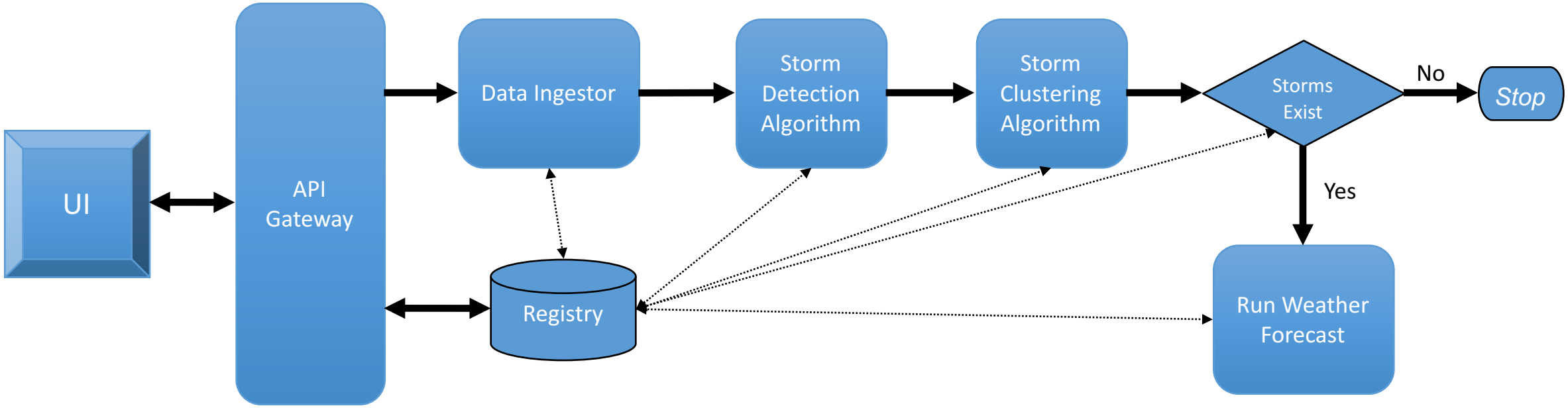
Representational State Transfer (REST)

Applications to Science Gateways

Project Milestone 1 Microservices



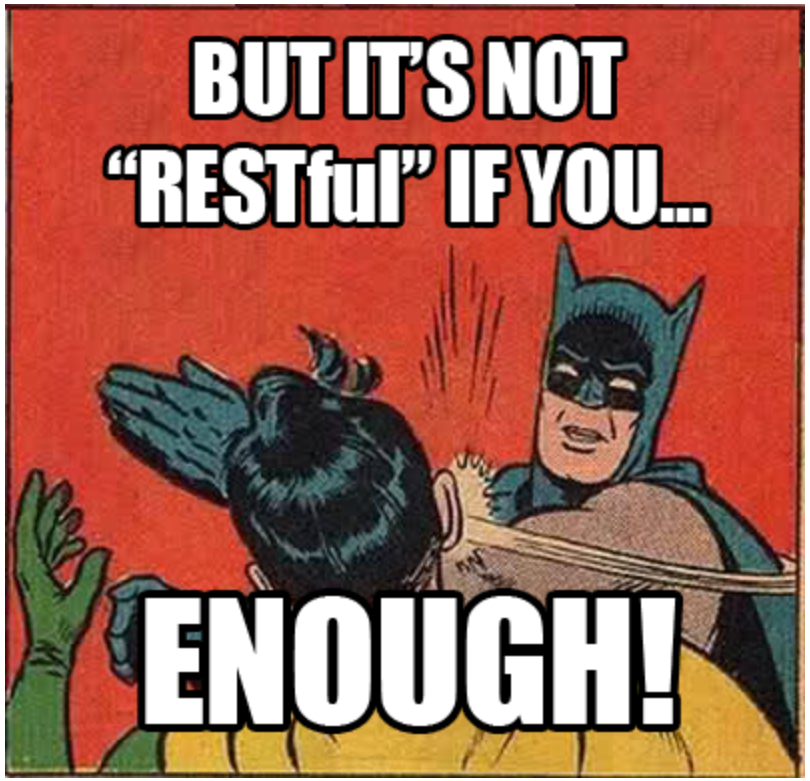
Components need to communicate. Options include Apache Thrift, REST, and Hybrids

State in Distributed 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
  - 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.



