# AJAX
# and
# WebAPI

# Agenda

- JSON
- Web storage
- AJAX
- Clint side of AJAX
- Server side of AJAX
- Web.API
- REST

# JSON
# JavaScript Object Notation

Is described in [RFC 4627]

# JSON

- Is a lightweight data-interchange format
- Is easy for humans to read and write
- Is easy for machines to parse and generate
- **Is based on a subset of JavaScript**
- Is a text format that is completely language independent
  - but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others
- These properties make JSON an ideal data-interchange language
  - And JSON is widely used in data communication
  - And several NoSQL databases uses JSON (or BSON) as data storage format.
    - BSON is a binary representation of JSON

# JSON Structure

- JSON is built on two structures:
  - A collection of name/value pairs
    - In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array
  - An ordered list of values.
    - In most languages, this is realized as an array, vector, list, or sequence

```
{
    name : "10gen HQ",
    address : "578 Broadway 7th Floor",
    city : "New York",
    zip : "10011",
    tags : [ "business", "tech" ]
}
```

# Native JSON

- **All modern browsers have native JSON support, via:**
  - **JSON.parse() and**
  - **JSON.stringify()**

- They were added to the Fifth Edition of the ECMAScript standard

- For older browsers, a compatible JavaScript library is available at JSON.org

# Json.NET

- Json.NET is a popular high-performance JSON framework for .NET
- Features:
  - Flexible JSON serializer for converting between .NET objects and JSON
  - LINQ to JSON for manually reading and writing JSON
  - High performance, faster than .NET's built-in JSON serializers
  - Write indented, easy to read JSON
  - Convert JSON to and from XML
- The serializer is a good choice when the JSON you are reading or writing maps closely to a .NET class
- LINQ to JSON is good for situations where you are only interested in getting values from JSON, you don't have a class to serialize or deserialize to, or the JSON is radically different from your class and you need to manually read and write from your objects

# Serialization Example

```
Product product = new Product();
product.Name = "Apple";
product.Expiry = new DateTime(2008, 12, 28);
product.Price = 3.99M;
product.Sizes = new string[] { "Small", "Medium", "Large" };
string json = JsonConvert.SerializeObject(product);
// . . .

Product deserializedProduct =
             JsonConvert.DeserializeObject<Product>(json);
```

- The serializer is a good choice when the JSON you are reading or writing maps closely to a .NET class

```
{
  "Name": "Apple",
  "Expiry": "2008-12-28T00:00:00",
  "Price": 3.99,
  "Sizes": [
    "Small",
    "Medium",
    "Large"
  ]
}
```

# Getting JSON Values

```
string json = @"{
  ""Name"": ""Apple"",
  ""Expiry"": "2008-12-28T00:00:00",
  ""Price"": 3.99,
  ""Sizes"": [
    ""Small"",
    ""Medium"",
    ""Large""
  ]}";

JObject o = JObject.Parse(json);

string name = (string)o["Name"];
// Apple

sizes = (JArray)o["Sizes"];
string smallest = (string)sizes[0];
// Small
```

- `Jobject` is good for situations where:
  - you are only interested in getting values from JSON
  - you don't have a class to serialize or deserialize to
  - or the JSON is radically different from your class and you need to manually read and write from your objects

# LINQ to JSON

- JObject/JArray can also be queried using LINQ
  - Children() returns the children values of a JObject/JArray as an IEnumerable<JToken> that can then be queried with the standard Where/OrderBy/Select LINQ operators

```
string json = @"{. . .}";
JObject rss = JObject.Parse(json);

var postTitles =
        from p in rss["channel"]["item"]
        select (string)p["title"];

foreach (var item in postTitles)
{
  Console.WriteLine(item);
}
```

# WEB STORAGE

## ANOTHER USE OF JSON

You can save data on the client
– even between sessions!

