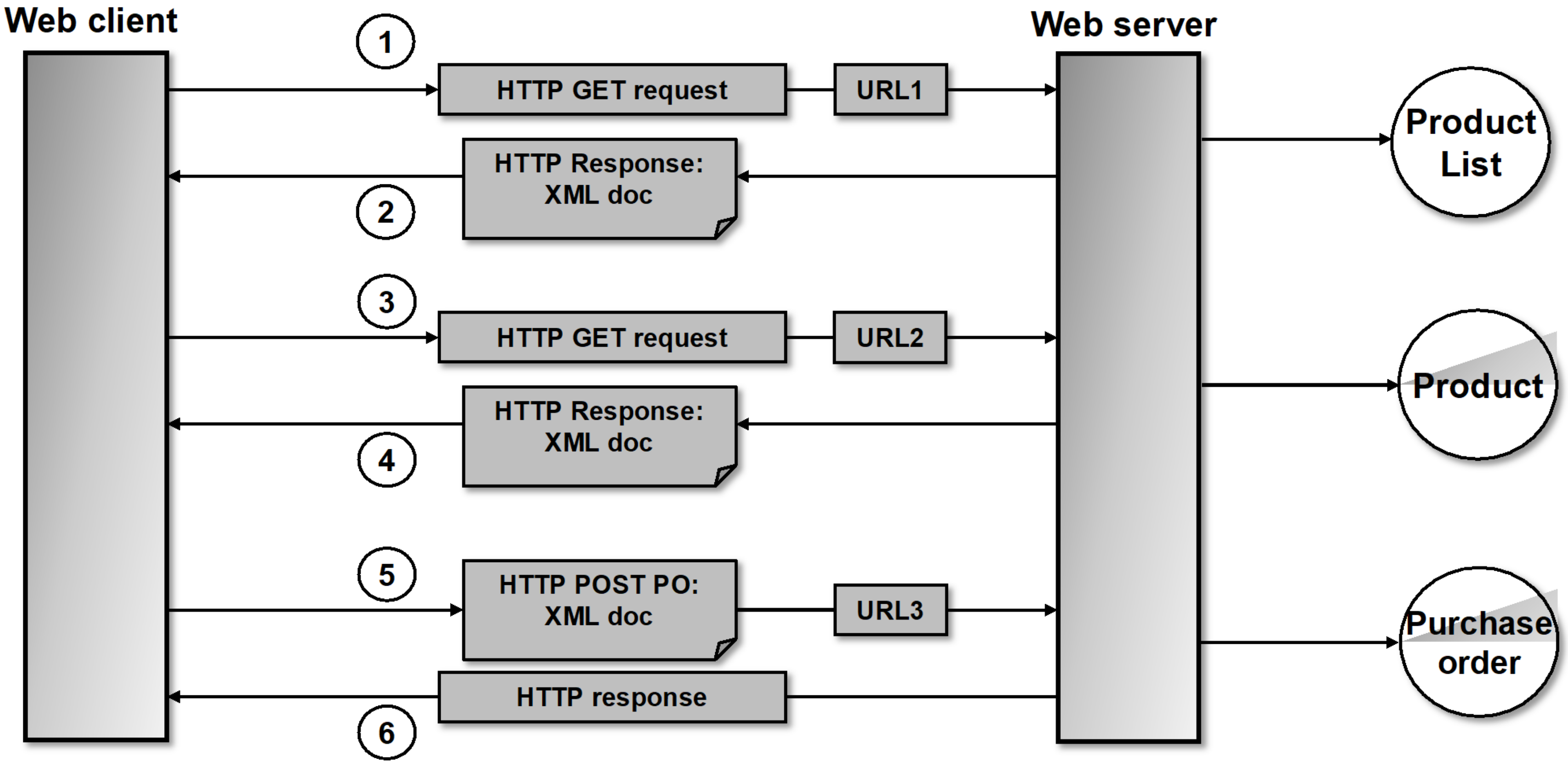
HTTP (Over TCP/IP)

Server (@ *host:port*)