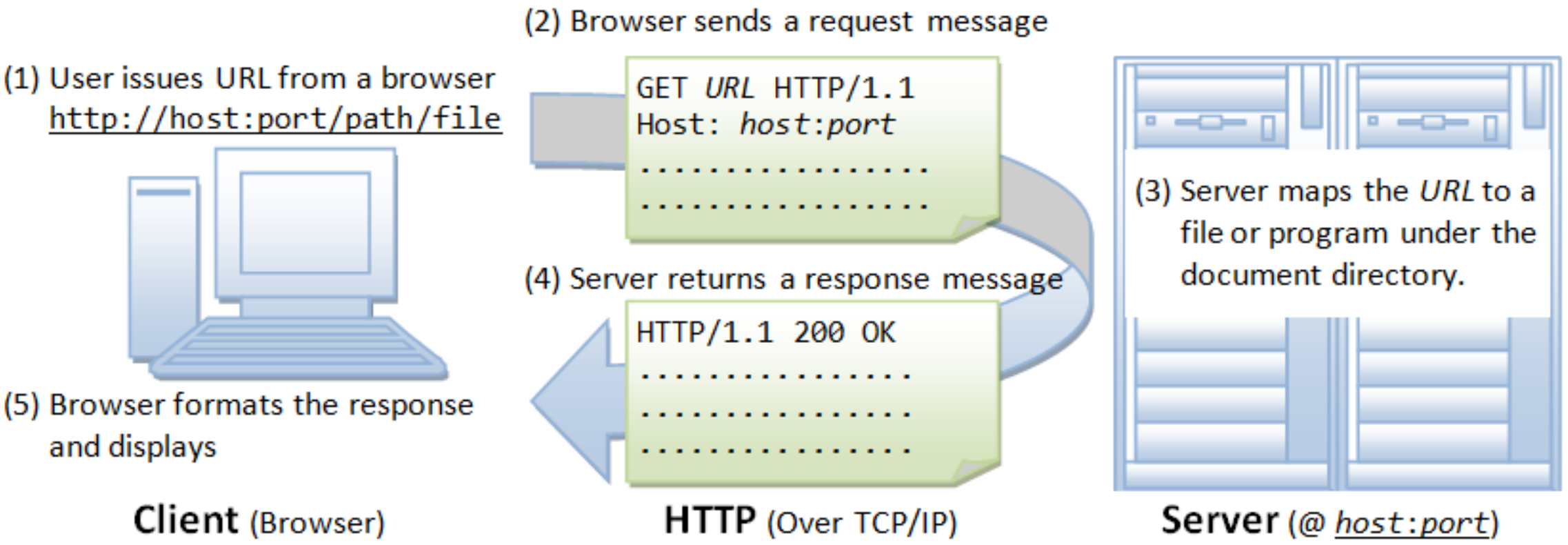
From the Source: Roy Fielding

"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…

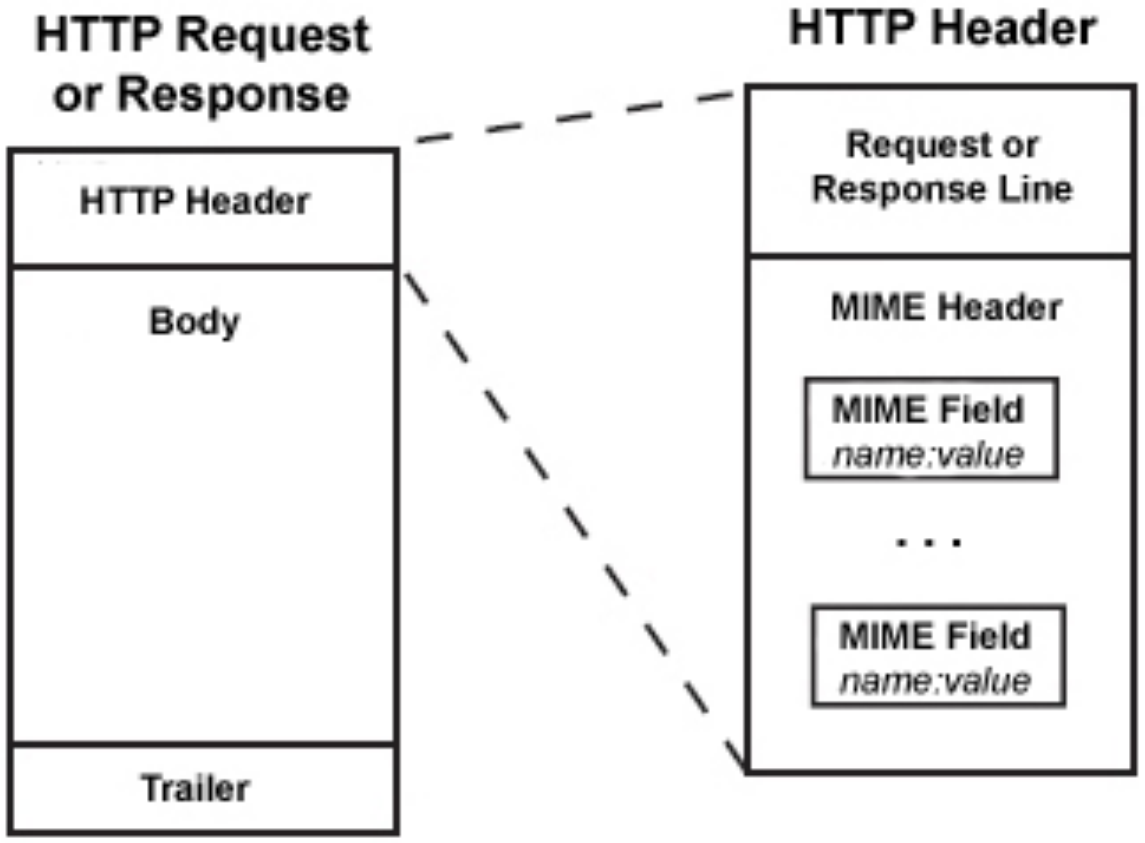
- REST is a generalization of the way the Web works

