INFO/CS 4302 Web Information Systems

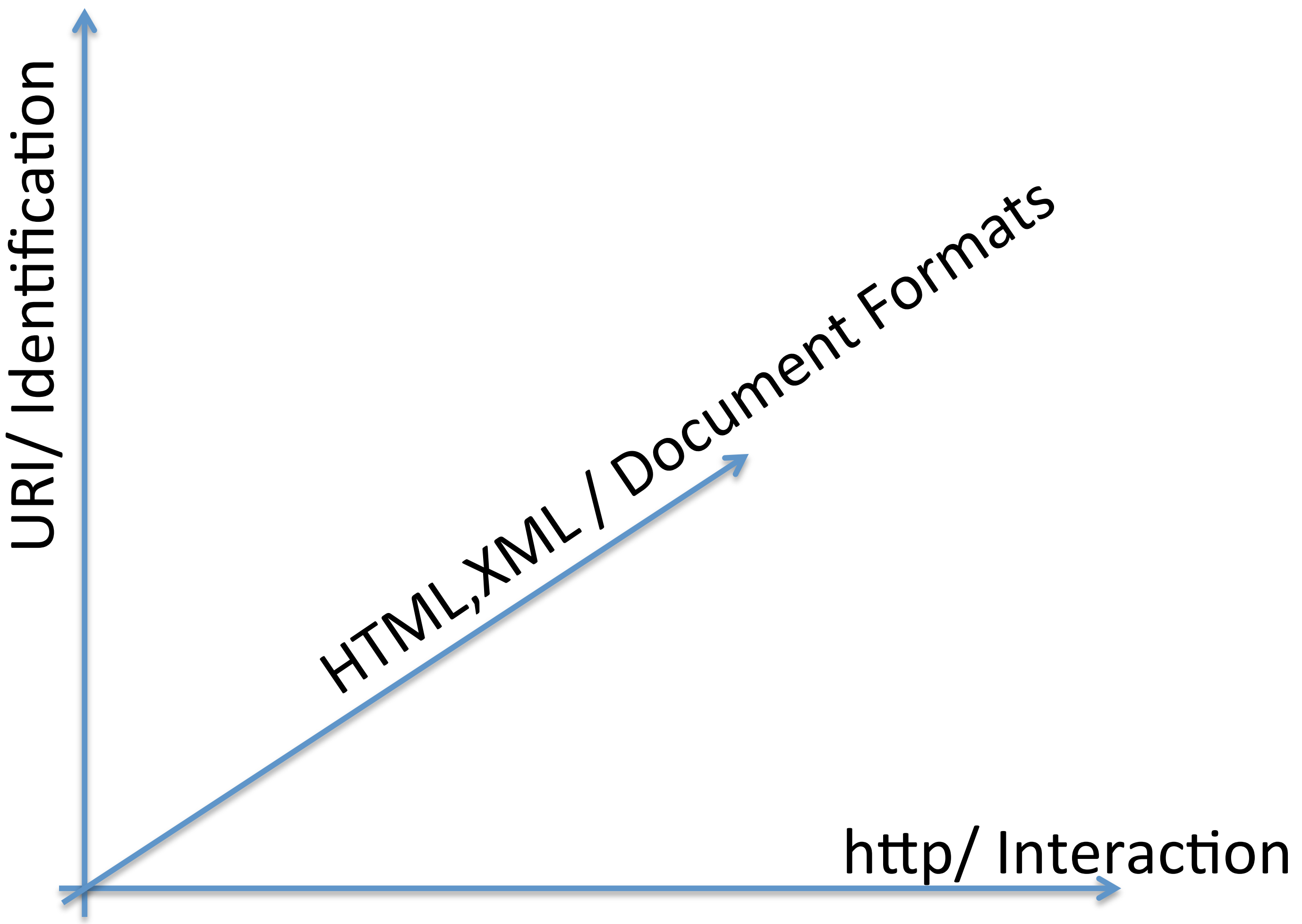
FT 2012
Week 7: RESTful Webservice APIs

- Bernhard Haslhofer -

Web Fundamentals

- Key Architectural Components
  - Identification: URI
  - Interaction: HTTP
  - Standardized Document Formats: HTML, XML, JSON, etc.

Principle ‘Orthogonal Specifications’