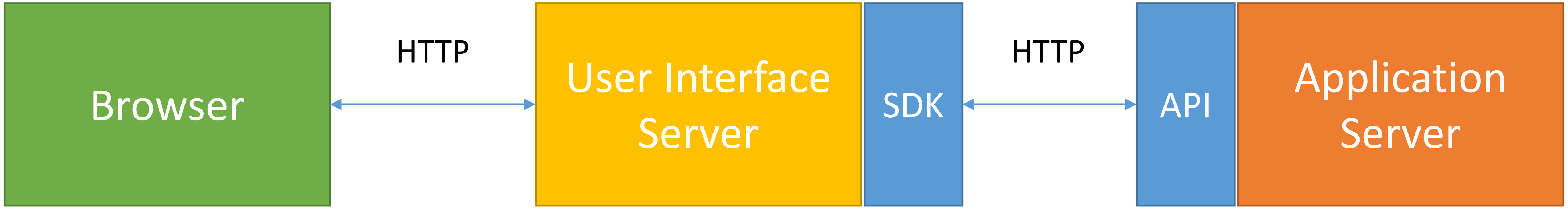
Generalize this for machine-to-machine.

4. REST 'protocol' (3/5)

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

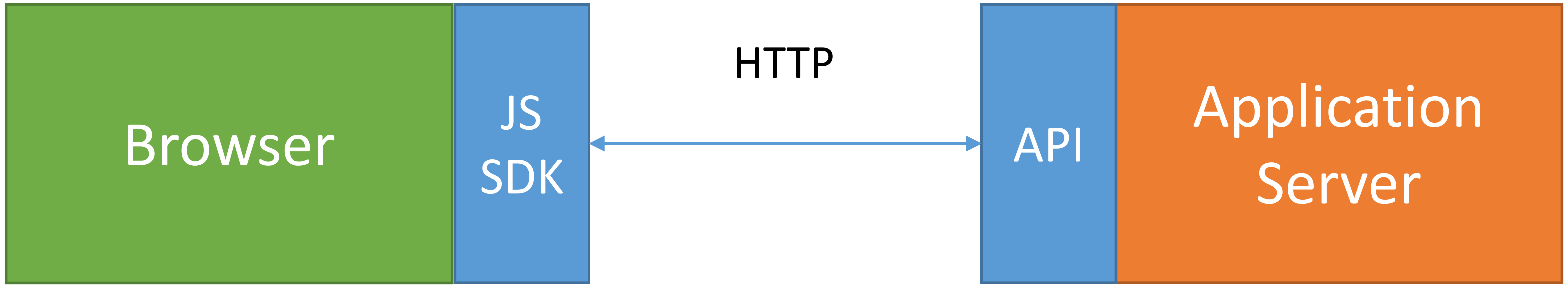


REST and APIs, Style #1



The API and the SDK can be implemented in different programming languages.