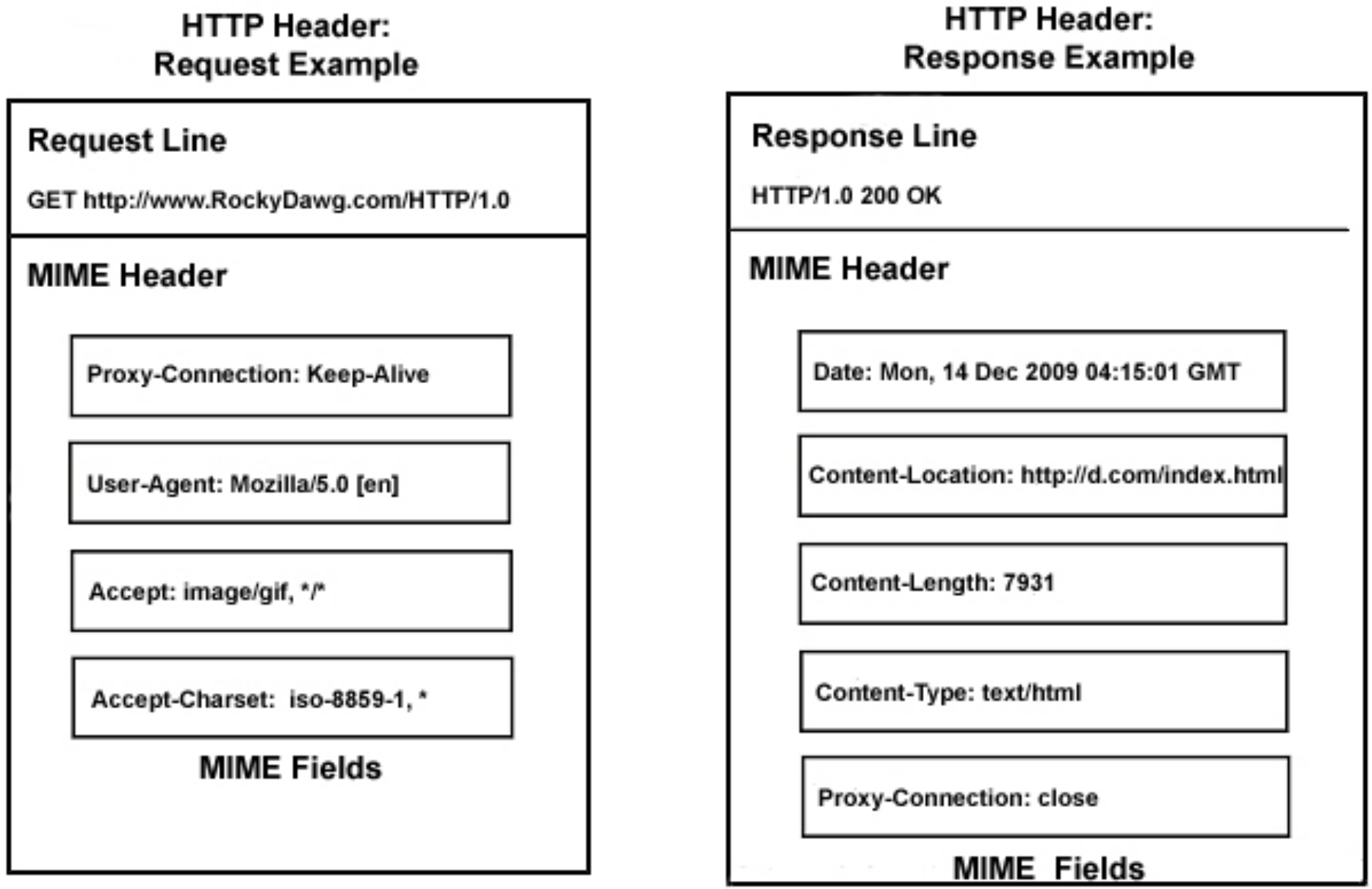


Generalize this for machine-to-machine.

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





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

REST and HTTP

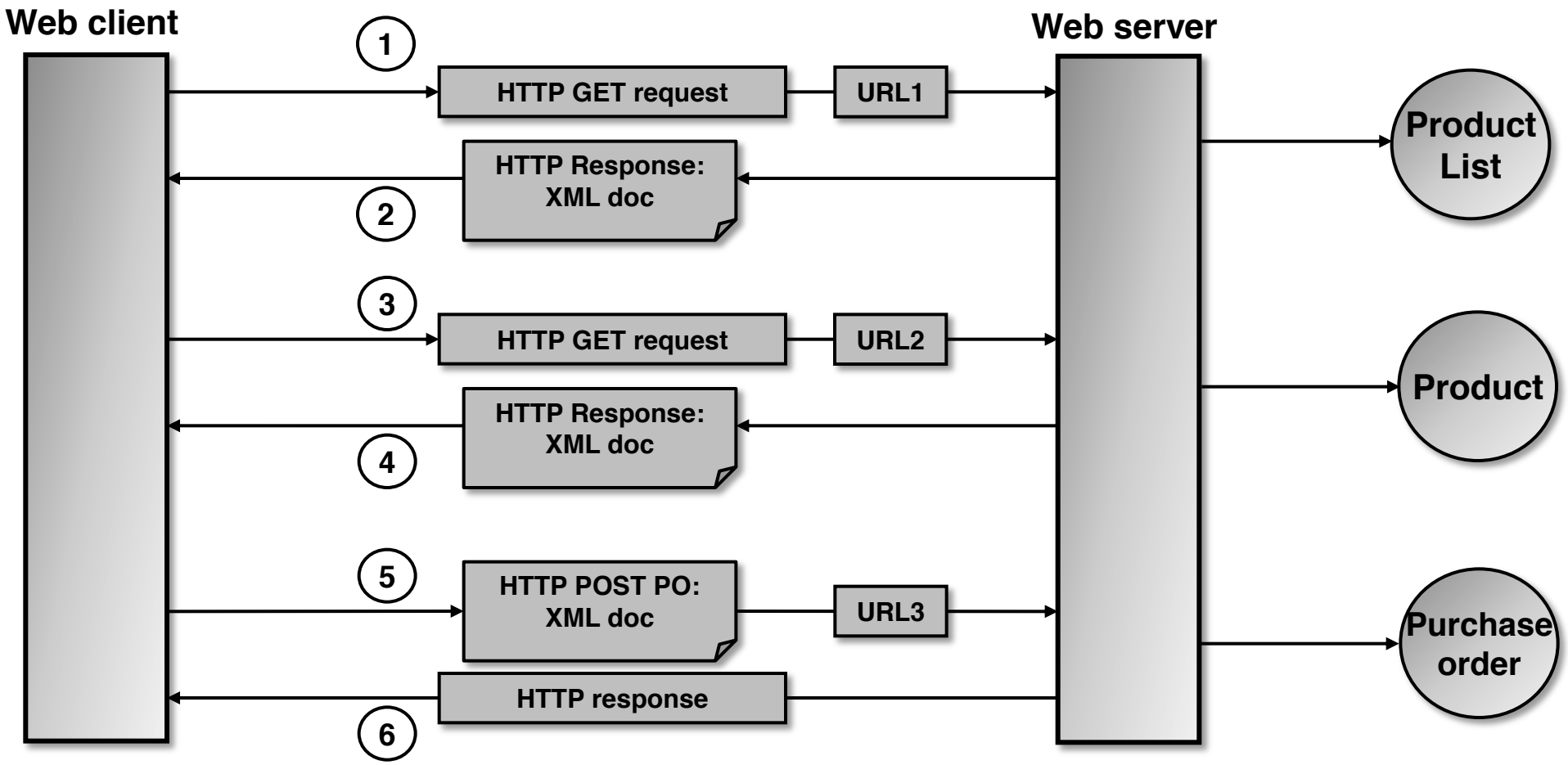
- HTTP is a formally specified protocol for network interactions.
- REST is an architectural style that uses HTTP.
  - Because the Web scales
- “Architectural style”
  - Art + technical skill
- The art of REST is to define APIs using HTTP that are beautiful and elegant as well as useful.

