44-563: Unit 08

Developing Web Applications and Services

Includes

- APIs
- REST
- SOAP
- AJAX
- No class Friday

APIs

API

An **application programming interface** (**API**) is a set of subroutine definitions, protocols, and tools for building software and applications.

A good API makes it easier to develop a program by defining all the building blocks, which are then put together by the programmer.

APIs may be private, public, or for partners.

https://en.wikipedia.org/wiki/Application_programming_interface

Purpose & Examples

An API is designed to make a developer's life easier. It abstracts away all the details of how a library works -- the implementation -- only revealing those objects/actions that the developer requires to do their work.

What examples of APIs have you seen?

https://en.wikipedia.org/wiki/Application_programming_interface

Web API

A **web API** is an application programming interface (API) for either a web server or a web browser.

- A **server-side web API i**s a programmatic interface consisting of publicly-exposed endpoints (methods) to a defined request–response message system, typically expressed in JSON or XML, which is exposed via the web—most commonly by means of an HTTP-based web server.
- A **client-side web API** is a programmatic interface to extend functionality within a web browser or other HTTP client.

Reusability: We can use a single server-side web API to support a web client, an Android client, and an iOS client.
Microservices: Have you heard of microservices? It's an emerging trend towards small, modular services that can be configured as needed.

https://en.wikipedia.org/wiki/Web_API

Web Resource vs Web Service

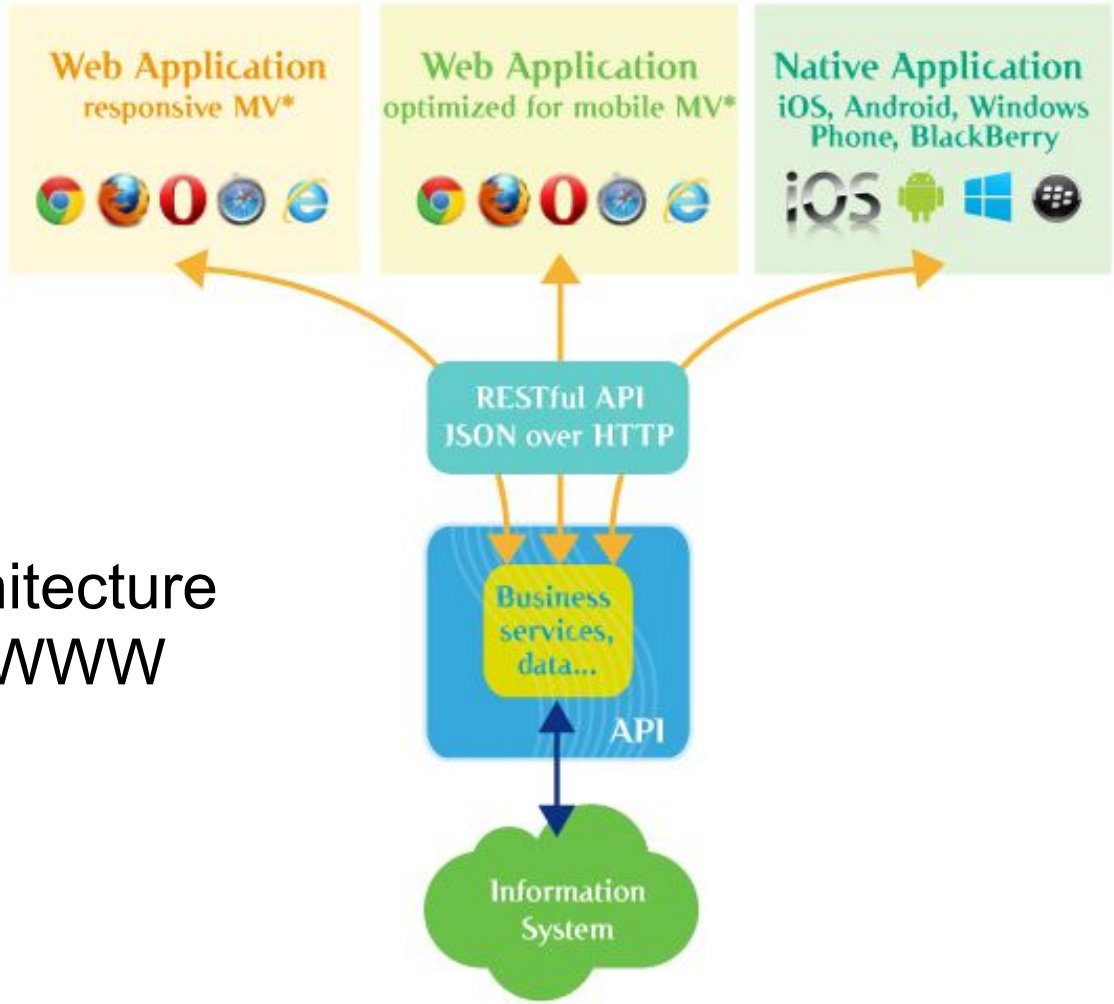
Web API is moving from **SOAP-based web services** towards more cohesive collections of **RESTful web resources**.

- RESTful web APIs are accessible via *standard HTTP methods* by a variety of HTTP clients including browsers and mobile devices.
- They have advantages over web services in that they tend to be less difficult to develop and less resource intensive (and thus usually run faster) since they do not need to perform as many data conversions as required by SOAP-based service APIs.

https://en.wikipedia.org/wiki/Web_API


REST

REST is an architectural style

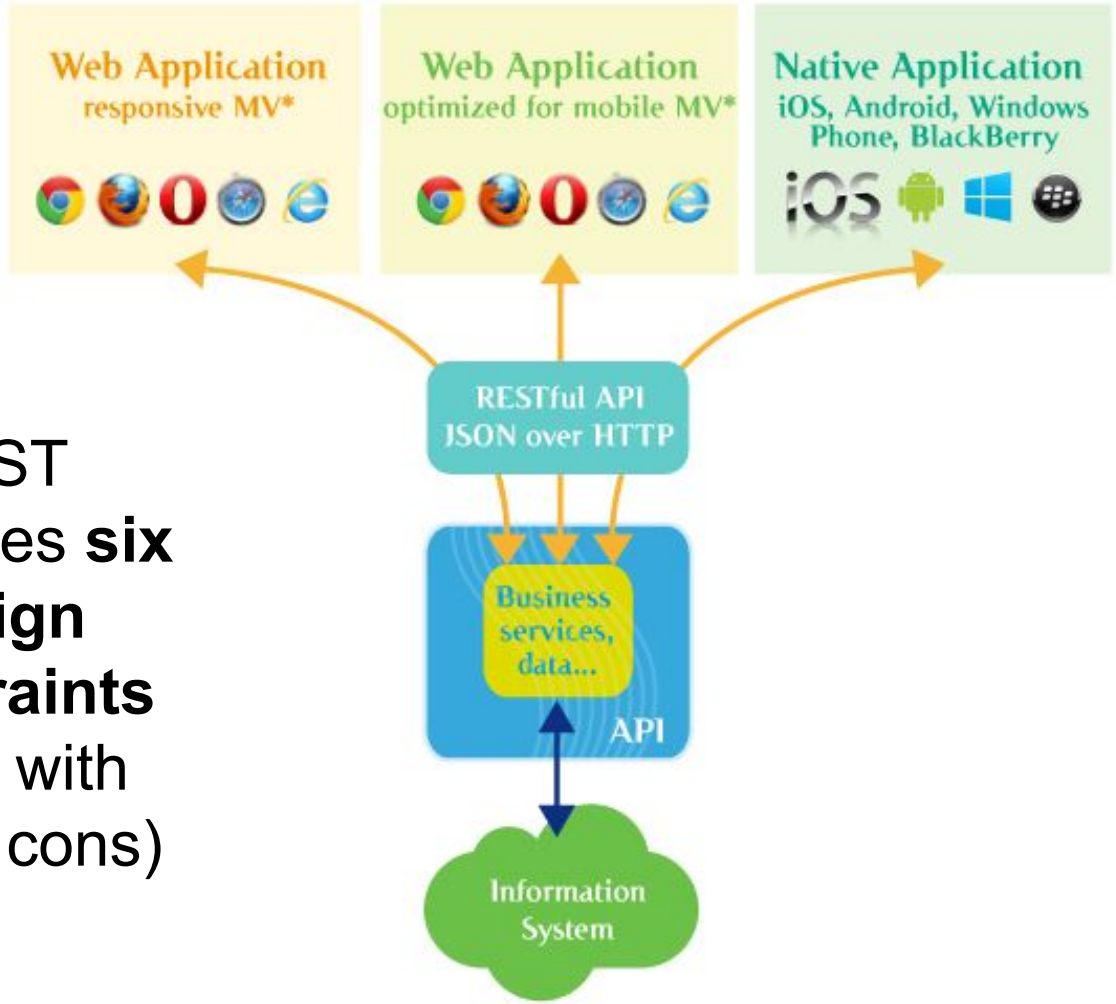


The architecture of the WWW

Many heterogeneous clients + common back-end code

http://blog.octo.com/en/new-web-application-architectures-and-impacts-for-enterprises-2/