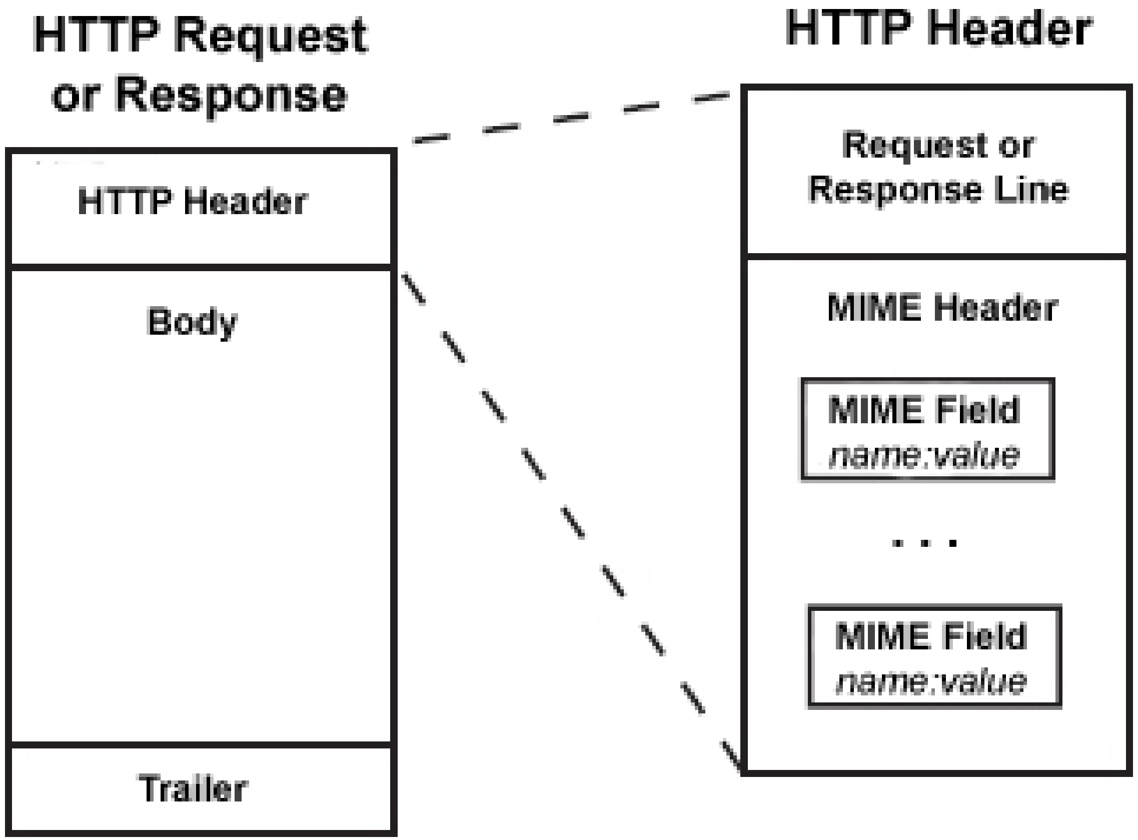
REST and APIs, Style #2



Use JavaScript in the browser as the REST client. Have the Application Server send JSON.

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.
- 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



REST and HTTP

- In REST, HTTP operations are VERBS.
  - There are only ~4 verbs.
- URLs are NOUNS
  - Right: ”/userID”
  - Wrong: “/getUserID”, “/updateUserID”.
  - Why?

- VERBS act on NOUNS to change the resource state.
- Client states are contained in the response message.
- Resource states are maintained by the server

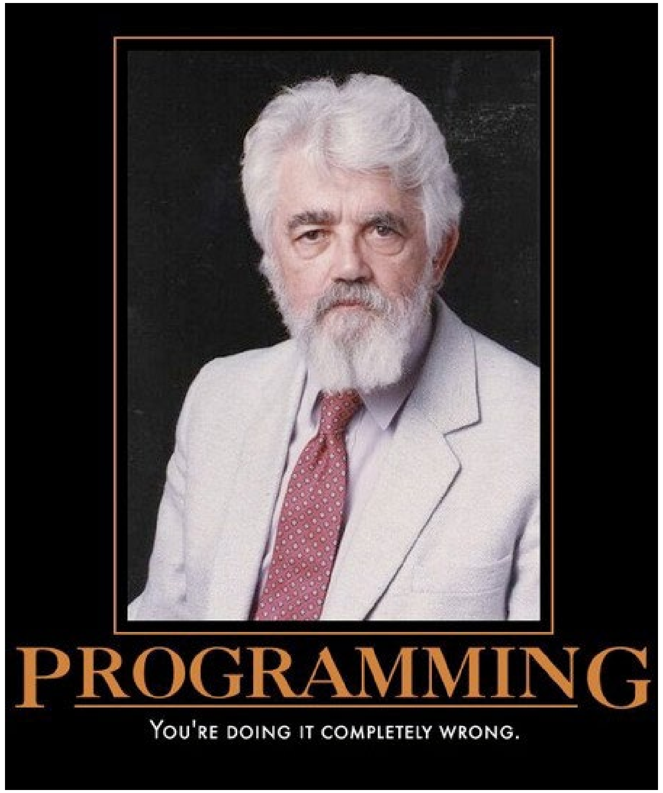
Status Codes and Errors

- REST services return HTTP status codes.
- 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.

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?