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 to change the 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
- Best practice is to use structured, not flat URLs.
- Think of your resources as collections.
- 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
- 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

Some REST Advantages

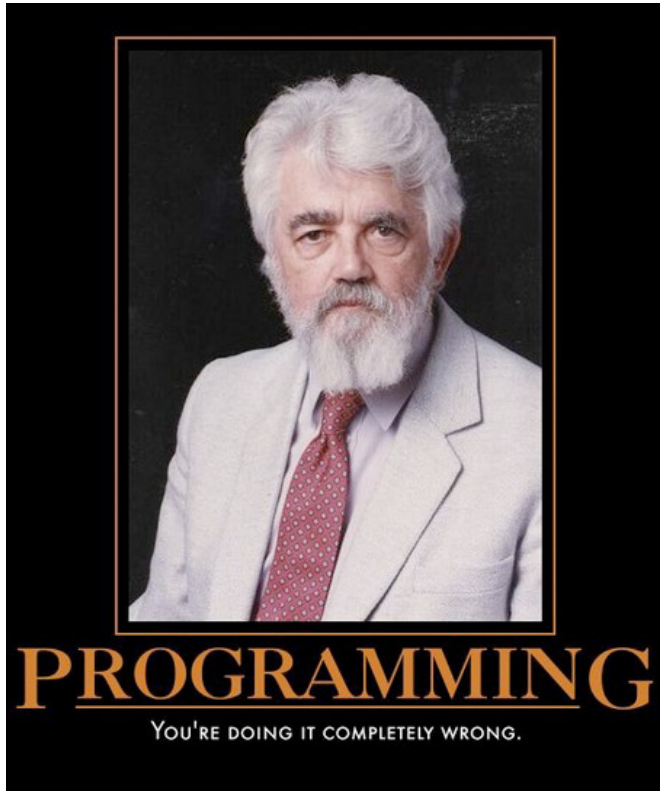
- Leverage 25 years of HTTP investments
  - Security, extensibility, popularity
- Low entry barrier to get people to try your service
  - Use curl command to try things out
- Message format independent
  - Like JSON? Use JSON
  - Like XML? Use XML
  - Like CSV?

REST Challenges

- Data models for your messages
  - 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 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

```
<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

```yaml
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/

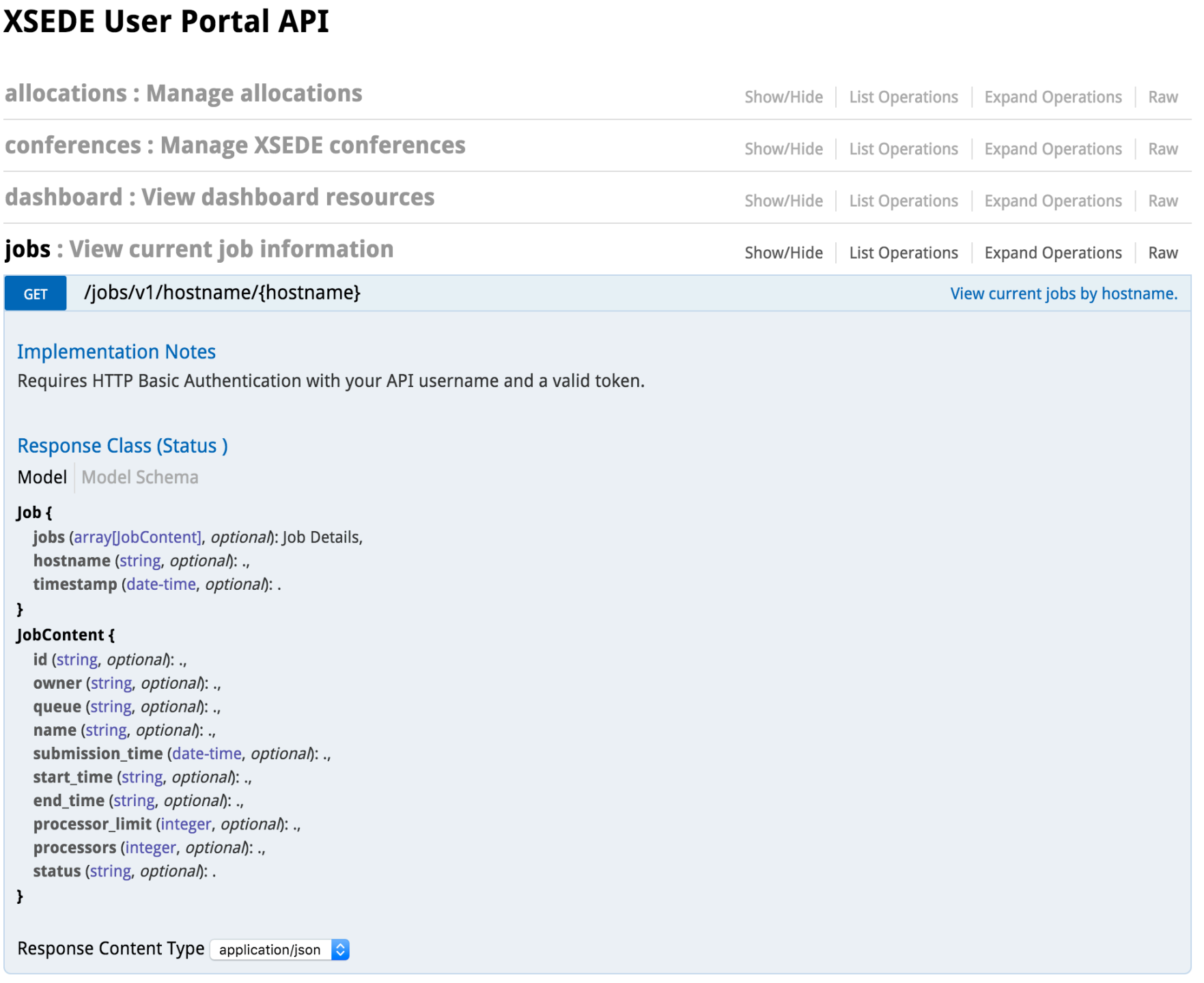
Creating OAI Definitions

- 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/

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.

A Case Study: CIPRES Gateway and REST

Miller, Mark A., Terri Schwartz, Brett E. Pickett, Sherry He, Edward B. Klem, Richard H. Scheuermann, Maria Passarotti, Seth Kaufman, and Maureen A. O'Leary. "A RESTful API for Access to Phylogenetic Tools via the CIPRES Science Gateway." *Evolutionary bioinformatics online* 11 (2015): 43.

