Handling HTTP Requests

SWE 432, Fall 2017

Design and Implementation of Software for the Web

GEORGE MASON UNIVERSITY

Today

- Status checkpoint
- Demo: Building a microservice with Express
- Design considerations in identifying resources
- REST
  - What is it?
  - Why use it?

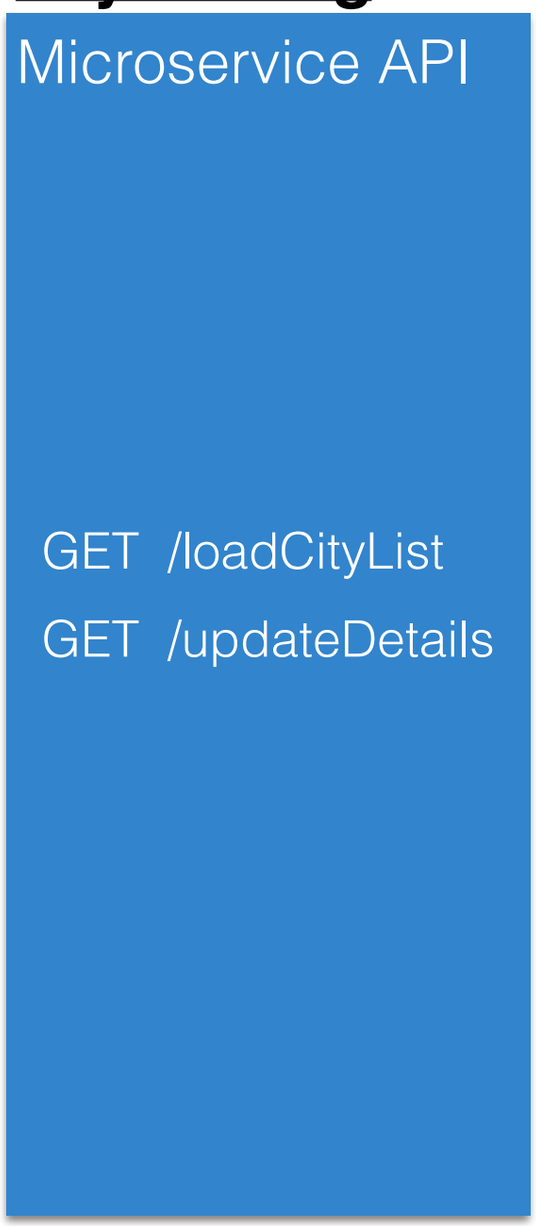
(Some) topics covered so far

- What is the web
- Basic JavaScript syntax
- Organizing code with closures, classes, packages
- Using Git, NPM, Node, Heroku to assemble web apps
- Building a webserver with Express; routing