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

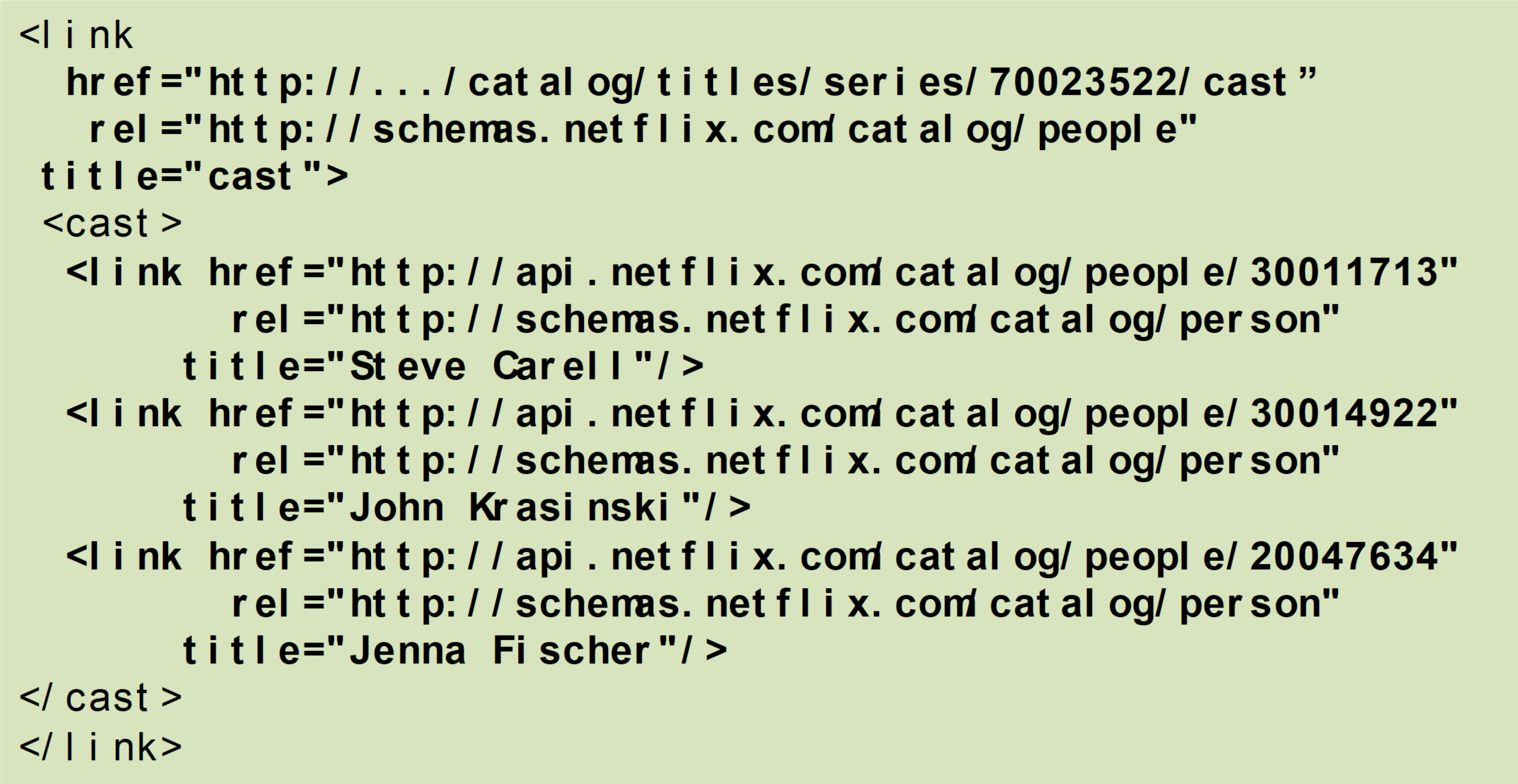
- Main idea of HATEOAS: **REST services return *hypermedia* responses.**
- Hypermedia is just a document with links to other documents.
- In ”proper” REST, hypermedia documents contain links to what the client can do.
- Semantics of the API need to be understood and defined up front.
- Specific details (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 a noun. You should have persistent, consistent “rels” for all your nouns.
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