There are highly-evolved legacy desktop/browser applications that help with matrix assembly, but have no tree inference tools or are under powered:

MORPHOBANK
HOMOLOGY OF PHENOTYPES OVER THE WEB

MESQUITE A modular system for evolutionary analysis
W.P. Maddison University of British Columbia
D.R. Maddison Oregon State University

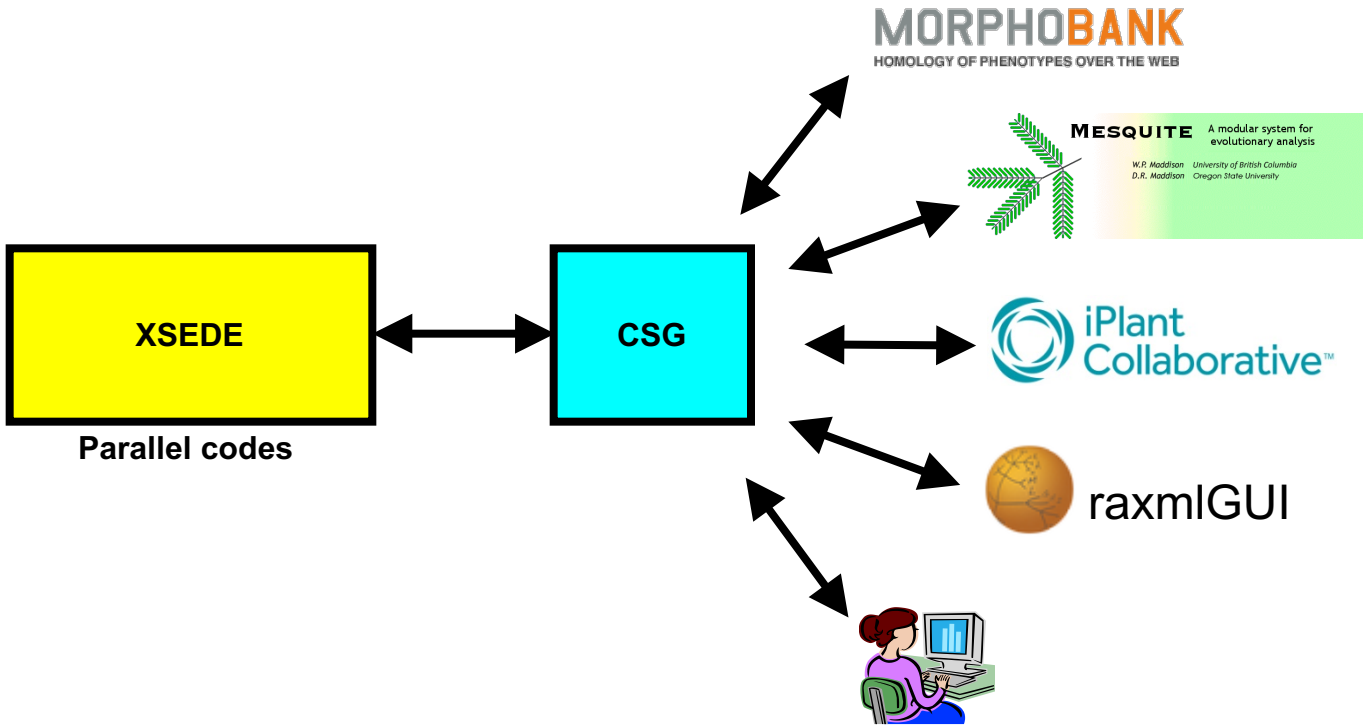
raxmlGUI

ViPR Virus Pathogen Resource

IRD Influenza Research Database

CIPRES slides courtesy of Mark Miller, SDSC

We received funding to create a public CIPRES RESTful API (CRA) to help with these use cases….



CIPRES

Cyberinfrastructure for Phylogenetic Research

MORPHOBANK

HOMOLOGY OF PHENOTYPES OVER THE WEB

Home | Browse Projects | FAQ | In the News | Documentation | Ask Us

Log In | Register

Home

Building the Tree of Life with phenotypes

FOR SCIENTISTS
Use the Tools

FOR SCIENTISTS & THE PUBLIC
See Published Research

Comparative biologists at work with these tools now....

SEE TOTAL ACTIVITY

103	1553	145282	6789	25653/1410	13286/174	81210/88
SCIENTISTS WORKING	SITE VISITORS	CELLS SCORED	IMAGES UPLOADED	PROJECT	MATRIX	MEDIA