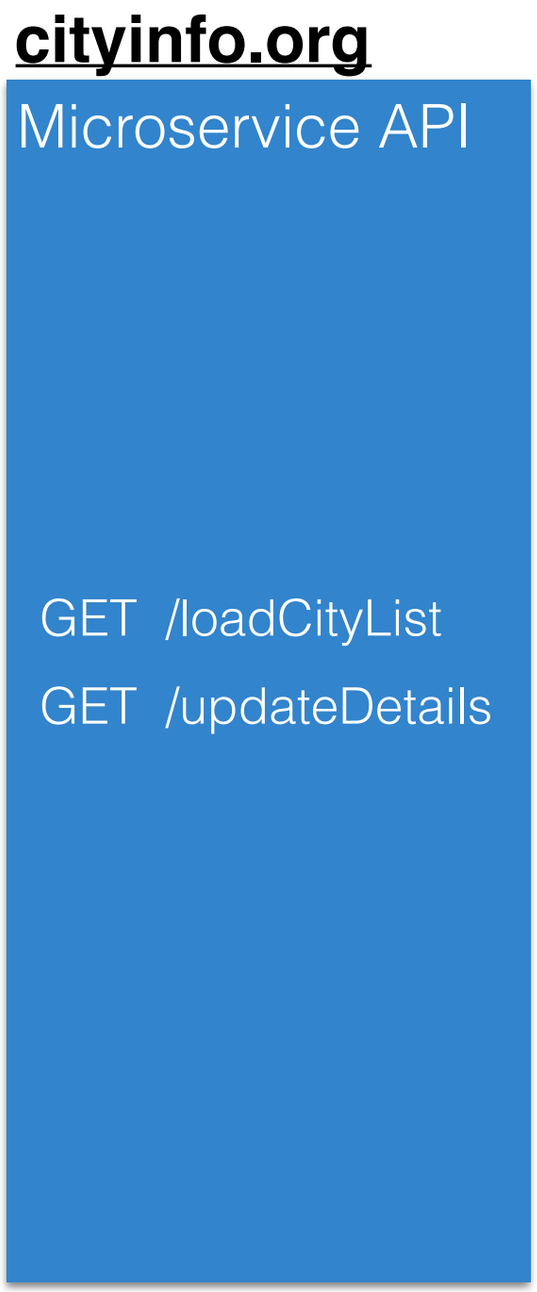
Some topics we have yet to cover

- How should HTTP requests be identified? (today)
- When do callbacks execute?
- How do you ensure clients can only do what they are allowed to do?
- How do you store data in a backend?
- How do you ensure your web app stays available?
- How do you design and implement a frontend using HTML, CSS, DOM, templates, front-end frameworks, information visualizations? (Part 2)
- How do you design a web app that enables humans to successfully accomplish their tasks? (Part 3)

Demo: Building a microservice w/ Express



API: Application Programming Interface



- Microservice offers public **interface** for interacting with backend
  - Offers abstraction that hides implementation details
  - Set of endpoints exposed on micro service

- Users of API might include
  - Frontend of your app
  - Frontend of other apps using your backend
  - Other servers using your service

APIs for functions and classes

V1

```
function sort(elements)
{
    [sort algorithm A]
}
```

```
class Graph
{
    [rep of Graph A]
}
```

Implementation change

Consistent interface

V2

```
function sort(elements)
{
    [sort algorithm B]
}
```

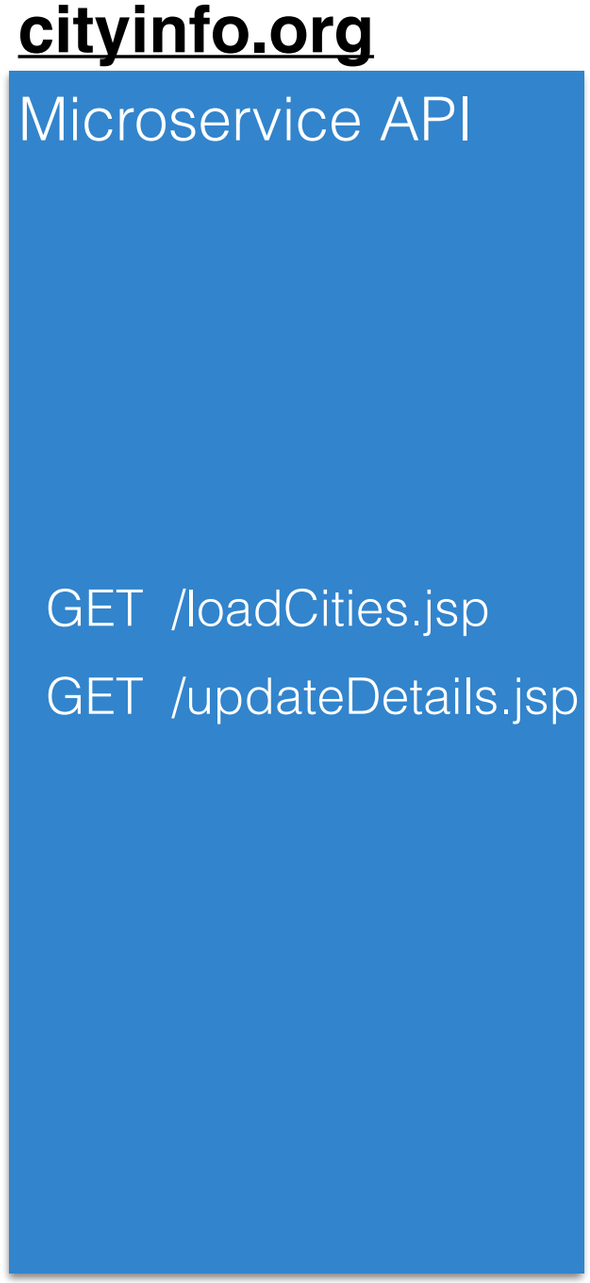
```
class Graph
{
    [rep of Graph B]
}
```

What makes a good microservice API design?