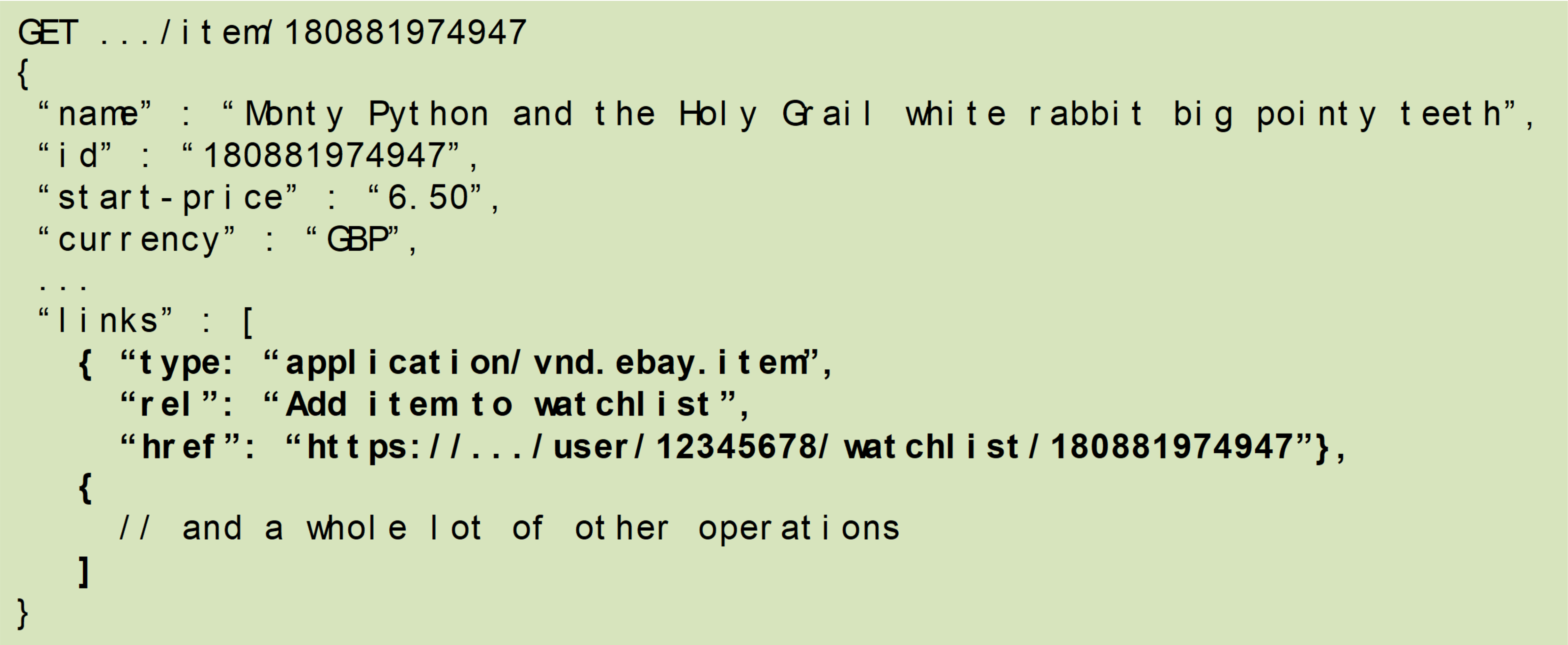
~"Netflix"API"~"



```
<link
   href="http://.../catalog/titles/series/70023522/cast”
    rel="http://schemas.netflix.com/catalog/people"
  title="cast">
  <cast>
   <link href="http://api.netflix.com/catalog/people/30011713"
             rel="http://schemas.netflix.com/catalog/person"
         title="Steve Carell"/>
   <link href="http://api.netflix.com/catalog/people/30014922"
             rel="http://schemas.netflix.com/catalog/person"
         title="John Krasinski"/>
   <link href="http://api.netflix.com/catalog/people/20047634"
             rel="http://schemas.netflix.com/catalog/person"
         title="Jenna Fischer"/>
</cast>
</link>
```

