Microsoft

REST API Design

Crystal Tenn
Crystal.Tenn@microsoft.com

Why build REST APIs?

Broad range of clients from disparate platforms can consume HTTP services

Benefits of REST

Scalability

Simplicity

Evolvability

Heterogeny

Efficiency

Reliability

Portability

Visibility

Manageability

Performance

What is !REST?

- An architecture that is not SOAP
- Not a URI style
- Not a standard
- In theory - Not HTTP

Compare REST and RPC

Feature	RPC	REST
Contract	Service and its operations	Uniform interface
Actions	Specified separately using something like WSDL	Specified by the uniform interface. Hypermedia used to move through the workflow
Errors	Specified out of band	Specified by the uniform interface
Caching	Optional and not guaranteed	Supported at each layer
URLs	Client knows the URL prior to deployment	Server determined.
Inputs and Outputs	Tied to underlying runtime types	Tied to the media type specification
Protocol	Multiple protocols	Tied to the protocol of the uniform interface

What about HTTP?

- HTTP provides a uniform interface
- The uniform interface includes all the REST constraints

So, in practice…

- REST is all about HTTP

What is REST?

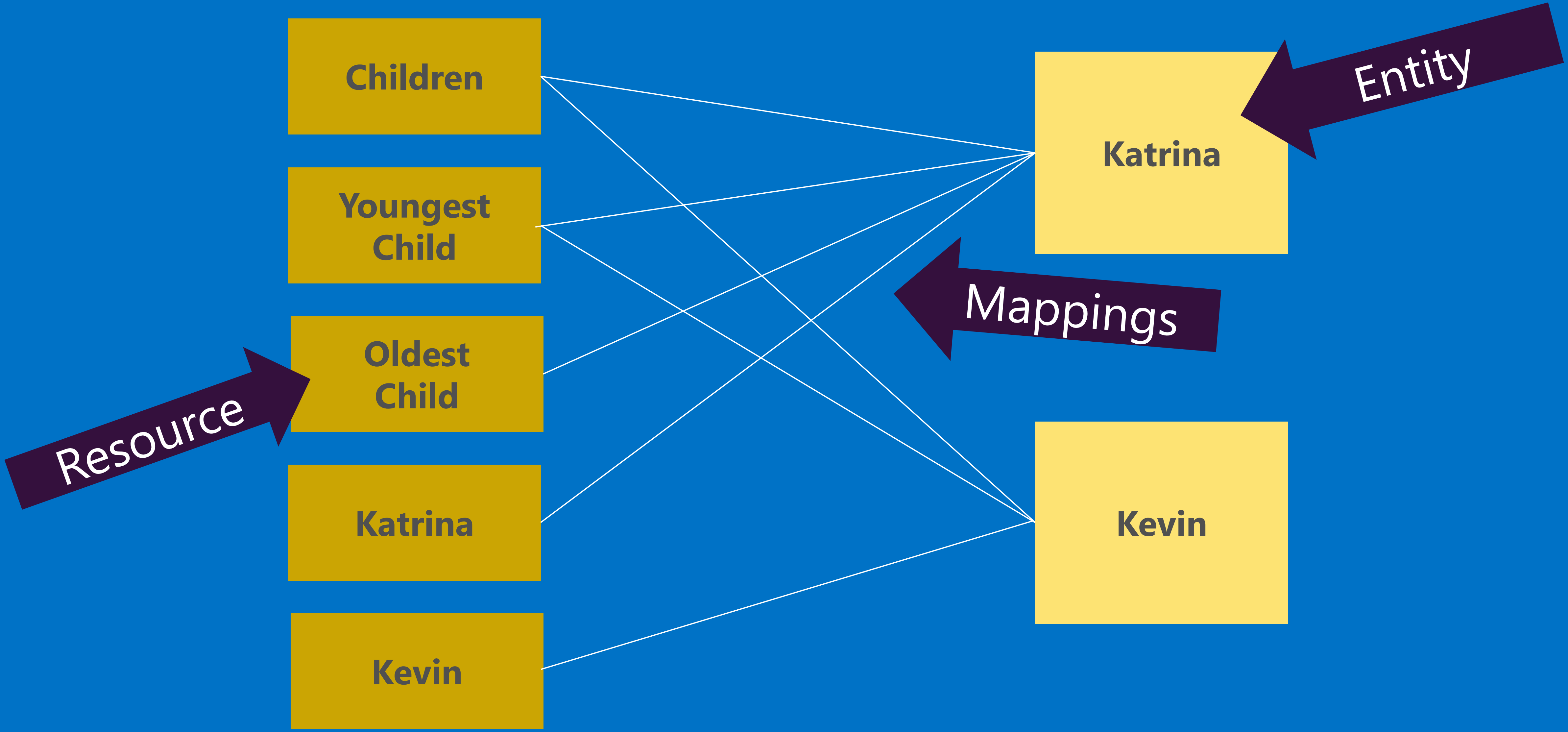
- REpresentational State Transfer
- Architectural style for service design
- Way of thinking
- Resource based
- 6 Constraints
  - Uniform interface
  - Stateless
  - Client-Server
  - Cacheable
  - Layered System
  - Code-on-demand

Basic Concepts

What is a Resource?

- A concept, a thing, a *noun*
- *Addressable* by URI
- Multiple URIs can refer to the same resource
- Expressed as a *representation*
- Separate from the representation

What is a Resource?