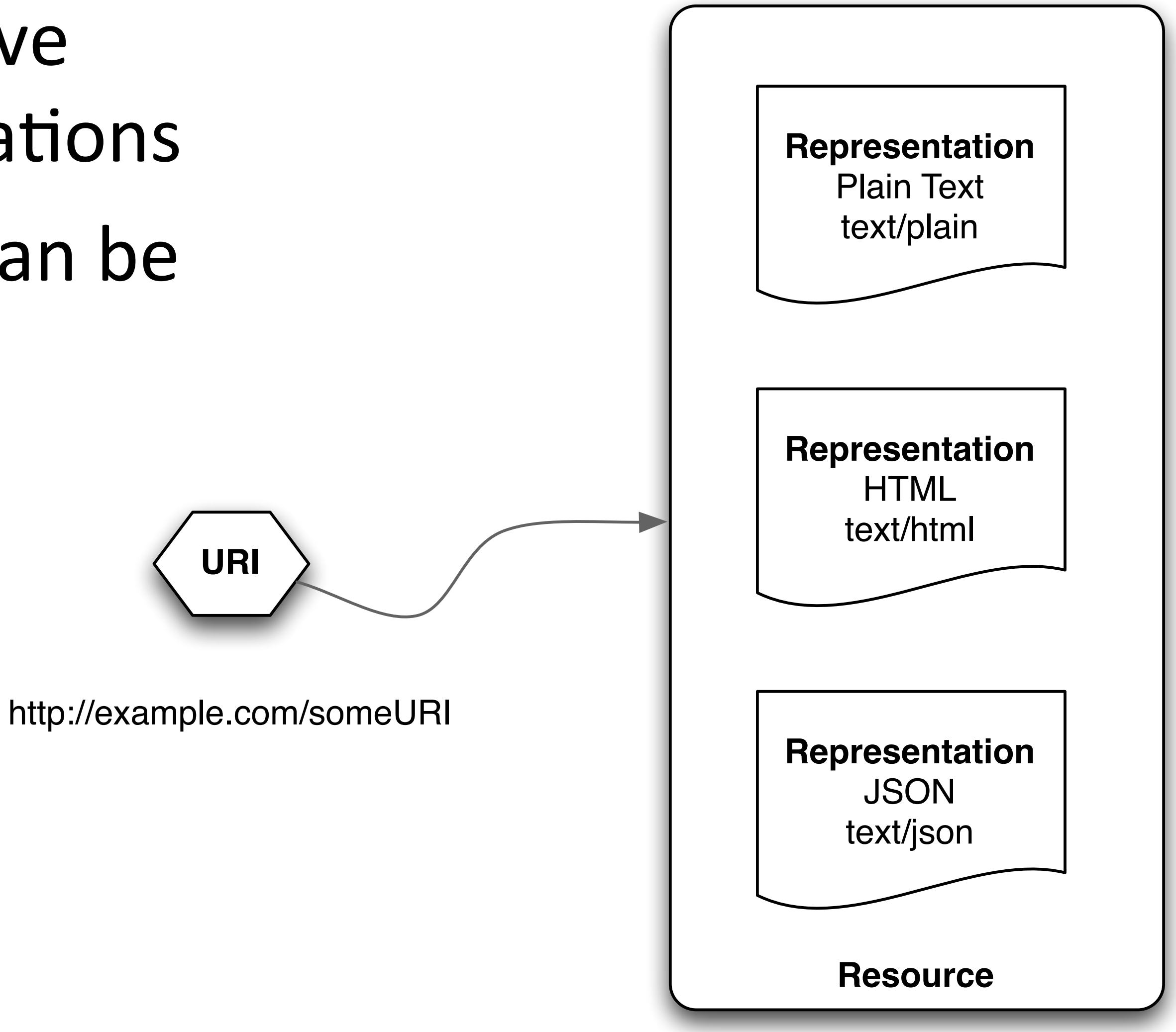


URIs / Resources

- URIs identify interesting things
  - documents on the Web
  - relevant aspects of a data set
- HTTP URIs name and address resources in Web-based systems
  - a URI names and identifies one resource
  - a resource can have more than one name
    - http://foo.com/software/latest
    - http://foo.com/software/v1.4

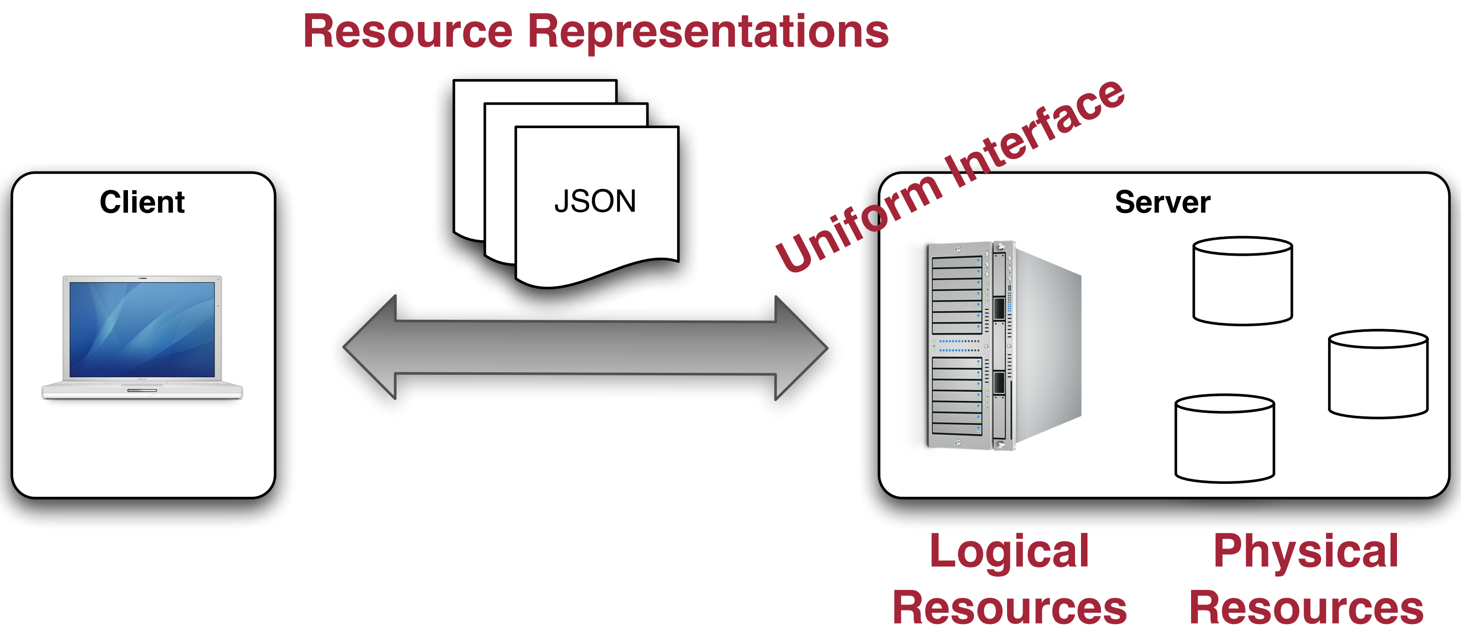
Resource Representation

- A resource can have several representations
- Representations can be in any format
  - HTML
  - XML
  - JSON
  - …