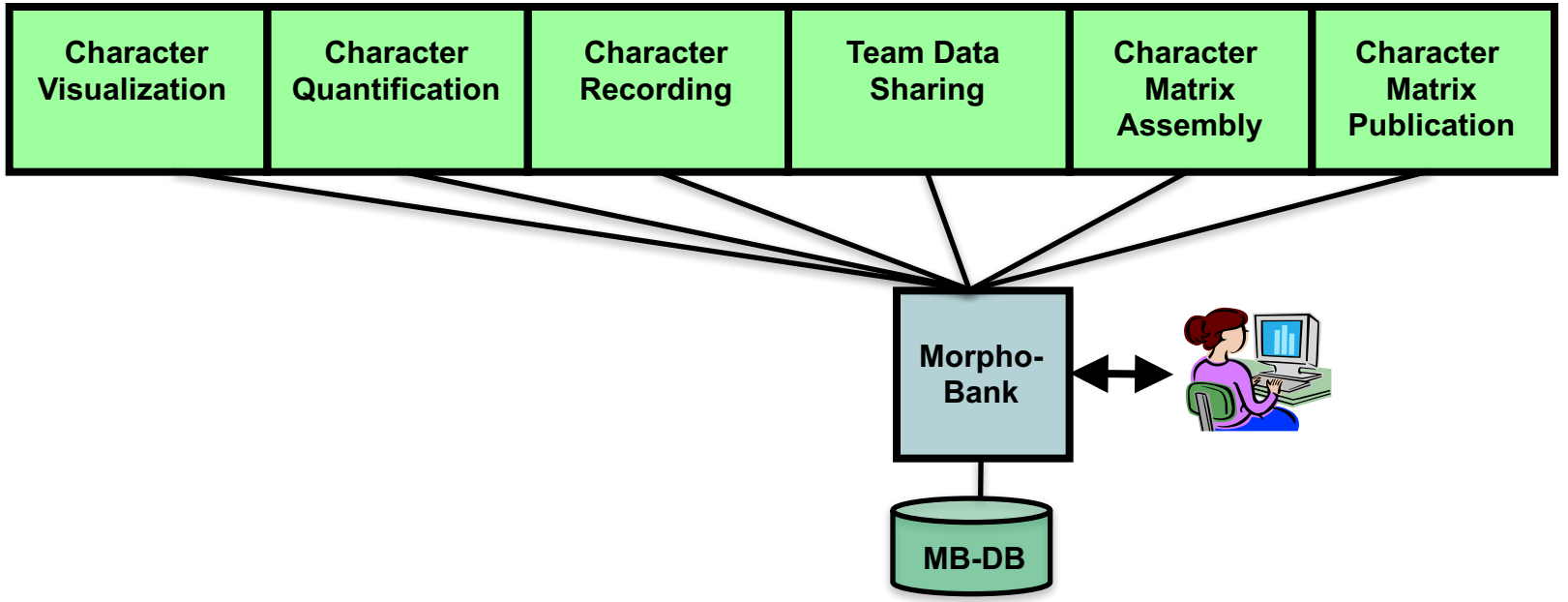
XSEDE

Extreme Science and Engineering Discovery Environment

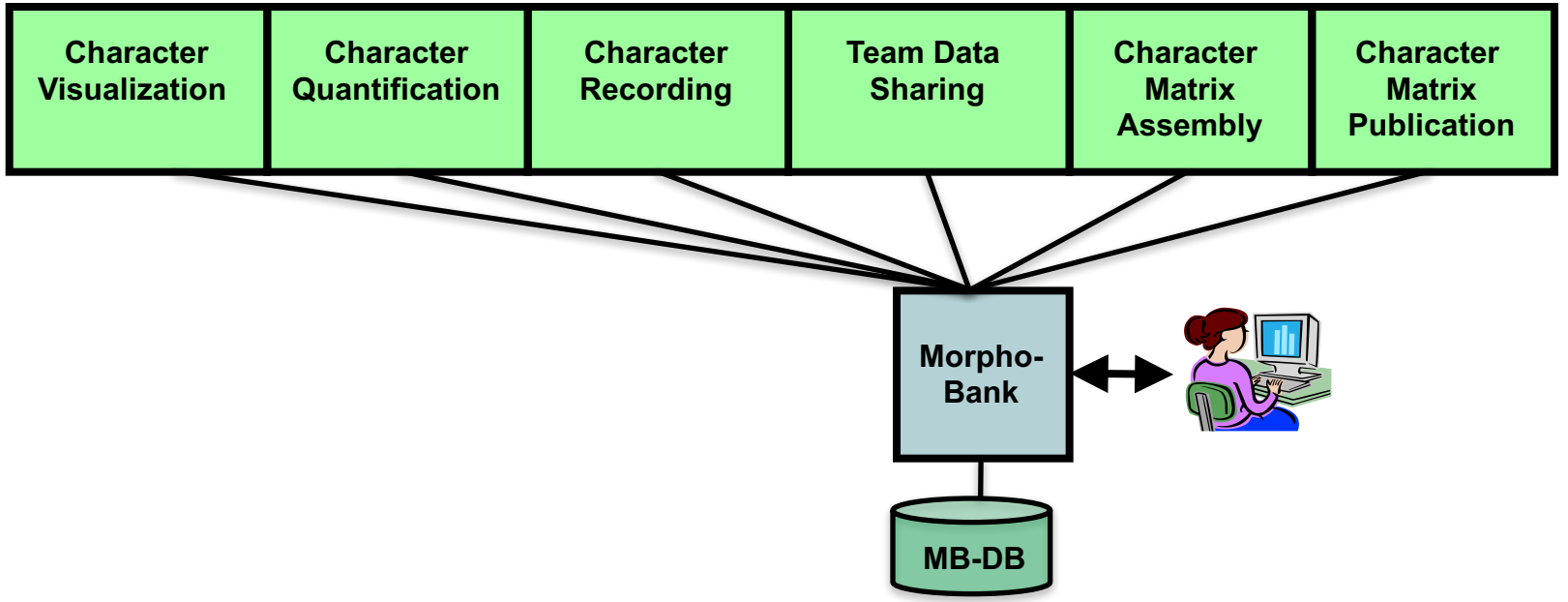
SDSC

Use Cases: MorphoBank and REST Services

MorphoBank provides powerful visual tools for creating and sharing data matrices among large teams……