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

~"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

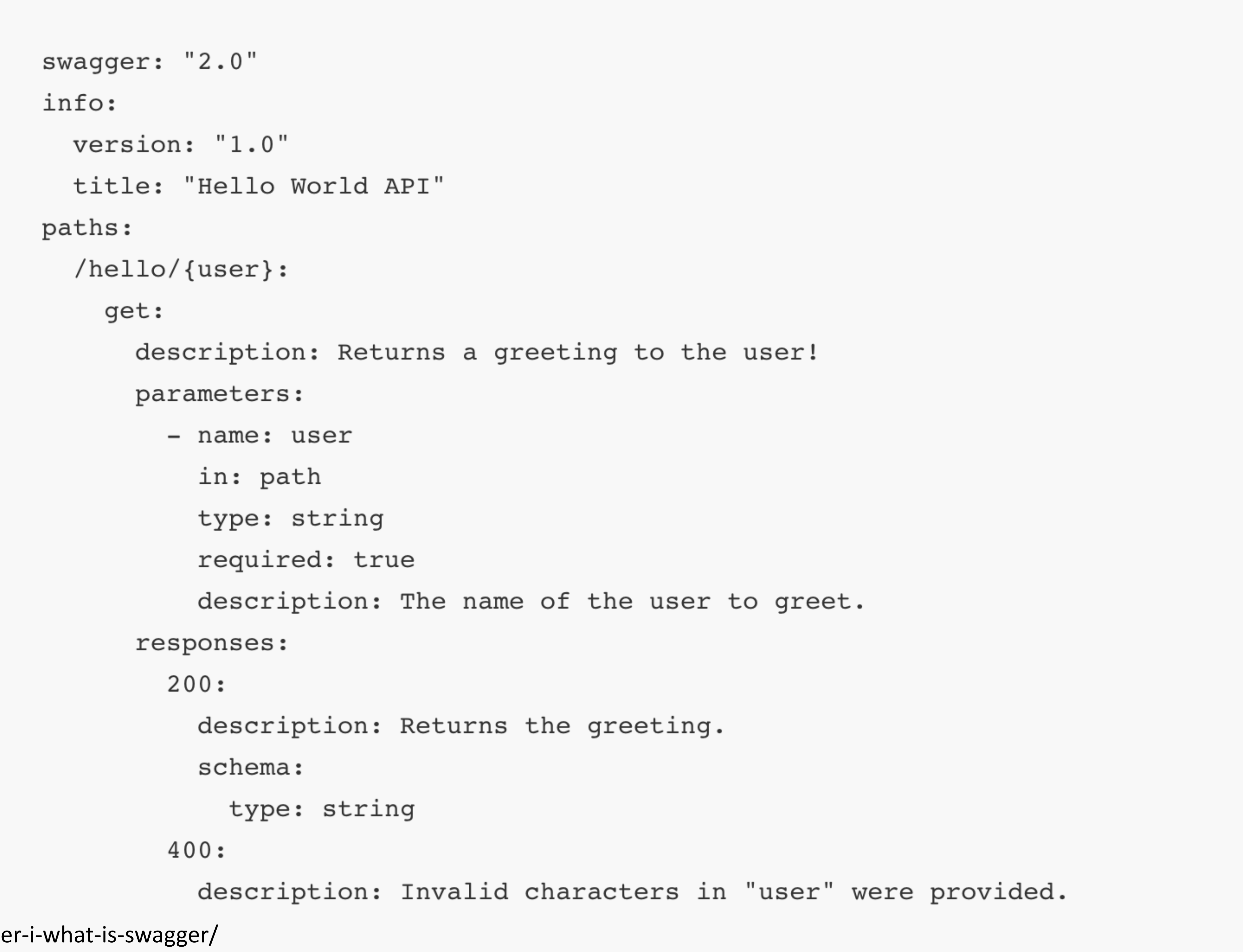
OAI Goals

- 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

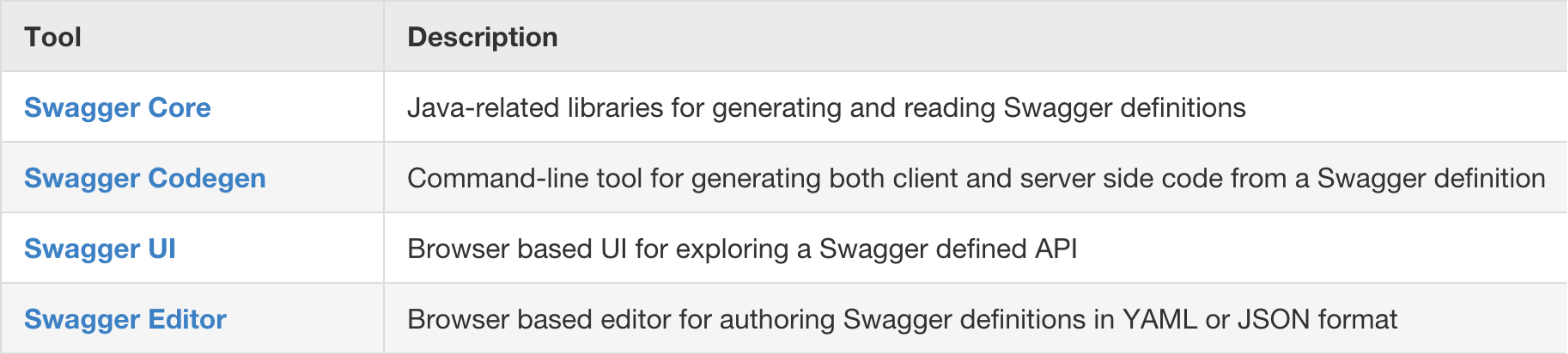
More examples: https://github.com/OAI/OpenAPI-Specification/tree/master/examples/v2.0/json



```
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

