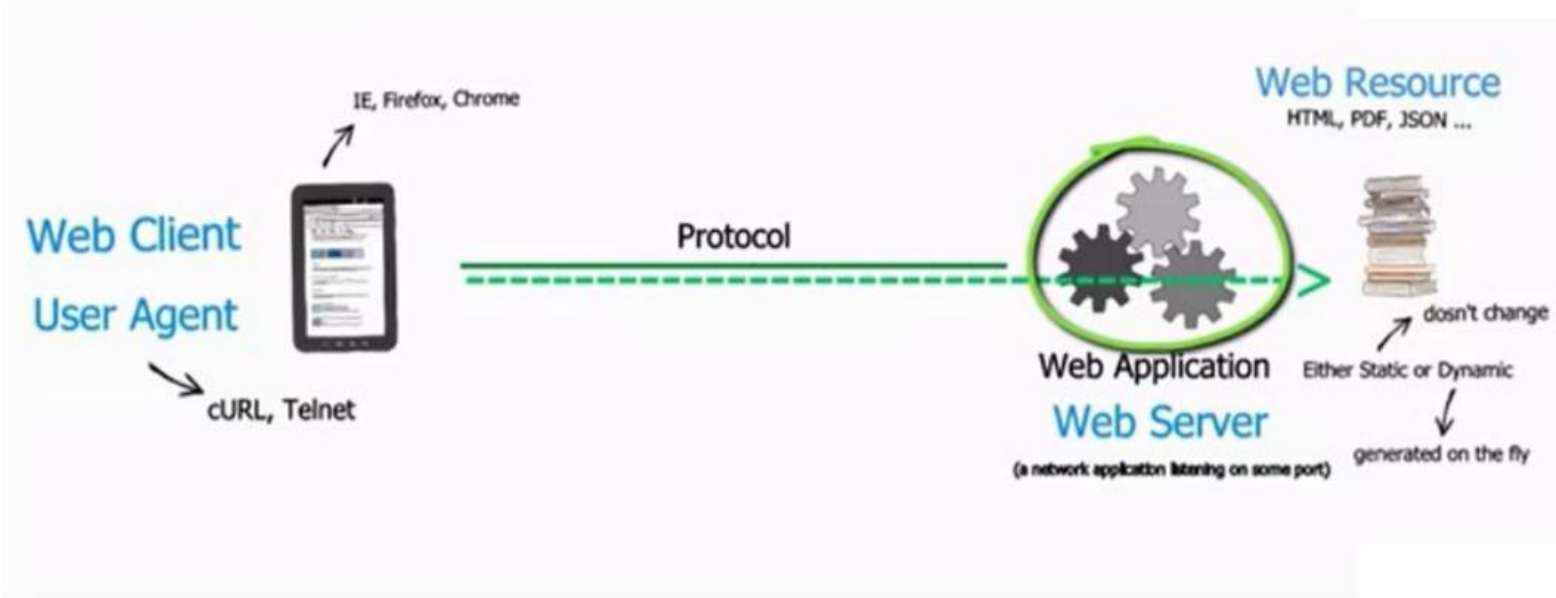
REST

Spring 2017
Software Development
Lab 6

Slides by Matthew Wright


IE, Firefox, Chrome
Web Client
User Agent
cURL, Telnet
Protocol
Web Resource
HTML, PDF, JSON ...
Web Application
Web Server
(a network application listening on some port)
dosn't change
Either Static or Dynamic
generated on the fly

Web – system of internet servers that support exchange of specially formatted documents.

Special format – HTML – HyperText Markup Language

Client and Server

HTTP – HyperText Transfer Protocol – how messages are formatted and transmitted

What is REST?

REST:

- Representational State Transfer
- Rest is an architectural style
- REST is about resources and how to represent resources in different ways
- REST is about client-server communication
- RESTful systems have *Six Architectural Constraints*, that give the service desirable properties such as scalability, simplicity, portability, reliability

REST: Architectural Constraints

1. Client-Server

a. Clients are not concerned with data storage, which remains internal to each server, so that the *portability of client code is improved*.

b. Servers are not concerned with the user interface or user state, so that *servers can be simpler and more scalable*.

c. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

REST: Architectural Constraints

2. Stateless

a. Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers.

b. *Every client request has enough information for the request to stand independent of other requests.* Only the client knows (or could ever need to know) the session state.