REpresentational **S**tate **T**ransfer



REST describes **six design constraints** (each with pros & cons)

Many heterogeneous clients / common back-end code

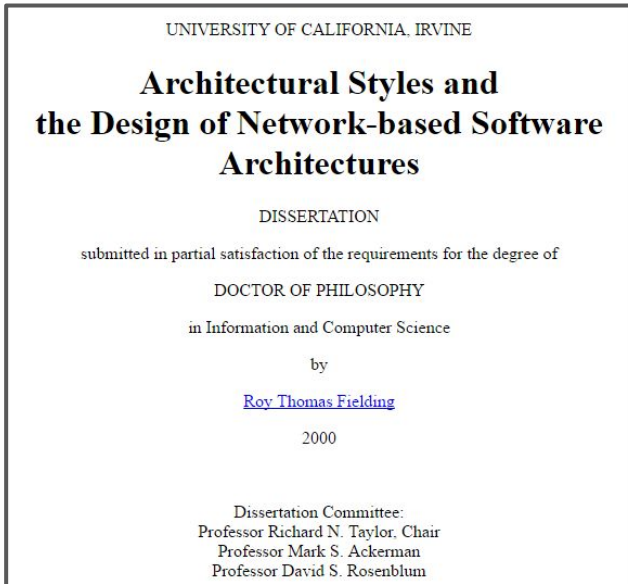
http://blog.octo.com/en/new-web-application-architectures-and-impacts-for-enterprises-2/

Roy Fielding

RESTful principles were developed by Roy Fielding in his PhD thesis, which was written in 2000.

He also:

- wrote RFC 2616, which specifies how HTTP 1.1 works,
- co-founded the Apache HTTP Web Server Project
- served as principal scientist at Adobe Systems



UNIVERSITY OF CALIFORNIA, IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

2000

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

REST: 6 constraints

6 **design rules** establish distinct characteristics of REST architectural style:

1. **Client-Server** (different; separation of concerns)
2. **Stateless** (state must be maintained by the client)
3. **Uniform Interface**
    - Resource-based
    - Clients can manipulate resources on the server through representations
    - Self-descriptive messages
    - Hypermedia as Engine of Application State (HATEOAS) - clients deliver state through requests / servers through responses

The RESTful Developer's Pledge

We solemnly swear to use GET, POST, UPDATE and DELETE only for retrieval, creation, updating and deleting, respectively.

REST: 6 constraints

6 **design rules** establish the distinct characteristics of the REST architectural style:

4. **Cache** (clients can cache responses)
5. **Layered System** - looks the same to the client (regards of addl layers for things like intermediaries, load balancing, security)
6. **Code-On-Demand** - server can supply logic to be executed on client (enables better performance, scalability, simplicity, modifiability, visibility, portability and reliability)

http://www.restapitutorial.com/lessons/whatisrest.html#

REST: Resource-based

- Resources are **nouns**.
- Resources are named from the perspective of the API user or **consumer**.
- Identify resources, then identify the actions needed.
- Leverage existing HTTP methods, e.g.,
  - GET
  - PUT
  - POST
  - DELETE
- Parameters go in the URL (no XML needed)

The RESTful Developer's Pledge

We solemnly swear to use GET, POST, UPDATE and DELETE only for retrieval, creation, updating and deleting, respectively.

REST: Resource-based

For example, if you're creating a website for **support tickets**, you might create an API that looks like the following:

- `GET /tickets` - Retrieves a list of tickets
- `GET /tickets/12` - Retrieves a specific ticket
- `POST /tickets` - Creates a new ticket
- `PUT /tickets/12` - Updates ticket #12
- `PATCH /tickets/12` - Partially updates ticket #12
- `DELETE /tickets/12` - Deletes ticket #12

http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

REST: Relations

One **support ticket** can have **many** messages:

- `GET /tickets/12/messages` - Retrieves list of messages for ticket #12
- `GET /tickets/12/messages/5` - Retrieves message #5 for ticket #12
- `POST /tickets/12/messages` - Creates a new message in ticket #12
- `PUT /tickets/12/messages/5` - Updates message #5 for ticket #12
- `PATCH /tickets/12/messages/5` - Partially updates msg #5 for ticket #12
- `DELETE /tickets/12/messages/5` - Deletes message #5 for ticket #12

http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

REST: Non-CRUD

Not everything fits easily into standard CRUD.

Think of the **user**.

- GitHub's API lets you star a gist with
  a. `PUT /gists/:id/star` and unstar with
  b. `DELETE /gists/:id/star`.
- a multi-resource search doesn't really make sense to be applied to a specific resource's endpoint. In this case, `/search` may be simple and clear.

CRUD=
Create
Read
Update
Delete

http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api


SOAP

SOAP

- Simple Object Access **Protocol** is an XML-based **protocol specification** for exchanging **structured** information in the implementation of **web services** in computer networks (could use HTTP, SMTP, others)
- Tightly coupled. Client- and server-side typically written together
- Often complex (banking, power plants, etc.)

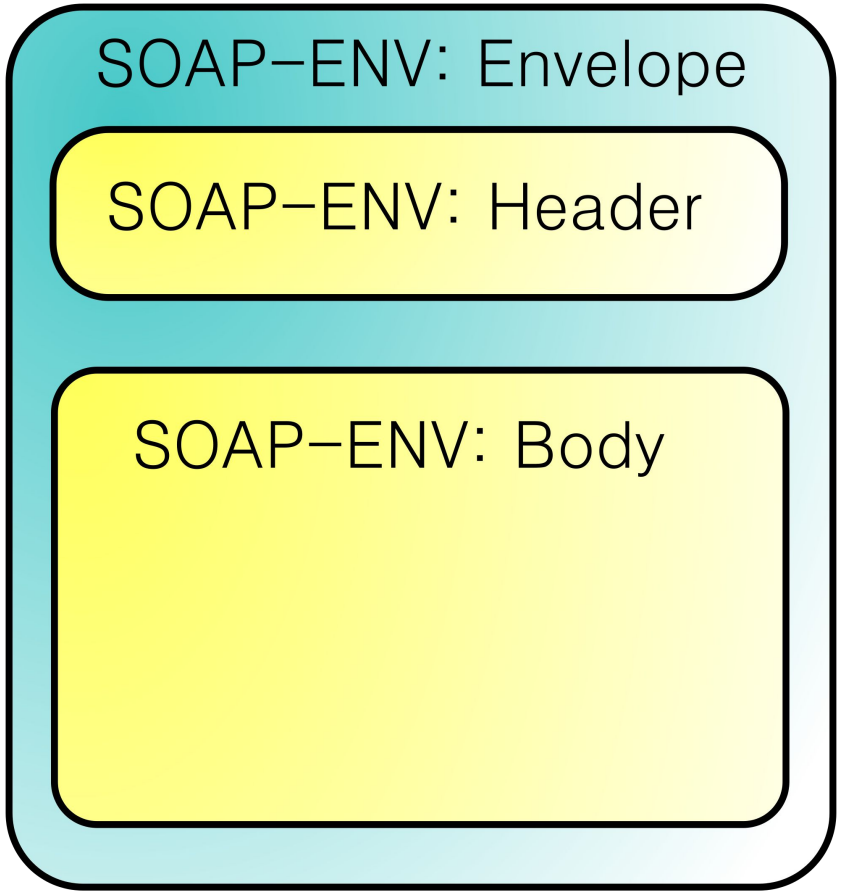
Web services use XML to **encode** data & SOAP to **transfer** it

https://en.wikipedia.org/wiki/SOAP
https://stackify.com/soap-net-core/

SOAP Message

A SOAP **message** is an **XML document** with:

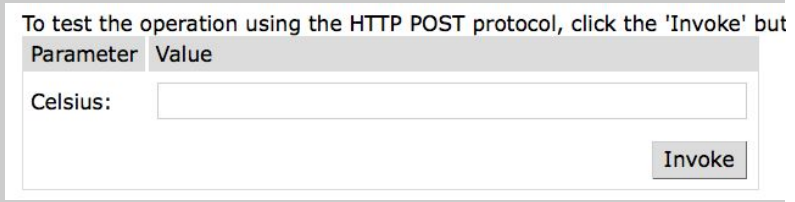
- An **Envelope** element that identifies the XML document as a SOAP message
- A **Header** element that contains header information
- A **Body** element that contains call and response information
- A **Fault** element containing errors and status information. It is optional.