# HTML5 Client-Side Storage

Browser Support

| |  |  |  |  |  |  |  | Mini | Mobile |
|---|---|---|---|---|---|---|---|---|---|
| *Web Storage - name/value pairs* | 4+ | 3.5+ | 4+ | 10.5+ | 8+ | 3.2+ | 2.1+ | — | 11.5+ |
| *IndexedDB* | 23+ | 10+ | 9+ | 15+ | 10+ | 9+ | 4.4 | — | 0 |
| *Web SQL Database* | 4+ | — | 3.1+ | 10.5+ | — | 3.2+ | 2.1+ | — | 11.5+ |

- Web Storage simply provides a key-value mapping,
  - e.g. localStorage["name"] = username;
- Web SQL Database gives you all the power of a structured SQL relational database
- Indexed Database is somewhere in between Web Storage and Web SQL Database
  - Like Web Storage, it's a straightforward key-value mapping, but it supports indexes, so searching objects matching a particular field is fast



Table from http://www.html5rocks.com/it/features/storage

# Web Storage

- Supports persistent data storage
  - similar to cookies but with a greatly enhanced capacity (5-10MB per origin) and no information stored in the HTTP request header
- There are two main web storage types:
  - `localStorage`
    - Permanent storage (may be deleted by user)
    - Data placed in local storage is per origin (protocol+hostname+port number)
  - `sessionStorage`
    - Limited to the lifetime of the window
- Browsers that support web storage have the global variables 'sessionStorage' and 'localStorage' declared at the window level
- Web storage is being standardized by the World Wide Web Consortium

# How to Use?

- Web Storage only supports string-to-string mappings
  - so you need to serialise and de-serialise other data structures

- You can use JSON.stringify() and JSON.parse() to store other data types than string

```
// Store an object/array using JSON
localStorage.numbers = JSON.stringify(numbers);
```

```
if (localStorage.hasOwnProperty("numbers")) {
   // Read an object/array using JSON
   numbers = JSON.parse(localStorage.numbers);
}
```

# AJAX
# ASYNCHRONOUS JAVASCRIPT AND XML

Or Json

# Background

- In the 1990s, most web sites were based on complete HTML pages:
  - each user action required that the page be re-loaded from the server (or a new page loaded)
  - This process is inefficient: all page content disappears then reappears
  - Each time a page is reloaded due to a partial change, all of the content must be re-sent instead of only the changed information
  - This can place additional load on the server and use excessive bandwidth

# Ajax

- Is a group of interrelated web development techniques used on the client-side to create asynchronous web applications
- With Ajax:
  - **web applications can send data to, and retrieve data from, a server asynchronously** (in the background)
- Data can be retrieved using the **XMLHttpRequest** object
  - Despite the name, the use of XML is not required
    - **JSON is often used instead**
- ***Ajax is not a single technology, but a group of technologies used in combination***

# AJAX Communication

Web Client

HTML dokument ⟷ JavaScript

Get or Post

XMLHttpRequest (**Ajax calls**)

Web Server

# CLINT SIDE OF AJAX

# HTTP request

- To make an HTTP request you create a new XMLHttpRequest object:

```
var request = new XMLHttpRequest();
request.open("GET", "files/data.txt", false);
request.send(null);
alert(request.responseText);
```


- The **open** method is used to configure the request
- The **send** method perform the actual request to the server
  - When the request is a POST request, the data to be sent to the server can be passed to this method as a string
  - For GET requests just pass null
- The **responseText** property contains the content of the retrieved document (after the request has been made)
- The headers that the server sent back can be inspected with the getResponseHeader and getAllResponseHeaders functions

# Asynchronous HTTP Request

- When the third argument to open is true, the request is set to be **'asynchronous'**
  - This means that send will return right away, while the request happens in the background

```
var request = new XMLHttpRequest();}
request.open("GET", "files/data.txt", true);
request.send(null);
request.onreadystatechange = function() {
  if (request.readyState == 4)
    alert(request.responseText.length);
};
```

# XMLHttpRequest - load Event

- Use of the load event is often a better solution

```
var req = new XMLHttpRequest();
req.addEventListener("load", transferComplete);
req.addEventListener("error", transferFailed);
req.open("GET", "files/data.txt", true);
req.send(null);

function transferFailed(evt) {
 alert("An error occurred while transferring the file.");
 }

function transferComplete(evt) {
 console.log("The transfer is complete.", req.status);
}
```