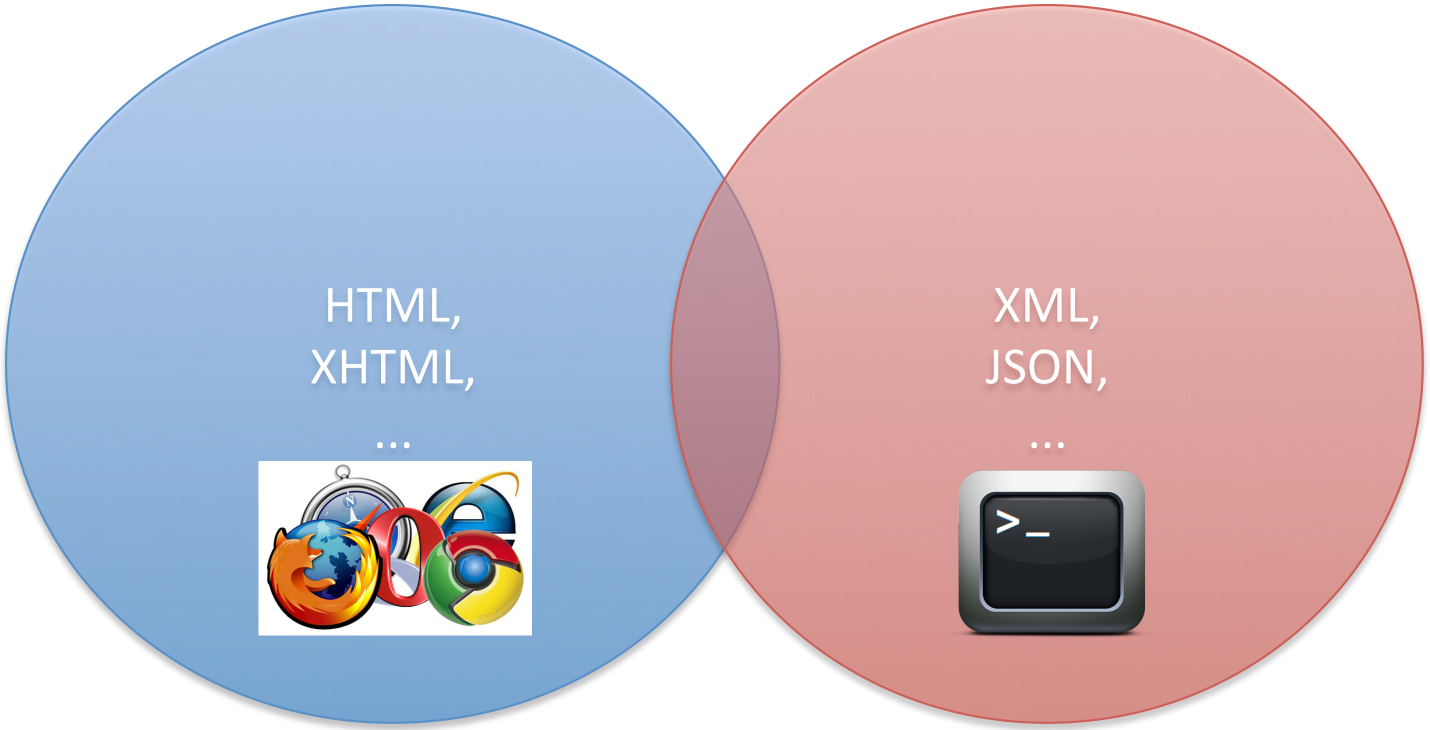


Interacting with Resources

- We deal with resource representations
  - not the resources themselves (pass by value)
  - representations can be in any format (defined by media-type)
- Each resource implements a standard uniform interface (HTTP)
  - a small set of verbs applied to a large set of nouns
  - verbs are universal and not invented on a per-application basis



Document/Data Formats



Display data

Transport and store data

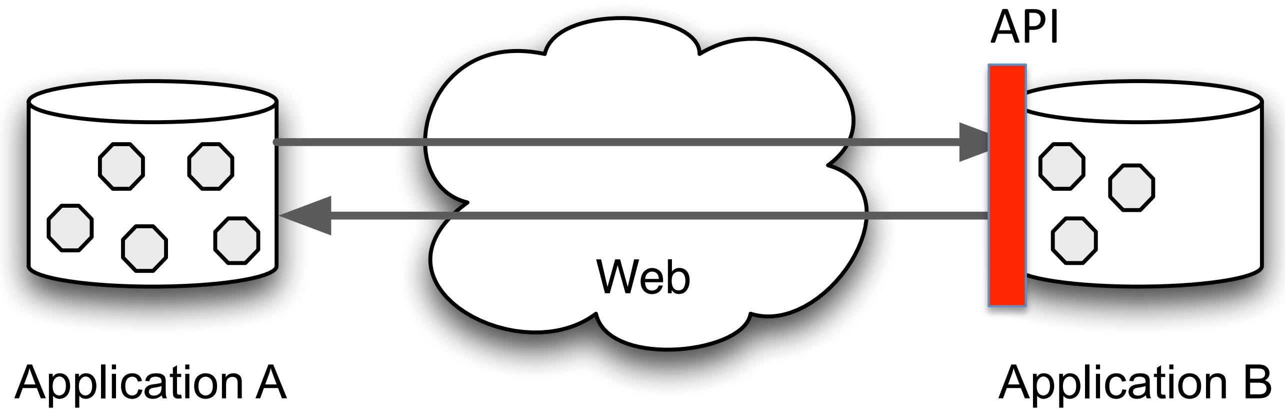
APIS, WEB SERVICES

(Web) APIs

- Application Programming Interface
- Specifies how software components communicate with each other
  - e.g., Java API, 3rd party library APIs
  - usually come with documentation, howtos

- Web API: specify how applications communicate with other over the Web (HTTP, URI, XML, etc.)

Web Services



- Example operations:
  - Publish image on Flickr
  - Order a book at Amazon
  - Post a message on your friend's Facebook wall
  - Update user photo on foursquare

Web Services

- “Web Services” ≅ “Web APIs”
- Build on the design principles and architectural components of the Web
- Provide certain operations
- Exchange structured data in standard formats (JSON, XML, etc)

GROUP BRAINSTORMING

RESTFUL APIS – ARCHITECTURAL PRINCIPLES

The Resource-Oriented Architecture

- A set of design principles for building RESTful Web Services
  - Addressability
  - Uniform interface
  - Connectedness
  - Statelessness

Web Services for the Real World
RESTful
Web Services
O'REILLY
Leonard Richardson & Sam Ruby

Addressability

- An addressable application
  - exposes the interesting aspects of its dataset as resources
  - exposes a URI for every piece of information it might serve
  - which is usually an infinite number of URIs

Addressability

- A resource
  - is anything that is important enough to be referenced as a thing in itself
  - usually something
    - you want to serve information about
    - that can be represented as a stream of bits
      - actors
      - movies
  - a resource must have at least one name (URI)

Addressability

- Resource names (URIs)
  - the URI is the name and address of a resource
  - a resource's URI should be descriptive

http://example.com/movies

instead of

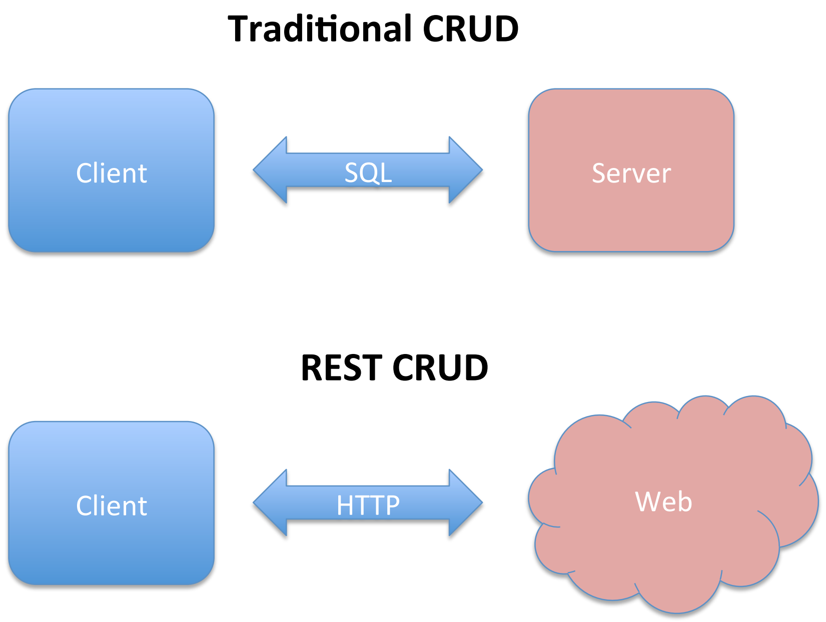
http://example.com/overview.php?list=all,type=movie

Uniform Interface

- The same set of operations applies to everything (every resource)
- A small set of verbs (methods) applied to a large set of nouns (resources)
  - verbs are universal and not invented on a per-application base
- Natural language works in the same way (new verbs rarely enter language)