Support scaling

- Yesterday, cityinfo.org had 10 daily active users. Today, it was featured on several news sites and has 10,000 daily active users.
- Yesterday, you were running on a single server. Today, you need more than a single server.

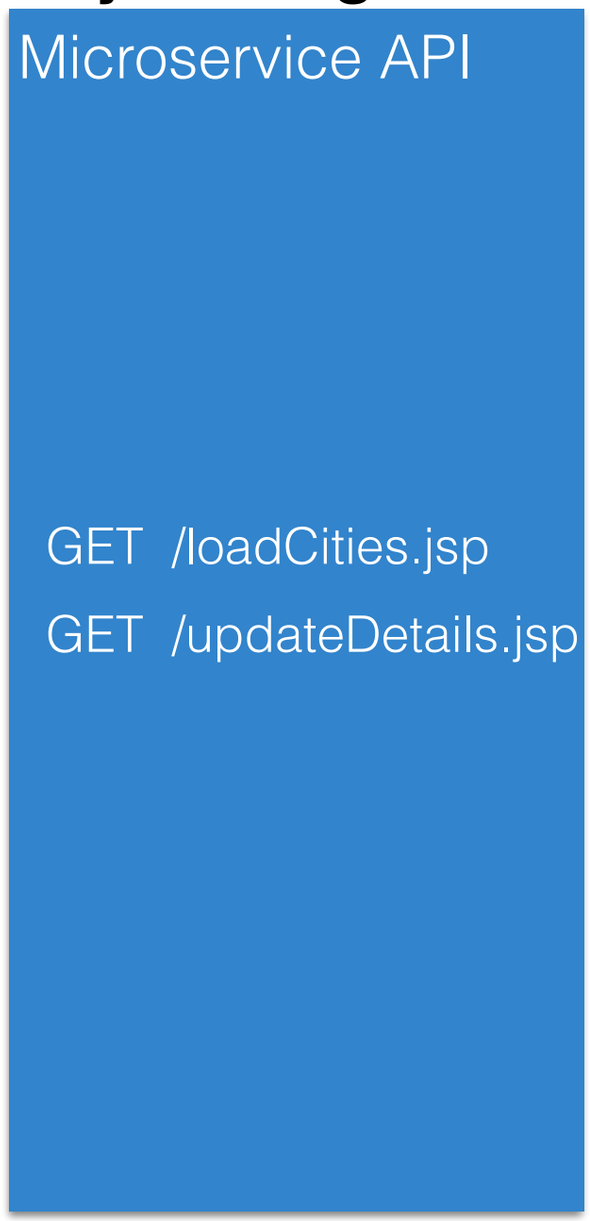
- Can you just add more servers?
  - What should you have done yesterday to make sure you can scale quickly today?



Support change

- Due to your popularity, your backend data provider just backed out of their contract and are now your competitor.
- The data you have is now in a different format.
- Also, you've decided to migrate your backend from PHP to node.js to enable better scaling.

- How do you update your backend without breaking all of your clients?



Support reuse

- You have your own frontend for cityinfo.org. But everyone now wants to build their own sites on top of your city analytics.

- Can they do that?

Design Considerations for Microservice APIs

- API: What requests should be supported?
- Identifiers: How are requests described?
- Errors: What happens when a request fails?
- Heterogeneity: What happens when different clients make different requests?
- Caching: How can server requests be reduced by caching responses?
- Versioning: What happens when the supported requests change?

REST: REpresentational State Transfer

- Defined by Roy Fielding in his 2000 Ph.D. dissertation
  - Used by Fielding to design HTTP 1.1 that generalizes URLs to URIs
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- "Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do… I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience. That process honed my model down to a core set of principles, properties, and constraints that are now called REST."
- Interfaces that follow REST principles are called RESTful

