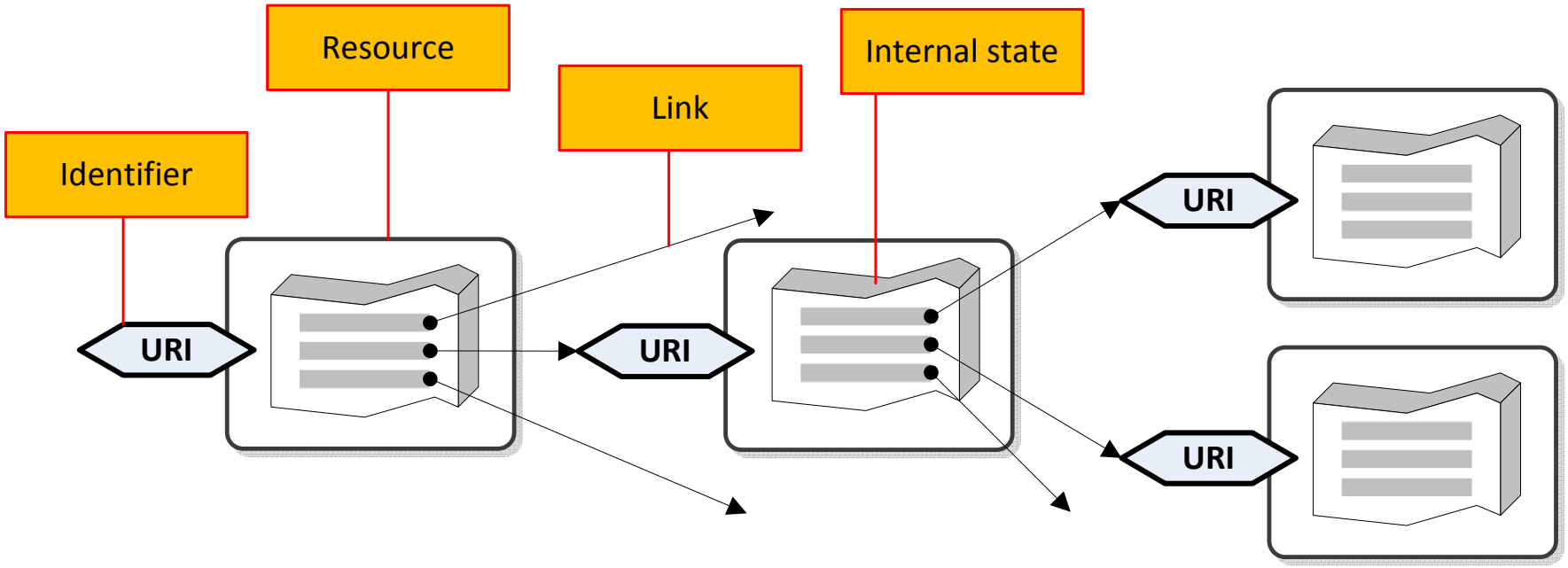
Short introduction to REST

Bartosz Baliś

REST on one slide

RESTful system = a set of linked Resources



Only four actions on a resource (CRUD):

- **C**reate resource (**POST**)
- **R**ead current resource state (**GET**)
- **U**pdate resource state (**PUT**, **POST**)
- **D**elete resource (**DELETE**)

Hypermedia as the Engine of Application State

- Application workflow is driven by hypermedia
- Links = possible state transitions
- A client moves the application from one state to another by following the links

Resources

- Work in a RESTful system = side effect of placing documents at URIs
  - Don't *invoke methods* through the Web!