Uniform Interface

- With HTTP we have all methods we need to manipulate Web resources (**CRUD** interface)
  - **Create** = POST (or PUT)
  - **Read** = GET
  - **Update** = PUT
  - **Delete** = DELETE

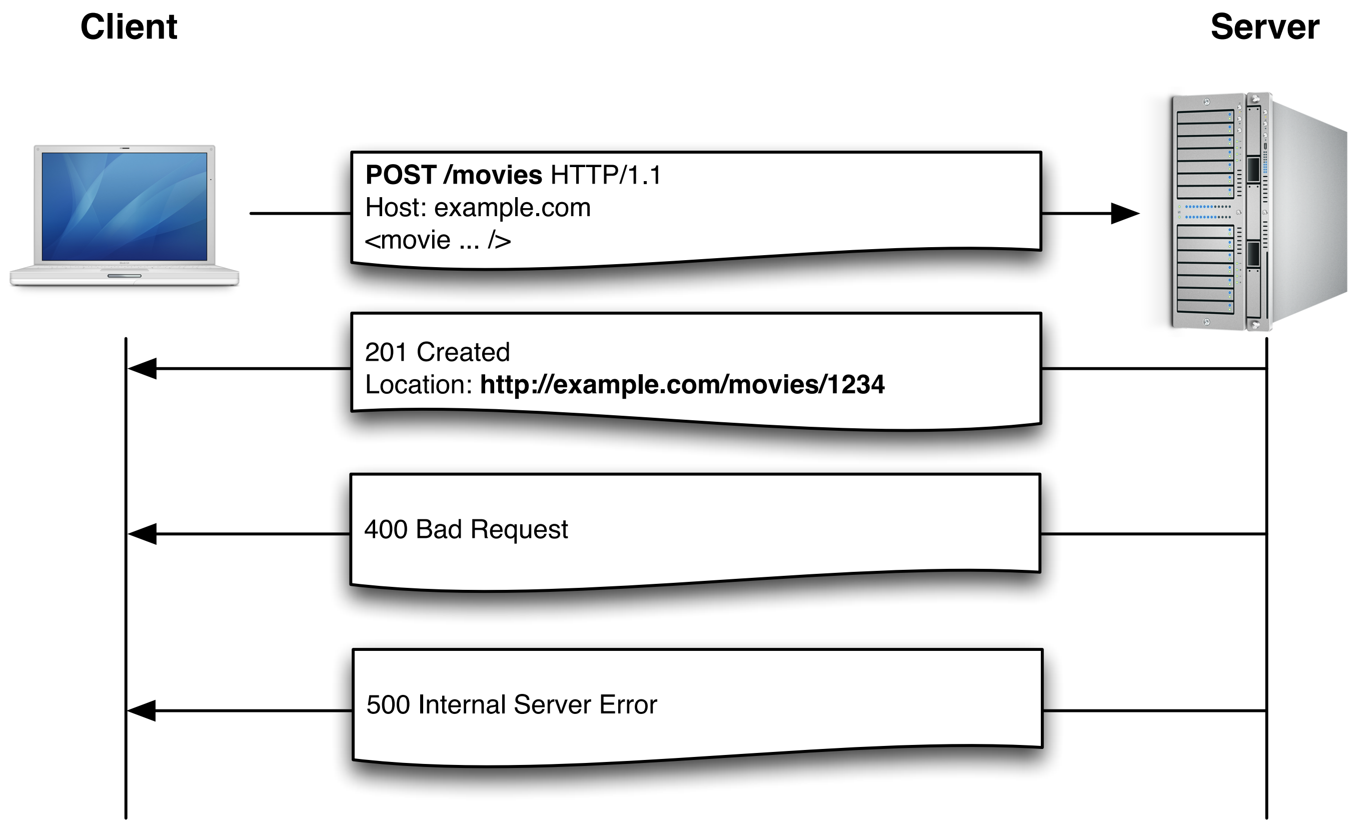


Safe and Idempotent Behavior

- Safe methods can be ignored or repeated without side-effects: GET and HEAD
- Idempotent methods can be repeated without side-effects: PUT and DELETE
- Unsafe and non-idempotent methods should be treated with care: POST