# Working With XML

- When the file retrieved by the request object is an XML document, the request's **`responseXML`** property will hold a representation of this document
- Such XML documents can be used to exchange structured information with the server
- Their form — tags contained inside other tags — is often very suitable to store things that would be tricky to represent as simple flat text
- The DOM interface is rather clumsy for extracting information though, and XML documents are notoriously wordy

```
var catalog = request.responseXML.documentElement;
alert(catalog.childNodes.length);
```

# Working With JSON

- A JSON document is a file containing a single JavaScript object or array
  - which in turn contains any number of other objects, arrays, strings, numbers, booleans, or null values

```
request.open("GET", "files/fruit.json", true);
request.addEventListener("load", function(){
  console.log(request.responseText);
});
request.send(null);
```

```
{banana: "yellow", lemon: "yellow",
cherry: "red"}
```

# Working With JSON

- A string that contains JSON data can be converted to a normal JavaScript value by using the JSON.parse function

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.json", false);
req.send(null);
console.log(JSON.parse(req.responseText));
```

```
{banana: "yellow", lemon: "yellow",
cherry: "red"}
```

# jQuery AJAX

- jQuery's low level interface:
  - jQuery.ajax()
    - Perform an asynchronous HTTP (Ajax) request
    - Note: has lots of possible settings
  - Examples:
    - Save some data to the server and notify the user once it's complete

```
$.ajax({
  type: "POST",
  url: "some.php",
  data: { name: "John", location: "Boston" }
}).done(function( msg ) {
  alert( "Data Saved: " + msg );
});
```

    - Retrieve the latest version of an HTML page.

```
$.ajax({
  url: "test.html",
  cache: false
}).done(function( html ) {
  $("#results").append(html);
});
```

# jQuery AJAX Shortcuts

- jQuery has several functions to ease with AJAX calls:
  - jQuery.get()
    - Load data from the server using a HTTP GET request.
  - jQuery.getJSON()
    - Load JSON-encoded data from the server using a GET HTTP request.
  - jQuery.getScript()
    - Load a JavaScript file from the server using a GET HTTP request, then execute it.
  - jQuery.post()
    - Load data from the server using a HTTP POST request.
  - .load()
    - Load data from the server and place the returned HTML into the matched element.

```
$.get('ajax/test.html', function(data) {
  $('.result').html(data);
  alert('Load was performed.');
});
```

# SERVER SIDE OF AJAX

# Microsoft Frameworks

- ASP.NET MVC
  - A variation of the Controllers in the MVC framework
- WCF REST (Windows Communication Foundation)
  - Lets developers build contract-first services that leverage transport protocols such as TCP, HTTP and MSMQ. Originally built for SOAP-based services that want WS-* capabilities, WCF eventually added a handful of REST-friendly capabilities
- **ASP.NET Web API**
  - Differentiates itself from the previous Microsoft in-box HTTP service solutions in that it was built from the ground up around the HTTP protocol and its messaging semantics

# ASP.NET AJAX

- If you use ASP.NET MVC then the server side of AJAX is very simple
- The AJAX request will hit a controller just like an ordinary web-request
  - **The difference is in what the controller returns!**

# A Controller Returning Html

- A normal view will include all the shared “chrome” from _Layout.cshtml. We don’t want that so we **return a PartialView**
  - This ensures that the surrounding chrome that’s inside the layout page is not included in the markup returned from our controller

Index.cshtml

```
<a href="/Home/PrivacyPolicy“
   id="privacyLink">
   Show the privacy policy</a>

<div id="privacy"></div>
```

AjaxDemo.js

```
$('#privacyLink').click(function (event) {
    event.preventDefault();
    var url = $(this).attr('href');
    $('#privacy').load(url);
```

```
// In the controller class
public ActionResult PrivacyPolicy()
{
    return PartialView();
}
```

PrivacyPolicy.cshtml