- How to perform complex work, e.g. modify a collection of resources in a transaction?
  - Create a new resource that represents this collection and place appropriate document at its URI

CRUD to HTTP

Operation	Method	Semantics
Create	**POST**	Create a new resource
Read	**GET**	Read resource state
Update	**PUT**	Replace resource state
Update	**POST**	Update resource state
Delete	**DELETE**	Delete resource
	OPTIONS	Read the list of supported methods
	HEAD	Read HTTP headers (resource metadata)

Example: ordering system Web Services vs. REST

WS-*

OrderManagementService

+getOrders()
+submitOrder()
+getOrderDetails()
+getOrdersForCustomers()
+updateOrder()
+addOrderItem()
+cancelOrder()
+cancelAllOrders()

CustomerManagementService

+getCustomers()
+addCustomer()
+getCustomerDetails()
+updateCustomer()
+deleteCustomer()
+deleteAllCustomers()

REST

Resource
<<interface>>
GET
PUT
POST
DELETE

/orders
GET: list all orders
PUT: N/A
POST: add a new order
DELETE: cancel all orders

/orders/{id}
GET: get order details
PUT: update (replace) order
POST: update order (e.g. add item)
DELETE: cancel all orders

/customers
GET: list all customers
PUT: N/A
POST: add a new customer
DELETE: delete all customers

/customers/{id}
GET: get customer details
PUT: N/A
POST: update customer
DELETE: delete customer

/customers/{id}/orders
GET: get all orders of customer
PUT: N/A
POST: add a new order
DELETE: cancel all orders of customer

Hypermedia: linking resources

```
POST /orders HTTP/1.1

<order>
  <products>
    <link rel="product" href="/products/p1"/>
    <link rel="product" href="/products/p5"/>
  </products>
</order>
```

```
HTTP/1.1 201 Created
Location: http://www.example.com/orders/1234

<order id="1234" href="/orders/1234">
  <products>
    <link rel="product" href="/products/p1"/>
    <link rel="product" href="/products/p5"/>
  </products>
  <total>1000</total>
  <link rel="cancel"  href="..."/>
  <link rel="payment" href="..."/>
</order>
```

Example: Hypermedia API in three easy steps

- Step one: Data structure
- Step two: Hypermedia semantics
- Step three: Protocol implementation
- Example: RESTful task list ("to-do")

Full article:

Mike Amundsen, A RESTful Hypermedia API in Three Easy Steps

http://www.amundsen.com/blog/archives/1041

Step 0: identify operations

Example – TODO list:

- GetList()
- GetItem(id)
- AddItem(name, description, date-due, completed)
  UpdateItem(id, name, description, date-due, completed)
  DeleteItem(id)
- GetOpenItems()
- GetTodaysItems()
- GetItemsByDate(date-start,date-stop)

Step 1: define data structures

Example for XML data format

- Define XML elements

Name	Appearance	Attributes	Children
`list`	MUST	href="{collection-uri}"	`item` (0..+)
`item`	MAY	href="{item-uri}"	`data` (1..+)
`data`	MAY	name="{data-name}"	none
`query`	MAY	href="{query-uri}"	`data` (0..+)

Step 1: define data structure

```
<?xml version="1.0" encoding="utf-8"?>
<list>
  <item>
    <data name="title">First Task</data>
    <data name="description">Produce first draft of Task</data>
    <data name="date-due">2010-03-21</data>
    <data name="completed">false</data>
  </item>
  <item>
    <data name="title">Second Task</data>
    <data name="description">Implement REST version</data>
    <data name="date-due">2010-03-22</data>
    <data name="completed">false</data>
  </item>
</list>
```

Step 2: Define URIs and add links

For TODO list we will have

- Collection URI
- Item URI
- Query URI

Step 2: Define URIs and links

```
<list href="{collection-uri}">
  <item href="{item-uri}">
    <data name="title">First Task</data>
    <data name="description">Produce first draft</data>
    <data name="date-due">2010-03-21</data>
    <data name="completed">false</data>
  </item>
  <item href="{item-uri}">
    ...
  </item>
  <query href="{query-uri}" rel="today" />
  <query href="{query-uri}" rel="open" />
  <query href="{query-uri}" rel="date-range" >
    <data name="date-start"></data>
    <data name="date-stop"></data>
  </query>
</list>
```