Uniform Interface

- **CREATE** a new resource with HTTP POST

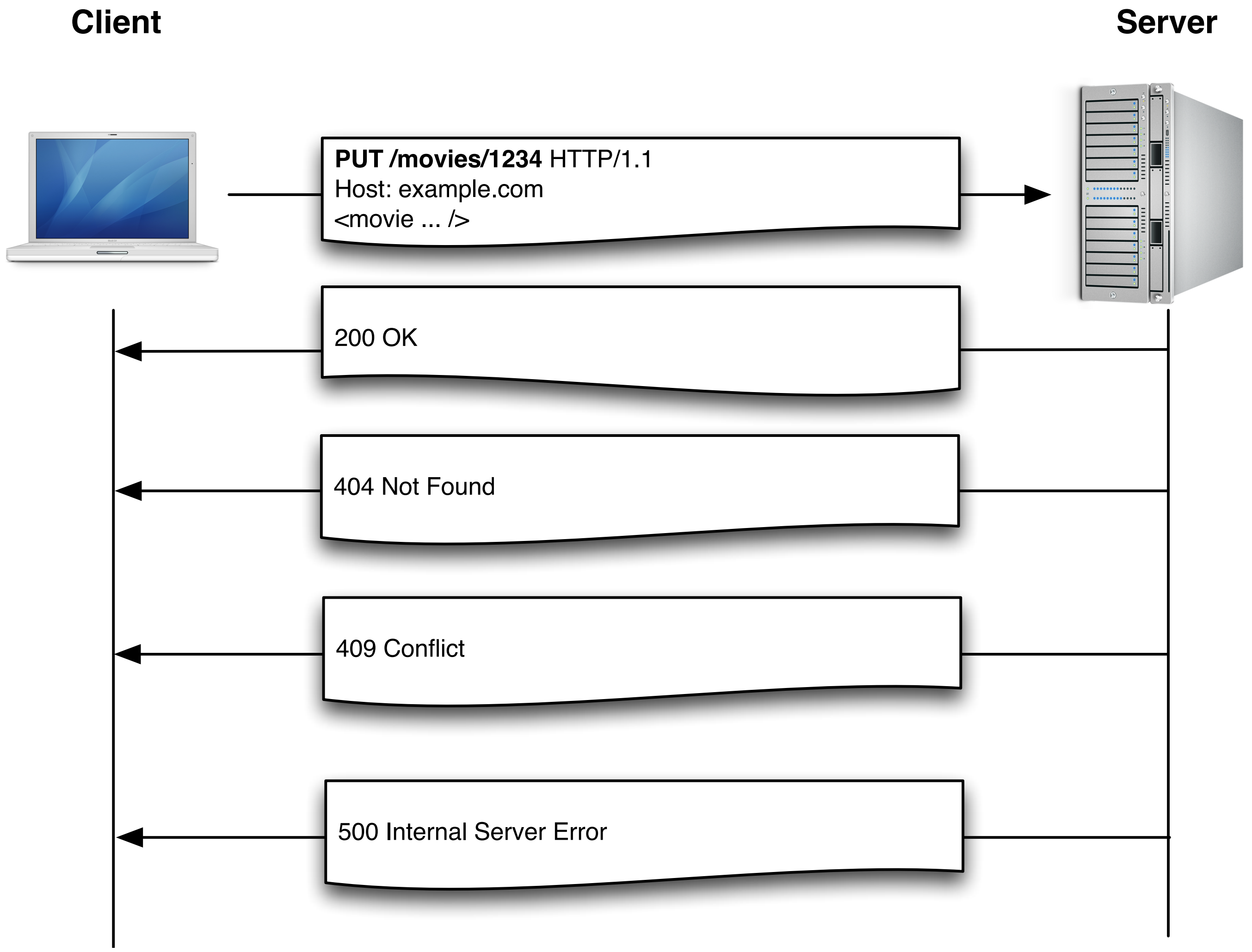


POST Semantics

- POST creates a new resource
- The server decides on the resource’s URI
- POST is **not idempotent**
  - A sequence of two or more POST requests has side-effects
  - Human Web:
    - “Do you really want to post this form again?”
    - “Are you sure you want to purchase that item again?”
  - Programmatic Web:
    - if you post twice, you create two resources

Uniform Interface

- **CREATE** a new resource with HTTP PUT



PUT Semantics

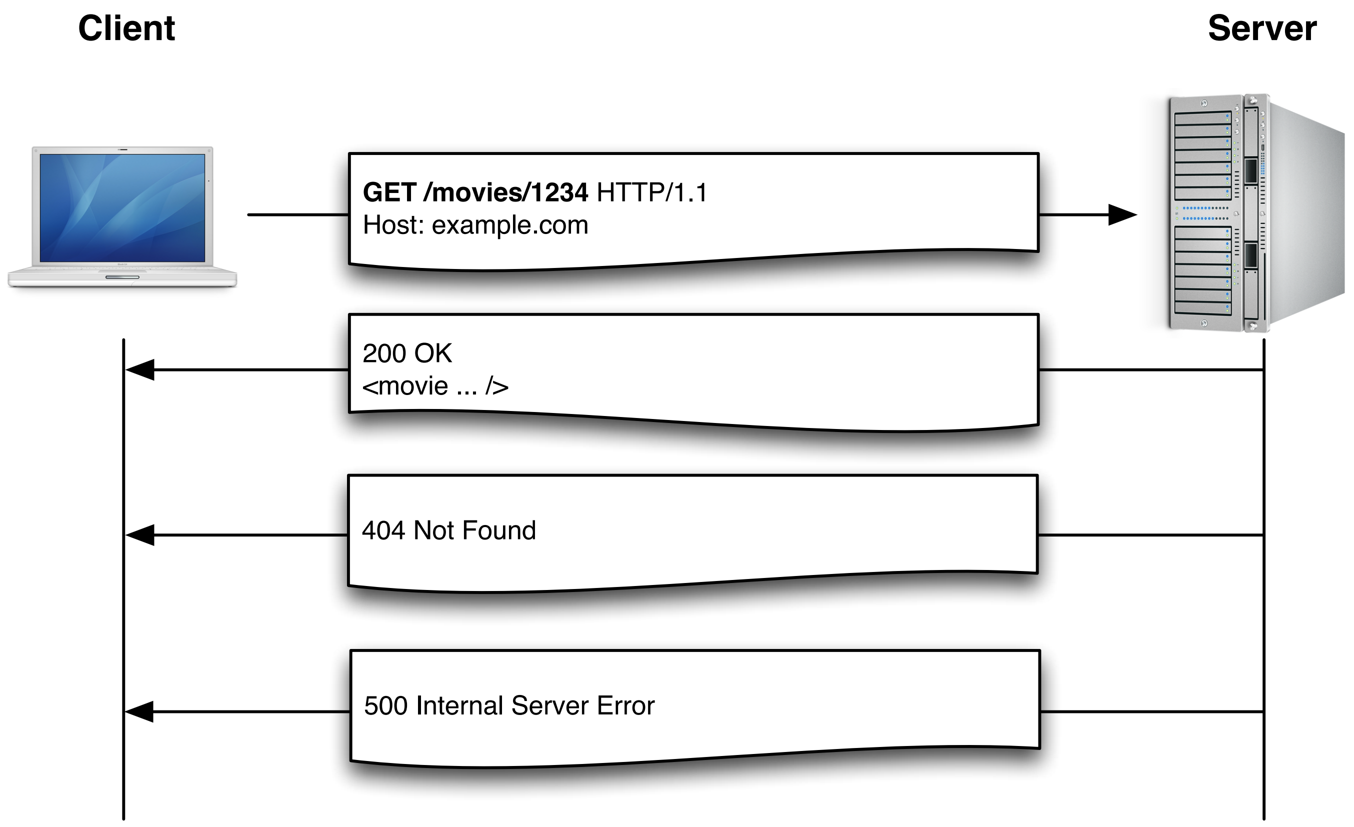
- PUT creates a new resource
- The client decides on the resource’s URI
- PUT is **idempotent**
  – multiple PUT requests have no side effects

Create with PUT or POST?

- The generic answer: it depends ☺
- Considerations
  - PUT if client
    - can decide on the URI
    - sends complete representation to the server
  - POST if server creates the URI (algorithmically)