Resource Identifier

- Identifies a resource

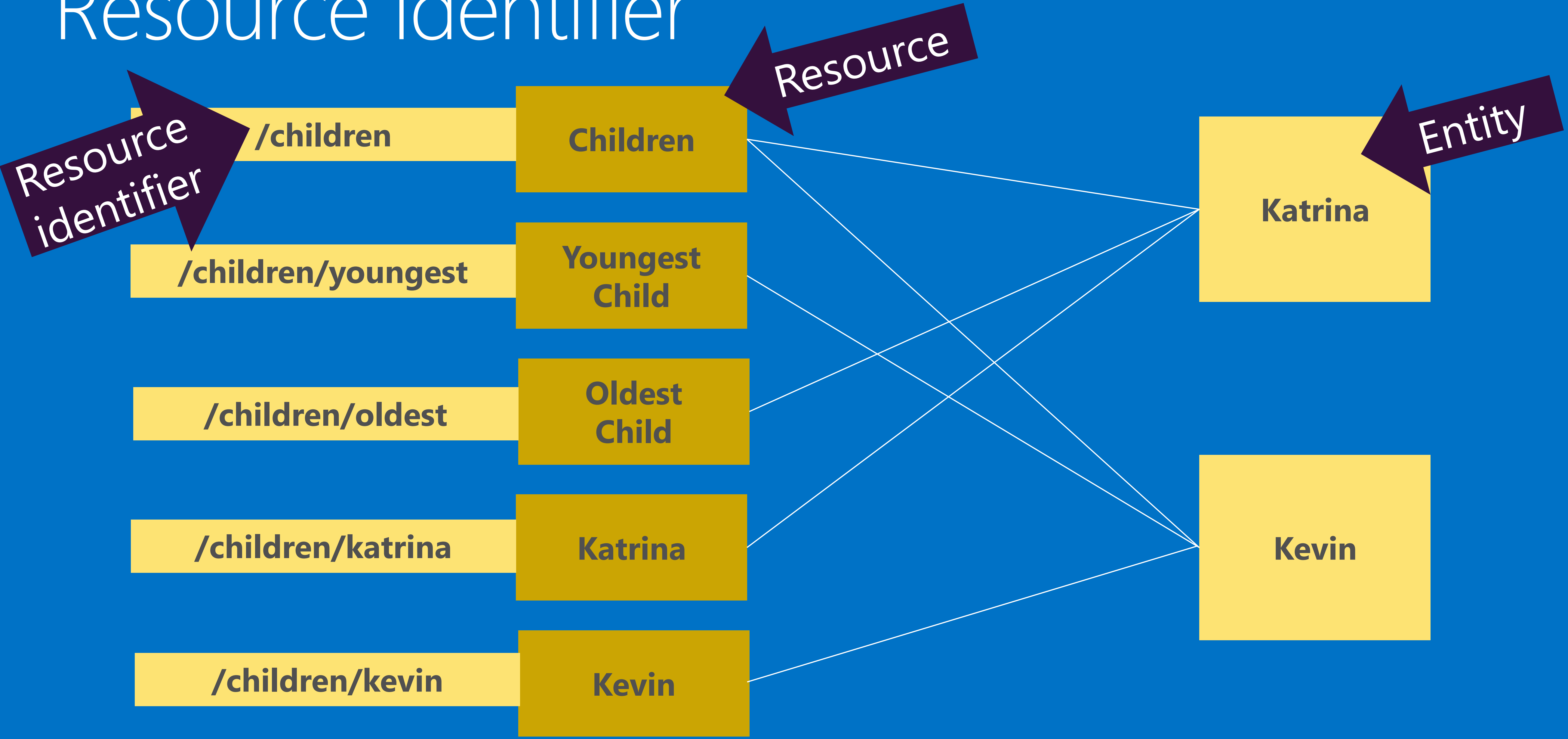
Book with ISBN of 2739129

Resource

http://myapi.com/books/27391290

Resource Identifier

Resource Identifier



Representation

- View of the state of a resource at an instant in time.

```
<Person>
  <Id>72430</Id>
  <FirstName>Fred</FirstName>
  <LastName>Flinstone</LastName>
</Person>
```

```
{
  "Id": "72430",
  "FirstName": "Fred",
  "LastName": "Flinstone"
}
```

- Data format of a representation is its media type.

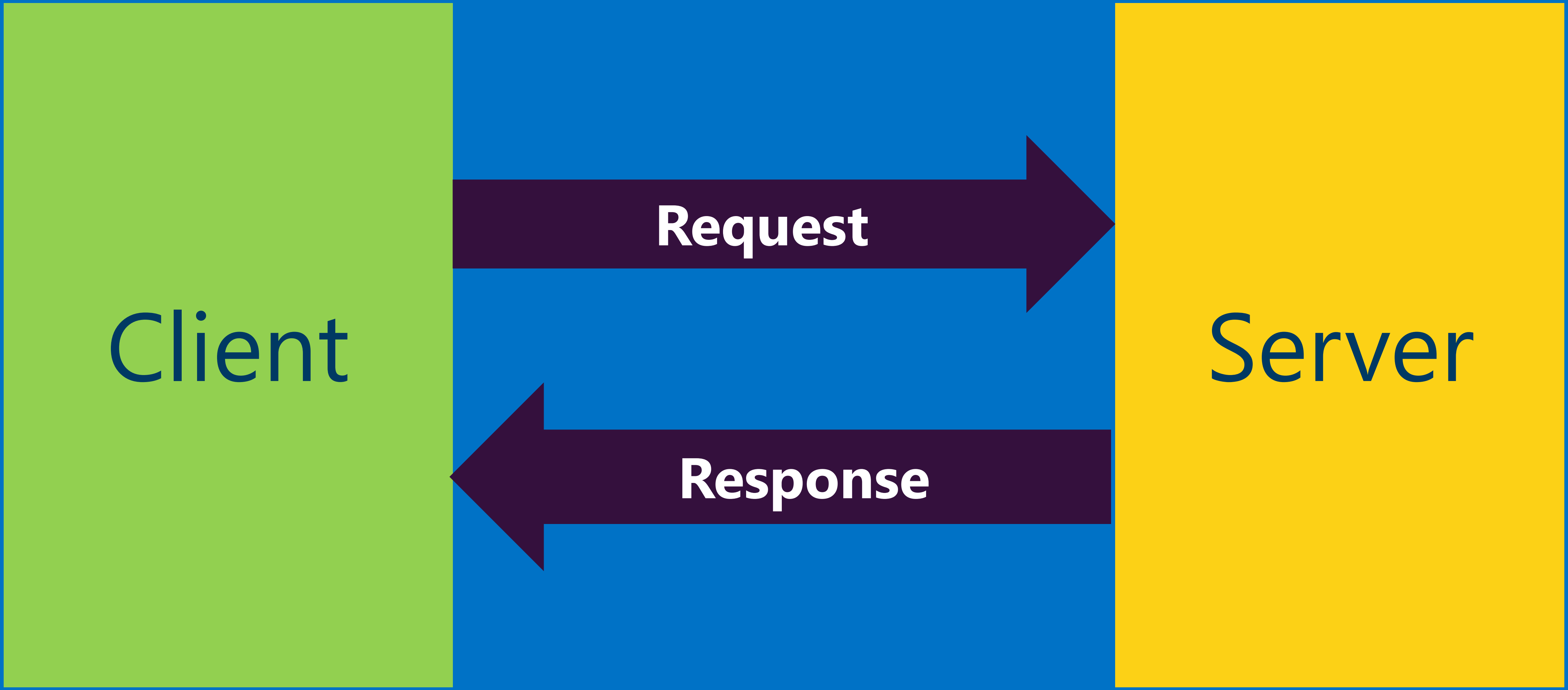
JSON XML CSV PDF

REST Data Elements

Data Element	Web Examples
Resource	the intended conceptual target of a hypertext reference
Resource Identifier	URL
Representation	HTML document, JPEG image
Representation Metadata	media type, last-modified time
Resource Metadata	source link, alternates
Control Data	if-modified-since, cache-control