3. Cacheable

a. As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests.

b. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance

REST: Architectural Constraints

4. Uniform Interface: Simplifies and decouples the architecture, enabling each part to evolve independently. There are four guiding principles to Uniform Interface

    a. Resource Based

    b. Manipulation of Resources through Representations

    c. Self-descriptive Messages

    d. Hypermedia as the Engine of Application State

5. Layered System

    a. A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. *Intermediary servers may improve system scalability* by enabling load-balancing and by providing shared caches.

    b. Layers can be used to ensure server security.

REST: Architectural Constraints

6. Code on Demand (optional)

    a. This is the only optional constraint, every other constraint is necessary for a system to be considered RESTful

    b. Servers can temporarily extend or customize the functionality of a client by the transfer of executable code.

    Complying with these constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability and reliability.

Why REST?

- REST is a simple, lightweight alternative to bulkier webservices like SOAP, WSDL
    - Feel free to read into REST and SOAP more, these are good topics to explore and can be talked about during some of your projects.
    - Furthering this, REST is not a standard, and there will likely never be a “standard” in W3C (World Wide Web Consortium), but most people find it a great way to produce scalable, responsive services.
- REST is full featured.
    - You can do just about anything with REST that you can do with other *Web Services*, though some actions may be easier or faster to implement through other means due to the lightweight nature of REST.

HTTP Verbs

- HTTP Verbs provide the action for our noun-based resources. They are used for many objects, and can be used for many nouns.
- We use four main verbs, GET, PUT, POST, and DELETE. These are not the only verbs, OPTIONS and HEAD are probably used the next most frequently, and there are some other HTTP verbs out there too.
- Think of an HTTP verb like you do a verb in standard english. It lets us provide action to an object or data.
    - We can GET ice cream from the freezer just as easily as we can GET our charger for our laptop. Two different objects have the same verb associated with them.

GET, PUT, POST, DELETE

1. GET

    a. GET is used to READ.

    b. GET requests read data without changing it. When they are used this way, they are considered to be *safe, as the request cannot modify or corrupt the data.*

    c. Because of this, GET should NEVER modify any resources on the server. Never, ever.

    d. Put all this together, and we find that GET is *idempotent, or the result of calling get will not change if called identically over and over.*

2. PUT

    a. PUT is used to UPDATE.

    b. PUT is not *safe*. But it *can be idempotent* and it is strongly recommended you let PUT remain idempotent (use POST for non-idempotent requests)

        i. If you update or create a resource with PUT, then make the same call again, it should have the same state as before. PUT can be non-idempotent if you use it to increment or something like that, but don't do that because you won't be cool if you use it that way.

GET, PUT, POST, DELETE

3. POST

a. POST is used to CREATE.

b. POST is neither *safe* nor *idempotent*. Calling it twice will likely result in two resources containing the same information.

c. Submits data to be processed to a specific resource. Use POST when the server or service is in charge of deciding the URI for the newly-created resource.

4. DELETE

a. DELETE is used to DELETE (Seems straight-forward enough, right?).

b. DELETE can be *idempotent* and it is recommended you use it that way. But this has a caveat

   i. Using it *idempotently*: Use it not to decrement, but to remove a resource. If you remove a resource, then remove it again, the end result of both will be the same.

   ii. The Caveat: If you DELETE a resource twice, the second call will likely return 404 (NOT FOUND). So it really isn't *idempotent,* but it is close. We're compromising with the term *idempotent* here, though it can be *idempotent* if you do it right.

GET, PUT, POST, DELETE

Here's a chart I stole from Wikipedia. I really should donate to them sometime.

HTTP methods

Uniform Resource Locator (URL)	GET	PUT	POST	DELETE
Collection, such as `http://api.example.com/resources/`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[17]	**Delete** the entire collection.
Element, such as `http://api.example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry within it.[17]	**Delete** the addressed member of the collection.