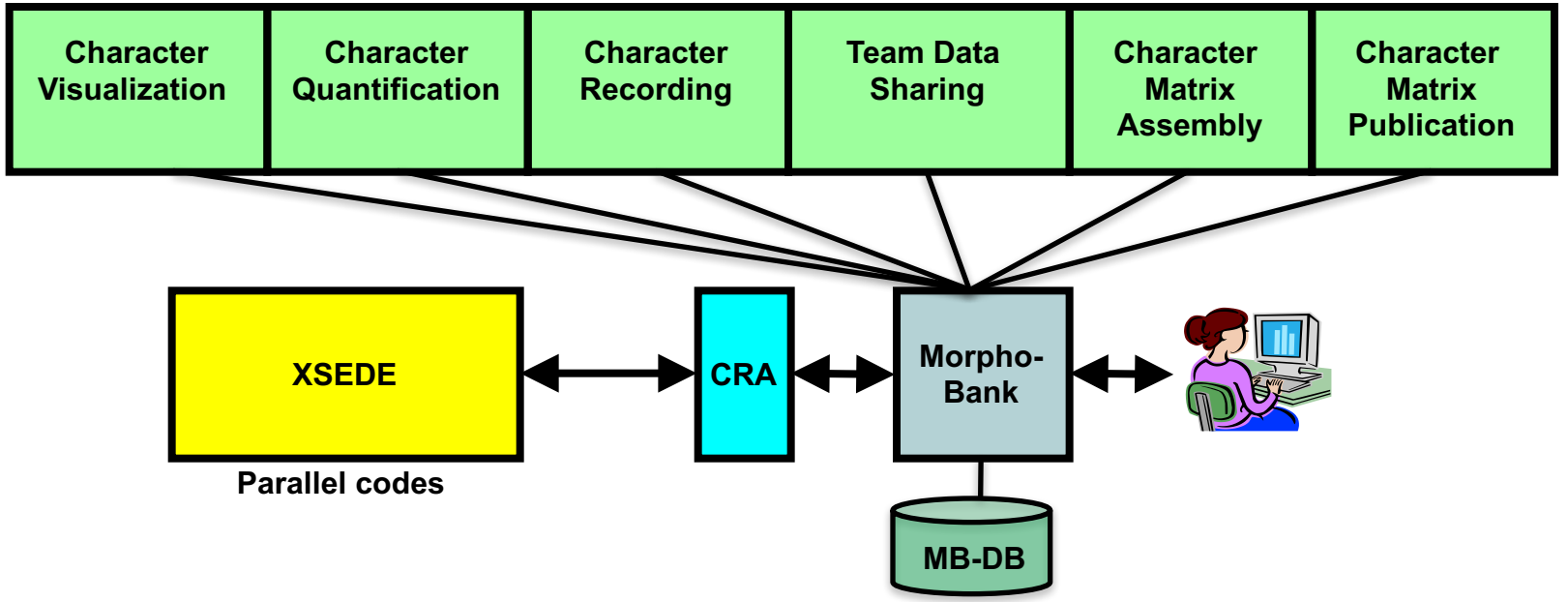


Use Cases: MorphoBank and REST Services



But its has no concept of trees or tree inference……

Use Cases: MorphoBank and REST Services



CIPRES RESTful API allows users to proceed with their workflow within the MorphoBank environment……

Many advanced developers find the workflow supported by the CIPRES browser too restrictive.

!!!

Use Cases: Individual developers and REST Services

Advanced phylogenetic researchers want:

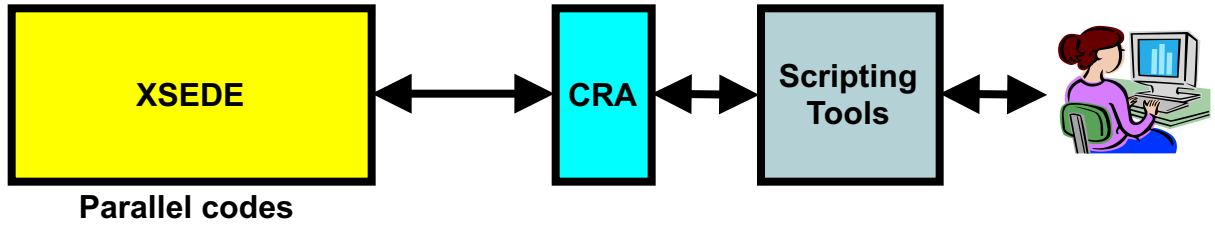
- to run many jobs simultaneously
- create ad hoc workflows

Advanced phylogenetic researchers don't want:

- to assemble and click each job one at a time
- to manually port the output of one job to the subsequent job in their workflow

Use Cases: Individual developers and REST Services

Assuming modest scripting skills, an advanced researcher can accomplish this goal using the CIPRES RESTful API to avoid the clumsy browser interface



iPlant Collaborative™

Advantages of offering REST services:

- **Preserves the investment in creating and learning to use complex software environments.**
- **Makes interaction with the application more flexible for individuals with scripting skills.**

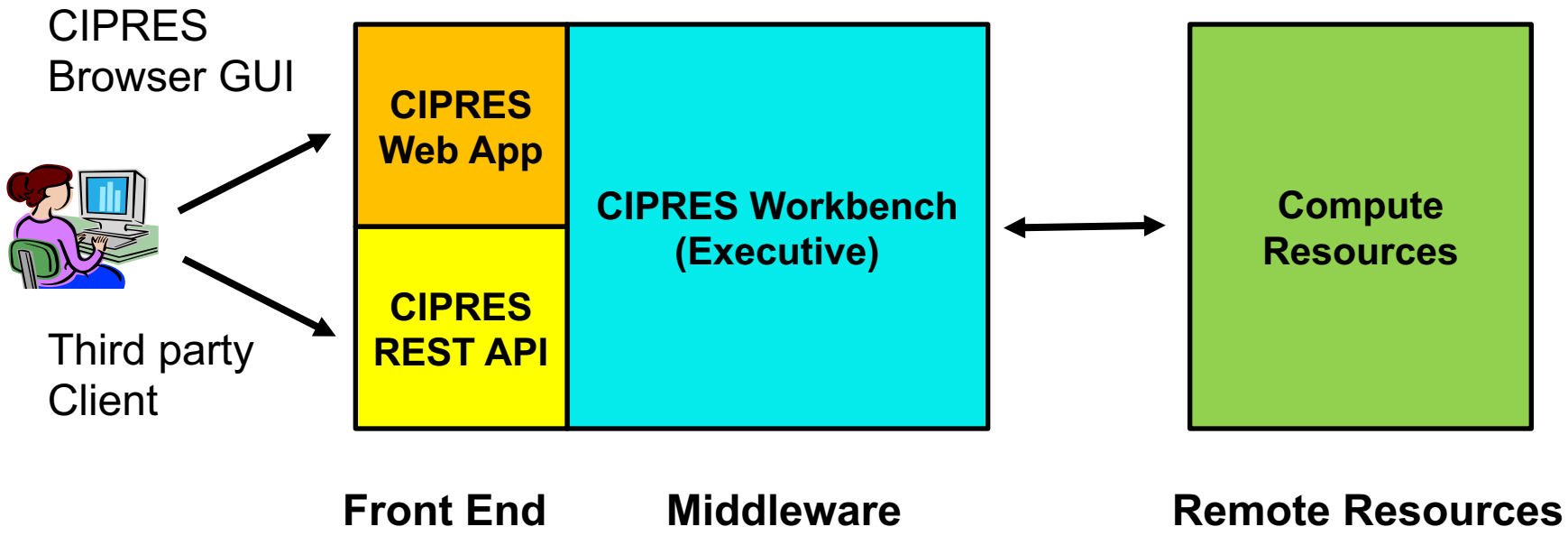
There are several immediate consequences of providing this kind of access:

- Responsibility for the end-user interface is shifted from the CIPRES development group to outside developers.
- Shift in responsibility for provenance and persistence of the user's work products from CIPRES to user.
- Developers of unknown intent and skill level will have access to powerful computing resources.