Uniform Interface

- **READ** an existing resource with HTTP GET

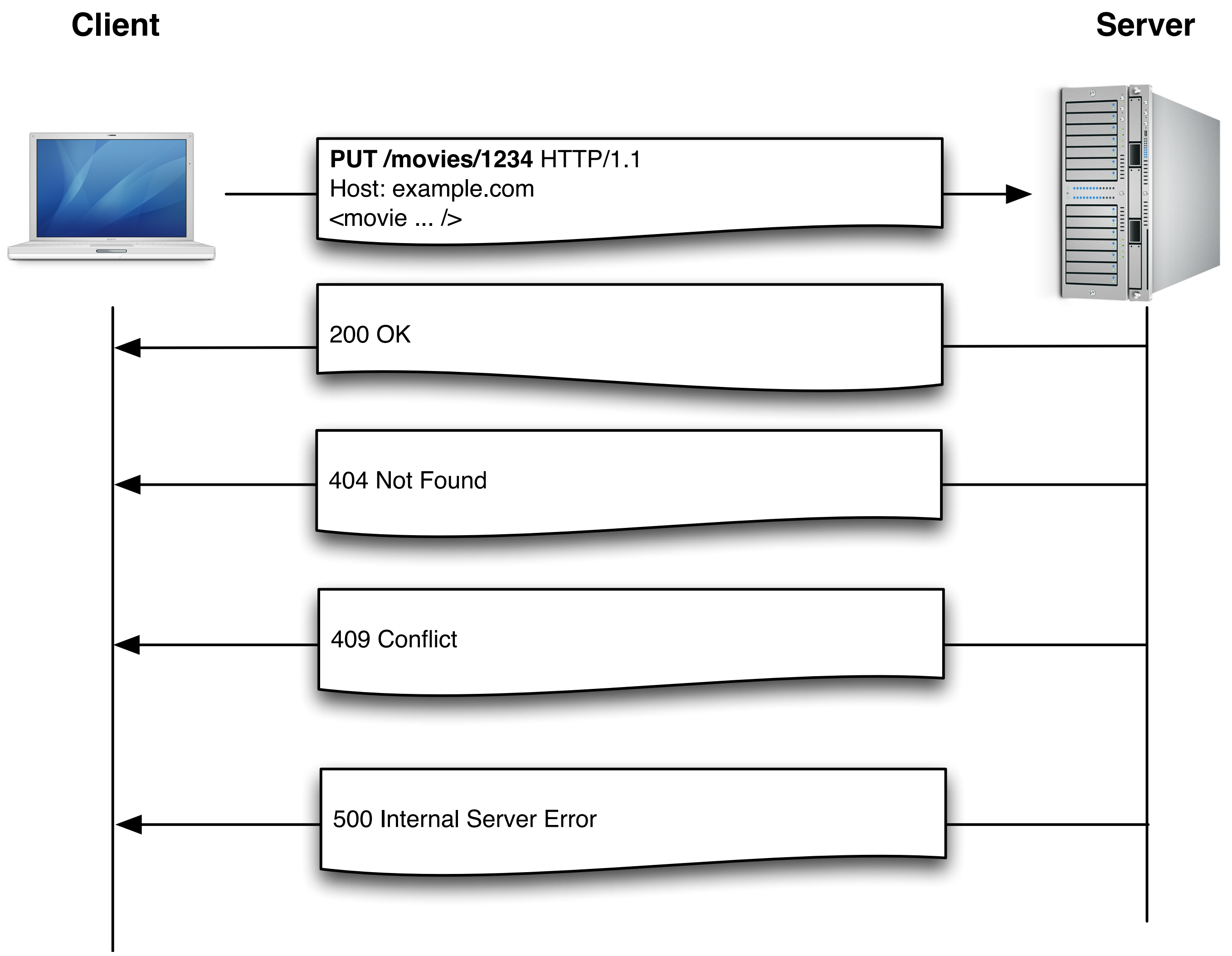


GET Semantics

- GET retrieves the representation (= the current state) of a resource
- GET is safe (implies idempotent)
  - does not change state of resource
  - has no side-effects
- If GET goes wrong
  - GET it again!
  - no problem because it safe (and idempotent)

Uniform Interface

- **UPDATE** an existing resource with HTTP PUT

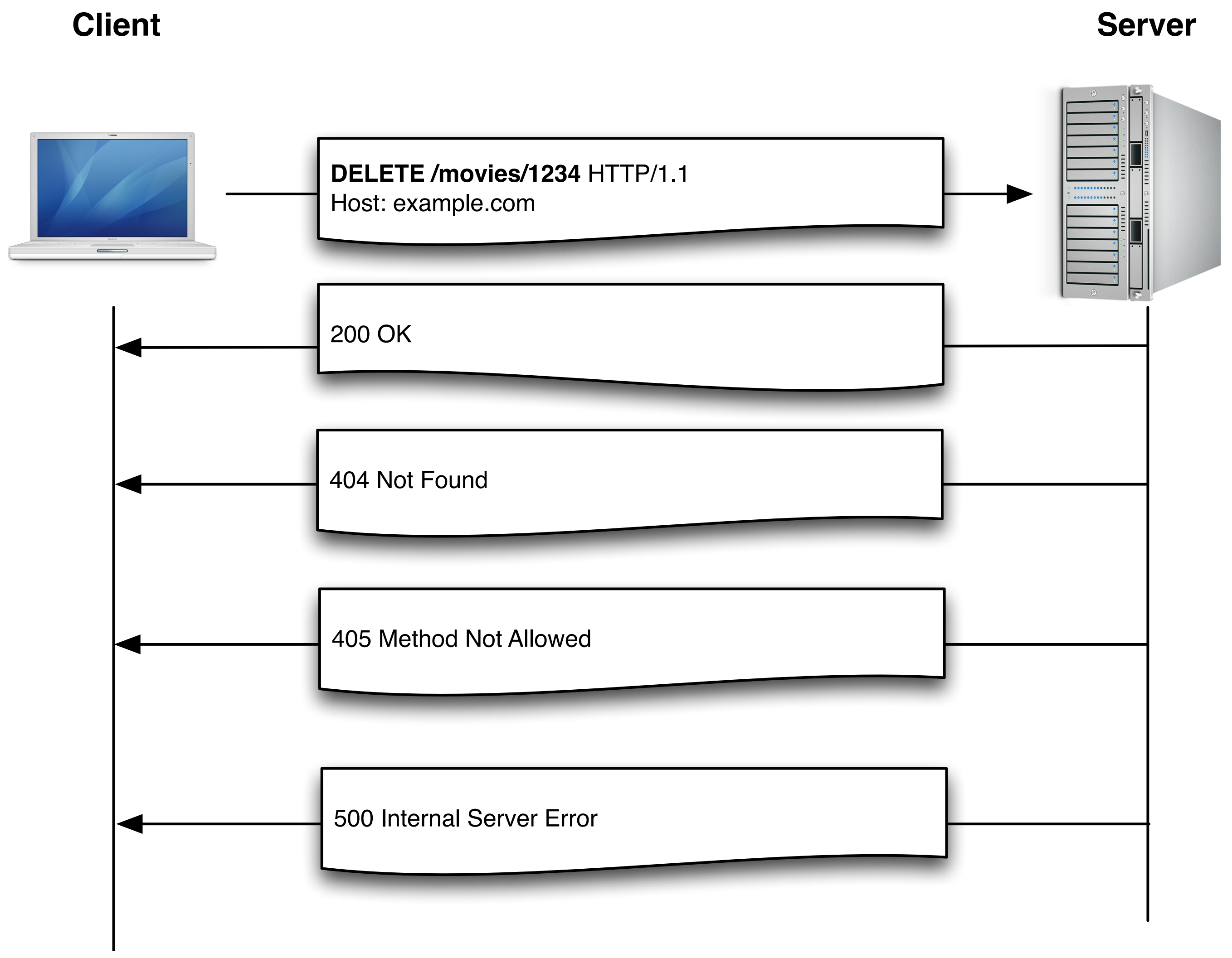


When PUT goes wrong

- If we get 5xx error, or some 4xx errors
  - simply PUT again!
  - no problem, because PUT is idempotent
- If we get errors indicating incompatible states then do some forward/backward compensation work and maybe PUT again
  - 409 Conflict (e.g., change your username to a name that is already taken)
  - 417 Expectation Failed (the server won't accept your representation – fix it, if possible)