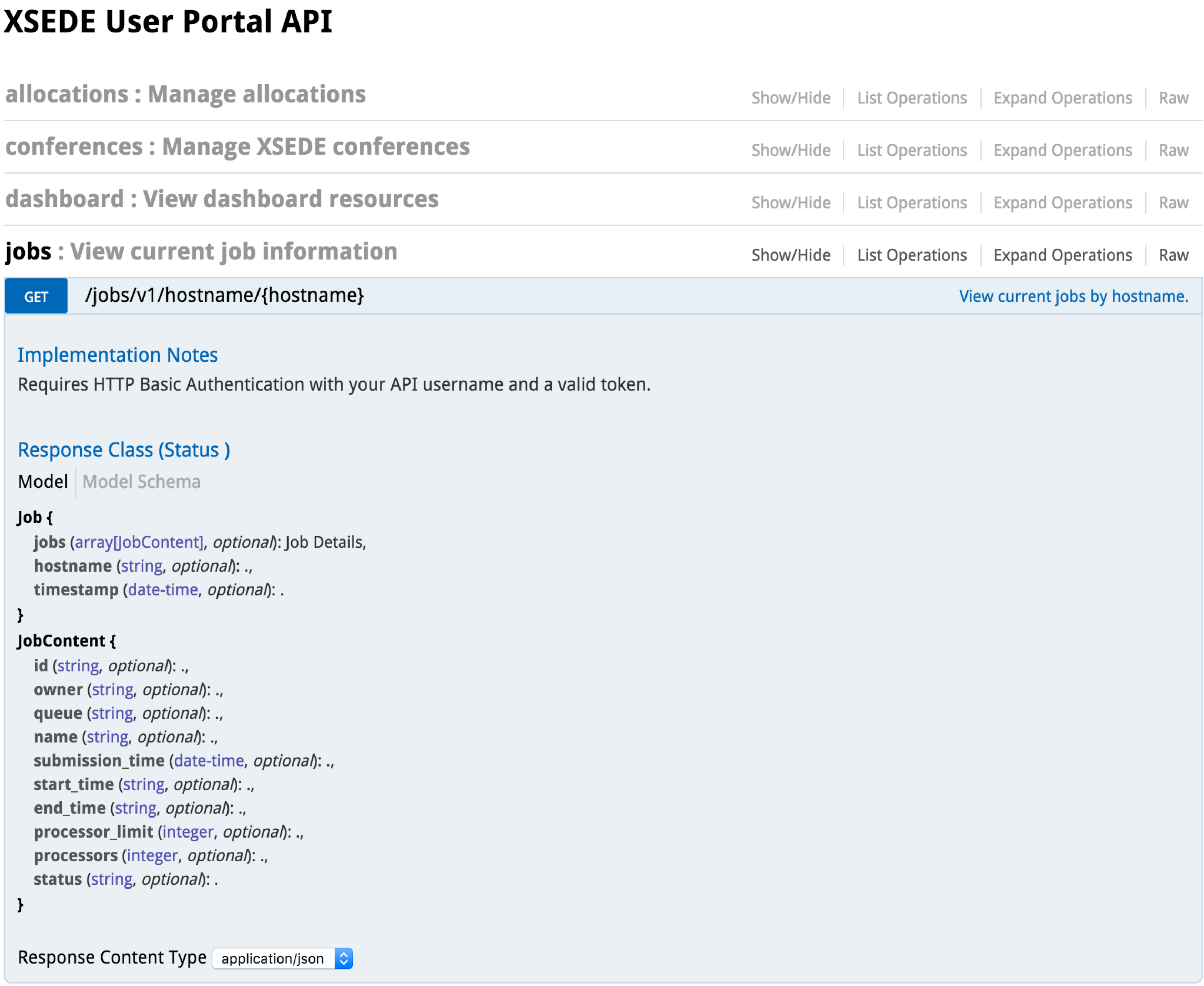
http://swagger.io/tools/

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

conferences : Manage XSEDE conferences

dashboard : View dashboard resources

jobs : View current job information

/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)

```
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, etc
- 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.