There are several immediate requirements for providing this kind of access:

- The interface between “outside” developers and the CRA software must be versatile and simple.
- Changes in phylogenetic codes accessed by the CRA must be easy to propagate to client applications.
- Changes in phylogenetic codes must be incorporated strictly at the discretion of the client application.
- Resources must be protected from unintentional (and intentional) abuse.

Basic structure of the CRA



The web application provides a browser GUI based on Struts2, whereas the CRA was created using Jersey.

To manage the requirement for preventing abuse and retaining accountability each job submission must include a unique user ID and a unique application ID

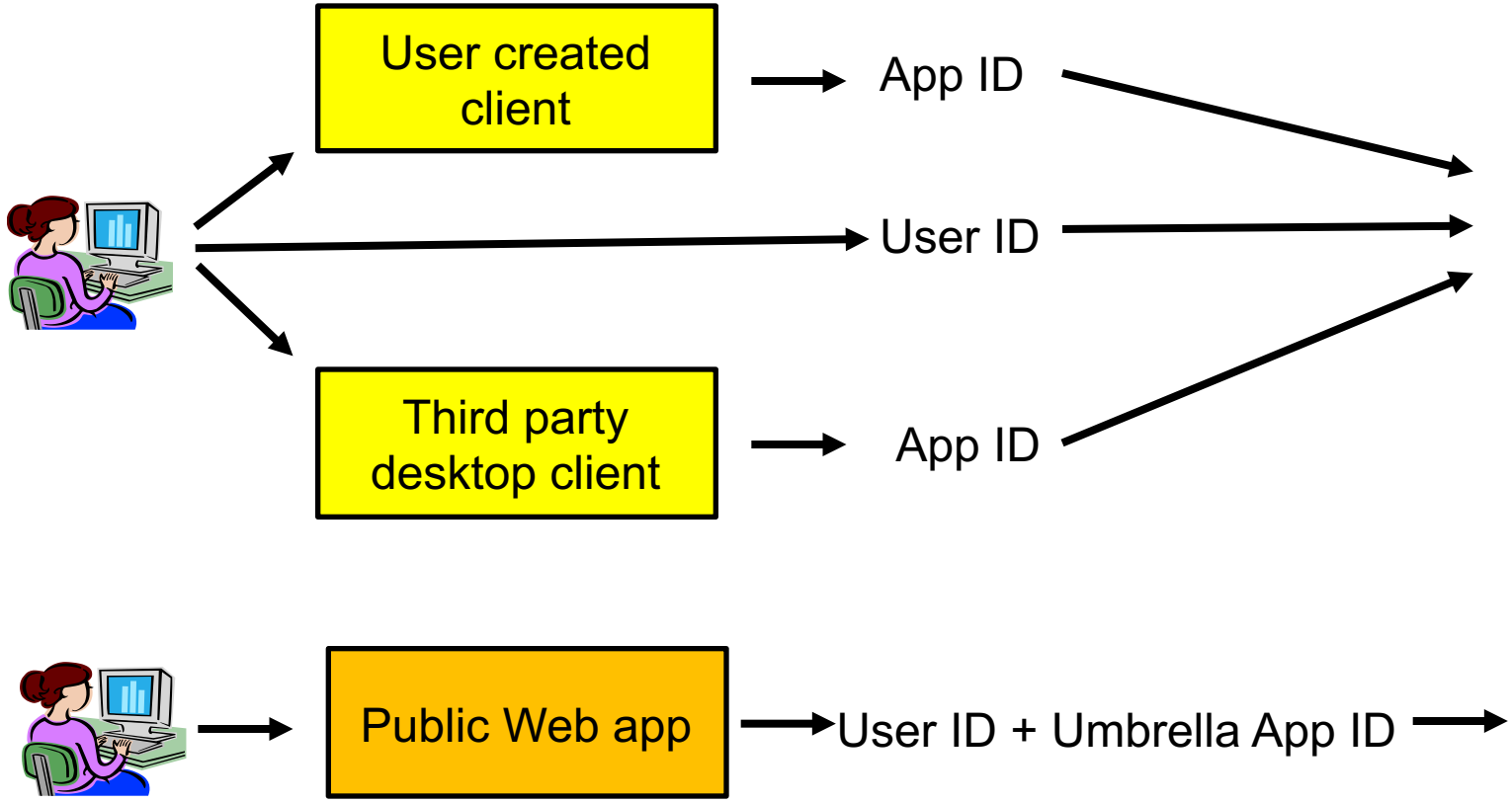
These IDs must be registered either with the CRA, or an application that has a trust relationship with the CRA.

Registration:

There are three specific use cases:

- An individual user/scripter who wishes to submit a job through their own scripts
- An individual wants to submit through a third party software package that can talk to CRA
- A public website that has registered users that access the CRA

User validation/job submission in CRA



The first two use cases require registration of the user and the application they wish to use.

User Registration:

- A user must register at the CRA web site, and provide their institution and country affiliation. This influences how much time they may access.
- Any registered user may create and register an application at the CRA web site.
- Any registered user may use any application that is registered with the CRA.

In the third use case, the CRA uses “umbrella” accounts to support public web sites which have registered user who wish to access the CRA.

A trust relationship exists between CRA and the third party web application, and the third party application provides unique user identifiers as part of the regular job submission. These users do not need to be registered with the CRA.

Applications in this category register and are vetted as “umbrella” providers.

Throttling:

Job submissions are controlled as follows:

- **Any individual user is allowed to submit only a certain number of jobs**
- **Any individual user can only encumber as many core hours as they currently have in their account.**
- **A given application can only submit a certain number of jobs**
- **Submissions by specific users or applications can be blocked**

Other consequences of REST v Browser apps

- The CRA provides asynchronous job submission, which eliminates the latency associated with synchronous browser-based submissions.
- The client app must poll for results if they want to detect changes in state of the job.
- With the Web Ap, CIPRES is responsible for data provenance and archiving; but with the CRA, the user is responsible for both.