Uniform Interface

- **DELETE** an existing resource with HTTP DELETE



DELETE Semantics

- Stop the resource from being accessible
  - logical delete
  - not necessarily physical
- If DELETE goes wrong
  - try it again!
  - DELETE is idempotent

Connectedness

- In RESTful services, resource representations are hypermedia
- Served documents contain not just data, but also links to other resources

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

<?xml...>
<movie>
    <title>The Godfather</title>
    <synopsis>...</synopsis>
    <actor>http://example.com/actors/567</actor>
</movie>
```

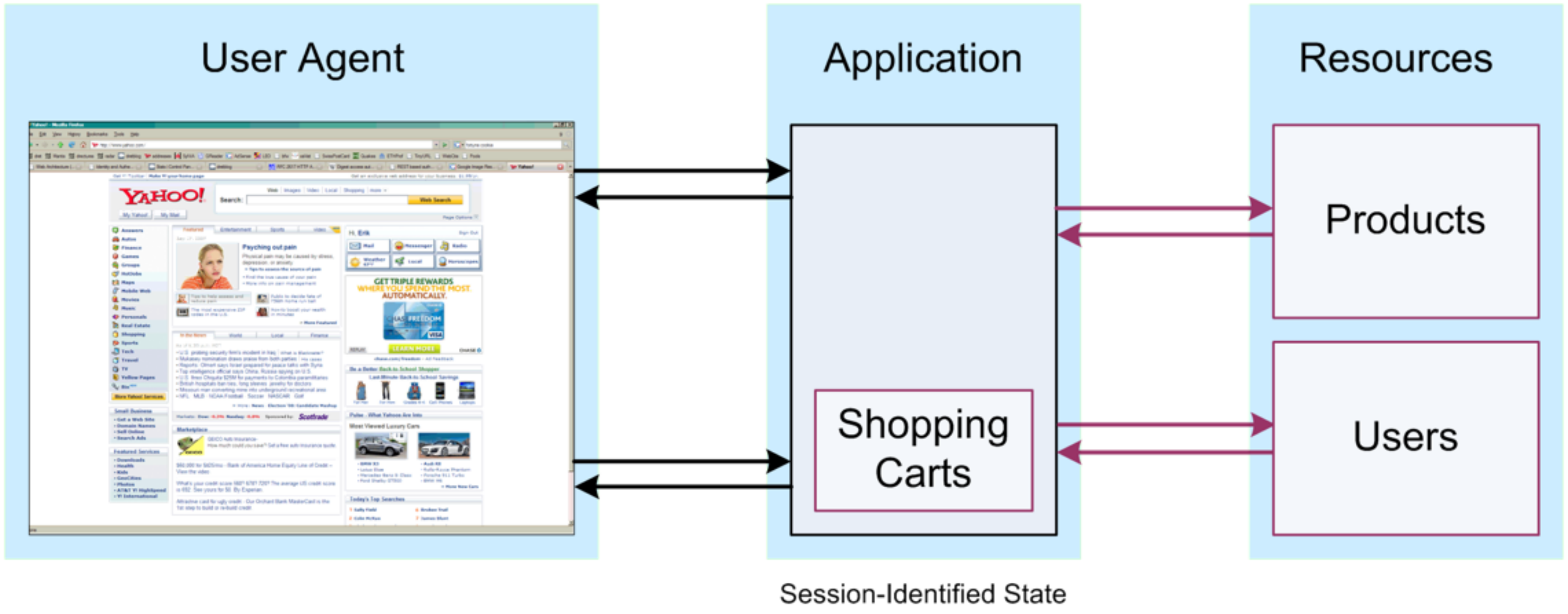
Statelessness

- Statelessness = every HTTP request executes in complete isolation
- The request contains all the information necessary for the server to fulfill that request
- The server never relies on information from a previous request
  - if information is important (e.g., user-authentication), the client must send it again

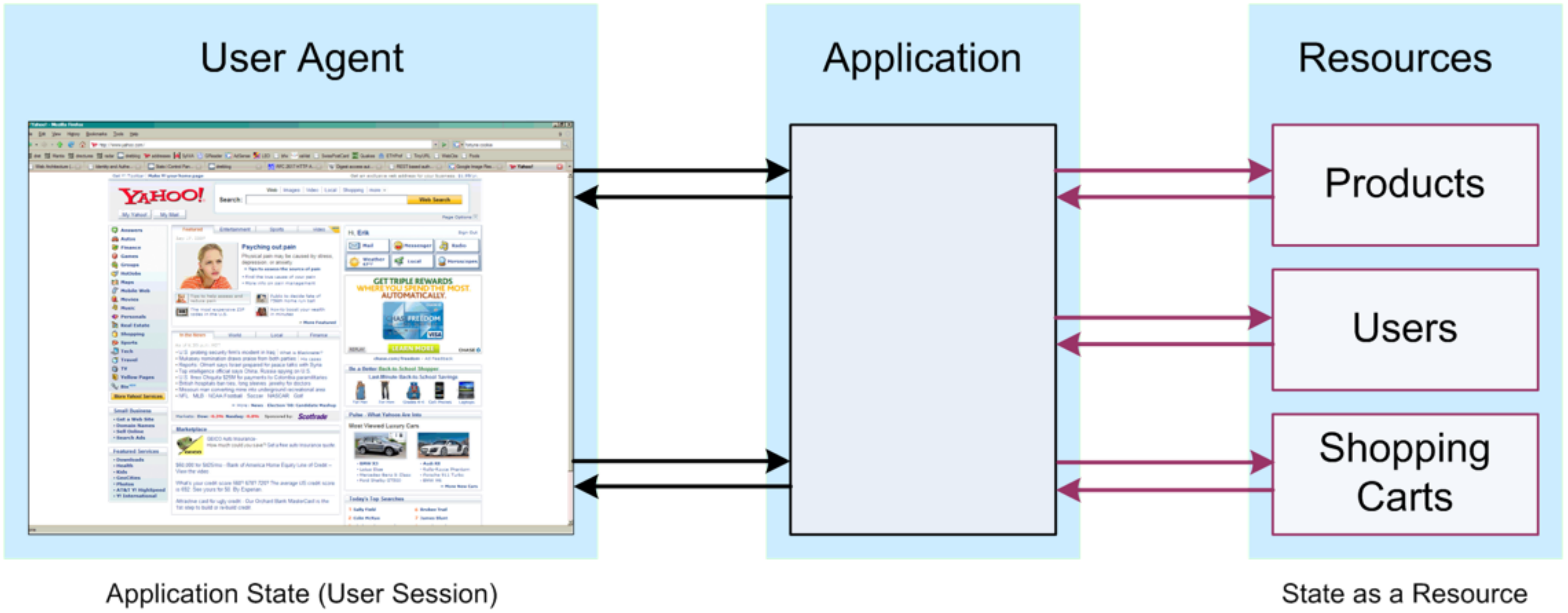
Statelessness

- This constraint does not say “stateless applications”!
  - for many RESTful applications, state is essential
  - e.g., shopping carts
- It means to move state to clients or resources
- State in resources
  - the same for every client working with the service
  - when a client changes resource state other clients see this change as well
- State in clients (e.g., cookies)
  - specific to client and has to be maintained by each client
  - makes sense for maintaining session state (login / logout)

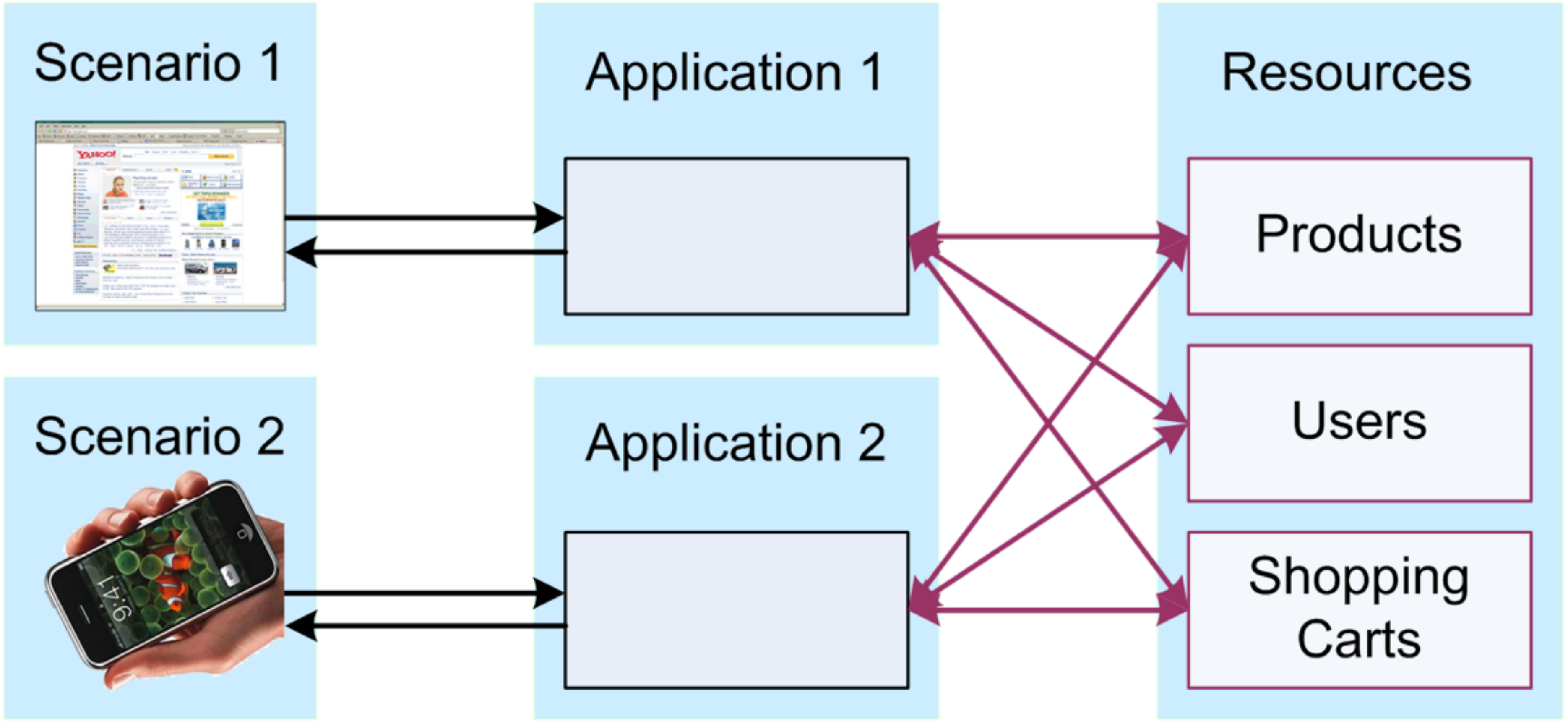
State in the Application



Statelessness



Statelessness



© Erik Wilde: http://dret.net/netdret/docs/rest-icwe2010/

RESTFUL SERVICE DESIGN – IN BRIEF

Design Methodology

- Identify and name resources to be exposed by the service
  - actors and movies
- Model relationships between resources that can be followed to get more details
  - an actor can play in several movies
  - several actors are playing in a movie
- Define “nice” URIs to address the resources

Design Methodology

- Map HTTP verbs to resources
  - e.g., GET movie, POST movie, etc...
- Design and document resource representations

- Implement and deploy Web Service
- Test with cURL or browser developer tools

REST API Design Principles

- Make application developer as successful as possible
- Primary design principle: .“...maximize developer productivity and success“ (Mulloy)
- Keep simple things simple
- Take the developer‘s point of view

Nouns are good; verbs are bad

- Simple and intuitive base URLs
  - /actors
  - /peopleplayingin80iesmovies
- 2 base URLs per resource
  - /actors (collection)
  - /actors/1234 (specific element in collection)
- Keep verbs out of your base URLs
  - /getAllActors

Simplify associations

- Relationships can be complex
  - movie -> actor -> pets -> ...
  - URL levels can become deep
- In most cases URL level shouldn‘t be deeper than: resource/identifier/resource
  - /actor/1234/movies
  - /movies/1234/actors

Filtering

...sweep complexity behind the ?