What is API?

Application Programming Interface:

An API (Application Programming Interface) is best thought of as a *contract* provided by one piece of computer software to another.

Pieces of software can interact with or without an API.

Without API:
An app finds the current weather in London by opening **http://www.weather.com/** and reading the webpage like a human does, interpreting the content.

With API:
An app finds the current weather in London by sending a message to the **weather.com** API (in a structured format like JSON). The **weather.com** API then replies with a structured response.

In the lab, you'll use...

1. jq: Essentially, jq processes (filters) JSON files. JSON stands for JavaScript Object Notation. JSON provides notation for data that both people and computers can read easily

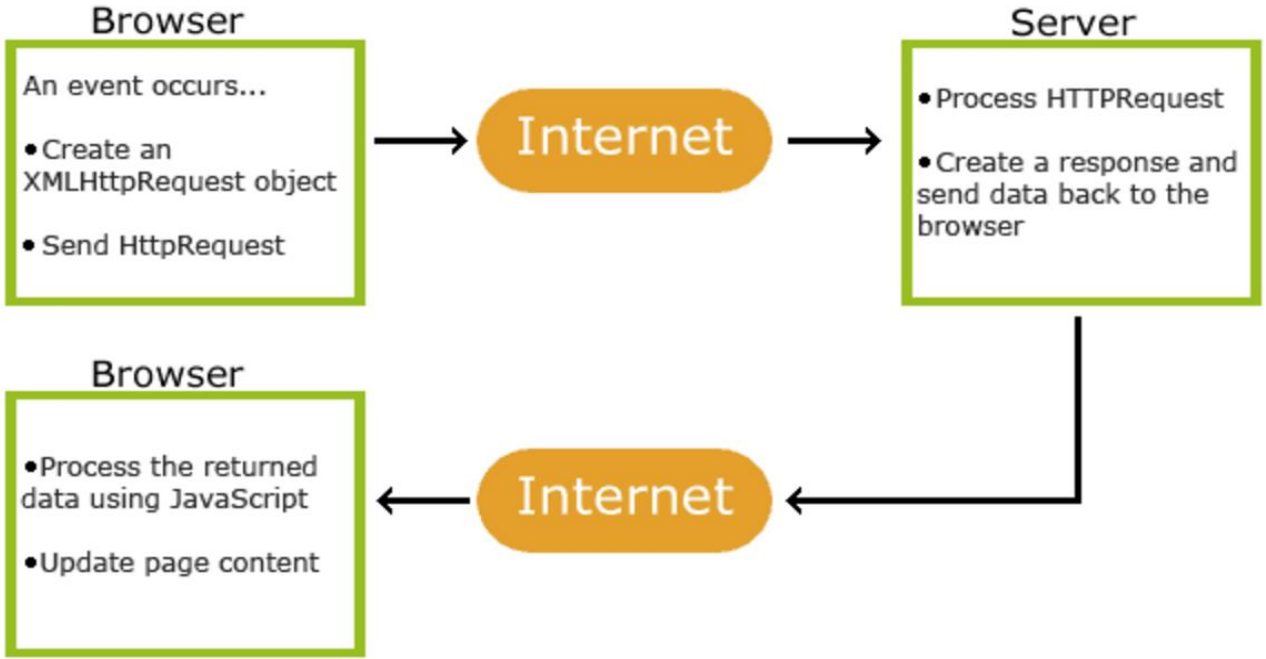
    a. https://stedolan.github.io/jq/ ← If you wanna know more.

2. API: You'll be using the API for darksky.net. API lets developers utilize services without having to do a bunch of research. We all like good API and good code commenting, however rare they might be.

3. Localhost: 127.0.0.1 the address for loopback Internet Protocall (IP). It establishes an IP connection with machine that is being used.

    a. 127.0.0.1 is common practice, any address like 127.*.*.* should work similarly, but just use 127.0.0.1 and you'll be one of the cool kids.

In the lab, you'll use... AJAX



Source:

http://www.w3schools.com/xml/ajax_intro.asp