```
<h2>Our Commitment to Privacy</h2>
<p>
    Your privacy is important to us.
    Bla Bla Bla
</p>
```

# Progressive Enhancement

- Progressive enhancement means that we begin with basic functionality (in this case, a simple hyperlink) and then layer additional behavior on top (our Ajax functionality)
- This way, if the user doesn't have JavaScript enabled in their browser, the link will gracefully degrade to its original behavior and instead send the user to the privacy policy page without using Ajax
- We can check to see whether the action has been requested via Ajax or not:

```
// In the controller class
public ActionResult PrivacyPolicy()
{
    if (Request.IsAjaxRequest())
    {
        return PartialView();
    }
    return View();
}
```

# A Controller Returning Plain Text

- You can also just return string if you know that's the only thing the method will ever return

```
public string AsText()
{
    return "apples, oranges, bananas";
}
```

- You can't return a string from a method which returns an ActionResult, so in this case you return Content("")

```
public ActionResult AsText2()
{
    return Content("apples, bananas");
}
```

# A Controller Returning XML

- You can also use the Content class to return XML

```
public ActionResult AsXML()
{
    var xmlString = "<catalog> <fruit><name>apple</name>
                        <color>red</color> </fruit>
                    </catalog>
    return Content(xmlString, "text/xml");
}
```

- But there are other smarter solutions
  - For example use of the WebAPI controller

# A Controller Returning JSON

- You can use the Json class to return data (an object) as JSON

```
public ActionResult AsJSON()
{
    var obj = new Myclass(…);
    return Json(obj, JsonRequestBehavior.AllowGet);
}
```

- Or you can use a WebAPI controller

# WEB.API

# Why WebAPI?

- HTTP is not just for serving up web pages
- It is also a powerful platform for building APIs that expose services and data
- **HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications**

# Web.API in ASP

- You can create a Web.API project in VS or you can just add a Web.API controller in one of the other web-projects

- In Web API, a *controller* is an object that handles HTTP requests
  - Web API controllers are similar to MVC controllers, but inherit the **ApiController** class instead of the **Controller** class

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> GetAllProducts() {
        return products;
    }
public IHttpActionResult GetProduct(int id) {
```

# Action Results

A Web API controller action can return any of the following:

| Return type | How Web API creates the response |
|---|---|
| void | Return empty 204 (No Content) |
| HttpResponseMessage | Convert directly to an HTTP response message |
| IHttpActionResult | Call ExecuteAsync to create an HttpResponseMessage, then convert to an HTTP response message |
| Other type | Write the serialized return value into the response body; return 200 (OK) |

Web API uses the Accept header in the request to choose the formatter

# IHttpActionResult Example

- The **ApiContoller** class defines helper methods that return these built-in action results:

```
public IHttpActionResult Get(int id)
{
  Product product = _repository.Get(id);
  if (product == null)
  {
     return NotFound(); // Returns a NotFoundResult
  }
  return Ok(product); // Returns an OkNegotiatedContentResult
}
```