Properties of REST

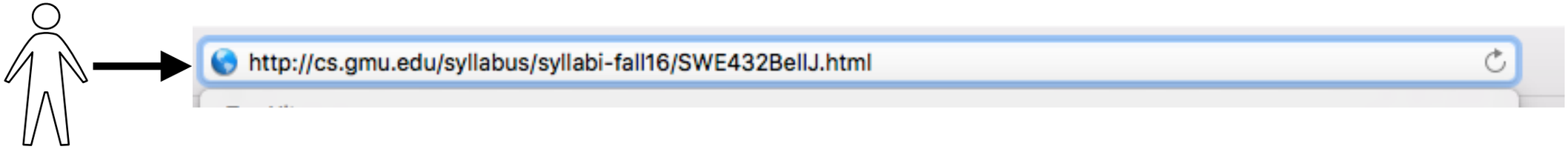
- Performance
- Scalability
- Simplicity of a Uniform Interface
- Modifiability of components (even at runtime)
- Visibility of communication between components by service agents
- Portability of components by moving program code with data
- Reliability

Principles of REST

- Client server: separation of concerns (reuse)
- Stateless: each client request contains all information necessary to service request (scaling)
- Cacheable: clients and intermediaries may cache responses. (scaling)
- Layered system: client cannot determine if it is connected to end server or intermediary along the way. (scaling)
- Uniform interface for resources: a single uniform interface (URIs) simplifies and decouples architecture (change & reuse)

HTTP: HyperText Transfer Protocol

High-level protocol built on TCP/IP that defines how data is transferred on the web



HTTP Request

```
GET /syllabus/syllabi-fall16/SWE432BellJ.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```

web server

Reads file from disk

HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head>...
```

SWE 432 Section 002 Fall 2016 Syllabus and Schedule

"Design and Implementation of Software for the Web"

Class Hours: Tuesdays and Thursdays, 12:00pm-1:15pm **Robinson Hall B228**
Grades, Readings available as pdfs: Blackboard
Resources (Announcements, Schedule, Assignments, Discussion):
Piazza - https://piazza.com/gmu/fall2016/swe432001/home

Instructor: Prof. Jonathan Bell
bellj@gmu.edu
http://jonbell.net
Twitter: @_jon_bell_
Office: 4422 Engineering Building; (703) 993-6089
Office Hours: Anytime electronically, **Tues 10:30am-12:00pm**, or by appointment

Uniform Interface for Resources

- Originally files on a web server
  - URL refers to directory path and file of a resource
- But… URIs might be used as an identity for any entity
  - A person, location, place, item, tweet, email, detail view, like
  - *Does not matter* if resource is a file, an entry in a database, retrieved from another server, or computed by the server on demand
  - Resources offer an *interface* to the server describing the resources with which clients can interact

URI: Universal Resource Identifier

- Uniquely describes a resource
  - https://mail.google.com/mail/u/0/#inbox/157d5fb795159ac0
  - https://www.amazon.com/gp/yourstore/home/ref=nav_cs_ys
  - http://gotocon.com/dl/goto-amsterdam-2014/slides/StefanTilkov_RESTIDontThinkItMeansWhatYouThinkItDoes.pdf
  - Which is a file, external web service request, or stored in a database?
    - It does not matter
- As client, only matters what actions we can *do* with resource, not how resource is represented on server

Intermediaries

Web “Front End”

“Origin” server

HTTP Request

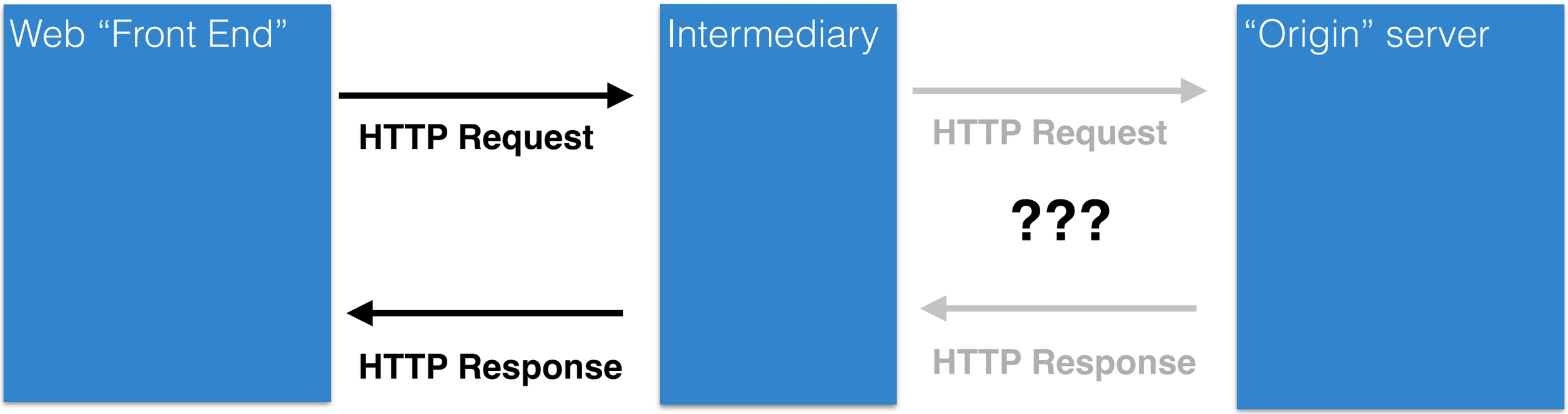
```
HTTP GET http://api.wunderground.com/api/
3bee87321900cf14/conditions/q/VA/Fairfax.json
```