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

A SOAP Message Example

```
POST /xml/tempconvert.asmx HTTP/1.1
Host: www.w3schools.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <CelsiusToFahrenheit xmlns="https://www.w3schools.com/xml/">
      <Celsius>20</Celsius>
    </CelsiusToFahrenheit>
  </soap12:Body>
</soap12:Envelope>
```



To invoke a web service, the body, in a POST message, must specify the name of the action to be called on the web service, and any needed arguments.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CelsiusToFahrenheitResponse xmlns="https://www.w3schools.com/xml/">
      <CelsiusToFahrenheitResult>68</CelsiusToFahrenheitResult>
    </CelsiusToFahrenheitResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP Use Cases

- Provide **reusable** application components (e.g., a dashboard -- very specific to a particular application/client)
- **Connect** distributed software applications
- **Transactions** (all or none)
- **Security on data layer** as well as transport layer
- **WSDL** specifies data structure for server & client (name of each action method, parameters, and return values)
- Tools use WSDL to automatically generate code proxies, on the client, to interact with the web service

Partial SOAP Response

```
<CurrentPrice>8.99</CurrentPrice>
<Currency>USD</Currency>
```

Partial WSDL Describing SOAP Response

```
<xsd:element name="CurrentPrice" type="decimal" />
<xsd:element name="Currency" type="string" />
```

Example

- .NET makes SOAP easy
- VBScript
- Easy to build
- Don't write SOAP
- Don't write WSDL
- ASP.NET does it

```
<%@ WebService Language="VBScript" Class="TempConvert" %>

Imports System
Imports System.Web.Services

Public Class TempConvert :Inherits WebService

<WebMethod()> Public Function FahrenheitToCelsius
(ByVal Fahrenheit As String) As String
  dim fahr
  fahr=trim(replace(Fahrenheit,",","."))
  if fahr="" or IsNumeric(fahr)=false then return "Error"
  return ((((fahr) - 32) / 9) * 5)
end function

<WebMethod()> Public Function CelsiusToFahrenheit
(ByVal Celsius As String) As String
  dim cel
  cel=trim(replace(Celsius,",","."))
  if cel="" or IsNumeric(cel)=false then return "Error"
  return ((((cel) * 9) / 5) + 32)
end function

end class
```

tempconvert.asmx

Try it: http://www.w3schools.com/xml/tempconvert.asmx

SOAP Web Services

- are application **components**
- communicate using **open protocols**
- are **self-contained** & **self-describing (via WSDL)**
- can be discovered using **UDDI**
- can be used by other applications

HTTP and XML are the basis for SOAP Web services (can use other protocols as well)

UDDI is an XML-based standard for describing, publishing, and finding web services. **UDDI** stands for Universal Description, Discovery, and Integration. **UDDI** is a specification for a distributed registry of web services.

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

Try it: Which column describes SOAP? Which describes REST?

SOAP or REST?	SOAP or REST?
standard **protocol** for creating web services.	**architectural style** to serve web resources.
uses **WSDL** to expose supported methods and technical details.	exposes methods through URIs, there are no technical details.
web services and client programs are bound with WSDL **contract**	**no contract** between server & client
web services & client are **tightly coupled** with contract.	Server and client are **loosely coupled.**

http://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-web-services

SOAP	REST
standard **protocol** for creating web services.	**architectural style** to serve web resources.
uses **WSDL** to expose supported methods and technical details.	exposes methods through URIs, there are no technical details.
web services and client programs are bound with WSDL **contract**	**no contract** between server & client
web services & client are **tightly coupled** with contract.	Server and client are **loosely coupled.**

http://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-web-services

Try it: Which column describes SOAP? Which describes REST?

SOAP or REST?	SOAP or REST?
supports **only XML**	supports **any data type** such as XML, JSON, image etc.
services are hard to maintain, any change in WSDL contract requires us to create client stubs again and make changes to client code.	services are easier to maintain, new methods can be added without any client- side change for existing resources.
tested through programs or software such as Soap UI.	tested with CURL command, Browsers & extensions, e.g., Postman.

http://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-web-services

SOAP	REST
supports **only XML**	supports **any data type** such as XML, JSON, image etc.
services are hard to maintain, any change in WSDL contract requires us to create client stubs again and make changes to client code.	services are easier to maintain, new methods can be added without any client- side change for existing resources.
tested through programs or software such as Soap UI.	tested with CURL command, Browsers & extensions, e.g., Postman.

http://www.journaldev.com/9193/web-services-interview-questions-soap-restful#rest-web-services

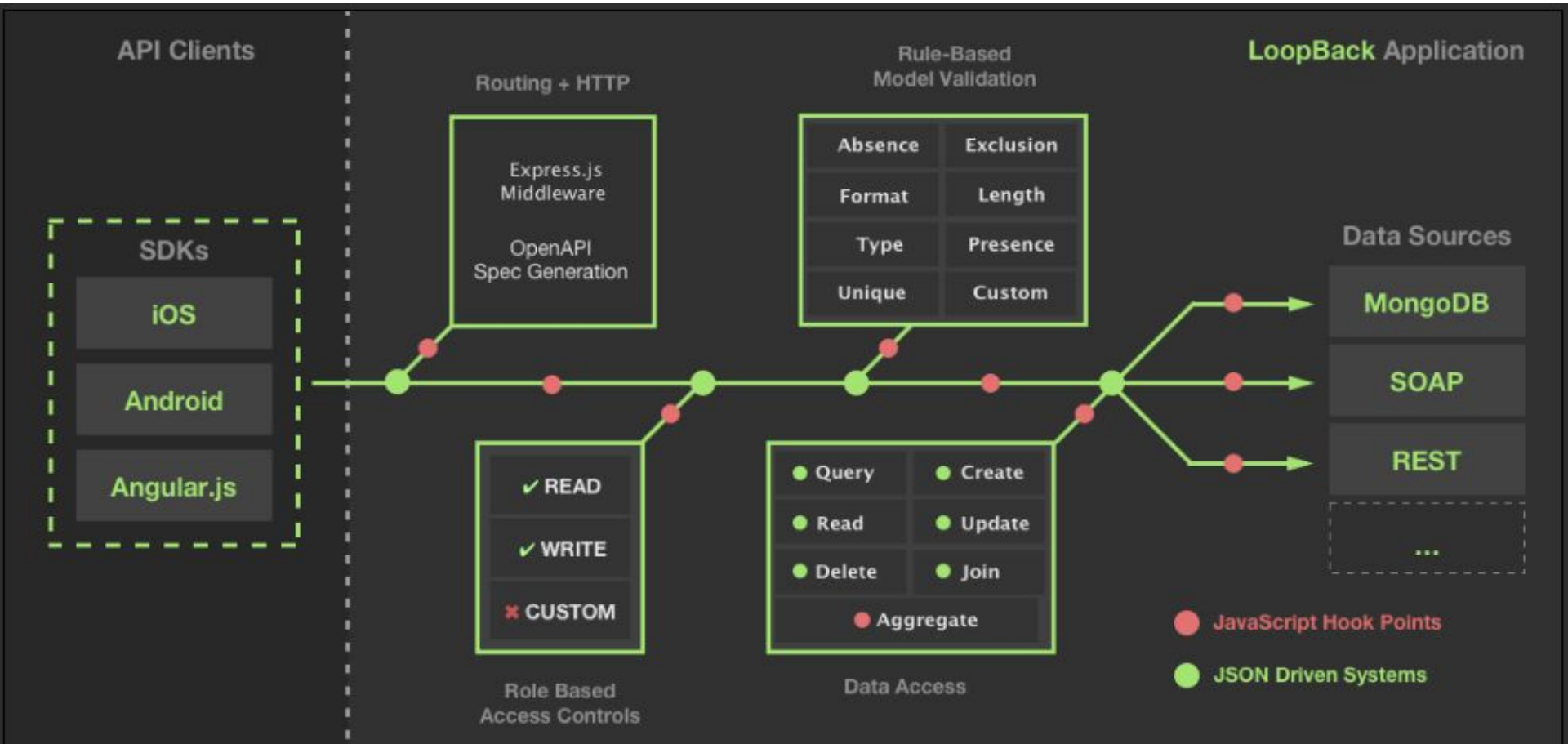
Loopback API Framework

- **API Framework** for Node.js
- Allows us to quickly create **REST APIs**.
- Connect devices & browsers to data & services.
- Use Android, iOS, and AngularJS SDKs to easily create client apps.
- Add-on components for file management, 3rd-party login, and OAuth2.
- Juggler connects to back-end data stores (mySQL, Oracle, MongoDB, etc.)

https://strongloop.com/strongblog/compare-express-restify-hapi-loopback/

Loopback

http://loopback.io/



```
var express = require('express');
var Item = require('models').Item;
var app = express();
var itemRoute = express.Router();

itemRoute.param('itemId', function(req, res, next, id) {
  Item.findById(req.params.itemId, function(err, item) {
    req.item = item;
    next();
  });
});

// Create new Items
itemRoute.post('/', function(req, res, next) {
  var item = new Item(req.body);
  item.save(function(err, item) {
    res.json(item);
  });
});

itemRoute.route('/:itemId')
  // Get Item by Id
  .get(function(req, res, next) {
    res.json(req.item);
  })
  // Update an Item with a given Id
  .put(function(req, res, next) {
    req.item.set(req.body);
    req.item.save(function(err, item) {
      res.json(item);
    });
  })
  // Delete and Item by Id
  .delete(function(req, res, next) {
    req.item.remove(function(err) {
      res.json({});
    });
  });

app.use('/api/items', itemRoute);
app.listen(8080);
```

The Loopback Advantage

At left is what's needed to set up basic CRUD functionality with Express. Below is the StrongLoop/LoopBack equivalent.

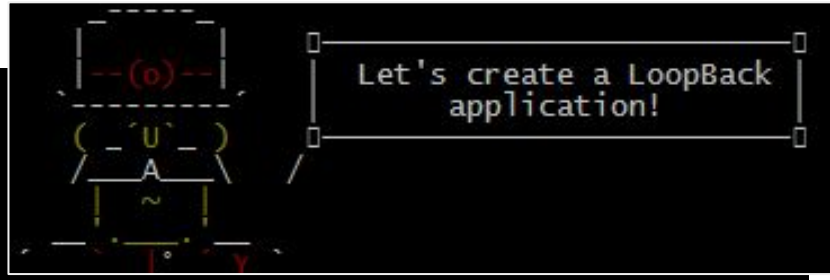
```
var loopback = require('loopback');
var app = module.exports = loopback();

var Item = loopback.createModel(
  'Item',
  {
    description: 'string',
    completed: 'boolean'
  }
);

app.model(Item);
app.use('/api', loopback.rest());
app.listen(8080);
```

https://strongloop.com/strongblog/compare-express-restify-hapi-loopback/

Let's Build It (M08)

Create C:\44563**m08** folder. Open a **cmd window*** here

Install IBM's **strongloop api platform** and create a loopback app.




```
> npm install -g strongloop

> slc loopback
[?] What's the name of your application? M08
  What version do you want to use? 2.x (long term support)
  What kind of application do you have in mind? api-server

  I'm all done. Running npm install for you to install
  the required dependencies.
  ...
```

Accept the defaults.

*using Git Bash here just led to tears (or at least errors)

Let's Build It (M08)

It tells us how to proceed..

```
Next steps:

  Create a model in your app
    $ slc loopback:model

  Run the app
    $ node .
```

We'll start with building a **model** - one model for each noun / resource in our app. We'll do a person - you can create any model you like. Models have **properties**.

Let's Build It (M08)

Create a loopback model.

slc = "Strong Loop Command" utility
Enter the values highlighted in green. To accept the default, press **Enter**.
lb may work now, too.

```
> cd m08 # if needed
> slc loopback:model

[?] Enter the model name: person
[?] Select the data-source to attach person to: db (memory)
[?] Select model`s base class (PersistedModel)
[?] Expose person via the REST API? Yes
[?] Custom plural form (used to build REST URL): people
[?] Common model or server only? common
Let's add some person properties now.
```

Let's Build It (M08)

```
Enter an empty property name when done.
[?] Property name: firstname

[?] Property type: (Use arrow keys)
› string
  number
  boolean
  object
  array
  date
  buffer
  geopoint
  any
  (other)

[?] Required? (y/N) y
```

1. Add firstname
2. **Repeat** for lastname

Accept the defaults (except firstname is required).

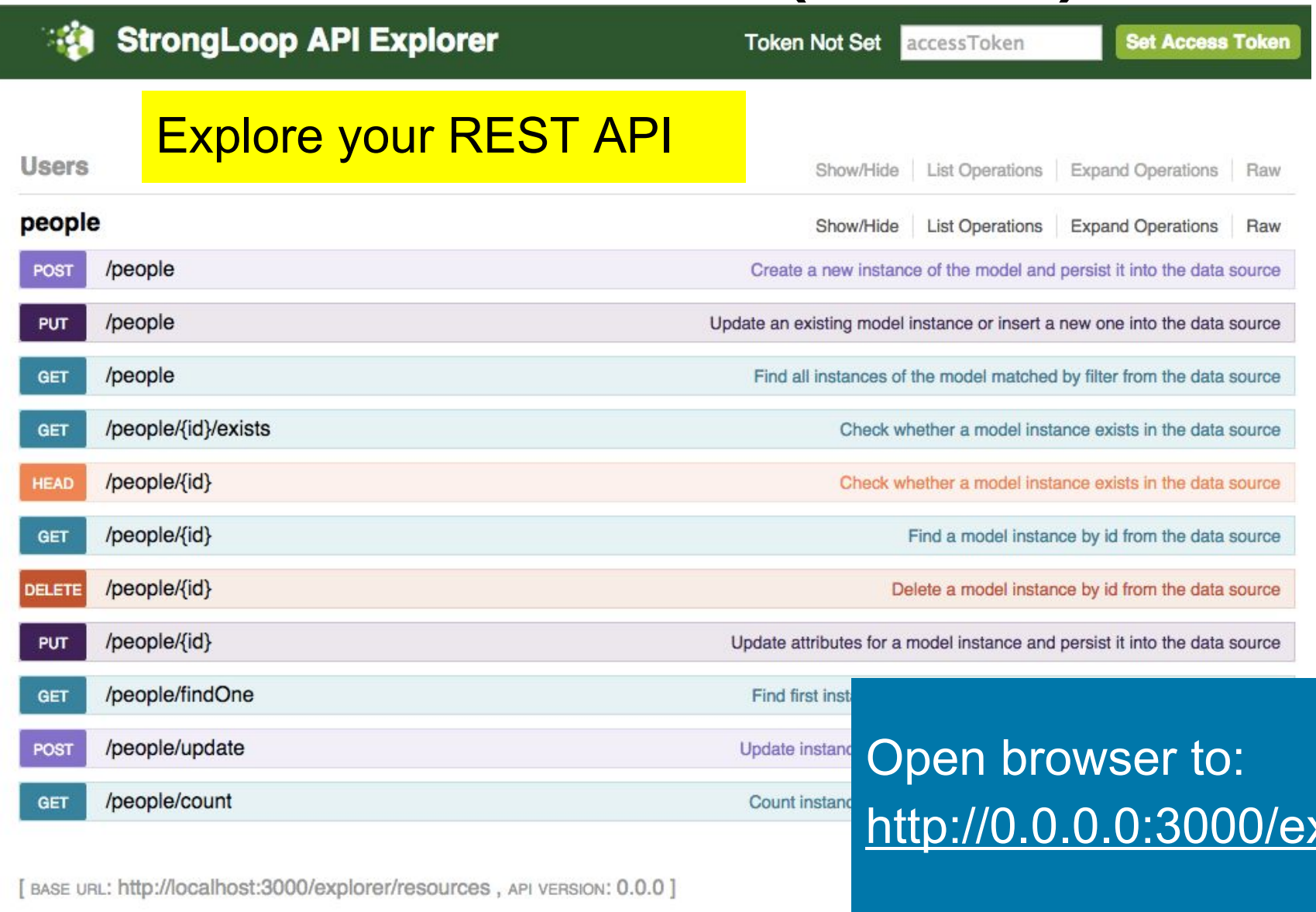
Let's Build It (M08)

Run the app.

The dot means "here" in the root folder.

```
// Run the app with node space dot

> node .
```

Let's Build It (M08)



Open browser to:
http://0.0.0.0:3000/explorer

Extra Fun!

Check out the Loopback Tutorial for more examples of what you can do with the technology.

http://loopback.io/doc/en/lb3/Tutorials-and-examples.html

AJAX

Reference:
"Introduction to AJAX for Java Web Applications"
http://netbeans.org/kb/docs/web/ajax-quickstart.html

AJAX

Asynchronous **Ja**vaScript and **X**ML (AJAX) is a group of interrelated web development techniques.

Allows a web browser to send messages back to the server without interrupting the flow of what's being shown in the browser - enables updating just a **part of the page**.

AJAX is a misnomer

AJAX: Asynchronous JavaScript and XML

- Not always asynchronous.
- Not always XML (JSON is common).
- ~~Always good at removing grease~~

Refreshing **part** of the page - not the whole thing.

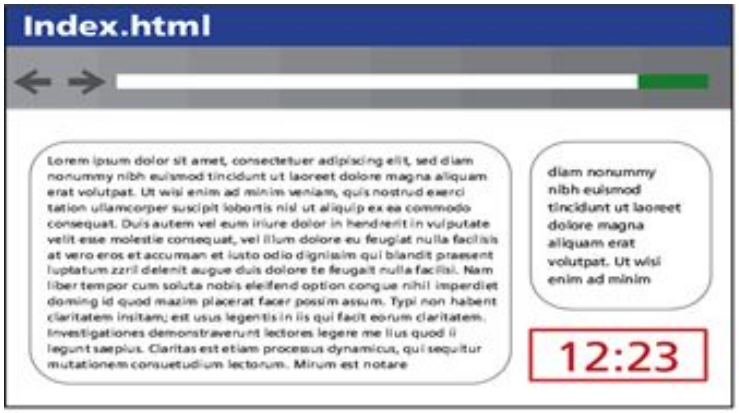
- Enables rich behavior (similar to that of a desktop application or plugin-based web application) using a browser.