REST Constraints

Client-Server Constraint

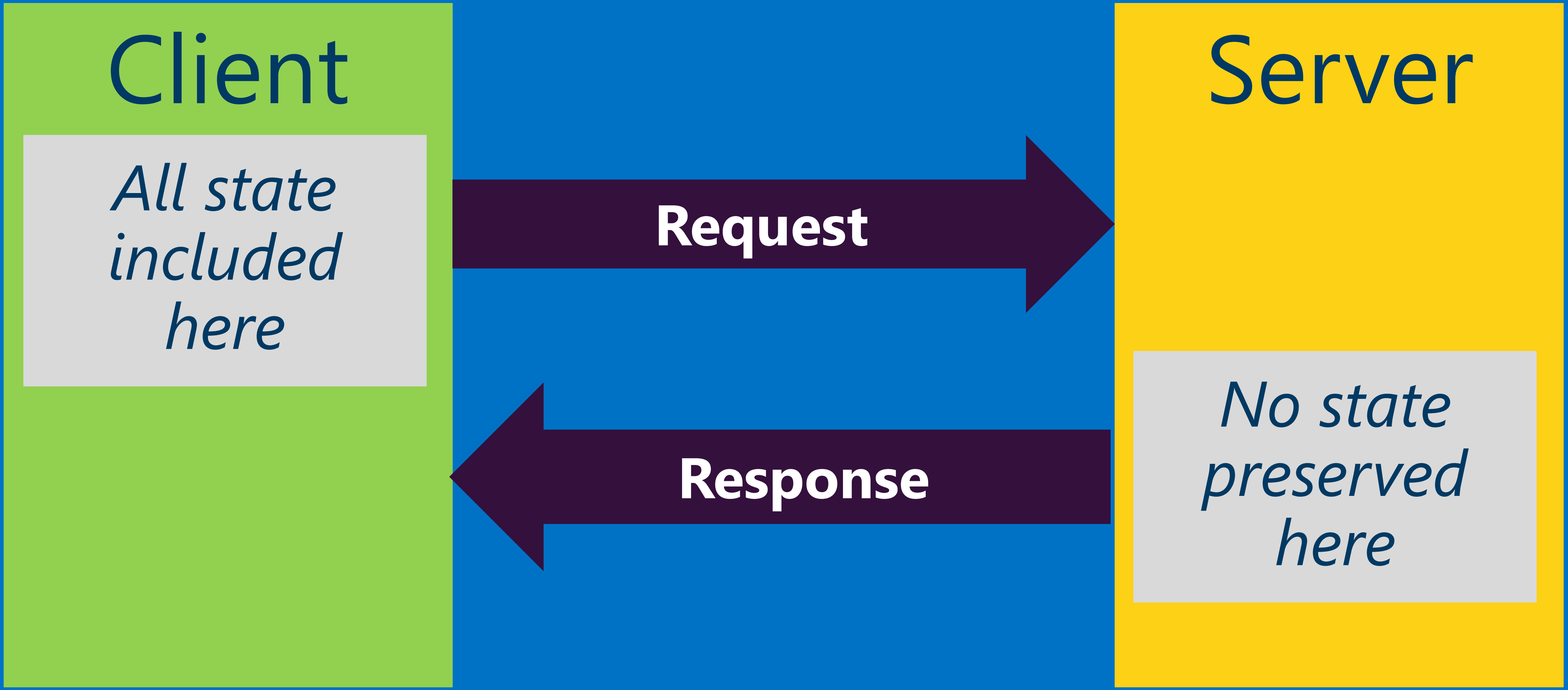


Client-Server Constraint

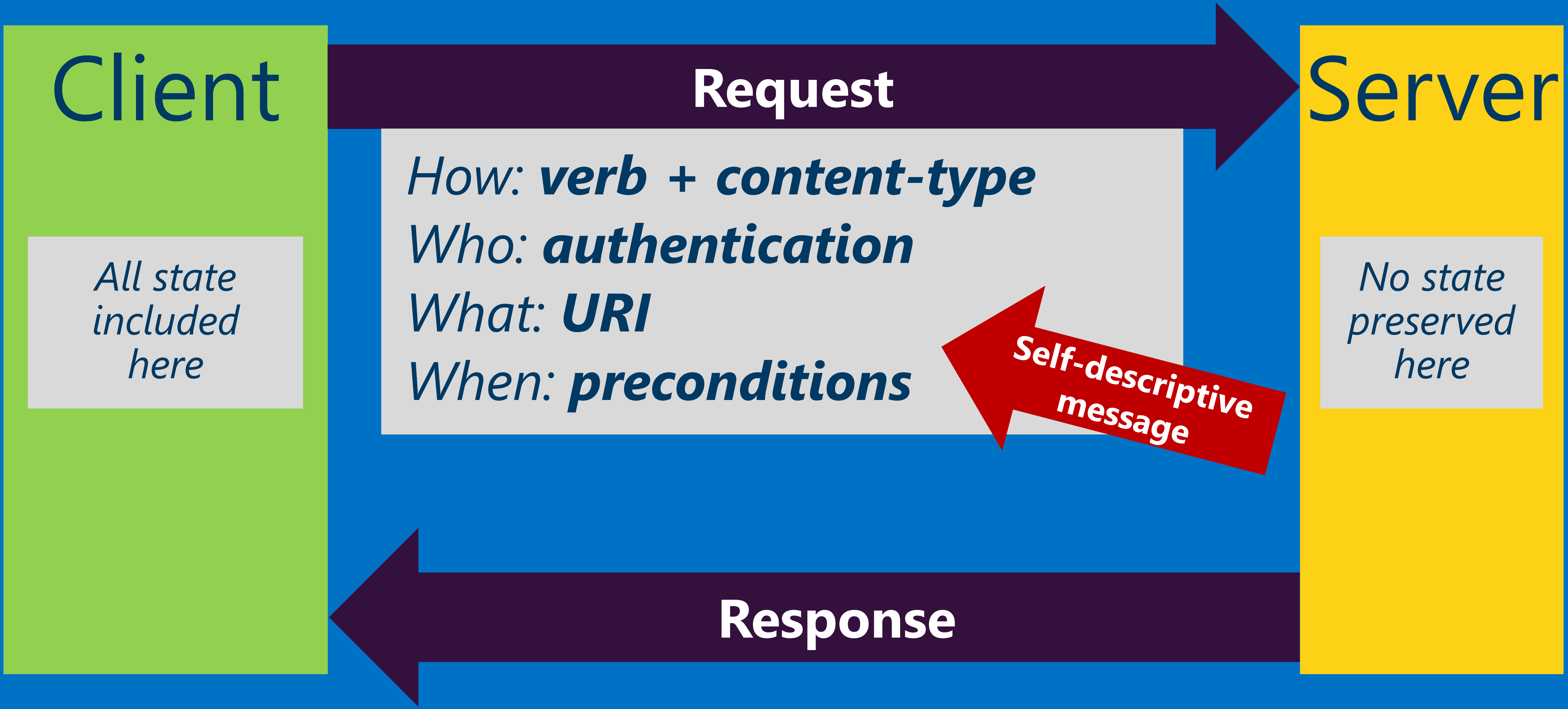
- Disconnected system
- Uniform interface is the connection
- Client initiates by sending a message to the server
- Server listens for incoming messages, does some processing, and returns a response to the client

- Goal: Separation of concerns

Stateless Constraint



Stateless Constraint



Self-descriptive Message - HOW

Verb

```
GET /api/persons/123 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8000
Authorization: Bearer 0b79bab50da…
If-None-Match: "289340187490"
```