HTTP Response

```
HTTP/1.1 200 OK
Server: Apache/2.2.15 (CentOS)
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
X-CreationTime: 0.134
Last-Modified: Mon, 19 Sep 2016 17:37:52 GMT
Content-Type: application/json; charset=UTF-8
Expires: Mon, 19 Sep 2016 17:38:42 GMT
Cache-Control: max-age=0, no-cache
Pragma: no-cache
Date: Mon, 19 Sep 2016 17:38:42 GMT
Content-Length: 2589
Connection: keep-alive


{
  "response": {
  "version":"0.1",
  "termsofService":"http://www.wunderground.com/weather/api/d/terms.html",
```

Intermediaries



- Client interacts with a resource identified by a URI
- But it never knows (or cares) whether it interacts with origin server or an unknown intermediary server
  - Might be randomly load balanced to one of many servers
  - Might be cache, so that large file can be stored locally
    - (e.g., GMU caching an OSX update)
  - Might be server checking security and rejecting requests

Challenges with intermediaries

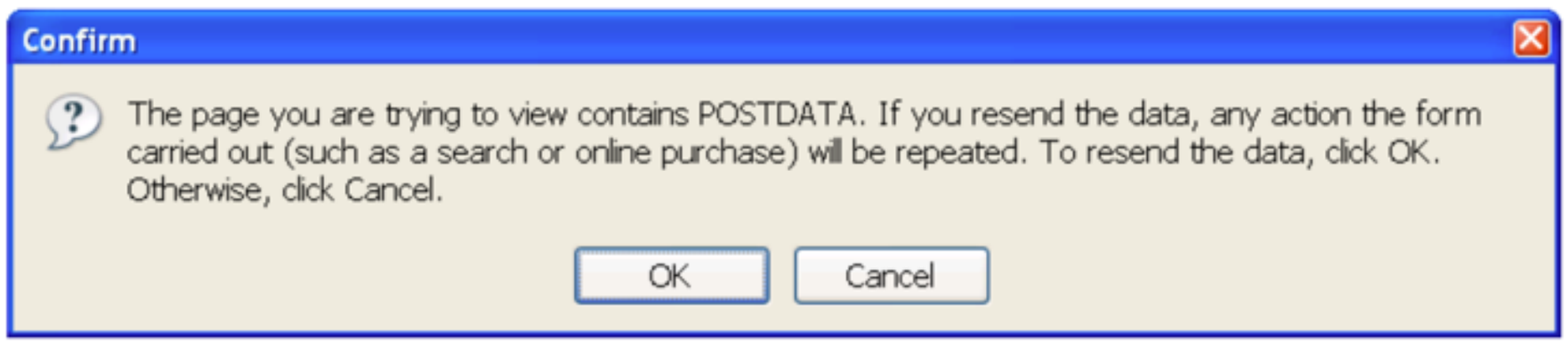
- But can all requests really be intercepted in the same way?
  - Some requests might produce a change to a resource
    - Can't just cache a response… would not get updated!
  - Some requests might create a change every time they execute
    - Must be careful retrying failed requests or could create extra copies of resources

HTTP Actions

- How do intermediaries know what they can and cannot do with a request?
- Solution: HTTP Actions
  - Describes what will be done with resource
  - GET: retrieve the current state of the resource
  - PUT: modify the state of a resource
  - DELETE: clear a resource
  - POST: initialize the state of a new resource