Step 3: Map CRUD operations to protocol (HTTP)

Collection URI

- *Example: http://www.example.org/list/*
- ***HTTP GET {collection-uri}*** *returns a list document with multiple items.*
- ***HTTP POST {collection-uri}*** *adds a new item to the list. POST body contains a single <item/> element*

Item URI

- *Example: http://www.example.org/list/1*
- ***HTTP GET {item-uri}*** *returns a list document containing the associated single <item/> element*
- ***HTTP PUT (item-uri}*** *updates the associated item. PUT body contains a single <item/> element*
- ***HTTP DELETE {item-uri}*** *removes the associated item from the list*

Query URI

- *Example: http://www.example.org/list/?today*
- *Example: http://www.example.org/list/?open*
- *Example: http://www.example.org/list/?date-start=2010-03-01&date-stop=2010-03-31*
- ***HTTP GET*** *{query-uri} returns a list document containing zero or more <item/> elements that match the query criteria.*
- *If the <query /> element in the list manager document has child <data /> elements, the name and value attributes of those elements should be added to the URI to form a valid query.*

RESTful good (and bad) practices

URIs: Universal Resource Identifiers

scheme "://" authority "/" path ["?" query] ["#" fragment]

- **Scheme**: more specific rules for constructing URIs within a 'family
  - Scheme is not a protocol!
- **Authority**
- **Path**: hierarchical part of resource identifier
- **Query**: non-hierarchical part of resource identifier
- **Fragment**: identifies a sub-resource (e.g. a section within a document)

URIs: basic rules

- No trailing slash
  - A different convention found on the Web: trailing slash denotes a collection, e.g. http://x.org/invoice/
- No underscores (use hyphens instead)
- Prefer lowercase
- No file extensions!!!
- Naming
  - Plural nouns for collections of things
  - Singular nouns for individual things
  - No verbs

URI design

- Forward slash (/) indicates hierarchical relationship
- Punctuations separate multiple pieces at the same hierarchy level
  - Comma when order matters, e.g. /earth/37.0,-95.2
  - Semicolon otherwise, e.g. /colorblends/red;blue
- Each 'subpath' should also be an addresable resource
  - http://api.example.com/clients/1/orders/2
  - http://api.example.com/clients/1/orders
  - http://api.example.com/clients/1
  - http://api.example.com/clients

URI design

- Use the *query* part for 'parameters' to an underlying algorithm
  - query/search-based resources
  - pagination of collections
- Examples:
  - http://x.com/cars – a collection of cars
  - http://x.com/cars/fords – a collection of Ford cars
  - http://x.com/cars?type=sedan – a collection of any sedan cars (different brands)
    - More 'transient' resource
  - http://x.com/cars?pageSize=10&pageStartIndex=50

Creating a resource

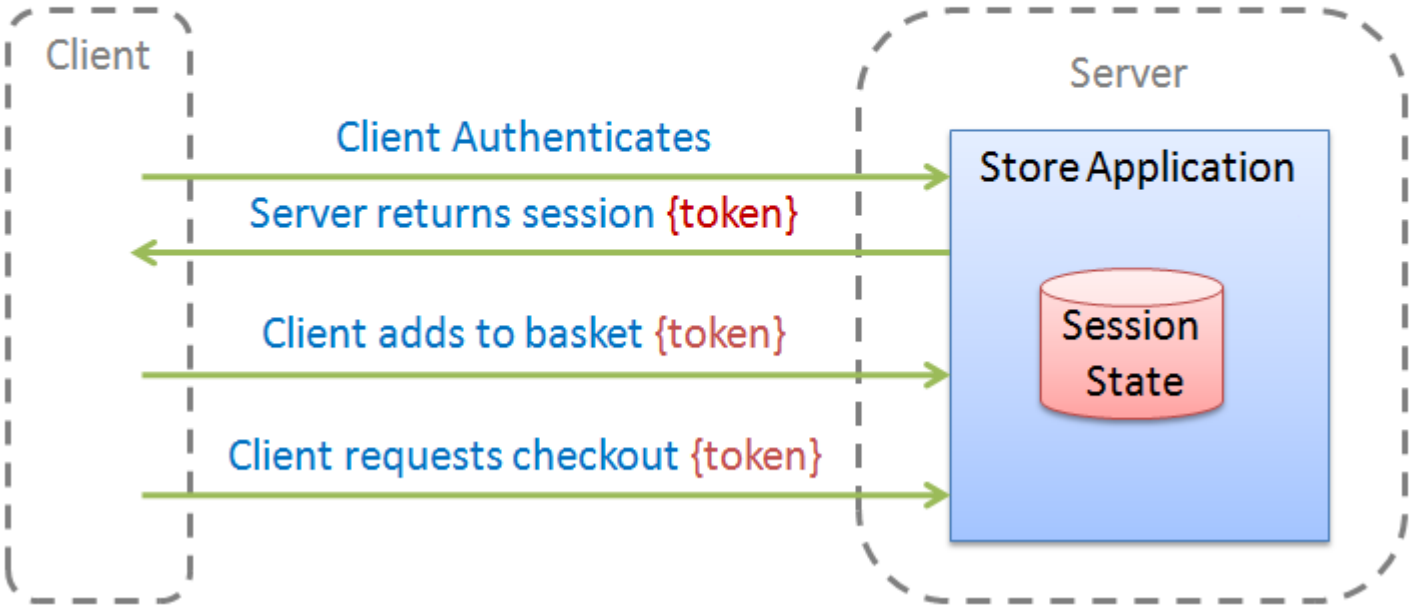
- POST to a **collection ('factory') resource**, e.g. www.example.com/books
  - Succesful response: 201 Created + Location
  - The server assigns the id of a new resource, e.g. www.example.com/books/1234
- Creating resources with PUT
  - Also possible when the client chooses the final URI of the resource
  - E.g. Amazon S3 service: the user chooses the name of a new bucket

```
PUT / HTTP/1.1
Host: BucketName.s3.amazonaws.com
```

- „Collection" resource vs. „Store" resource
  - Collection: new resource added with POST, the server chooses the id
  - Store: new resources added with PUT, the client chooses the id

Stateless shopping cart (1)

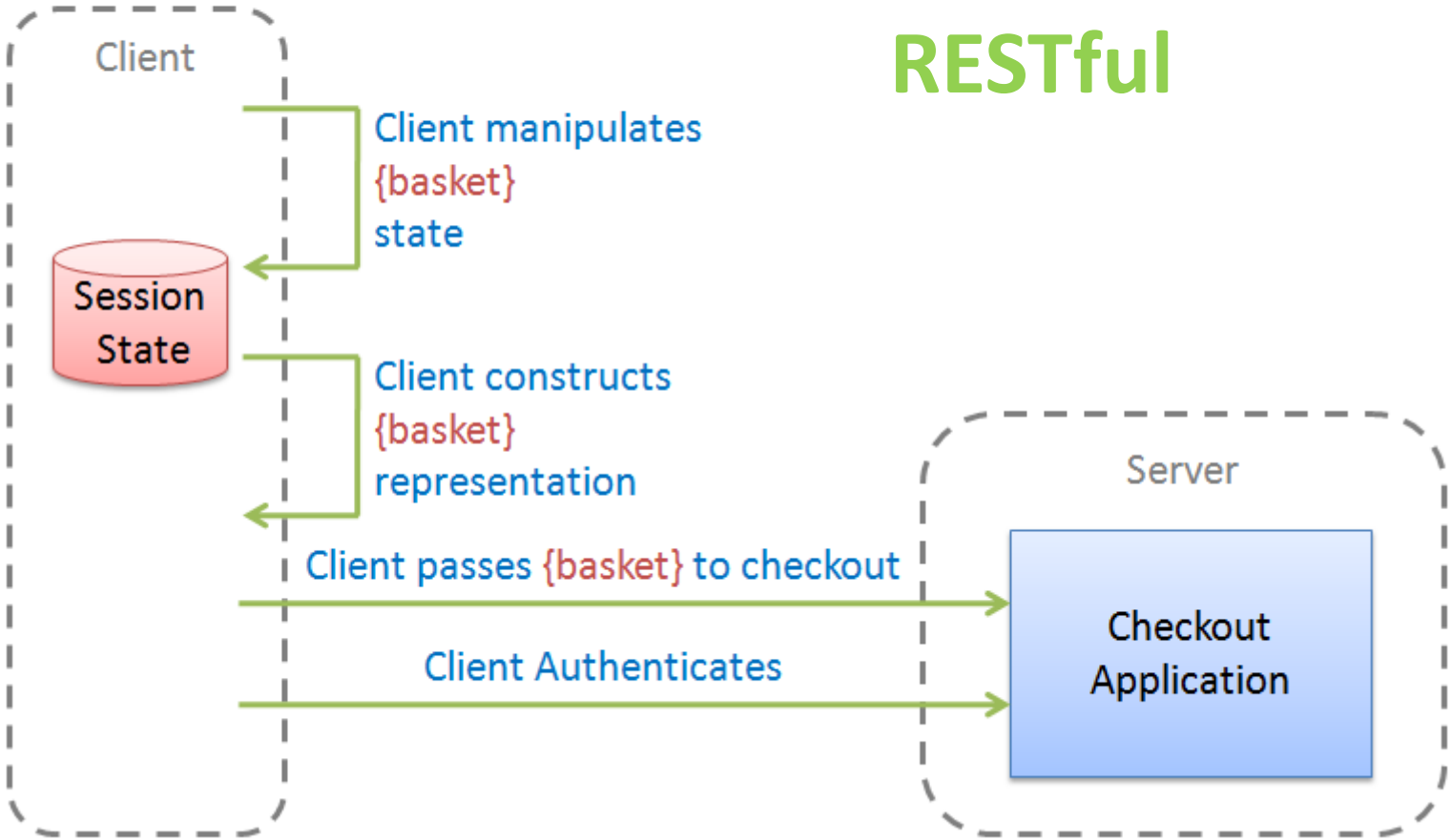
Server maintains Session



Not RESTful

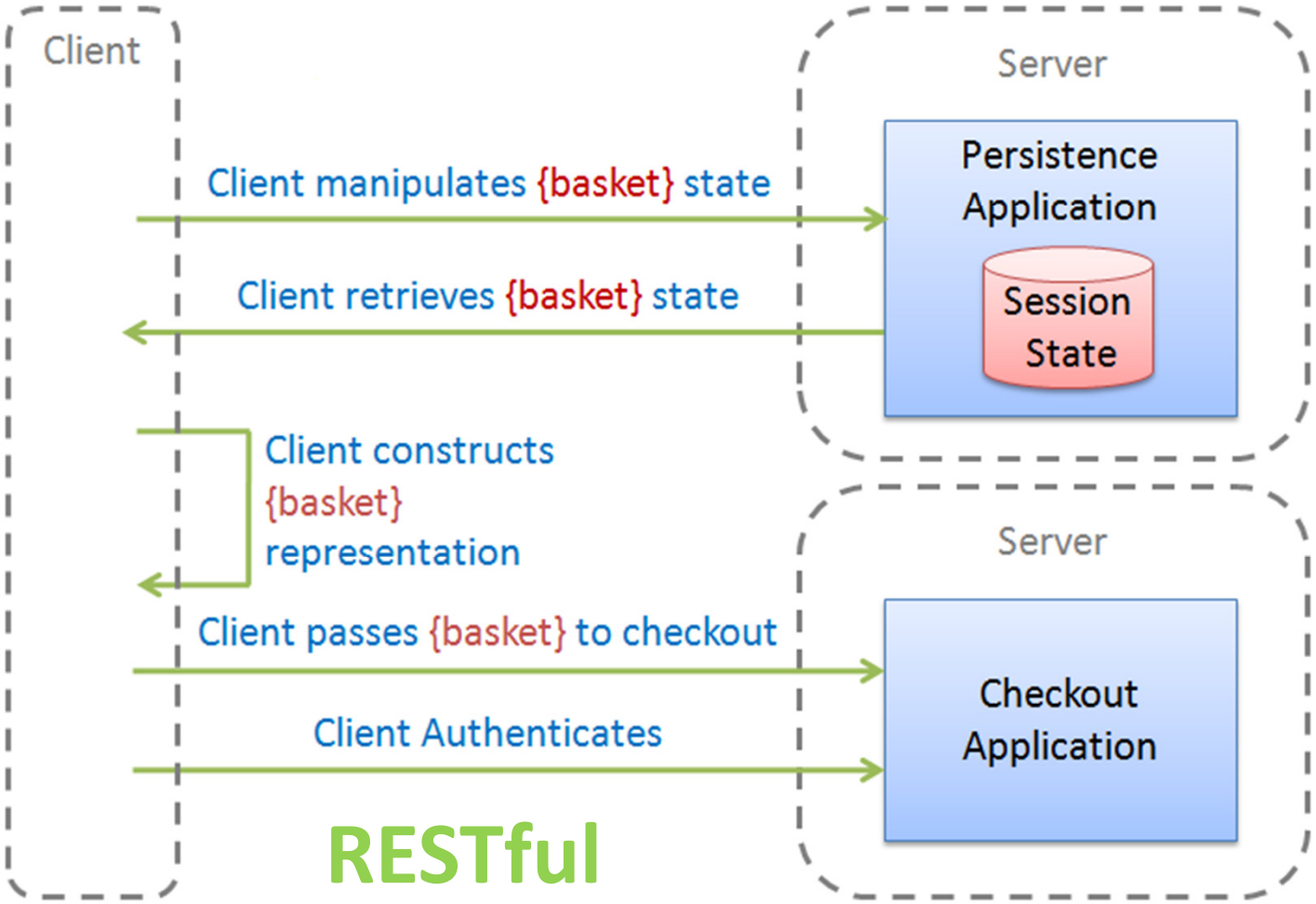
Stateless shopping cart (2)

Client maintains Session



Stateless shopping cart (3)

Client persists Session



Asynchronous invocation

Solution:

- POST + 202 Accepted
- polling with GET

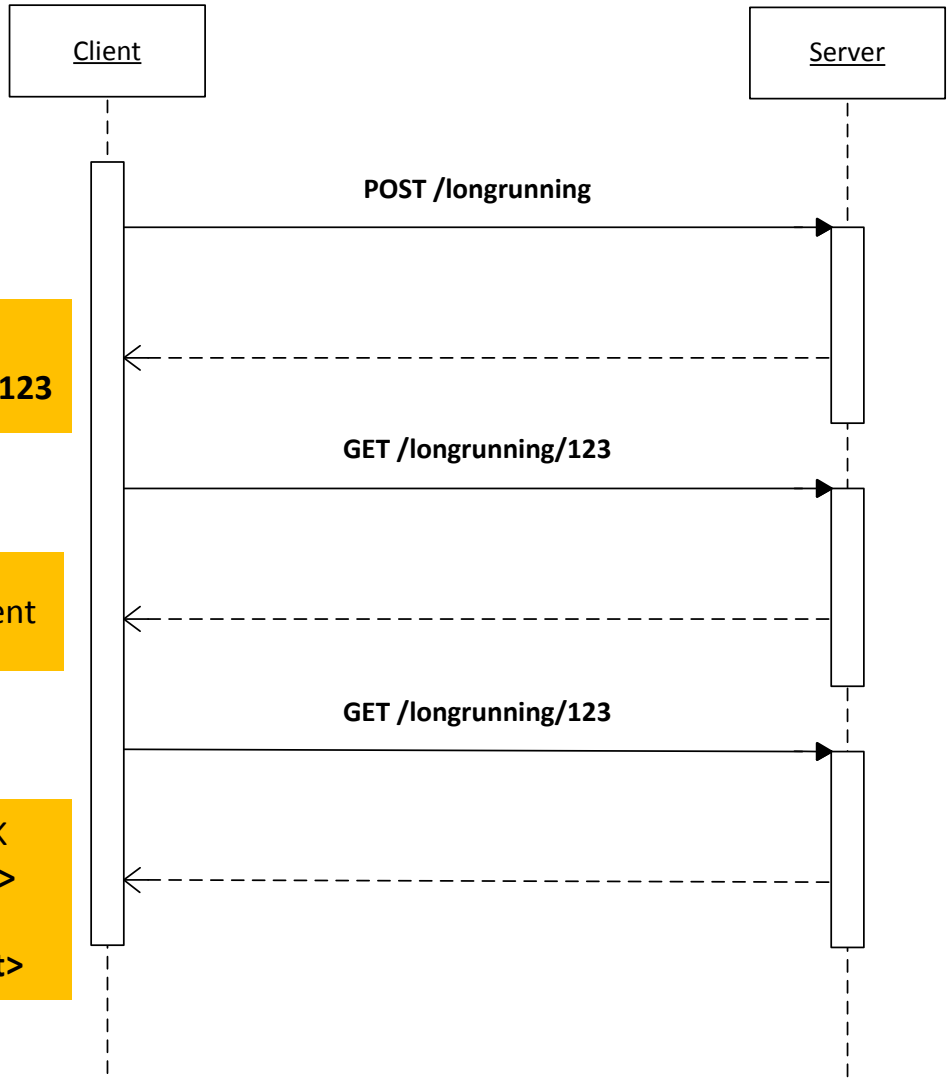
202 Accepted
Location: **http://x.com/longrunning/123**

Problem: how frequent polling?

- **longrunning/123** could return estimation of finishing time

204 No Content

200 OK
<result>
…
</result>



Using HTTP status codes

- Enrich semantics of client-server interactions
- Over 50 different status codes with uniform semantics!
  - 1xx – informational
  - 2xx – sucess
  - 3xx – redirection
  - 4xx – client error
  - 5xx – server error
- Use them, not just 200, 404 and 500

HTTP status codes

100 Continue
200 OK
201 Created
202 Accepted
203 Non-Authoritative
204 No Content
205 Reset Content
206 Partial Content
300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
304 Not Modified
305 Use Proxy
307 Temporary Redirect

400 Bad Request
401 Unauthorized
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed

500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported

Common REST antipatterns

- Tunneling everything through GET or POST
- Ignoring caching
- Ignoring response codes
- Misusing cookies
- Forgetting hypermedia
- Ignoring MIME types
- Breaking self-descriptiveness

Stefan Tilkov, REST antipatterns

http://www.infoq.com/articles/rest-anti-patterns

Video: http://www.parleys.com/#id=1397&st=5

Reading

- **rest-discuss** discussion group
  - http://tech.groups.yahoo.com/group/rest-discuss/
- **hypermedia-web** discussion web
  - https://groups.google.com/forum/?fromgroups=#!forum/hypermedia-web
- REST APIs must be hypertext-driven
  - http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- Classification of HTTP-based APIs
  - http://www.nordsc.com/ext/classification_of_http_based_apis.html
- A RESTful Hypermedia API in Three Easy Steps
  - http://www.amundsen.com/blog/archives/1041

Books

- Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures* (PhD Dissertation)
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- Jim Webber et al., *REST in Practice: Hypermedia and Systems Architecture*
- Thomas Erl et al., *SOA with REST. Principles, Patterns & Constraints for Building Enterprise Solutions with REST*
- Mike Amundsen, *Building Hypermedia APIs with HTML5 and Node*
- L. Richardson, M. Amundsen, S. Ruby, *RESTful Web APIs,* 2013