Content-Type

Self-descriptive Message - WHO

```
GET /api/persons/123 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8000
Authorization: Bearer 0b79bab50da…
If-None-Match: "289340187490"
```

Authorization header

Self-descriptive Message - WHAT

```
GET /api/persons/123 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8000
Authorization: Bearer 0b79bab50da…
If-None-Match: "289340187490"
```

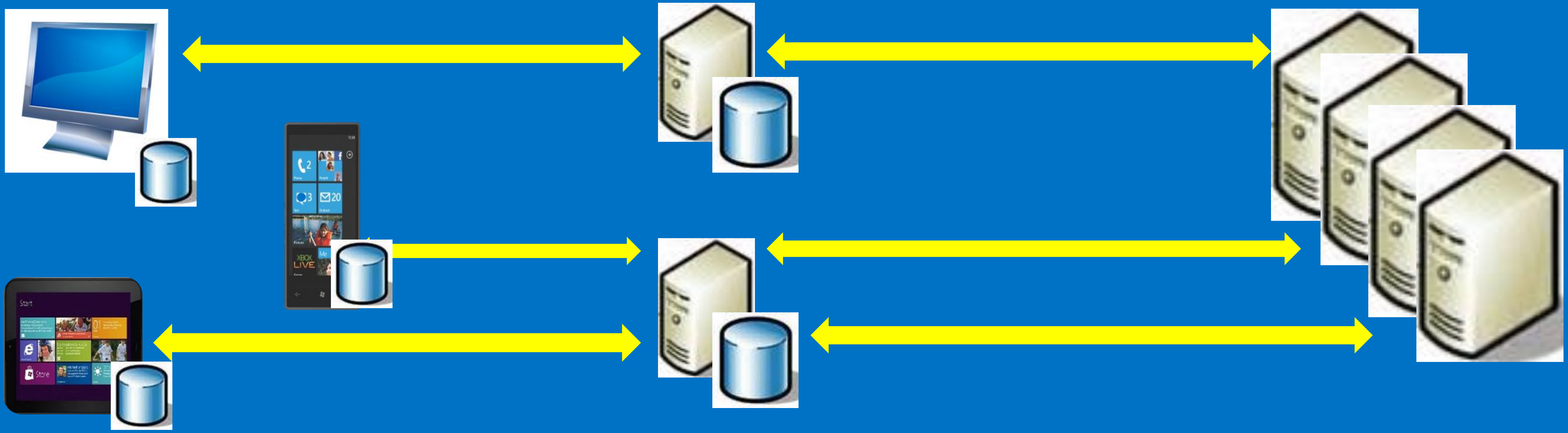
Self-descriptive Message - WHEN

```
GET /api/persons/123 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8000
Authorization: Bearer 0b79bab50da…
If-None-Match: "289340187490"
```

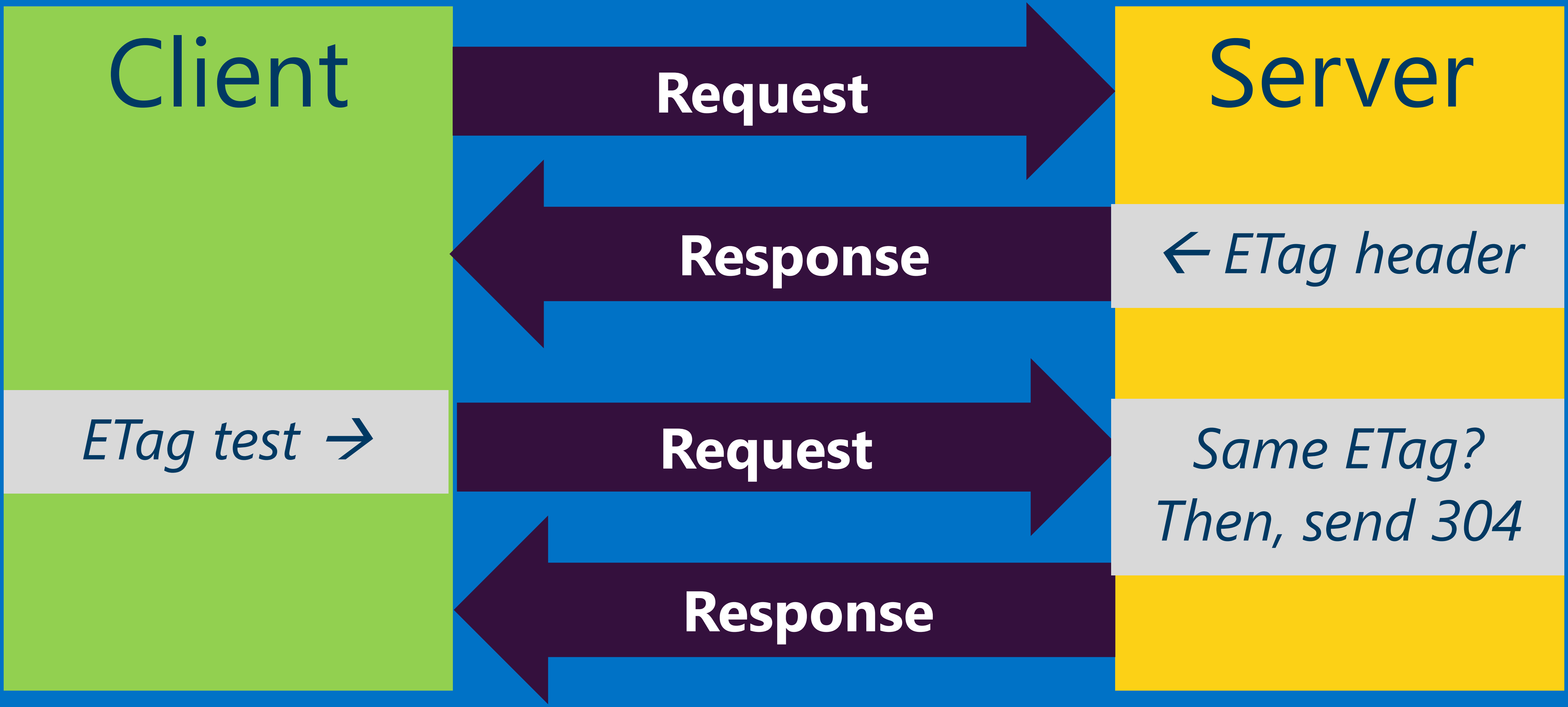
Precondition

Cacheable Constraint

- Representations from the server are cacheable on the client
- Responses from the server must be declared as cacheable or non-cacheable



ETags


Client
Server
Request
Response
← ETag header
ETag test →
Request
Same ETag?
Then, send 304
Response

ETags

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Wed, 17 June 2015 21;52:14 GMT
Etag: W/"289340187490"
Content-Length: 238
```

```
GET /api/persons/123 HTTP/1.1
Accept: application/json, text/xml
Host: localhost:8000
If-None-Match: "289340187490"
```

Uniform Interface Constraint

- Defines interface/contract between client and server
- REST does not require HTTP, but that will be our implementation, so Uniform interface in HTTP means:
  - URIs are the resources identifiers
  - HTTP verbs are the actions

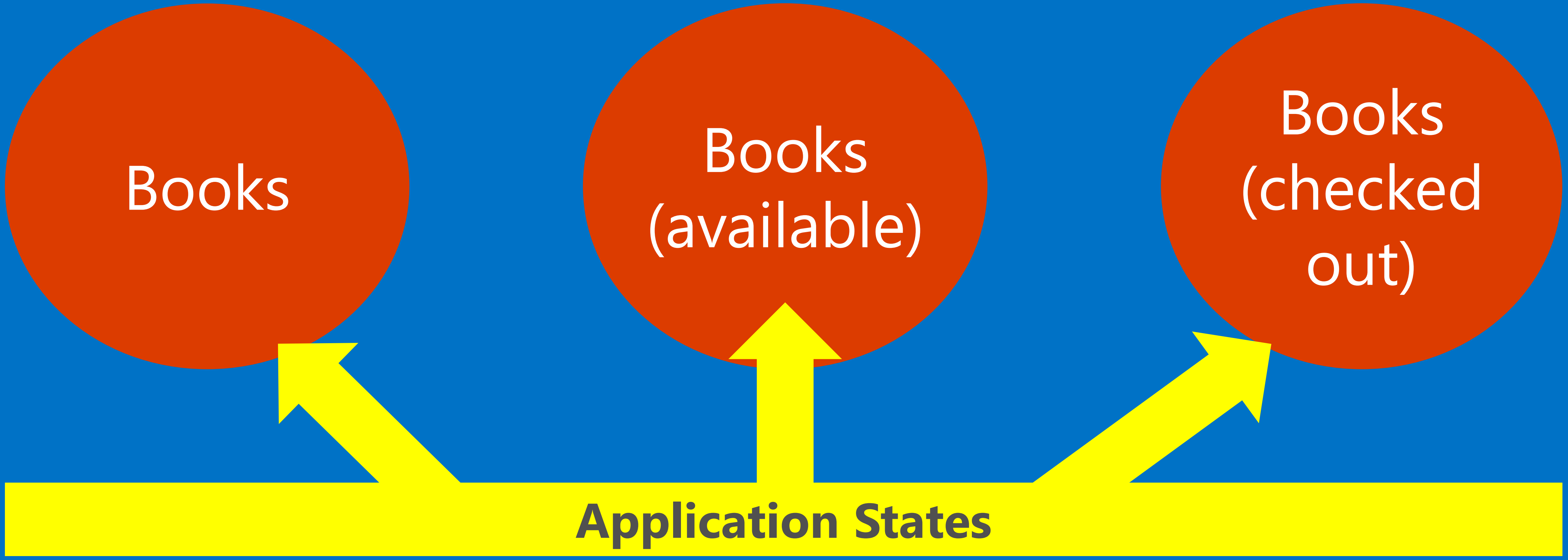
Guiding Principles of the Uniform Interface

- Identification of Resources
- Manipulation of Resources
- Self-Descriptive messages
- HATEOAS

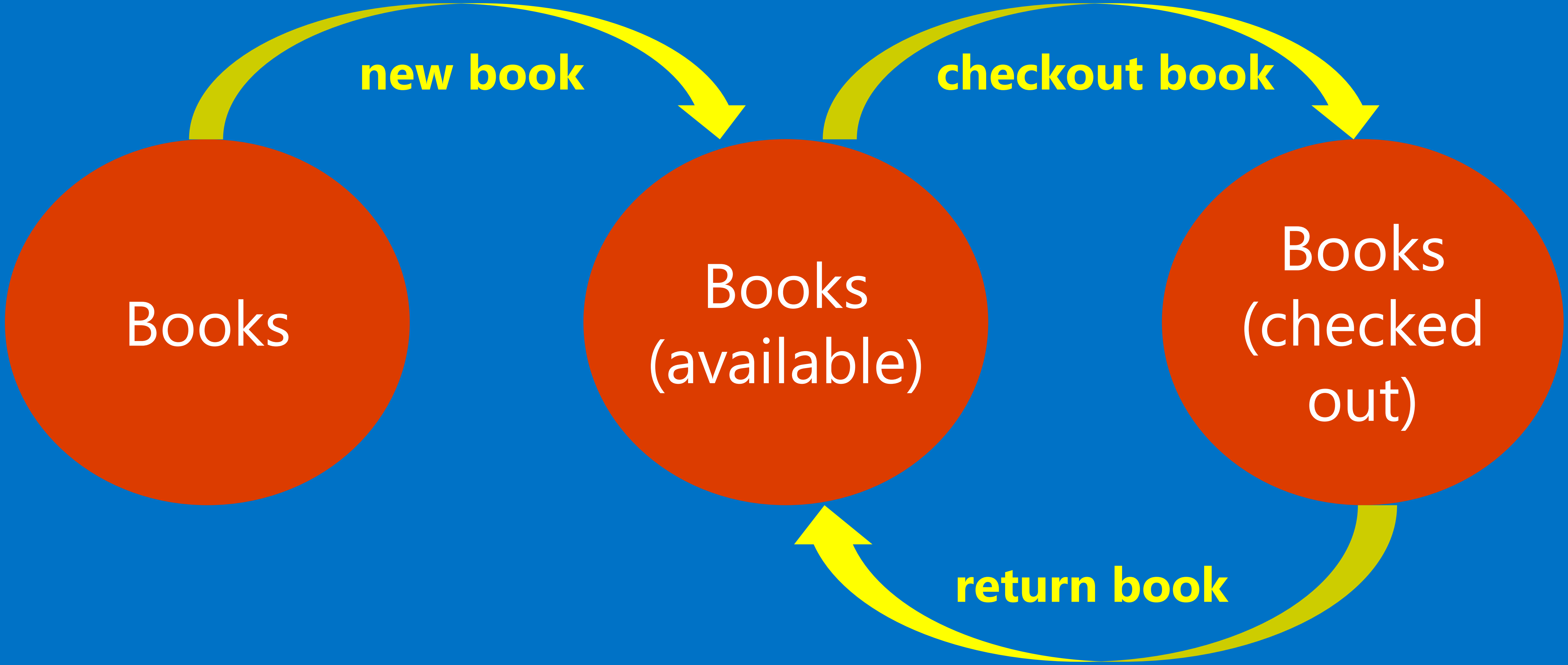
Library Example

- Given these requirements:
  - View all books in the library
  - View a specific book by id
  - Add a new book to the library
  - View books that are available to be checked out
  - Check out a book
  - Return a book

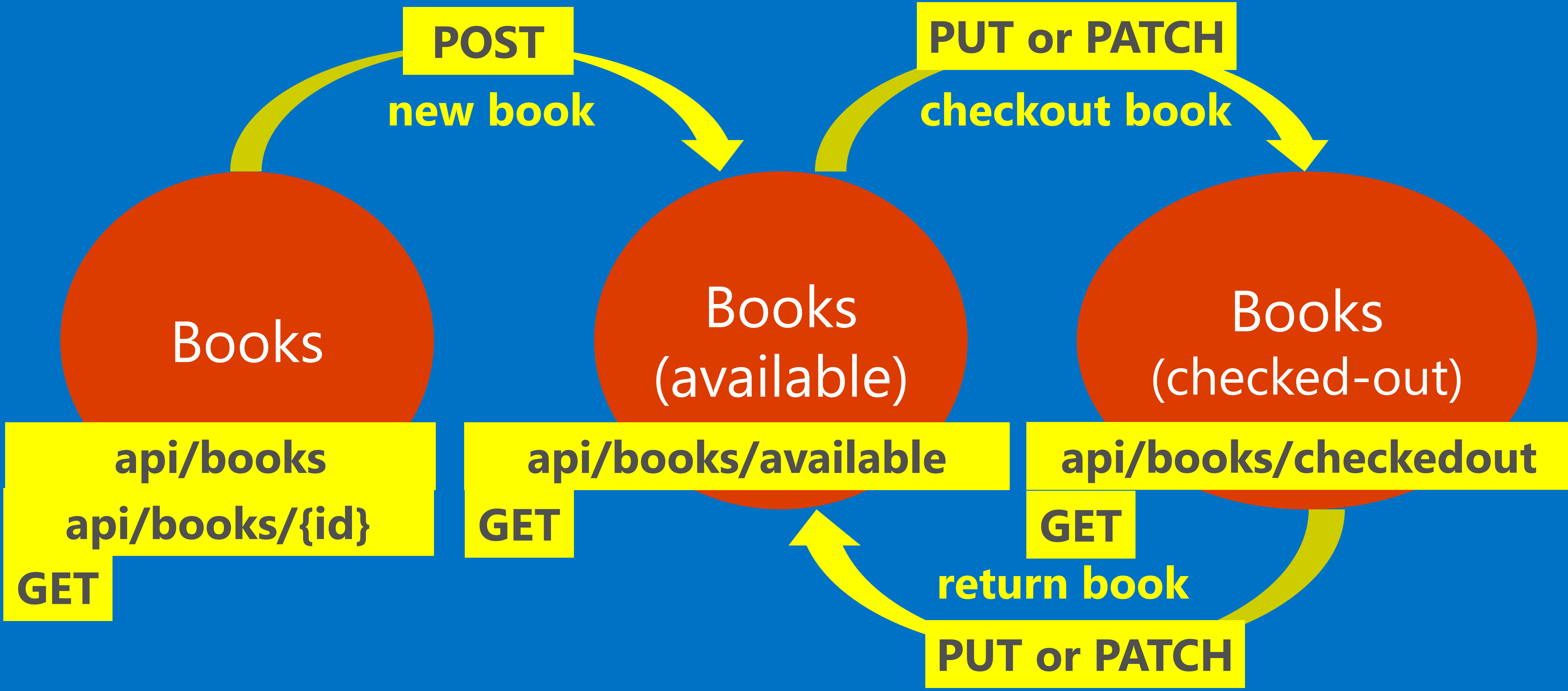
Identification of Resources



Manipulation of Resources



Map resources to HTTP



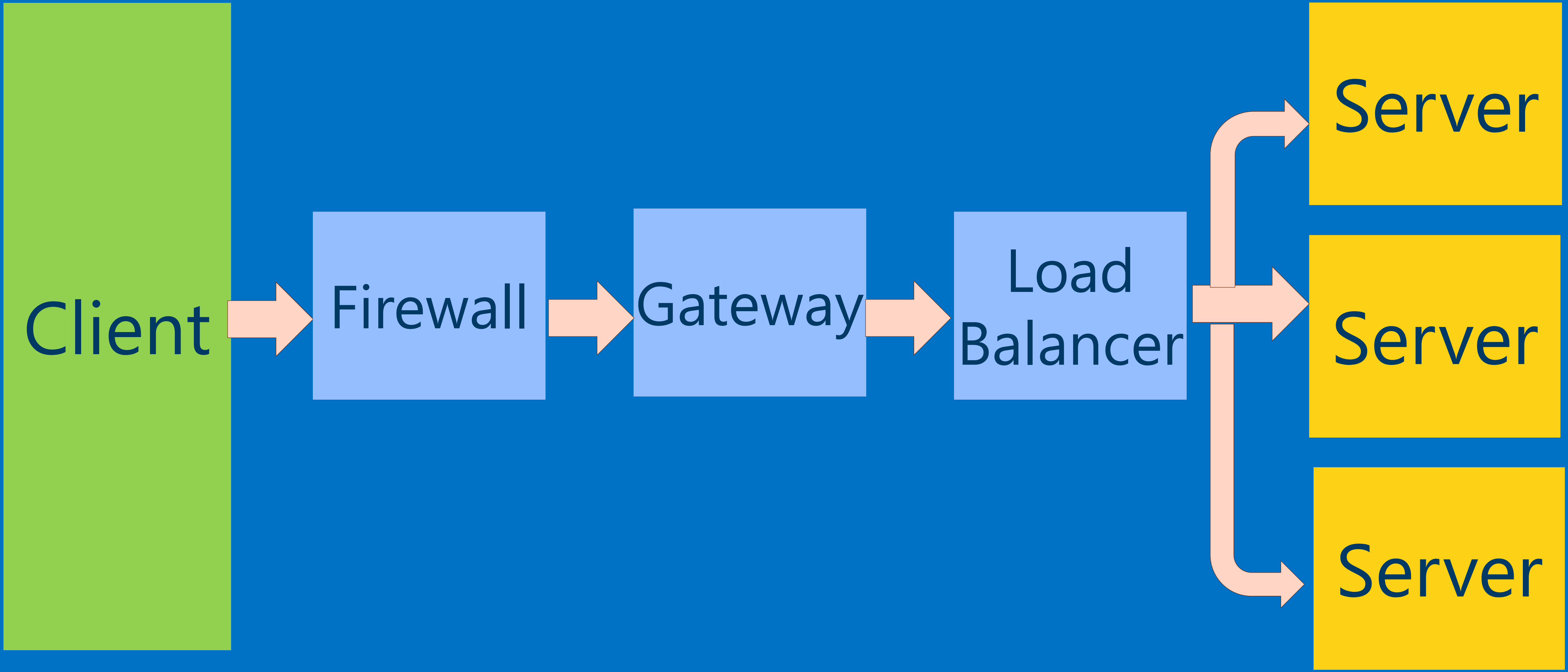
HATEOAS

- `GET api/books`
- Include the following in the response:
  - Link to create a new book
  - Link to get available books
  - Link to get checked out books
- `GET api/books/{id}`
- Include the following in the response:
  - Link to check out this book
  - Link to return this book

Common HATEOAS Links

- Paging
- Creating a new item
- Retrieving associations
- Performing actions

Layered System Constraint



REST Design Patterns

URI Structure

Not a requirement, but there are helpful patterns

URI Design

- Lowercase
- Substitute spaces with hyphens or underscores (pick one)
- Use nouns in your routes

URI Design

- No…
  - /api?action=getcomment&id=123
- Yes…
  - /api/comments/123
  - /api/articles/5/photos/4/comments/1

Cacheable and Readable

URI Design

- No…
  - /api/stories/orderby/date/limit/5

Don't filter via URI

- Yes…
  - /api/stories?orderby=date&limit=5

Do filter by query string

Verbs

- GET
  - retrieve whatever information (in the form of an entity) is identified by the Request-URI

Safe

Idempotent

Verbs

- POST
  - server accepts the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI

~~Safe~~

~~Idempotent~~

Verbs

- PUT
  - enclosed entity is stored under the supplied Request-URI
  - If the Request-URI refers to an already existing resource it is a modified version of the one residing on the origin server
  - Otherwise, the server creates the resource for that URI

~~Safe~~

Idempotent

Verbs

- PATCH
  - perform a partial update

~~Safe~~

Idempotent

Verbs

- DELETE
  - requests that the origin server delete the resource identified by the Request-URI

~~Safe~~

Idempotent

HTTP Status Code Categories

Code	Meaning
1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client error
5xx	Server error

HTTP 2xx Status Codes

Code	Meaning	
200	OK	Resource returned
201	Created	Resource created
204	No content	Resource deleted

HTTP 3xx Status Codes

Code	Meaning	
301	Moved Permanently	Resource reorganized
302	Found	Redirection for a specific object (i.e. search)
304	Not modified	Resource was not changed

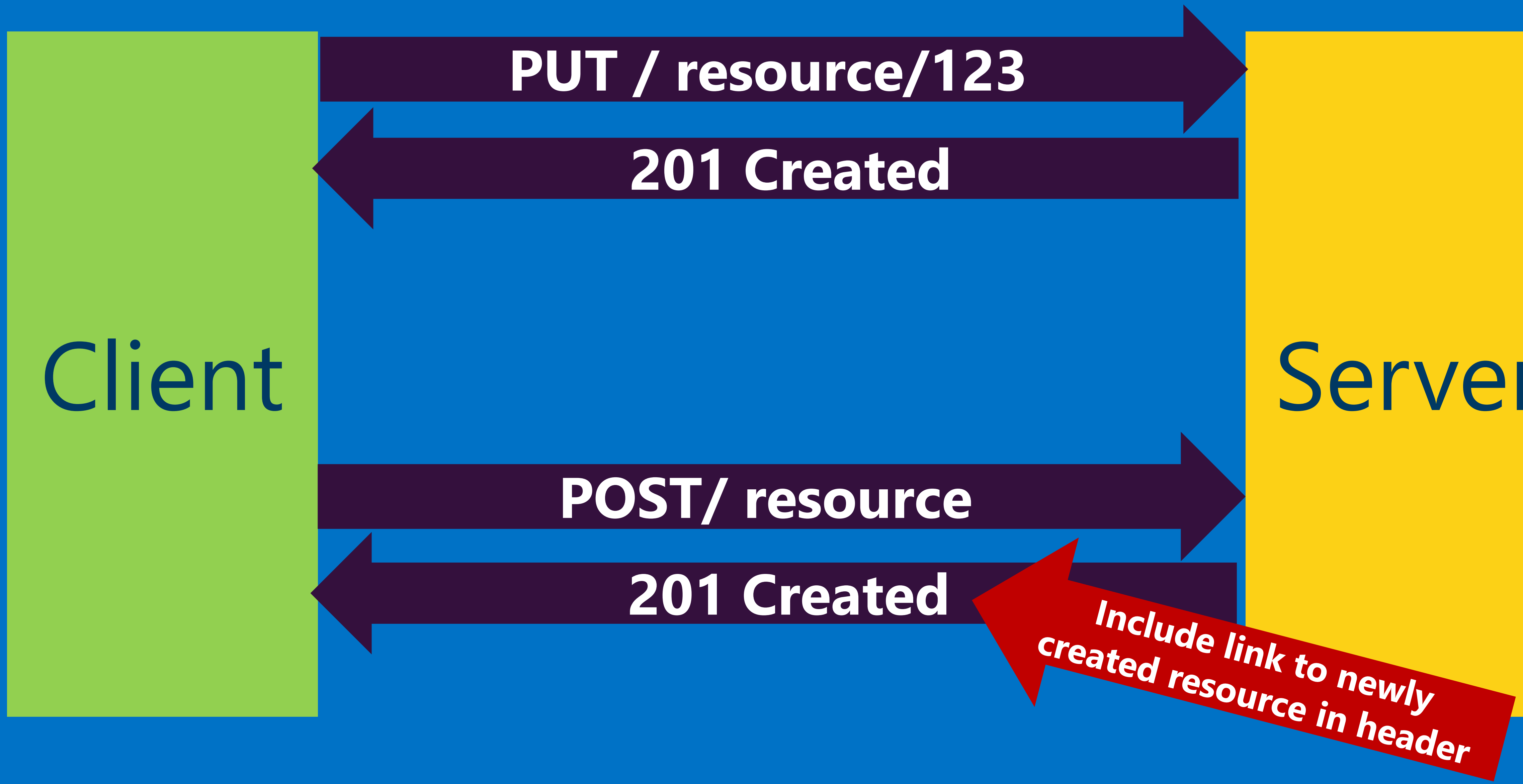
HTTP 4xx Status Codes

Code	Meaning	
400	Bad request	invalid request
401	Unauthorized	Request requires authentication
403	Forbidden	Server refuses the request
404	Not found	Resource not found
405	Method not allowed	Verb used is not allowed for resource
409	Conflict	Conflict with resource's current state

HTTP 5xx Status Codes

Code	Meaning	
500	Internal Server Error	Server encountered an unexpected condition which prevented it from fulfilling the request.

PUT or POST to create a new resource?