HTTP Actions

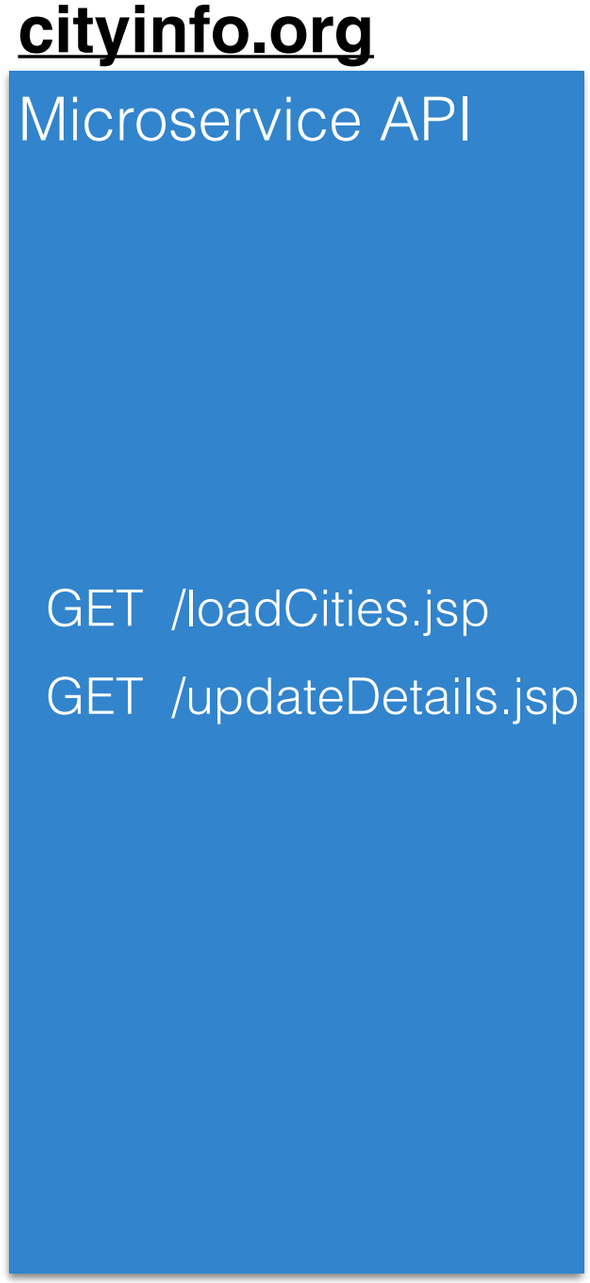
- GET: safe method with no side effects
  - Requests can be intercepted and replaced with cache response
- PUT, DELETE: idempotent method that can be repeated with same result
  - Requests that fail can be retried indefinitely till they succeed
- POST: creates new element
  - Retrying a failed request might create duplicate copies of new resource



Support scaling

- Yesterday, cityinfo.org had 10 daily active users. Today, it was featured on several news sites and has 10,000 daily active users.
- Yesterday, you were running on a single server. Today, you need more than a single server.

- Can you just add more servers?
  - What should you have done yesterday to make sure you can scale quickly today?



Support scaling

- Yesterday, cityinfo.org had 10 daily active users. Today, it was featured on several news sites and has 10,000 daily active users.
- Yesterday, you were running on a single server. Today, you need more than a single server.

- Can you just add more servers?
  - What should you have done yesterday to make sure you can scale quickly today?

Versioning

- Your web service just added a great new feature!
  - You'd like to expose it in your API.
  - But… there might be old clients (e.g., websites) built using the old API.
    - These websites might be owned by someone else and might not know about the change.
  - Don't want these clients to throw an error whenever they access an updated API.

Cool URIs don’t change

- In theory, URI could last forever, being reused as server is rearchitected, new features are added, or even whole technology stack is replaced.

- “What makes a cool URI?
  A cool URI is one which does not change.
  What sorts of URIs change?
  URIs don't change: people change them.”
  - https://www.w3.org/Provider/Style/URI.html
  - Bad:
    - https://www.w3.org/Content/id/50/URI.html (What does this path mean? What if we wanted to change it to mean something else?)
- Why might URIs change?
  - We reorganized our website to make it better.
  - We used to use a cgi script and now we use node.JS.