Example uses:

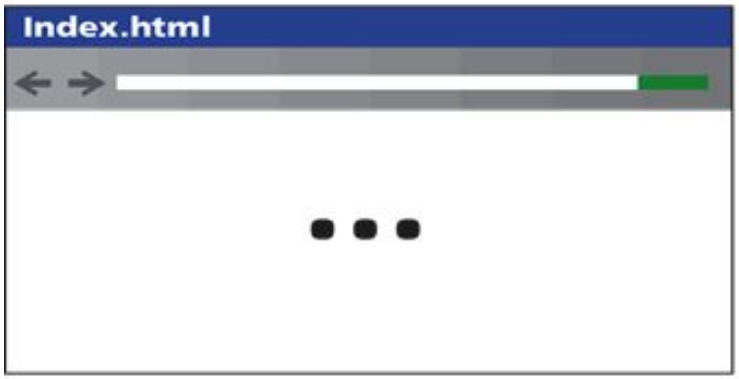
- validate form entries (while the user is entering them) using server-side logic
- retrieve detailed data from the server
- dynamically update data on a page
- submit partial forms

Consider a webpage that displays the server's time

Synchronous ("at the same time") means waiting for it to finish

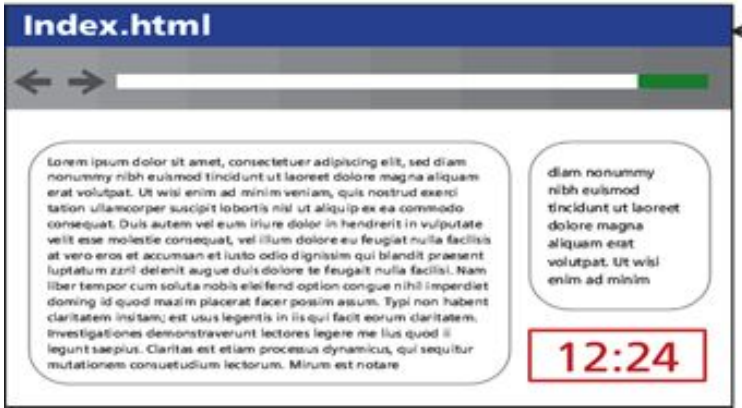


1 The page loads and shows the current server time as a small part of a larger page.



2 A **synchronous** JavaScript call makes an HTTP request for the "freshest" version of the page.

While waiting for the response, the browser goes into its waiting state.



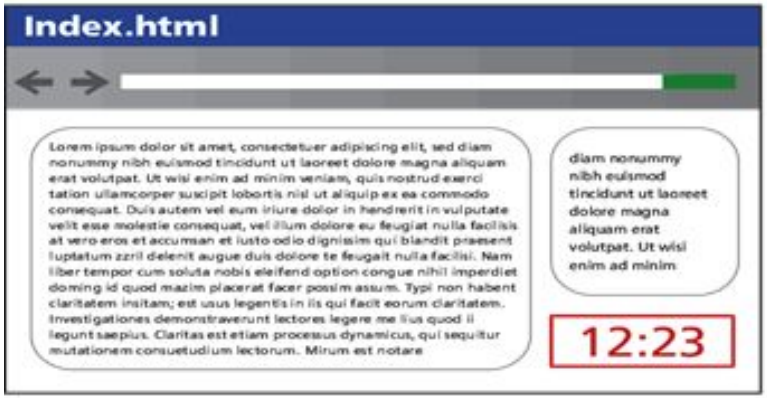
3 The response arrives, so the browser can render the new version of the page, and the functionality in the browser is restored.

```
<html>
        <head>
        ...
        </head>
        <body>
        ...
        <div id='serverTime'>
                12.24
        </div>
        ...
        </body>
</html>
```

Source: Connolly & Hoar

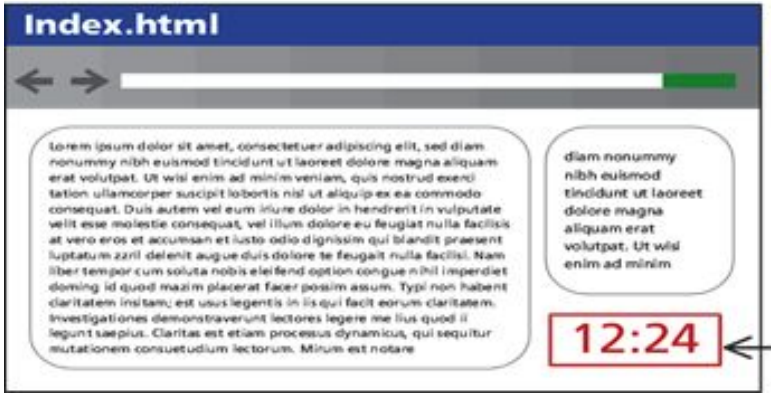
Consider a webpage that displays the server's time

1. The page loads and shows the current server time as a small part of a larger page.



2. An **asynchronous** JavaScript call makes an HTTP request for just the small component of the page that needs updating (the time).

   While waiting for the response, the browser still looks the same and is responsive to user interactions.



3. The response arrives, and through JavaScript, the HTML page is updated.

12.24

Asynchronous = NOT waiting for a response

Source: Connolly & Hoar

Why is AJAX popular?

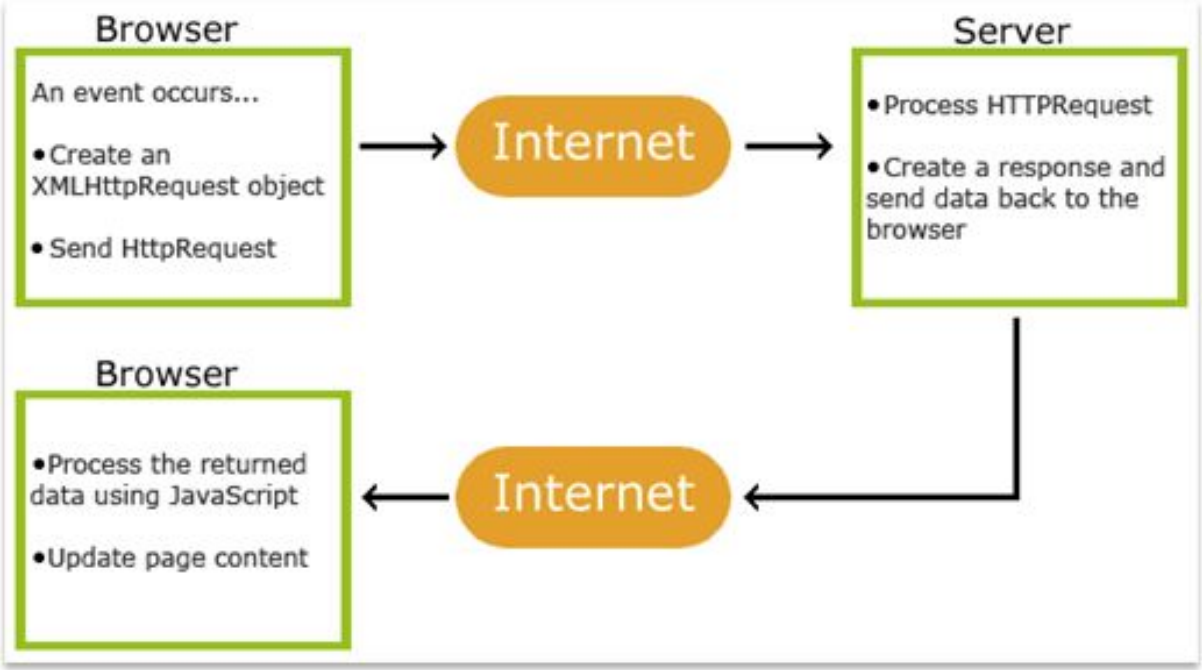
- No need to do a page refresh or full page reload for every user interaction
- Enables building **Rich Internet Applications** (RIA) -- applications that act like desktop applications running in a browser
- Allows dynamic interaction on the Web
- Improves **performance**
- Real-time updates

How does AJAX work?

- Initiated by JavaScript code
- When the AJAX interaction is complete, JavaScript updates the HTML source of the page
  - The changes are made immediately to just the affected parts without requiring a total page refresh

XMLHttpRequest object

- AJAX calls employ an **XMLHttpRequest** object to pass requests and responses between the client and server



Step 1: new request is created

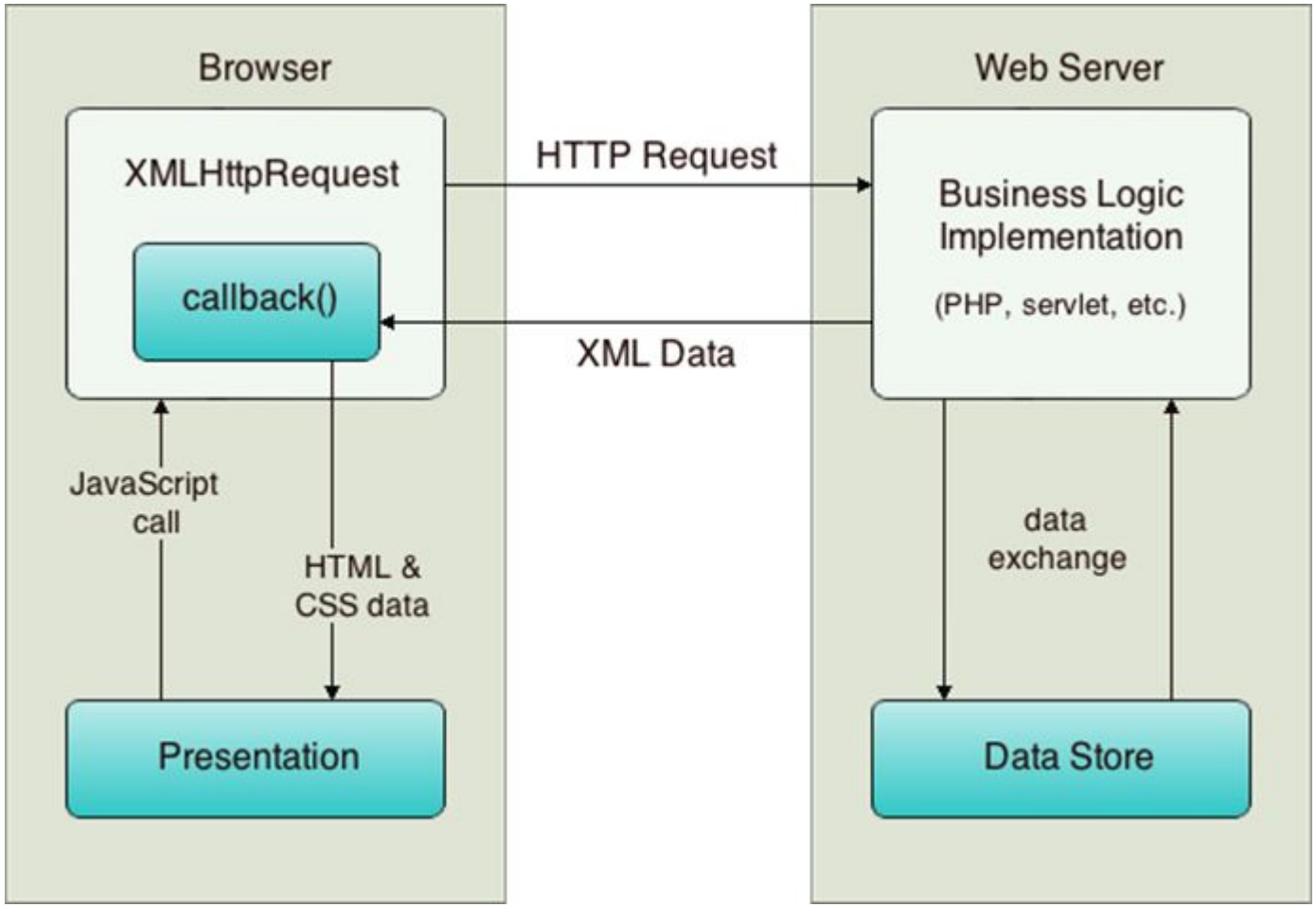
User triggers an event, for example, by releasing a key when typing in a name

–This results in a JavaScript call to a function that initializes an **XMLHttpRequest** object

```
let xhr = new XMLHttpRequest()
let  method = "GET"
let  url = "https://developer.mozilla.org/"
```

Step 1: trigger event creates an XMLHttpRequest object

(1)



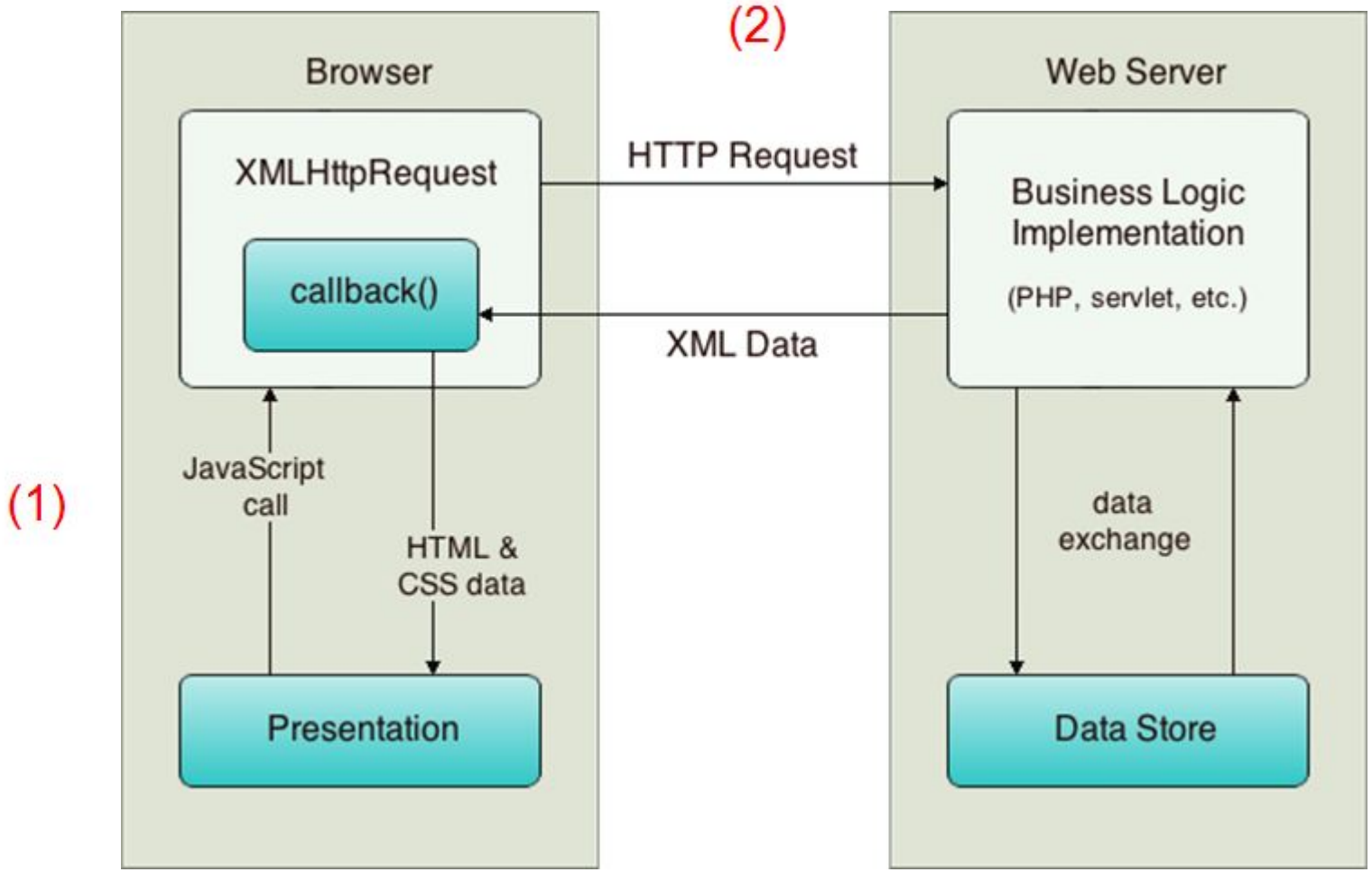
Step 2: This object sends a request

- The XMLHttpRequest object then sends an HTTP **request** to the web server

Method	Description
open(*method, url, async*)	Specifies the type of request *method*: the type of request: GET or POST *url*: the server (file) location *async*: true (asynchronous) or false (synchronous)
send()	Sends the request to the server (used for GET)

```
var xhr = new XMLHttpRequest(),
    method = "GET",
    url = "https://developer.mozilla.org/";
xhr.open(method, url, true);
xhr.send();
```

Step 2: This object sends a request



Step 3: server response

What kind of response is this example sending?

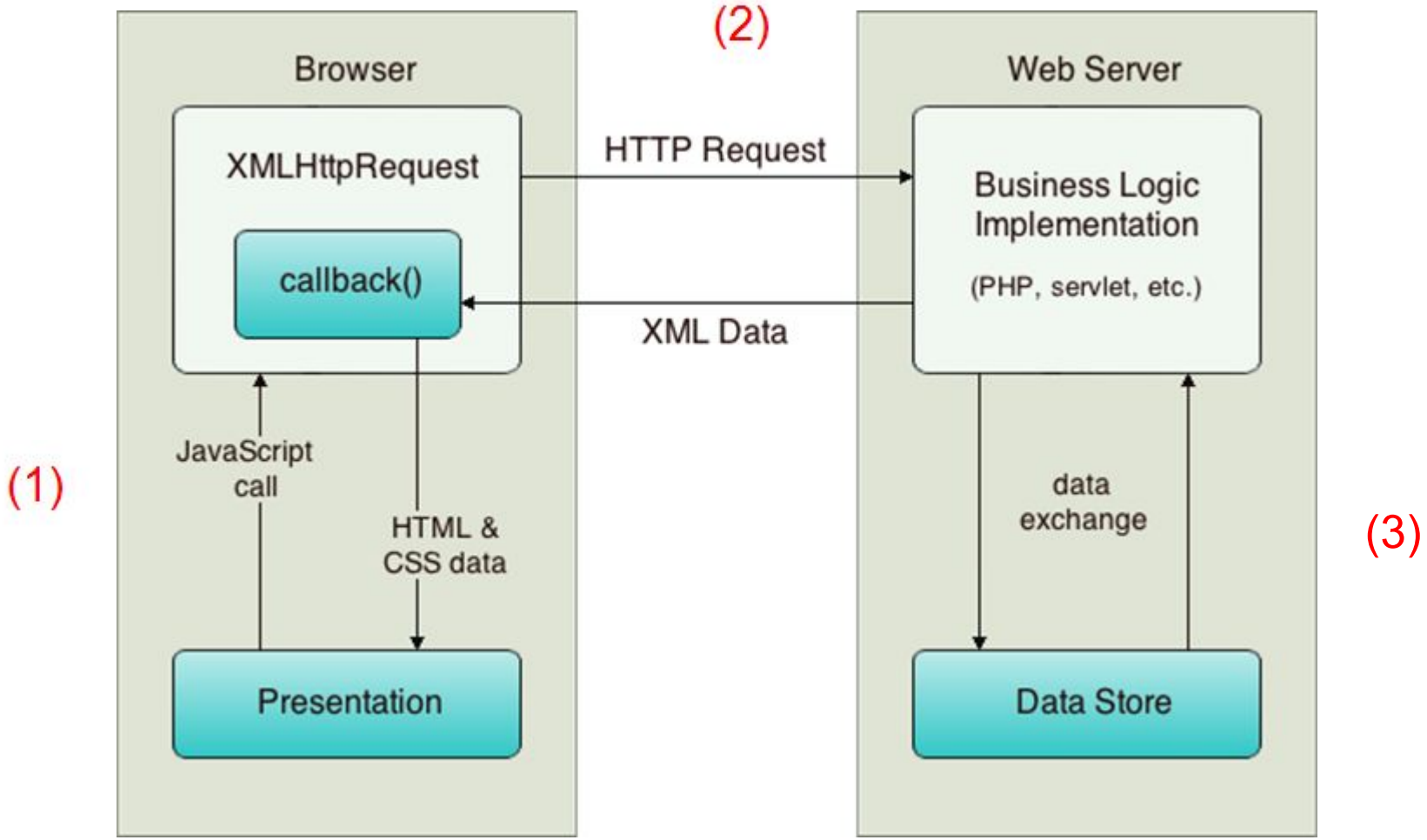
On a web server, a web service handles the request

–data is retrieved from the data store (if needed)

–a response is prepared containing the data in the form of XML, JSON, Text, HTML, or a View (code and HTML that dynamically creates HTML)

```
api.get('/findall', function(req, res){
    res.setHeader('Content-Type', 'application/json');
    var data = req.app.locals.waterproofingPrimers.query;
    res.send(JSON.stringify(data));
});
```

Step 3: server prepares and sends response



Step 4: client updates

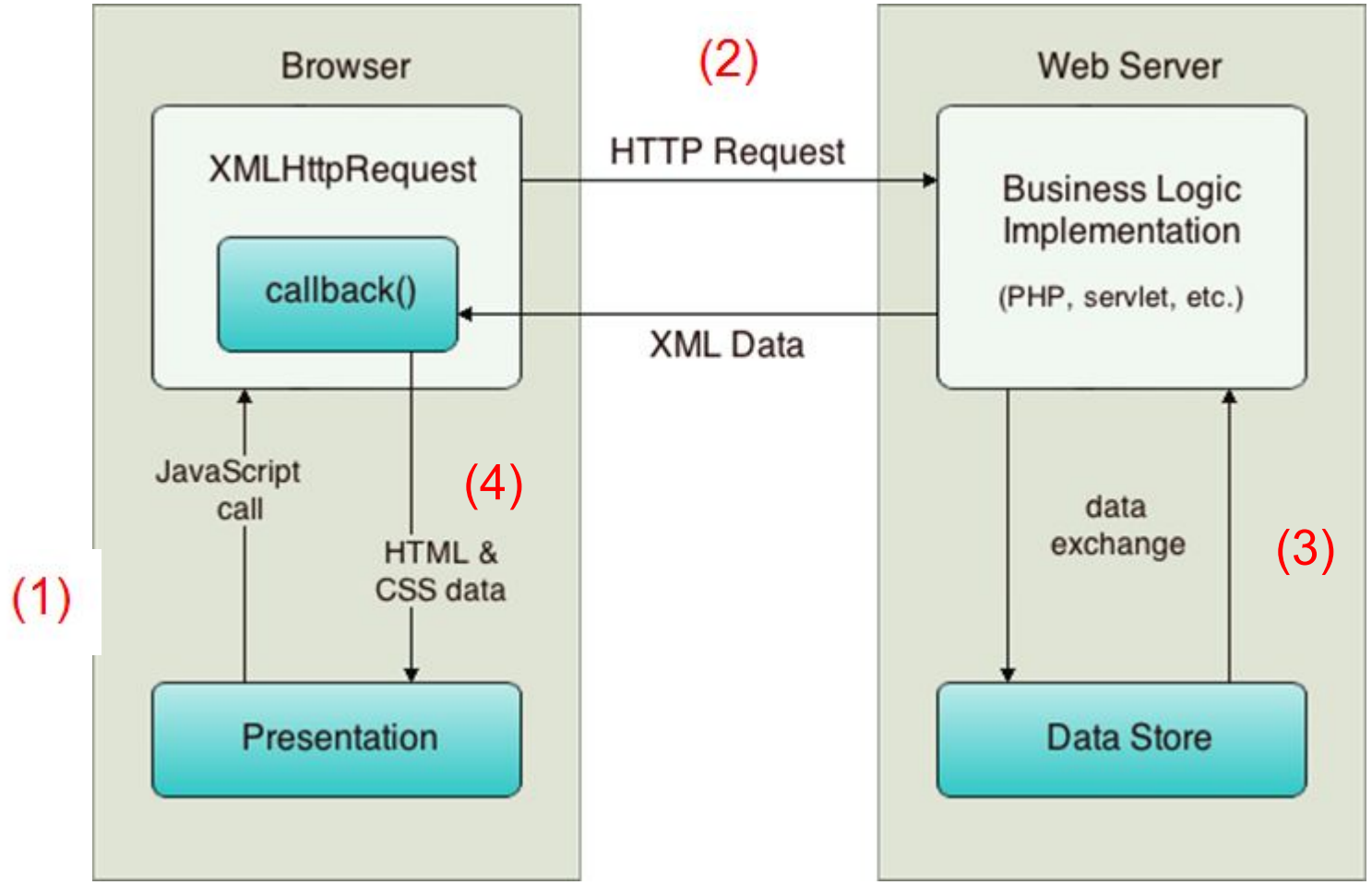
Finally, the XMLHttpRequest object

1. **receives the XML data** using a callback function
2. **processes** the data, and
3. **updates** the HTML DOM (Document Object Model) to display the page containing the new data

```
$(#message).html(xhr.responseText)
```

When should this execute?

Step 4: client updates



Onreadystatechange event

- When a request to a server is sent, we want to perform some actions based on the response
- The **onreadystatechange** event is triggered every time the **readyState** changes
- The **readyState property** holds the status of the XMLHttpRequest

Onreadystatechange event

Three important properties of the **XMLHttpRequest** object:

Property	**Description**
onreadystatechange	Stores a function (or the name of a function) to be called automatically each time the readyState property changes
readyState	Holds the status of the XMLHttpRequest. Changes from 0 to 4: 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
status	200: "OK" 404: Page not found

Onreadystatechange event

In the “onreadystatechange” event, specify what will happen when the server response is ready to be processed. When readyState = 4 (DONE) and status = 200, the response is ready

```
var xhr = new XMLHttpRequest(),
    method = "GET",
    url = "https://developer.mozilla.org/";

xhr.open(method, url, true);
xhr.onreadystatechange = function () {
  if(xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {
    console.log(xhr.responseText);
    $(#message).html(xhr.responseText)
  }
};
xhr.send();
```

All Together Now: Client, Server!

```
function tellMyFortune(){
    let xhr = new XMLHttpRequest()
    xhr.open("POST","fortune",true)
    xhr.send()
    xhr.onreadystatechange = function(){
        if(xhr.readyState === 4 && xhr.status === 200){
            document.getElementById("demo").innerHTML = xhr.responseText
        }
    }
}
```

HttpRequest

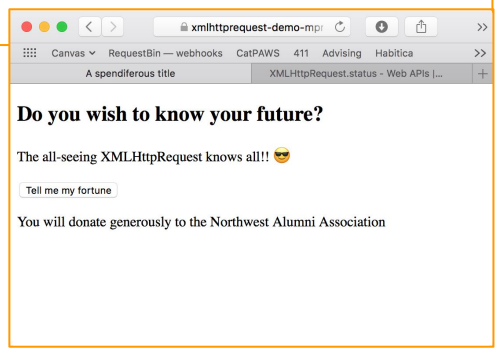
XML

JavaScript Call

```
<!DOCTYPE html>
<html>
    <head>
        <title>A splendiferous title</title>
    </head>
    <body>
        <h2>Do you wish to know your future?</h2>
        <p>The all-seeing XMLHttpRequest knows all!! □</p>
        <button onclick="tellMyFortune()">Tell me my fortune</button>
        <p id="demo"></p>
        <script src="test.js"></script>
    </body>
</html>
```

HTML & CSS

```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
  res.send('Hello,World!')
})

let fortunes = ["win the lotto", "fall in love", "fall out of love", "fall on the ground", "donate generously to the Northwest Alumni Association"]


app.post('/fortune', function(req,res){
  let choice = Math.floor(Math.random()*fortunes.length)
  res.send("You will " + fortunes[choice])
})

app.use(express.static('public'))

app.listen(process.env.PORT, function () {
  console.log('app listening on port ' + process.env.PORT)
})
```



https://ide.c9.io/mprogers/xmlhttprequest_demo

Or

https://bitbucket.org/professorcase/local_ajax_example

Let's Try It (W08)

Create C:\44563\w08 folder - create three client-side files:

- An empty **index.html**
- An empty **ajax.js**
- A file **show.txt** with content to display.
- In index.html,
  - Add boilerplate HTML
  - Include jQuery and our ajax.js script source files.
  - Add two buttons.
  - Add an element to display results.

Right-click your folder & *Open with Code.*

index.html

```
<!DOCTYPE html>
<html>
<head>
	<title>W08 Ajax</title>
</head>
<body>
	<h2>jQuery AJAX (open in Firefox)</h2>
	<h3> Get partial page content using:</h3>
	<button id="btnAjax" > .ajax() REST</button>
	<button id="btnLoadText">.load() Text File</button>
	<h2> Result</h2>
	<div id="showResult"></div>
	<hr>
	<a href="https://oscarotero.com/jquery/">jQuery Quick API Reference at
	https://oscarotero.com/jquery/</a>
	<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
	<script src="ajax.js"></script>
</body>
</html>
```

Where will we have our results display?

ajax.js

```
(function ($) {

  $('#btnLoadText').click(function () { $("#showResult").load("show.txt"); });
  $('#btnAjax').click(function () { callRestAPI() });

  // Perform an asynchronous HTTP (Ajax) API request.
  function callRestAPI() {
    var root = 'https://jsonplaceholder.typicode.com';
    $.ajax({
      url: root + '/posts/1',
      method: 'GET'
    }).then(function (response) {
      console.log(response);
      $('#showResult').html(response.body);
    });
  }
})($);
```

JavaScript self-executing anonymous Function

Just call (function xyx(arg){ }(arg);

Here we have switched to the handy jQuery ajax call. Earlier examples were in JavaScript to better illustrate the XMLHttpRequest object.

Where must show.txt be located for this to work?

show.txt

I can't believe I got fired from the calendar factory. All I did was take a day off.

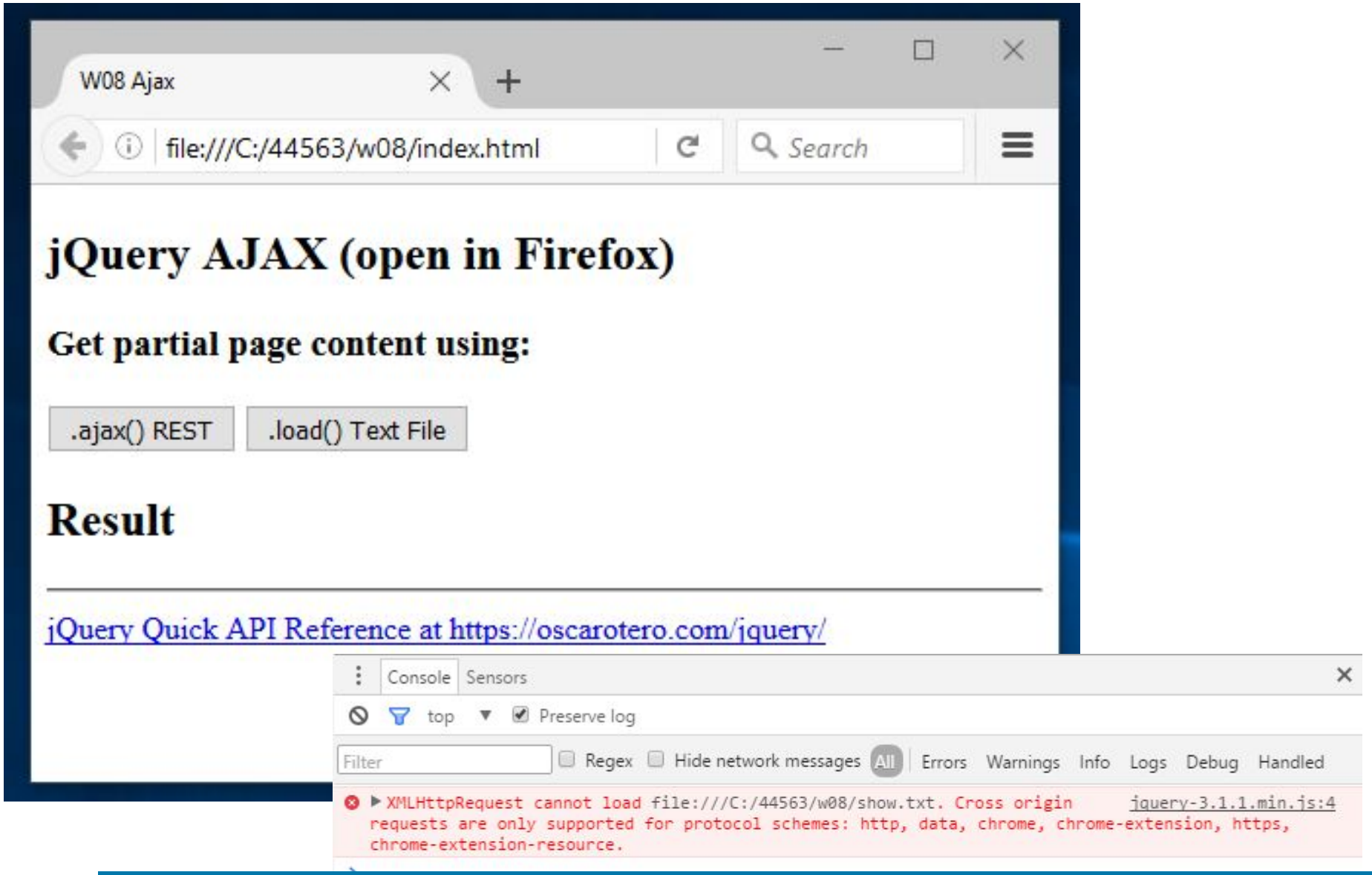
I'd tell you a chemistry joke but I know I wouldn't get a reaction.

I wasn't originally going to get a brain transplant, but then I changed my mind.

My math teacher called me average. How mean!

I wanted to buy a neutrino, but had to settle for a used one.

Customize your text content



Open in **Firefox**; Chrome blocks cross origin requests

Required Installations - please verify with course assistant

1. Visual Studio **Code** (be able to right-click a folder and "Open with Code"). If you don't have this, just reinstall VS Code.
2. **Git for Windows** (be able to right-click a folder and open a Git Bash window)
3. **TortoiseGit** (be able to right-click a folder and "Git create repository here")
4. Be able to **Open Command Window Here as Administrator** (See http://www.sevenforums.com/tutorials/47415-open-command-window-here-administrator.html (Links to an external site.))
5. **nodemon** (get live updates - install globally by opening a command prompt in any repo and running: npm install -g nodemon)

Next Wednesday

Next Wednesday October 25, we will have Nick Larson, the Chief Technologist from Object Partners, Inc come talk with us about a super-hot web technology, **React.js**.

All sections will meet in CH 3500 (the auditorium room on the third floor) at 4 PM that day. (Don't come at 11, don't come at noon, and don't come at 1 PM - and don't come to CH 1200 at 4 - come with all of us to CH 3500 at 4 PM).

Attendance is mandatory and assistants will be taking attendance. If you have a course conflict, please let us know immediately and we will figure out alternate plans.