REST Antipatterns

Services define methods

- Do not define more verbs or remote procedures such as /adduser or /updateuser
- Do not include method names or remote procedures in the body of the HTTP request. Body should only contain resource state.

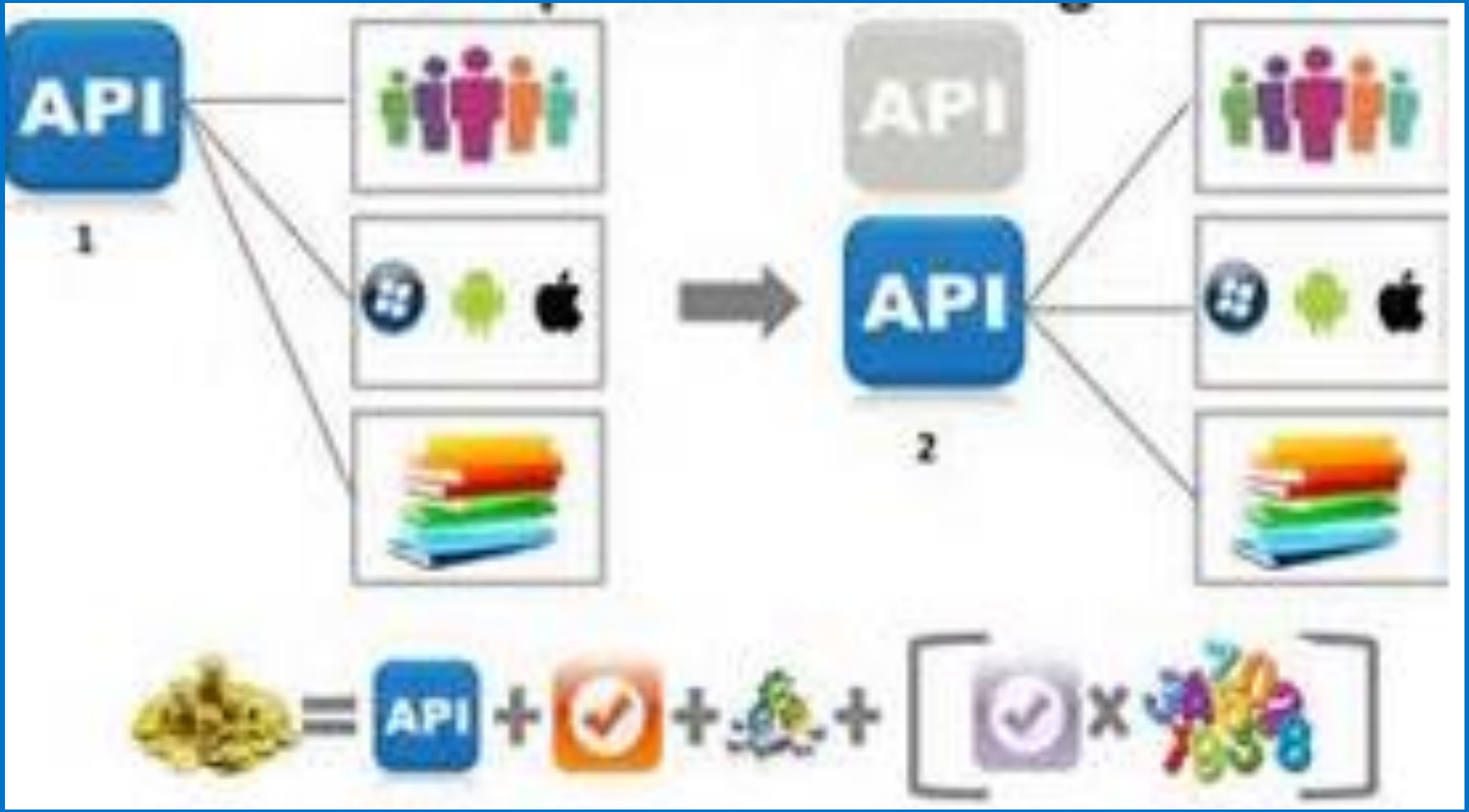
Using GET for everything

Verb	URI
GET	/api?method=addBook&title=Harry%20Potter
GET	/api?method=deleteBook&id=123
GET	/api?method=getBook&id=123
GET	/api?method=findBook&author=Rowling*

Using POST for everything

- Trying to solve problem of sending an arbitrary amount of data like RPC-style with SOAP
- POST is not idempotent and therefore unsafe
- Cannot be cached

Versioning



Versioning

- An API is an *explicit contract* with consumers
  - Developers write code against your method signatures and data shapes – changing it can break their applications
- However, to move forward, your service must support enhancements and bug fixes
- To do so, you must version your API to support changes without breaking clients
- API versioning is not product versioning

The Right way?

- No one right way
- Can examine existing APIs for options
- But, no single best option

When to Version?

- When adding functionality...
  - Example: Add spending limit property to customer
  - Versioning not always required
  - Add the new property and clients can typically ignore it
- Changing functionality
  - Removing or renaming resource content
  - Breaking change
  - Consider most appropriate versioning pattern

Versioning Examples

API	Pattern	Example
Tumbler	URI Path	http://api.tumbler.com/v2/user
Netflix	URI Parameter	http://api.netflix.com/catalog/titles/series/123?v=1.5
GitHub	Media Type	Content Type: application/vnd.github.1.param+json
Azure	Request Header	x-ms-version: 2015-01-01

URI Parameter Versioning Pattern

```
http://<host>/api/v3/order/1004
```

- Popular approach
- Include version number in URI path
- Can support large-scale API changes
- Everything after version number open to change

URI Parameter Versioning Pattern

```
http://<host>/api/v3/order/1004
```

- Advantages…
  - Simple to divide old API for backward compatibility
- Drawbacks…
  - Can end up with large amount of legacy code to support by maintaining entire code base for each version
  - Clients must change version numbers in their code

Query String Versioning Pattern

http://<host>/api/order/1004*?v=3*

- Add version number as *optional query string parameter* to URI

Query String Versioning Pattern

```
http://<host>/api/order/1004      ← Current Version, ver. 4
http://<host>/api/order/1004?v=3
```

- Advantages…
  - Without version parameter, consumers always get latest version of API
  - Users always stay current by not adding version number
- Drawbacks…
  - Surprise breakage can occur when clients do not keep up with API changes

Accept Header Parameter Pattern

- Version with ***header parameter*** in Accept Header
- Accept Header generally defines acceptable Content Types (data formats) for response that client supports

“text/plain” “image/jpeg” “application/xml” “application/json”

- But, can send additional parameters inside Accept Header to define other request aspects, such as version

```
GET /api/Order/1004
Host: Http://<host>
Accept: application/json;version=1
```

Send version parameter
Inside Accept Header

Accept Header Custom MIME Type

```
GET /api/Order/1004
Host: Http://<host>
Accept: vnd.contoso.v1.order
```

- Version with *custom type* in Accept Header
- Use vnd.* to indicate vendor

Accept Header Versioning Pattern

```
GET /api/Order/1004
Host: Http://<host>
Accept: vnd.contoso.v1.order
```

- Advantages…
  - API and versioning packaged together
  - Version separated from API call signature
- Drawbacks…
  - Complexity for clients by requiring header modifications

Custom Header Versioning Pattern

```
GET /api/Order/1004
Host: Http://<host>
X-MyAPI-Version: 2
```

```
GET /api/Order/1004
Host: Http://<host>
X-MyAPI-Version: 2015-01-01
```

Custom Header Versioning Pattern

```
GET /api/Order/1004
Host: Http://<host>
X-MyAPI-Version: 2
```

- Advantages…
  - API and versioning packaged together
  - Version separated from API call signature
  - Not tied to Content-Type
- Drawbacks…
  - Complexity for clients by requiring header modifications

Final thoughts on versioning

- Your API is a contract
- It will change over time
- Critical to handle changes in a structured and predictable manner
- Add versioning from the beginning, not as an afterthought

Tools

cURL

- an open source command line tool and library for transferring data with URL syntax

http://curl.haxx.se

POSTMAN

- Supercharge your API workflow with Postman! Build, test, and document your APIs faster.

Swagger

- Swagger is a simple yet powerful representation of your RESTful API.

Microsoft

Thank you!