URI Design

- URIs represent a contract about what resources your server exposes and what can be done with them
- Leave out **anything that might change**
  - Content author names, status of content, other keys that might change
  - File name extensions: response describes content type through MIME header not extension (e.g., .jpg, .mp3, .pdf)
  - Server technology: should not reference technology (e.g., .cfm, .jsp)
- Endeavor to make all changes backwards compatible
  - Add new resources and actions rather than remove old
- If you must change URI structure, support old URI structure **and** new URI structure

Support change

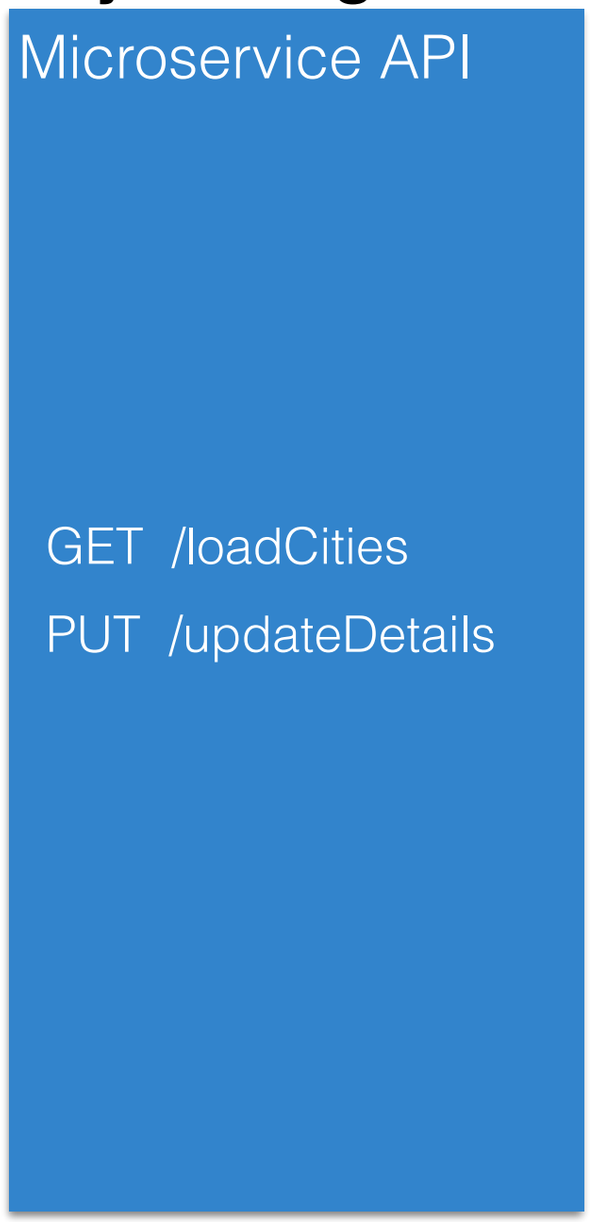
- Due to your popularity, your backend data provider just backed out of their contract and are now your competitor.
- The data you have is now in a different format.
- Also, you've decided to migrate your backend from PHP to node.js to enable better scaling.

- How do you update your backend without breaking all of your clients?

Support change

- Due to your popularity, your backend data provider just backed out of their contract and are now your competitor.
- The data you have is now in a different format.
- Also, you've decided to migrate your backend from PHP to node.js to enable better scaling.

- How do you update your backend without breaking all of your clients?



Nouns vs. Verbs

- URIs should hierarchically identify **nouns** describing **resources** that exist
- Verbs describing actions that can be taken with resources should be described with an HTTP **action**