```
/actors?gender=male&age=50
```

Handling Errors

- Use HTTP status codes
  - over 70 are defined; most APIs use only subset of 8-10
- Start by using
  - 200 OK (...everything worked)
  - 400 Bad Request (..the application did sth. wrong)
  - 500 Internal Server Error (...the API did sth. wrong)
- If you need more, add them
  - 201 Created, 304 Not Modified, 401 Unauthorized, 403 Forbidden, etc..

Handling Errors

- Make messages returned in HTTP body as verbose as possible

```
{"developerMessage" : "Verbose, plain
language description of the problem for
the app developer with hints about how to
fix it.”,

"userMessage":"Pass this message on to the
app user if needed.”,

"errorCode" : 12345,

"more info": http://example.com/errors/
12345”}
```

Versioning

- Never release an API without a version
- Suggested syntax
  - put version number in first path element
  - ‚v‘ prefix
  - simple ordinal number
  - /v1/actors
- Maintain at least one version back

Pagination

- It‘s almost always a bad idea to return every available resource
- Use limit and offset to allow pagination
  - /movies?limit=20&offset=0
- Include metadata about total number of resources in representation

Actions not dealing with resources

- Certain API calls don‘t send resource responses
  - calculate
  - translate
  - convert
- Use verbs and make it clear in the docs
- /convert?from=EUR&to=USD&amount=100

API subdomain

- Consolidate all API requests under one API subdomain
  - api.example.com
- Developer portal (documenation, etc...)
  - developer.example.com
- Web redirects
  - e.g., redirect browser requests to developer portal