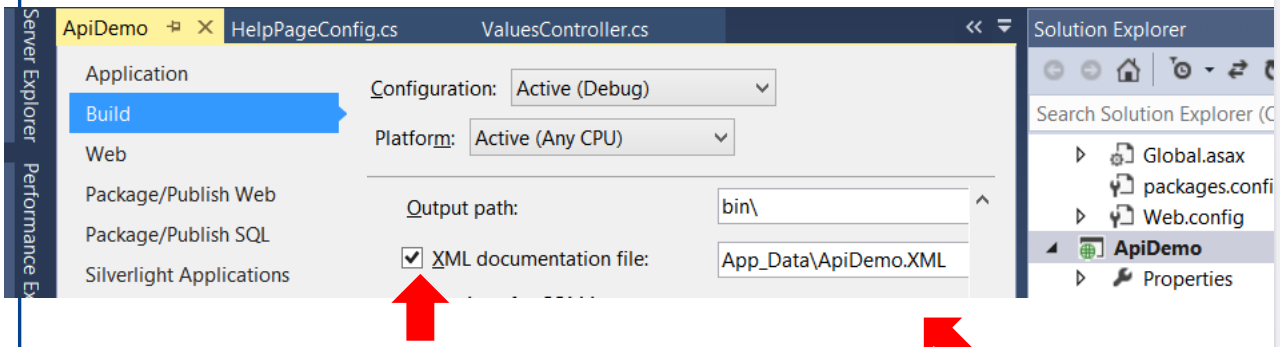
# Web API: methods corresponds to URIs

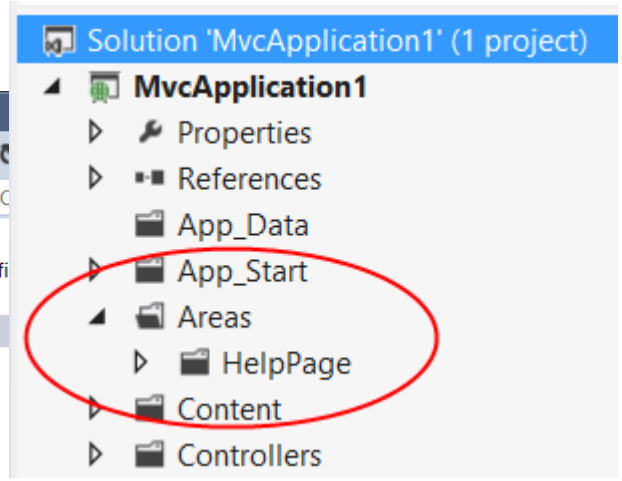
| Controller Method | URI |
| --- | --- |
| GetAllProducts | /api/products |
| GetProduct | /api/products/id |

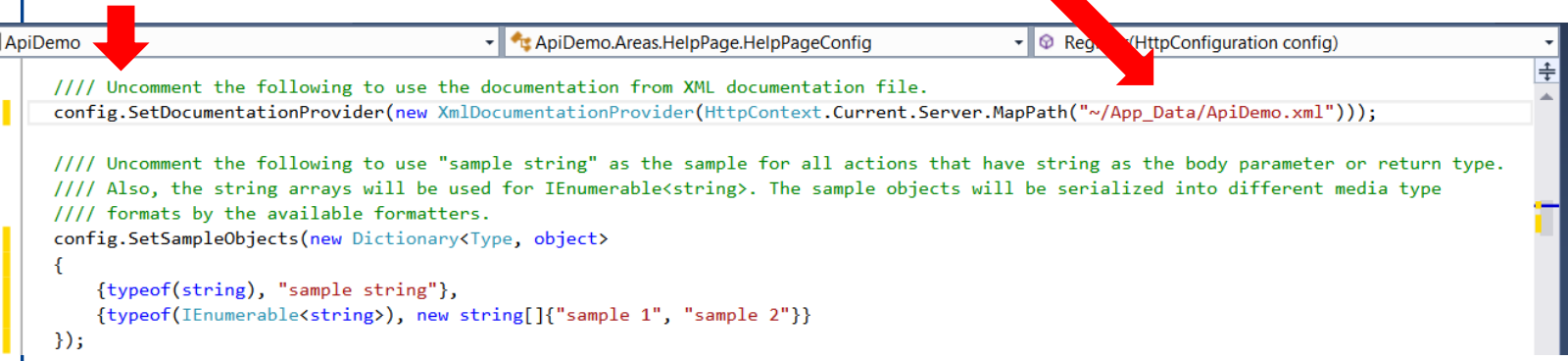
Each method on the controller corresponds to one or more URIs

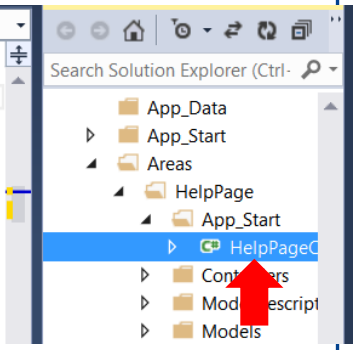
# Help Page

- ASP.NET Web API provides a library for auto-generating help pages at run time




**Add documentation with /// on each controller action**

# CORS

- Cross-origin resource sharing
- XMLHttpRequest (and Embedded web fonts) requests have traditionally been limited to accessing the same domain as the parent web page
- CORS defines a way in which a browser and server can interact to safely determine whether or not to allow the cross-origin request
- Client – prerequest:

```
Origin: http://www.foo.com
```