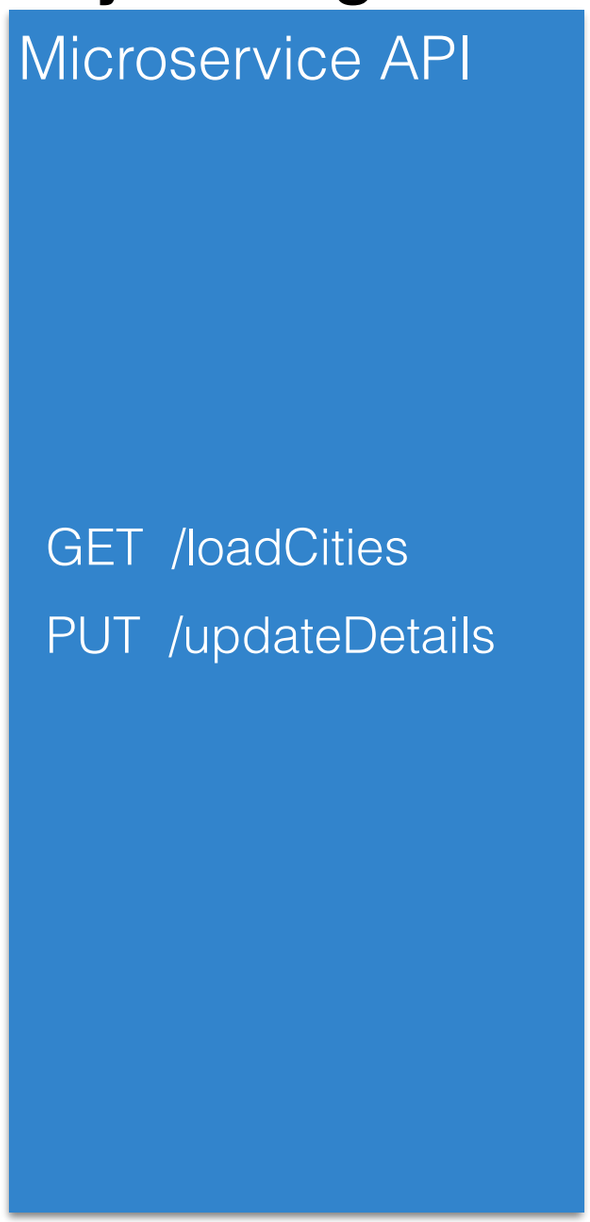
- PUT /cities/:cityID (nouns: cities, :cityID)(verb: PUT)
- GET /cities/:cityID (nouns: cities, :cityID)(verb: GET)

- Want to offer **expressive** abstraction that can be reused for many scenarios

Support reuse

- You have your own frontend for cityinfo.org. But everyone now wants to build their own sites on top of your city analytics.

- Can they do that?



Support reuse

cityinfo.org

Microservice API

/topCities GET
/topCities/:cityID/descrip PUT, GET

/city/:cityID GET, PUT, POST, DELETE
/city/:cityID/averages GET
/city/:cityID/weather GET
/city/:cityID/transitProvders GET, POST
/city/:cityID/transitProvders/:providerID GET, PUT, DELETE

What happens when a request has many parameters?

- /topCities/:cityID/descrip PUT

- Shouldn't this really be something more like
  - /topCities/:cityID/ descrip/:descriptionText/:submitter/:time/

Solution 1: Query strings

- PUT /topCities/Memphis?submitter=Dan&time=1025313

```
var express = require('express');
var app = express();

app.put('/topCities/:cityID', function(req, res){
    res.send(`descrip: ${req.query.descrip} submitter: ${req.query.submitter}`);
});

app.listen(3000);
```

- Use req.query to retrieve
- Shows up in URL string, making it possible to store full URL
  - e.g., user adds a bookmark to URL
- Sometimes works well for short params

Solution 2: JSON request body

- PUT /topCities/Memphis
  { "descrip": "Memphis is a city of ...",
    "submitter": "Dan", "time": 1025313 }
- Best solution for all but the simplest parameters (and often times everything)
- Use body-parser package and req.body to retrieve

```
$npm install body-parser
```

```
var express    = require('express');
var bodyParser = require('body-parser');

var app = express();

// parse application/json
app.use(bodyParser.json());

app.put('/topCities/:cityID', function(req, res){
    res.send(`descrip: ${req.body.descrip} submitter: ${req.body.submitter}`);
});

app.listen(3000);
```

https://www.npmjs.com/package/body-parser

How do you persist state?

- Can save state as global variables.
- Cons
  - State will be lost whenever server restarts
  - State will not be shared across multiple servers
- Sometimes useful as a cache
- We'll look at better approaches in a few lectures...

```
var express    = require('express');
var bodyParser = require('body-parser');

var app = express();

var cities = new Map();

// parse application/json
app.use(bodyParser.json());

app.put('/topCities/:cityID', function(req, res){
    cities.set(req.params.cityID, req.body);
});

app.listen(3000);
```