- Server – response:

```
Access-Control-Allow-Origin: http://www.foo.com
```

  - or

```
Access-Control-Allow-Origin: *
```

# Auth

- ASP Web.API has the same possibilities for autorization as ASP.MVC
  - But no authorization is default
- But no register or login page
  - This must be done by use of XMLHttpRequest  calls
- When you login you receive an access token that must be sent along with all the following requests

# REST

## REPRESENTATIONAL STATE TRANSFER

REST didn't attract much attention when it was first introduced in 2000 by Roy Fielding at the University of California, Irvine, in his academic dissertation, "Architectural Styles and the Design of Network-based Software Architectures," which analyzes a set of software architecture principles that use the Web as a platform for distributed computing.

Representational State Transfer (REST) has gained widespread acceptance across the Web as a simpler alternative to SOAP- and Web Services Description Language (WSDL)-based Web services.

# RESTful Web Services: the Basics

- REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages

- A concrete implementation of a REST Web service follows four basic design principles:
  - **Use HTTP methods explicitly**
  - **Be stateless**
  - **Expose directory structure-like URIs**
  - **Transfer XML, JSON,** or any other valid Internet media type content (such as an image or plain text)

- REST is not a standard but an architectural style

# Use HTTP Methods Explicitly

- One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616
- This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods
- According to this mapping:
  - **To create a resource on the server, use POST**
  - **To retrieve a resource, use GET**
  - **To change the state of a resource or to update it, use PUT**
  - **To remove or delete a resource, use DELETE**

# Use HTTP Methods Explicitly

**API Design Principles:**

- Use nouns in URIs instead of verbs.
  - In a RESTful Web service, the verbs—POST, GET, PUT, and DELETE—are already defined by the protocol
- And the Web service should not define more verbs or remote procedures, such as /adduser or /updateuser
- The body of an HTTP request should be used to transfer resource state
  - not to carry the name of a remote method to be invoked

# Use HTTP Methods Explicitly

- A non-RESTful API:

```
GET /adduser?name=Robert HTTP/1.1
```

- A RESTful API:

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/json
{name : Robert}
```

# Example - Collection URI

| | |
|---|---|
| **Resource** | Collection URI, such as **http://example.com/resources** |
| **GET** | **List** the URIs and perhaps other details of the collection's members. |
| **PUT** | **Replace** the entire collection with another collection. |
| **POST** | **Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation. |
| **DELETE** | **Delete** the entire collection. |

# Example - Element URI

| | |
|---|---|
| **Resource** | Element URI, such as **http://example.com/resources/item27** |
| **GET** | **Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type (typical JSON). |
| **PUT** | **Replace** the addressed member of the collection, or if it doesn't exist, **create** it. |
| **POST** | Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it. |
| **DELETE** | **Delete** the addressed member of the collection. |

# OData

- OData is a standardized protocol for creating and consuming data APIs
- OData builds on core protocols like HTTP and commonly accepted methodologies like REST
- The result is a uniform way to expose full-featured data APIs

- http://msopentech.com/odataorg/introduction/

# References & Links

- http://json.org/
- Json.net
  http://james.newtonking.com/projects/json-net.aspx
- An Introduction to JavaScript Object Notation (JSON) in JavaScript and .NET
  http://msdn.microsoft.com/en-us/library/bb299886.aspx
- **ASP.NET Web API**
  **http://www.asp.net/web-api**
- Gratis bog: **ASP.NET Web API Succinctly**
- ASP.NET Ajax
  http://www.asp.net/ajax
- **Postman** (a Chrome extension very useful for testing Web.API)
  https://www.getpostman.com/
- REST
  http://www.ibm.com/developerworks/webservices/library/ws-restful/
  http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm