Building RESTful Services with Synergy .NET

Presented by Steve Ives

Building RESTful Services with Synergy .NET

- Introduction to web services
- Technology primers
  - HTTP, JSON, Postman
- RESTful web services
- Building RESTful services with .NET
  - ASP.NET Web API 2
- Calling RESTful services

Introduction to Web Services

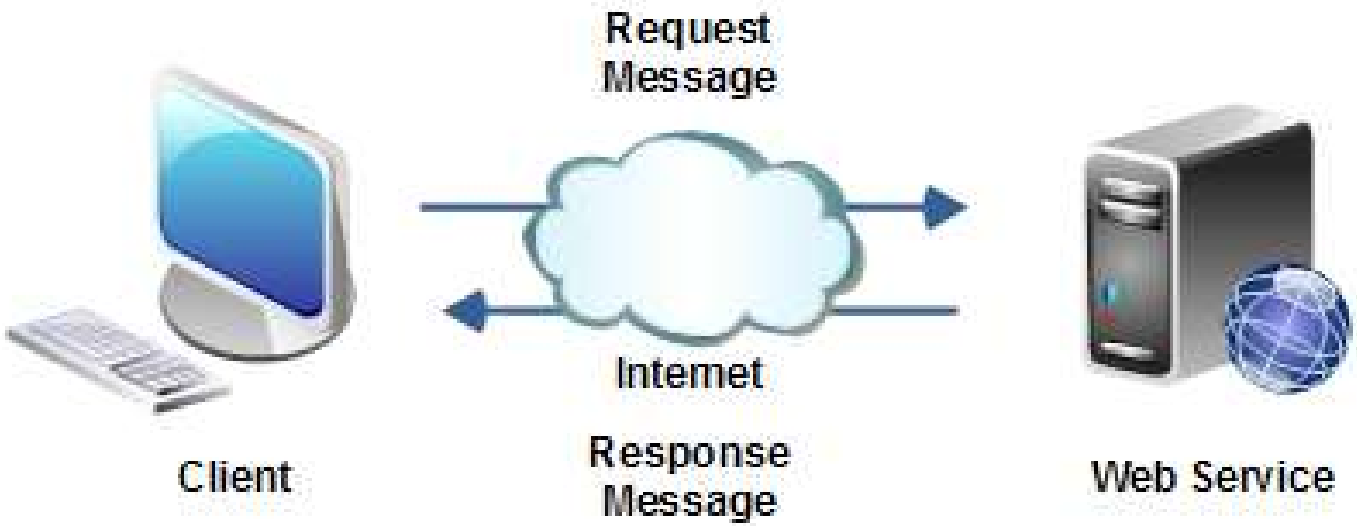
Building RESTful Services with Synergy .NET

What Is a Web Service?

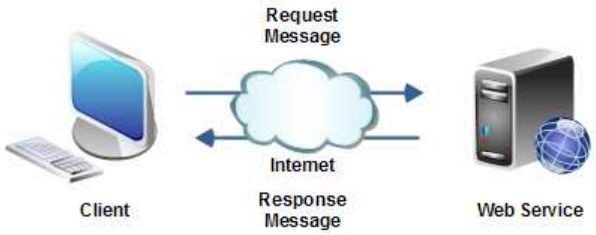
"A service offered by an electronic device to another electronic device, communicating with each other via the World Wide Web." (Wikipedia)

"A software system designed to support interoperable machine-to-machine interaction over a network." (W3C)

Visualizing a Web Service



Why Use Web Services?

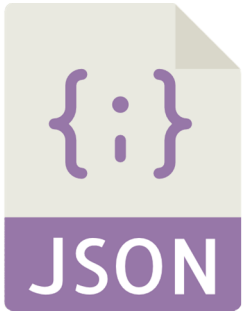


- Combine best aspects of component-based development and web
- Interoperability
- Platform and language agnostic
- Decoupled
  - Client doesn't care how service is implemented
  - Service doesn't care how client is implemented
- Scale well (if designed well)
- Multiple deployment options
- Great ROI
  - One server, multiple clients

What Accesses a Web Service?

- Software apps access web services
  - Desktop or server apps
  - Websites
  - Other web services
  - Mobile apps
- Devices also access web services
  - Internet of Things (IoT)
- Interaction with published APIs
  - Great alternative to custom components
  - Zero client footprint

How Are Web Services Called?



- Via ubiquitous web protocols
  - HTTP or HTTPS
  - Occasionally other protocols within a private LAN
- Using ubiquitous data formats
  - XML (SOAP) or JSON

Primary Use Cases

- Private APIs
  - Part of a larger overall solution
  - Highly distributed scenarios
    - Enterprise service bus
    - Service oriented architecture
  - One server, multiple clients
- Public APIs
  - Access parts of a larger application
  - The entire solution
  - No control over the client
  - No restrictions on potential clients

Developing Web Services in .NET

- Generation 1 : **ASP.NET Web Services**
  - .NET Framework 1.0 (2002)

- Generation 2 : **Windows Communication Foundation** (WCF)
  - First was WCF 3.0 (Visual Studio 2005)
  - Latest is WCF 4.5 (Visual Studio 2012)

- Generation 3 : **ASP.NET Web API**
  - First was Web API 1.0 (Visual Studio 2010)
  - Latest is Web API 2.2 (now NuGet, December 2014)

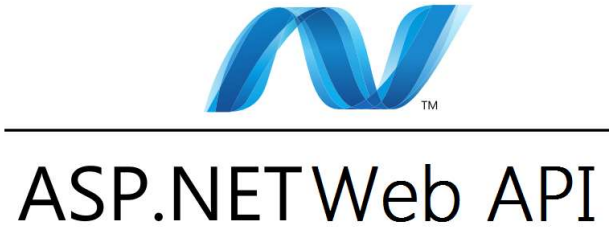
ASP.NET Web Services

- Always uses HTTP[S] as a transport
- SOAP-based
  - Verbose XML dialect
- WSDL
  - XML file describing the service, to custom tooling
- Easy to use from environments with WSDL tooling
  - .NET “add service reference” generates client-side code
- Difficult to use from other environments
  - Producing and parsing complex SOAP messages
- Performance and scalability could be challenging

Windows Communication Foundation

- More flexible and capable than ASP.NET web serv
- Support for multiple network transports
  - HTTP, TCP, named pipes, etc.
- Still SOAP-based, but with improved performance
- Easy to migrate—ASP.NET compatibility mode
- Heavily configuration based
  - Good and bad: simple to code, nightmare to configure!
- Can still be a great solution for some use cases

ASP.NET Web API



- Framework for building RESTful Web APIs
  - No SOAP, no WSDL
  - HTTP transport
  - Preference for JSON, XML supported
- Closely related to ASP.NET MVC
  - Models and controllers
- Convention over configuration
  - Naming conventions, verbs map to HTTP methods, etc.
- Easy to learn, use, and deploy… and FAST!
- HUGE improvement over earlier technologies
- Open source

Technology Primer - HTTP

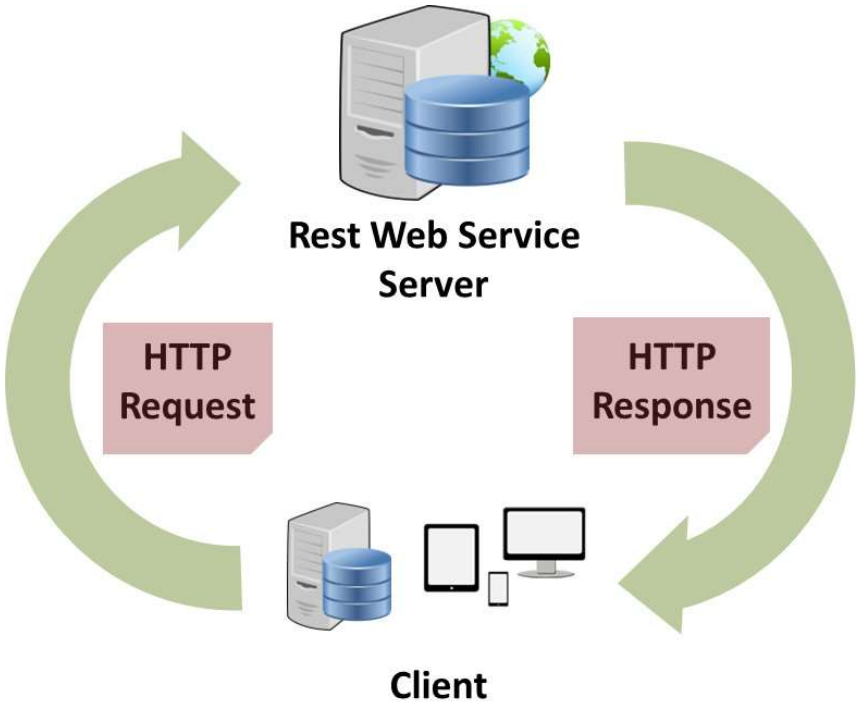
Building RESTful Services with Synergy .NET

Hypertext Transfer Protocol (HTTP)

- Data transfer protocol designed for distributed, collaborative, and hypermedia information systems
- Foundation of web data communication
- Most web services use HTTP[S]
- Specification defined in various RFCs
  - RFC 7230 - 7237

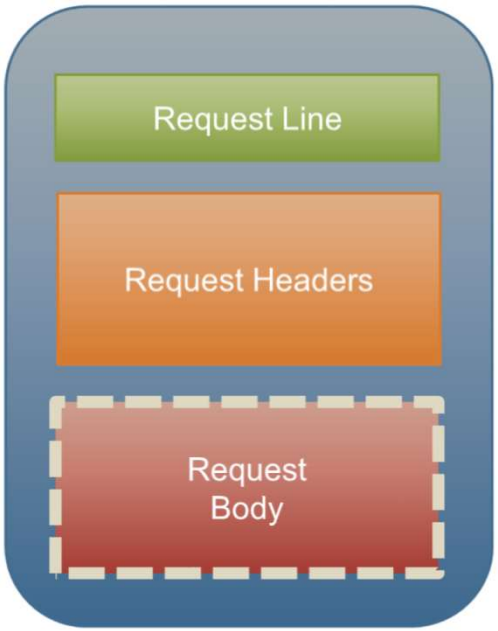
Request, Response, Disconnect

- A one shot deal—always
  - Client connects and sends request
  - Server does something and sends response
  - Client disconnects
- Anything beyond that is application-specific



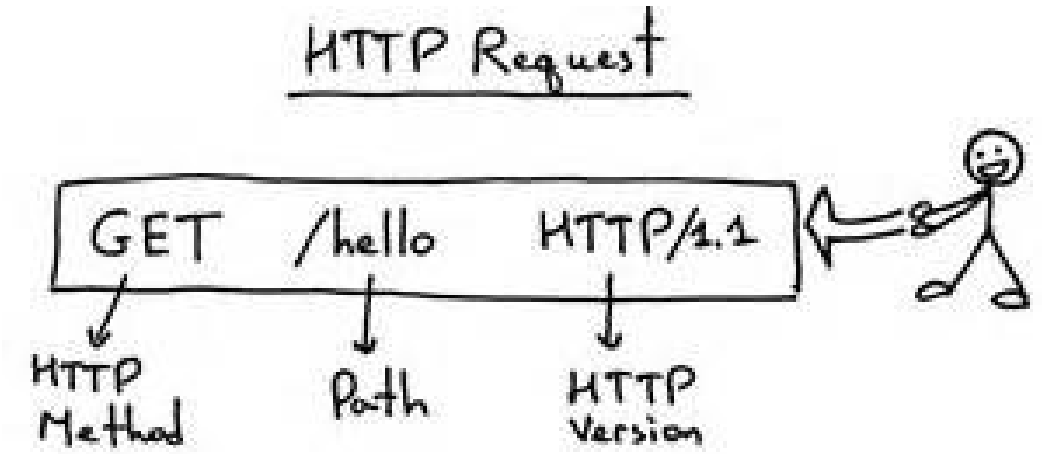
Anatomy of an HTTP Request

- Request line
- Request headers
- Request body (optional)



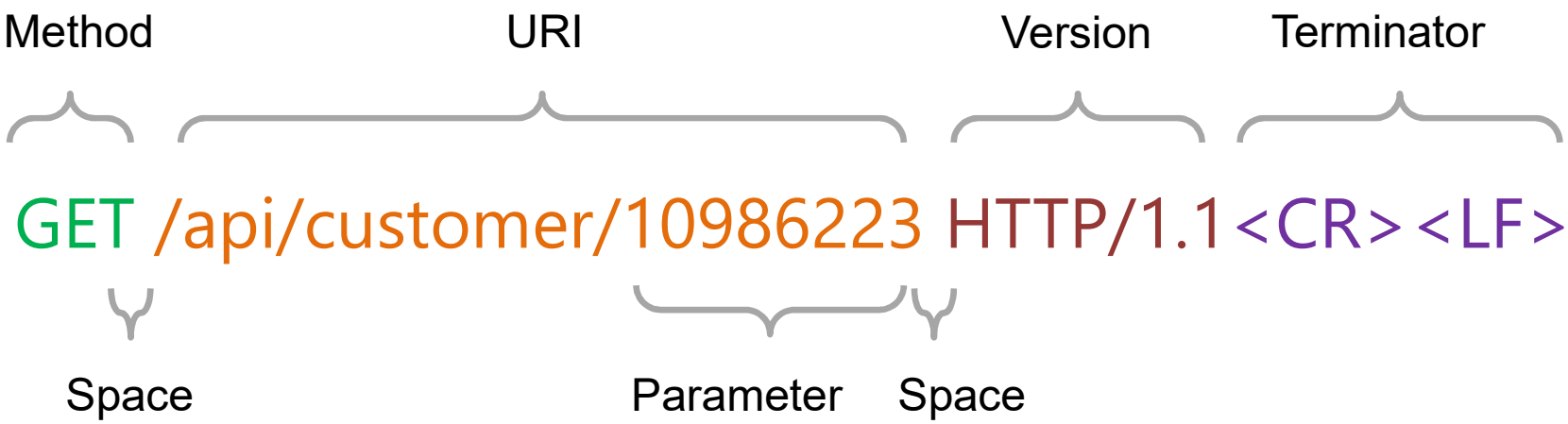

```
POST /api/orders HTTP/1.1
Host: www.acme.com:80
Content-Type: application/json
Accept: application/json
Content-Length: 108

{“account”:10986223,“ponumber”:19734,“items”:[{“sku”:“ABB701”,
“quantity”:1},{“sku”:“CRD100”,“quantity”:10}]}
```

HTTP Request Line



- Common methods (web)
  - GET - Download
  - POST - Upload
- Path
  - Identifies a resource
  - May include parameter data
- HTTP version
  - HTTP/1.0 or HTTP/1.1
  - Pretty much always HTTP/1.1 these days
- Terminated by a CR-LF pair

Example HTTP Request Line



HTTP URI Parameters

- Data can be included in the URI in several ways
  - /customers?id=12345
  - /customers/12345
  - /orders.aspx?customer=12345&order=1127
  - /customers/12345/orders/1127

HTTP Methods

- Common for the web
  - **GET**
    - Retrieve state
  - **POST**
    - Create state
- Additional with REST
  - **PUT**
    - Alter state
  - **DELETE**
    - Remove state
- Others (uncommon)
  - HEAD
  - CONNECT
  - OPTIONS
  - TRACE
  - PATCH

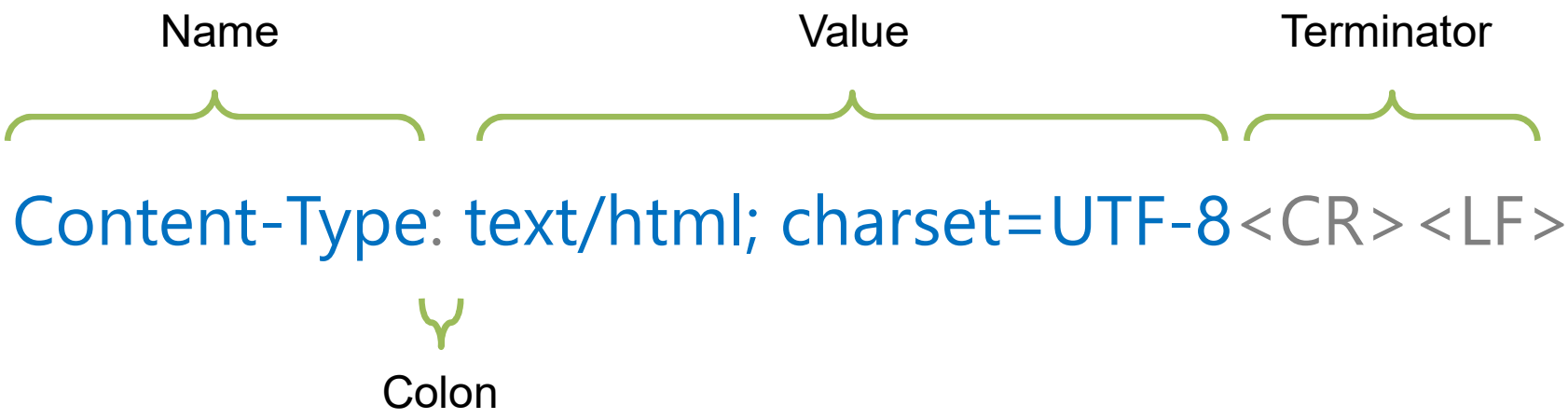
HTTP Request Headers

- Collection of name / value pairs sending information from client to server
  - Generally contextual information, not application data
- Immediately follow the request line
- Each is a colon-separated name / value pair, in clear text, terminated by a CR-LF pair
- Additional CR-LF pair indicates end of headers

Request Header Format

- Header names
  - Descriptive names, no spaces, hyphens commonly used to delimit words
  - Standard header fields (defined by several RFCs)
  - Custom header fields
    - Application specific
    - Convention used to be to prefix name with “x-”, but now deprecated
- Header values
  - Variable length “blob” of data
  - Frequently XML or JSON, but could be anything

Example Request Header



Mandatory HTTP Request Header

Host: <server name or ip> [:port]

- "Host" header is required
  - Domain name or IP address of the server being contacted
  - TCP port number on which the server is listening
- Port may be omitted if standard port for service being used

Common Standard Request Headers

- Content-Type
  - MIME type of data in the request body
  - Required with POST and PUT requests
- Content-Length
  - Length of data in request body, in octets (8-bit bytes)
- Accept
  - MIME type(s) acceptable in response body
- Cookie
  - Value of an HTTP cookie previously received from the server
  - Set-Cookie response header

HTTP Request Body

- A variable length “blob” of data
- Immediately follows the blank line after the headers
- Optional
  - GET and DELETE requests do not include a body
  - POST and PUT requests do

Example GET Request

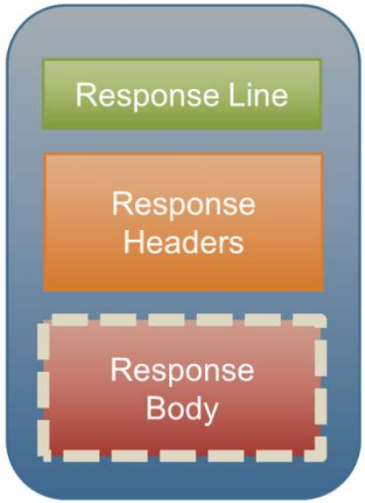
```
GET /api/orders/455320 HTTP/1.1<CR><LF>
Host: www.acme.com<CR><LF>
Accept: application/json<CR><LF>
<CR><LF>
```

Example POST Request

```
POST /api/orders HTTP/1.1<CR><LF>
Host: www.acme.com<CR><LF>
Content-Type: application/json<CR><LF>
Content-Length: 108<CR><LF>
<CR><LF>
{“account”:10986223,“ponumber”:19734,“items”:[{“sku”:“ABB701”,“quantity”:1},{“sku”:“CRD100”,“quantity”:10}]}
```

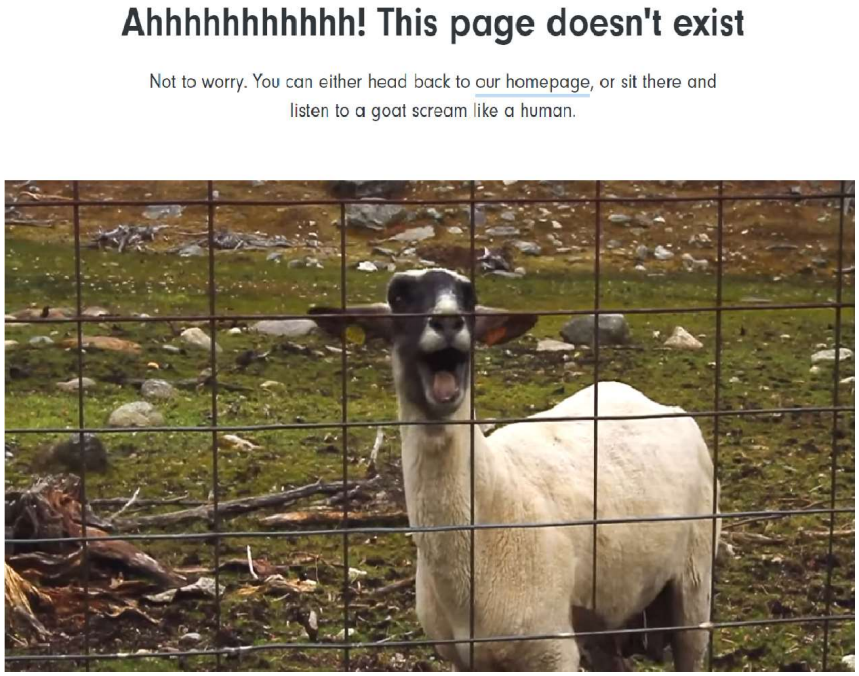
Anatomy of an HTTP Response

- Response line
  - HTTP version
  - Status code
- Response headers
- Response body (optional)



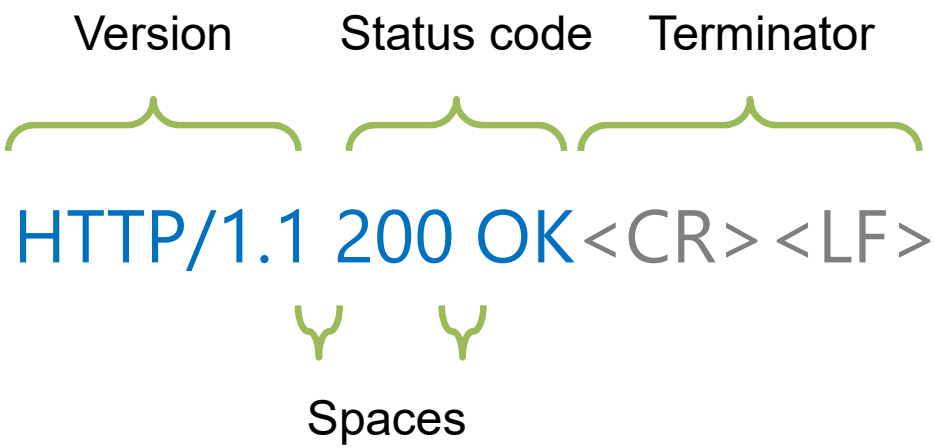
HTTP Status Codes

- Status codes are three-digit numeric values
  - 1xx – Informational
  - **2xx – Success**
  - 3xx – Redirection
  - **4xx – Client Errors**
  - **5xx – Server Errors**



http://bluegg.co.uk/404

Example HTTP Response Line



Common Success Codes

- 200 OK
  - Entity returned in response body
- 201 Created
  - No entity returned in response body
  - Location header should be returned (more later)
- 202 Accepted (but not processed yet)
  - May or may not be processed
  - May or may not succeed
- 204 No Content
  - Success, but no entity in response body

Common Client Error Codes

- 400 Bad Request
  - Invalid or missing data, etc.
- 401 Unauthorized
  - Authentication required and not yet done, or failed
- 403 Forbidden
  - User may be logged in but doesn't have necessary permissions
- 404 Not Found
  - Requested entity was not found

Common Server Error Codes

- 500 Internal Server Error
  - Unhandled exception in server code
  - Should never be allowed to happen, but we're not perfect!
- 501 Not Implemented
  - Usually implies future availability, e.g., new feature coming
- 503 Service Unavailable
  - Overloaded, down for maintenance, etc.

Response Headers

- Like request headers, but sent from server to client
  - Collection of name / value pairs transmitted after response line
  - Some are pretty useful, some are informational

- Common response headers

Content-Type	MIME type of data in the response body
Content-Length	Length of data in the response body in octets (8-bit bytes)
Location	After resource creation, contains the URI to retrieve the newly created resource
Set-Cookie	HTTP cookie to be preserved by the client and returned via the cookie header in subsequent requests

Example GET Request / Response

```
GET /api/orders/455320 HTTP/1.1<CR><LF>
Host: www.acme.com<CR><LF>
<CR><LF>

HTTP/1.1 200 OK<CR><LF>
Content-Type: application/json<CR><LF>
Content-Length: 107<CR><LF>
<CR><LF>
{“order”:455320,“customer”:10986223,“items”:[{“sku”:“ABB701”,“quantity”:
1},{“sku”:“CRD100”,“quantity”:10}]}
```

Example DELETE Request / Response

```
DELETE /api/orders/1052632 HTTP/1.1<CR><LF>
Host: www.acme.com<CR><LF>
<CR><LF>

HTTP/1.1 204 No Content<CR><LF>
<CR><LF>
```

Visualizing HTTP Messages

- Synergy HTTP API
  - Enable logging to record the EXACT request and response messages
- HTTP utilities
  - Developer tools in web browsers
    - GET requests
  - Fiddler
    - http://www.telerik.com/fiddler
  - Postman
    - https://www.getpostman.com

```
POST /price_consignment HTTP/1.1
Host: localhost
Content-Type: text/xml
Content-Length: 188

<?xml version='1.0'?>
<package>
  <origin>95670</origin>
  <destination>73455</destination>
  <width>100</width>
  <depth>45</depth>
  <height>15</height>
  <weight>95</weight>
</package>
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 104

<?xml version='1.0'?>
<quotation>
  <reference>98765</reference>
  <cost>135.00</cost>
  <currency>USD</currency>
</quotation>
```

HTTP and the Persistence of State

- HTTP is a stateless protocol
  - Not requirement for HTTP servers to retain information or status about each user for the duration of multiple requests
  - Request / response / done

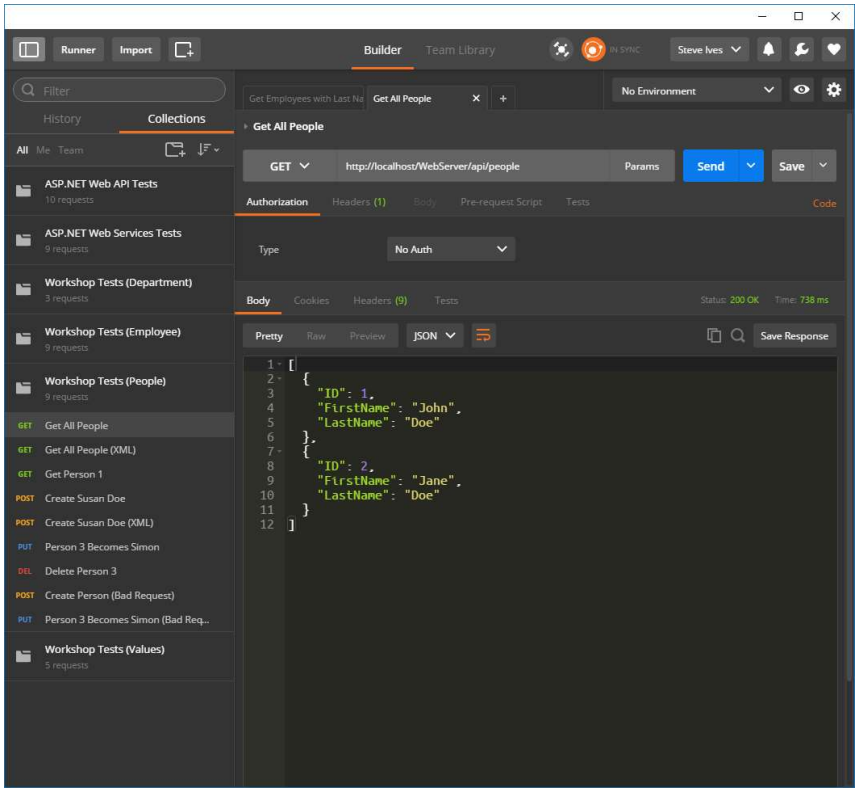
- Some web application frameworks implement state
  - Server side “sessions”
  - HTTP cookies allow state to be re-established for subsequent requests
  - RESTful services are stateless (more later)

Technology Primer - Postman

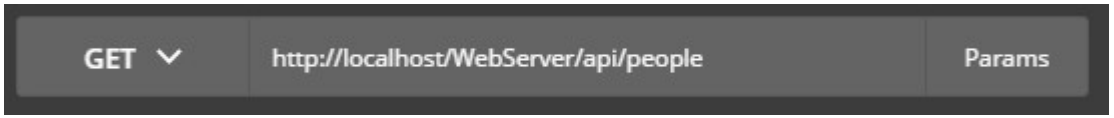
Building RESTful Services with Synergy .NET

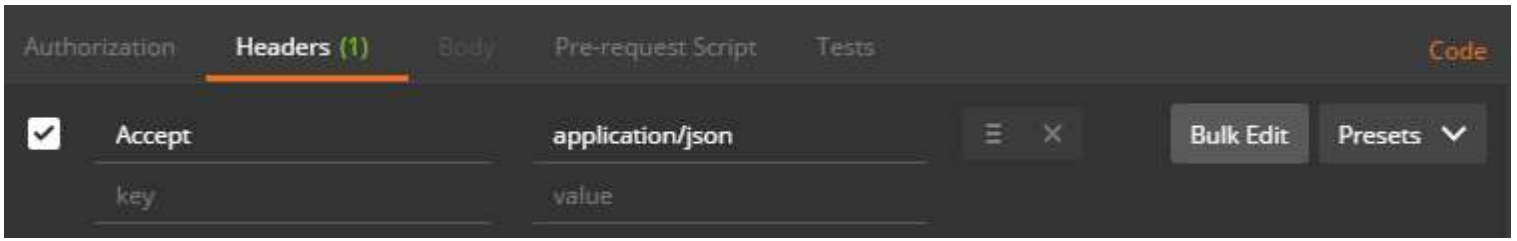
Postman

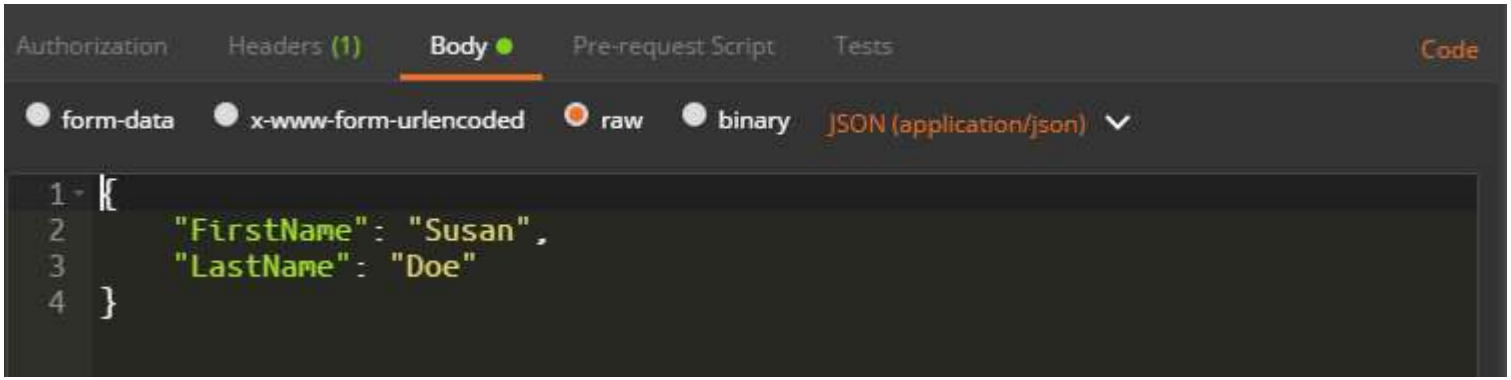
- Google Chrome App extension
- Build and issue HTTP requests
  - Define URL
  - Specify HTTP methods
  - Add request headers
  - Provide body data
  - Send request
  - Examine response
- Designed for testing RESTful APIs
- https://www.getpostman.com



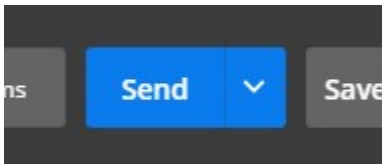
Postman—Defining an HTTP Call

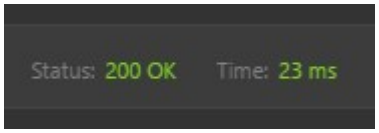


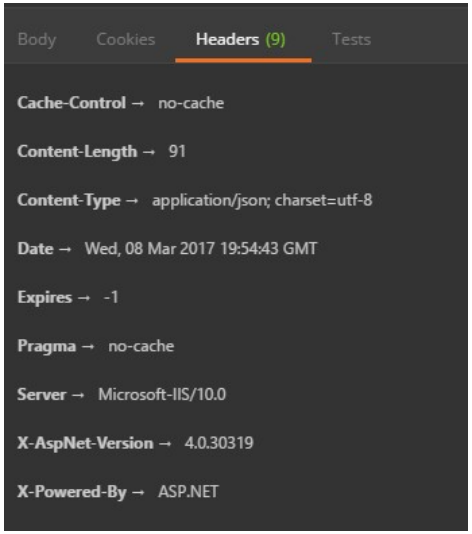


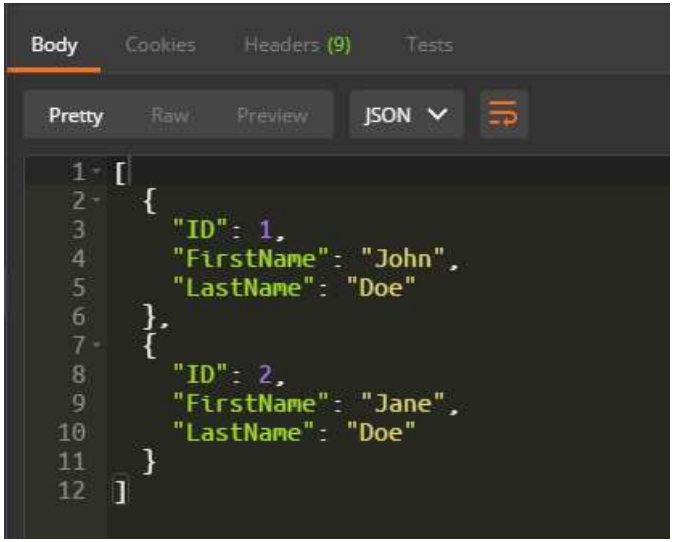


Postman—Making an HTTP Request



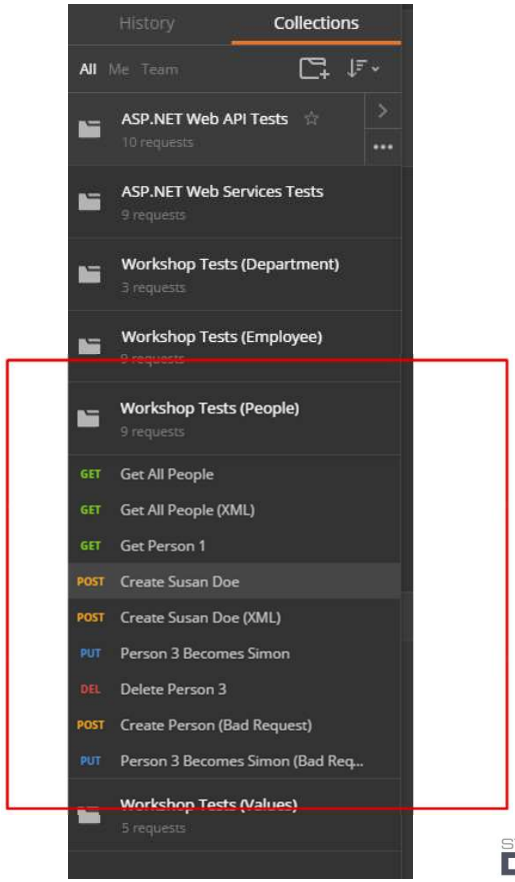






Postman Collections

- Collections are like projects
  - Group related tests together
- Export to JSON file
- Import on another PC
- Optionally create account and log-in
  - Postman or Google account
  - Synchronizes collections via cloud



Demo 1: Supporting Technologies

- Let's see HTTP in action, using Postman as our tool

Technology Primer - JSON

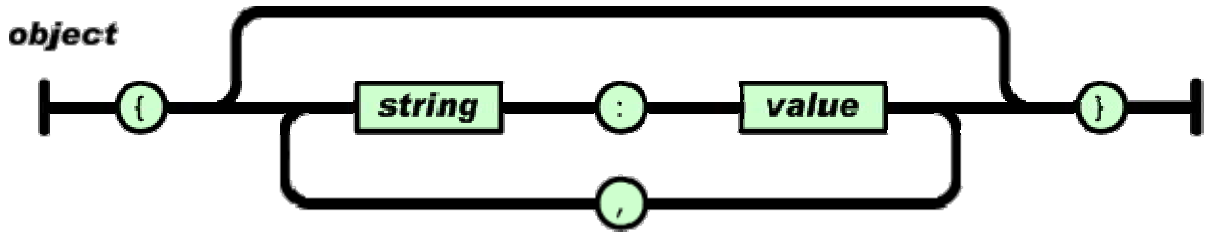
Building RESTful Services with Synergy .NET

JavaScript Object Notation (JSON)

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address":
    {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber":
    [
        {
          "type": "home",
          "number": "212 555-1234"
        },
        {
          "type": "fax",
          "number": "646 555-4567"
        }
    ]
}
```

- Text format facilitating structured data interchange between all programming languages
- Inspired by JavaScript object literals
- Braces, brackets, colons, and commas delimit data
- Agnostic about numbers; uses the representation that humans use: a sequence of digits
- Objects are simple collections of name/value pairs
- Also supports ordered lists of values (arrays)

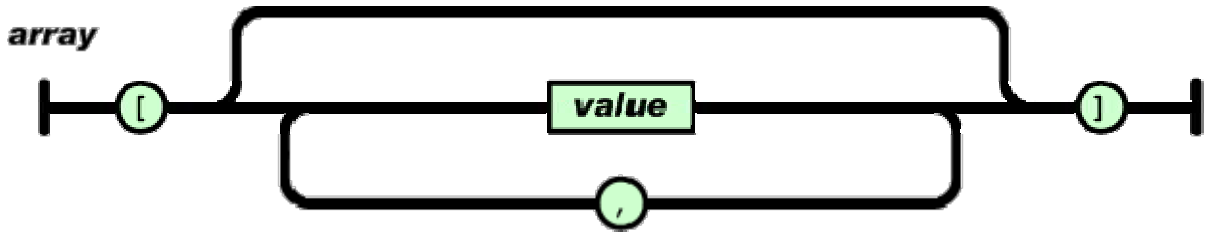
JSON Object Notation



- An *object* is an unordered set of name/value pairs
- Begins with left brace { and ends with right brace }
- Each name is followed by colon
- Name/value pairs are separated by a comma

```
{ "EmployeeID": 100, "FirstName": "Steve", "LastName": "Ives" }
```

JSON Array Notation

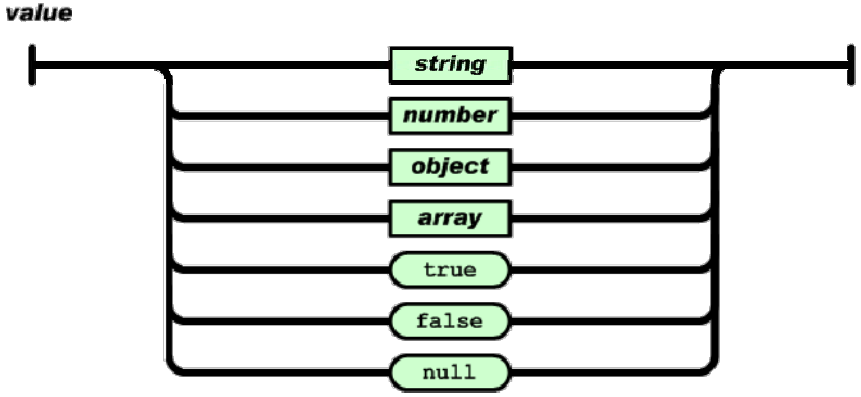


- An *array* is an ordered collection of values
- Begins with [(left bracket) and ends with] (right bracket)
- Values are separated by , (comma)

```
[ "Red", "Green", "Blue" ]
```

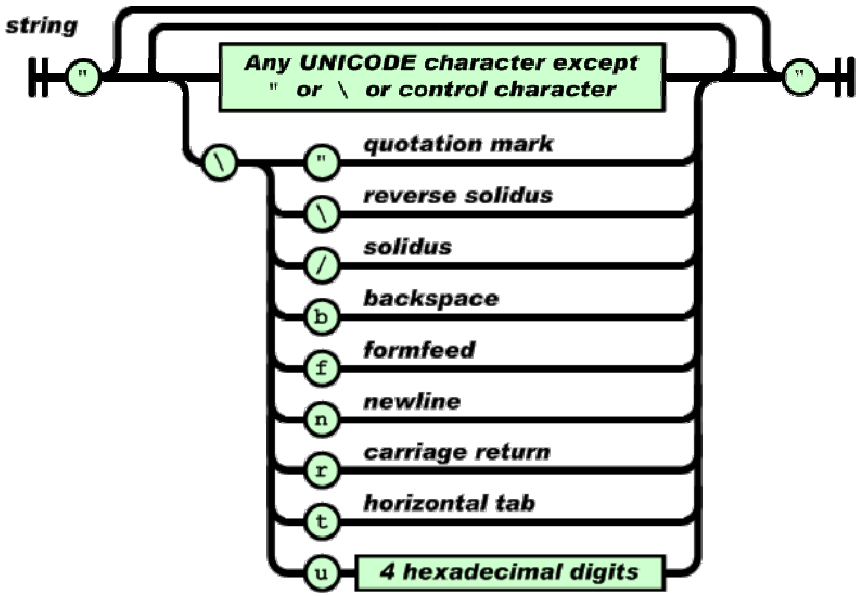
```
[ 1, 2, 3 ]
```

JSON Value Notation



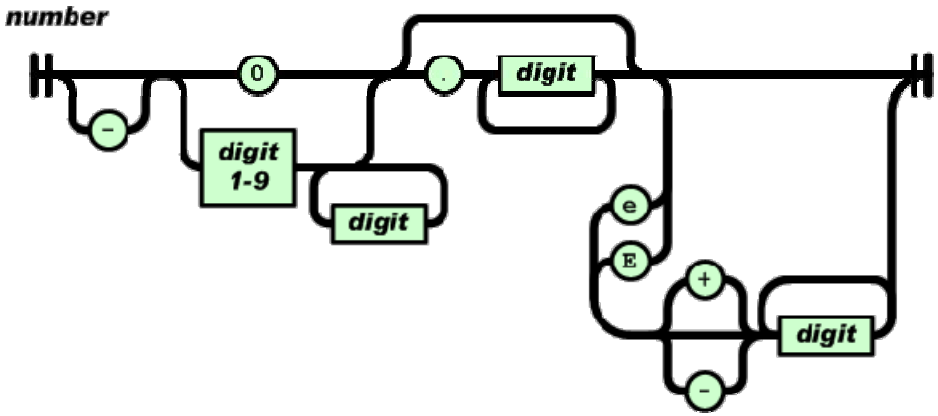
- A *value* can be
  - A string in double quotes
  - A number
  - An object
  - An array
  - True or false or null
- These structures may be nested

JSON String Notation



- A *string* is a sequence of zero or more Unicode characters wrapped in double quotes
- Backslash is an escape character
- A character is represented as a single character string
- Very much like a C or Java string

JSON Number Notation



- A *number* is very much like a C or Java number, except that the octal and hexadecimal formats are not used

All JSON Data Is an Object or an Array

Object

```
{
  "EmployeeID": 100,
  "FirstName": "Steve",
  "LastName": "Ives"
}
```

Array

```
[ "Red", "Green", "Blue" ]

[ 1, 2, 3 ]

[
  1,
  "Red",
  500
]
```

Array of Objects

```
[
  {
    "EmployeeId": 1,
    "FirstName": "John",
    "LastName": "Doe"
  },
  {
    "EmployeeId": 2,
    "FirstName": "Jane",
    "LastName": "Doe"
  },
  {
    "EmployeeId": 3,
    "FirstName": "William",
    "LastName": "Doe"
  }
]
```

JSON

- Whitespace may be inserted between any pair of tokens
- Because of its simplicity JSON grammar is never expected to change
- Further information
  - http://www.json.org

Serialization and Deserialization

- Serialization
  - Transform an object (property names and values) into a string
  - Common targets are XML and JSON

- Why?
  - Strings are easy to transmit to other places

- Deserialization
  - Transform a string back into an object

Working with JSON in .NET

- Json.NET by James Newton-King (Newtonsoft)
  - Open source library for processing JSON data
  - Free for commercial use
  - Easy to use and FAST
  - Windows, UWP, Windows Phone, Mono, and Xamarin
- Widely used, including Microsoft in project templates
- NuGet package Newtonsoft.Json
  - http://www.newtonsoft.com/json

Json.NET—Basic Use

```
;;Make an object
data p = new Person() { Id=1, FirstName="Steve", LastName="Ives" }

;;Serialize it to JSON
data json = JsonConvert.SerializeObject(p)
;json contains {"Id":1,"FirstName":"Steve","LastName":"Ives"}

;;De-serialize the JSON string back to a object
p = JsonConvert.DeserializeObject<Person>(json)
```

Representational State Transfer (REST)

Building RESTful Services with Synergy .NET

Representational State Transfer (REST)

- A way of providing interoperability between computer systems

- Access and manipulate textual representations of resources
  - In a uniform way
  - In a stateless way

- Architectural goals
  - Accessibility from any environment
  - Simplicity through use of a uniform interface
  - Performance and scalability

REST Is a Design Pattern

- Unlike SOAP-based web services, there is no official standard for RESTful Web APIs
- SOAP is a protocol, REST is an architectural style
- REST is not a standard in itself, but RESTful implementations make use of various standards
  - URI RFC 3986
  - HTTP RFC 7230 - 7237
  - JSON RFC 7159

Architectural Constraints

- Five guiding constraints define a RESTful system:
  - Client-server
  - Stateless
  - Cacheable
  - Uniform interface
  - Layered system
- By conforming, services gain desirable properties
  - Performance, scalability, simplicity, etc.
- Services violating these constraints not considered RESTful
  - How much does that matter?

Client – Server Constraint

- RESTful services must adopt a client-server style
- Separation of concerns
  - User interface from business logic and data storage concerns
- Supports independent evolution of client-side logic
- Improves portability of the UI across multiple platforms
- Improves scalability by simplifying the server components

Stateless Constraint

- No client context stored on the server between requests
  - Each request from any client contains all the information necessary to service the request
  - Session state is held in the client

- Server may transfer session state to another service to maintain a persistent state for a period and allow authentication
  - E.g., a database

Cacheable Constraint

- Clients and intermediaries may cache responses
- Responses must
  - Implicitly or explicitly define themselves as cacheable, or not
  - Indicate how long cacheable data is valid
- Clients should
  - Check for and persist cacheable data
- Well-managed caching partially or completely eliminates some client–server interactions
  - Improving scalability and performance

Uniform Interface Constraint

- Fundamental to the design of any REST service
  - Defines the interface between client and server
- Resources identified by consistently structured URIs
  - http://myserver.com/api/customers
  - http://myserver.com/api/customers/12345
  - http://myserver.com/api/customers/12345/orders
  - http://myserver.com/api/customers/12345/orders/3225
- Operations on resources specified via HTTP methods
  - GET, POST, PUT, and DELETE
- Self-descriptive messages
  - Messages include information on how to process message
    - E.g., Content-Type: application/json

Relationship between URI and HTTP Methods

URI	GET	POST	PUT	DELETE
/resources	Return all resources	Create one resource	Replace all resources	Delete all resources
/resources/1	Return resource 1	Not generally used	Replace resource 1	Delete resource 1

- Most APIs use plural URIs (resources not resource)
- Return all—data… or URIs
- Many APIs don't use "Replace all" and "Delete all"

HATEOAS

- **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate
  - Part of the uniform interface constraint

- REST service should behave like a hypermedia service (like the Web)
  - Base URI serves up collection of links to available resources
  - Responses include metadata—links to available operations

Layered System Constraint

- Clients can't tell whether they are connected directly to the end server or to an intermediary

- Intermediary servers may improve scalability
  - Load balancing
  - Shared caches, etc.

Why Use REST instead of SOAP Services?

- More open, reach more clients
- Less overhead
- Less duplication
- More standardized
- More human readable and testable
- Content negotiation—not forced into using XML

Demo 2: RESTful Web Service

- Goals
  - Demonstrate a RESTful web service in use
  - Show what we will be developing today
- Activities
  - Demo the final Web API solution

ASP.NET Web API 2

Building RESTful Services with Synergy .NET

Building RESTful Web Services in .NET

- WCF REST
  - WCF designed around a SOAP stack
  - Designed for XML—JSON possible but not natural
  - Performance and scalability were common issues
- ServiceStack
  - Popular third-party solution a couple of years ago
  - Designed for REST, easy, good performance, and scalability
  - Great alternative to WCF REST… before Web API
- ASP.NET Web API 2
  - Current “state of the art”

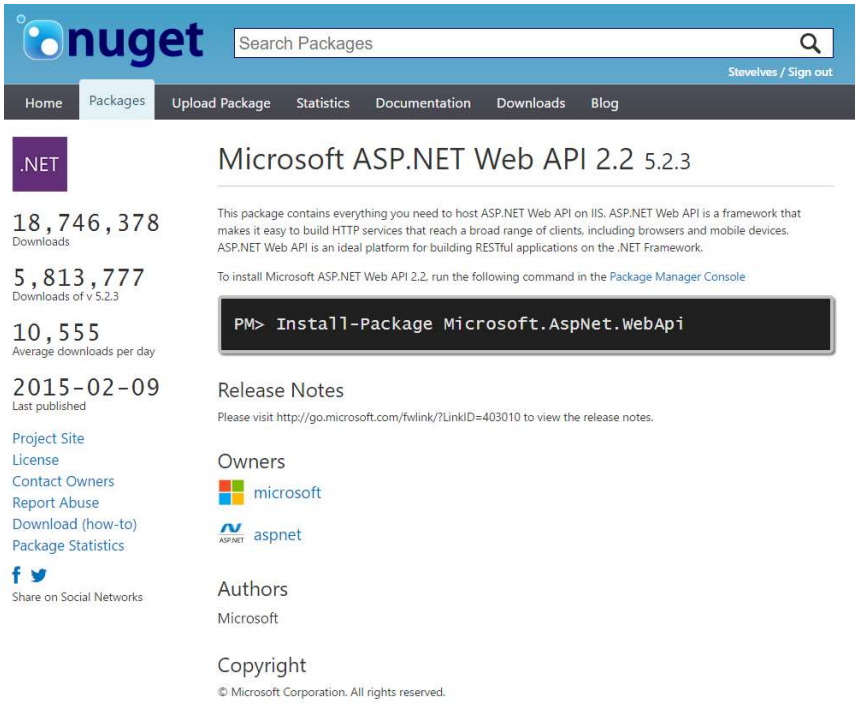
What Is ASP.NET Web API 2?

- Framework for building RESTful Web APIs
  - No SOAP or WSDL
  - HTTP communication
- Closely related to ASP.NET MVC
  - Models and controllers
- Convention over configuration
  - Naming conventions
  - Verbs map to HTTP methods
- Easy to learn, use, and deploy… and FAST!
- HUGE improvement over earlier technologies

- Desktop - **MVVM**
- Web - **MVC**
- **Web API**
  - Closely related to MVC
  - Without the V!

Obtaining Web API

- NuGet
  - **Microsoft.AspNet.WebApi**
  - Main package containing everything
- Many other packages have specific components
  - **Microsoft.AspNet.WebApi.Core**
    - Core hosting support
  - **Microsoft.AspNet.WebApi.Client**
    - Core client-side support
- NuGet packages auto-restored during first build
- Notice NuGet packages versioned differently as patches are released



Web API Controllers

- Controller classes contain operations
  - System.Web.Http.ApiController
- Operations defined by public methods
- Accept / return data
  - Parameters and return value
  - Simple value types
    - int, string, Boolean
  - Complex types
    - Model classes
  - Arrays and collections

```
;;; <summary>
;;; Exposes operations relative to a collection of people.
;;; </summary>
public class PeopleController extends ApiController

    private static people, @List<Person>

    public method PeopleController
    proc
        if (people == ^null)
        begin
            people = new List<Person>() {
            &   new Person() { ID=1, FirstName="John", LastName="Doe"},
            &   new Person() { ID=2, FirstName="Jane", LastName="Doe"}
            &   }
        end
    endmethod

    ;;;  <summary>
    ;;;  Get all people.
    ;;;  </summary>
    ;;;  <returns>A collection of all people.</returns>
    public method Get, @IEnumerable<Person>
    proc
        mreturn people
    endmethod
```

Web API Models

- Model classes define data
- Used to define inbound and outbound messages
  - Over and above simple types
- Some simple, some complex, but all just POCOs
  - All data, no code
- By convention, model classes go in a Models folder and Models sub-namespace

```
;;; <summary>
;;; Represents a person.
;;; </summary>
public class Person

    ;;; <summary>
    ;;; Person ID.
    ;;; </summary>
    public readwrite property ID, int

    ;;; <summary>
    ;;; First name.
    ;;; </summary>
    {Required}
    {StringLength(15)}
    public readwrite property FirstName, string

    ;;; <summary>
    ;;; Last name.
    ;;; </summary>
    {Required}
    {StringLength(15)}
    public readwrite property LastName, string

endclass
```

Routing Requests to Controllers

- Routes define how inbound requests map to controllers
  - Along with naming conventions
- Examples
  - Requests to www.myapi.com/api/**Employees** by convention will be processed by a class named **EmployeesController**
  - Requests to www.myapi.com/api/**Inventory** by convention will be processed by a class named **InventoryController**
- Discovery
  - Web API discovers controllers wherever they are located
  - By convention controllers go in a Controllers folder and in a Controllers sub-namespace

Default Routing Table

Services are exposed via HTTP at a “base URL”
e.g., **http://www.myapp.com/**

```
// Default Web API routing rules
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

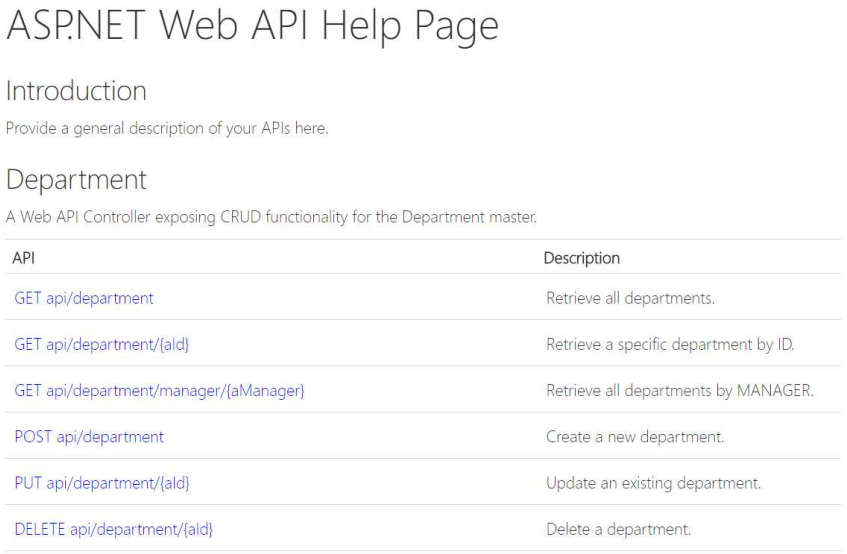
Routing Requests to Methods

- HTTP verbs determine which METHOD in a controller a request will be processed by
  - Naming conventions
  - Parameter matching
  - Method overloading rules

- Examples
  - HTTP **GET** to **www.myapi.com/api/Employees** will be processed by a method named **Get** which has no parameters
  - HTTP **PUT** to **www.myapi.com/api/Employees/1** will be processed by a method named **Put** which accepts 1 parameter
  - Both in **EmployeesController**

Self Documenting

- ASP.NET Web API projects have scaffolding pages and code that presents documentation for Web API services
- Decorate code to enhance documentation
  - XML doc comments
  - Data annotation attributes



ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

Department

A Web API Controller exposing CRUD functionality for the Department master.

API	Description
GET api/department	Retrieve all departments.
GET api/department/{ald}	Retrieve a specific department by ID.
GET api/department/manager/{aManager}	Retrieve all departments by MANAGER.
POST api/department	Create a new department.
PUT api/department/{ald}	Update an existing department.
DELETE api/department/{ald}	Delete a department.

Hosting Web API

- For development
  - IIS or IIS Express
- For deployment
  - IIS
  - Self host in an app or service (OWIN)
  - Azure website
  - Azure worker role (OWIN)
- Minimum requirements
  - Visual Studio 2010 / .NET 4.0 or later
  - .NET 4.5 added desirable async support

Demo 3: Create an ASP.NET Web API Service

- Goals
  - Create an ASP.NET web app to host our environment
  - Explore the default environment

- Activities
  - Create an ASP.NET project
  - Configure it to use IIS
  - See how API help is generated
  - Enhance the API help information
  - Create Postman tests for the default web API service


SYNERGY
2017
DEVPARTNER
CONFERENCE
2017SYNERGY
DEVPARTNER
CONFERENCE
ATLANTA, GA
MAY 9-12
2017
Synergy .NET Web API Services
Building RESTful Services with Synergy .NET

Synergy .NET Web API Services

- Web services frequently hosted by websites

- Websites are a combination of things
  - HTML, CSS, JavaScript—no Synergy support for that
  - Code—that we can do!

- Web SERVICES are just code
  - Implement with Synergy .NET
  - Interact with existing systems (*xf*NetLink and *xf*ServerPlus)
  - Leverage existing libraries
  - Access data (locally or via *xf*Server)

Development Workflow

- Create a Synergy .NET class library
- Add Web API support (NuGet)
- Use Synergy to create model classes
  - CLS… don't use alphas, decimals and integers!
  - CodeGen may be useful here
- Use Synergy to create controller classes
  - Ditto… CLS… and even CodeGen for basic CRUD
- Reference the Synergy assembly in the hosting application
- Magic happens
  - Web API discovers and exposes our services
  - No complex configuration (not missing you, WCF!)

Demo 4: Synergy .NET Web API Services

- Goals
  - Move the implementation of the sample service from C# to Synergy
  - Retain the extended API help established earlier

- Activities
  - Add a Synergy .NET class library
  - Configure it for Web API development
  - Convert ValuesController from C# to Synergy
  - Modify the source of extended API help information

So far we have…

- Created an ASP.NET web application to host Web API 2 services
  - C#, but we won't be writing any significant code here
  - IIS Express or IIS
  - Other hosting options available (OWIN later)

- Created a Synergy .NET class library to define and implement Web API 2 services
  - Discovered and hosted by the hosting app
  - We'll implement our services here

Exposing Complex Types

Building RESTful Services with Synergy .NET

Exposing Complex Types

- Model classes
  - Used when data requirements go beyond simple types
- Used to send / receive data
  - Single object
  - Array or collection
  - Complex hierarchy
- Serialized to be sent
  - Object to JSON or XML
- Deserialized when received
  - JSON or XML back to object
- Automatic
  - You code with objects

```
;;; <summary>
;;; Represents a person
;;; </summary>
public class Person

    ;;; <summary>
    ;;; Person ID
    ;;; </summary>
    public readwrite property ID, int

    ;;; <summary>
    ;;; First name
    ;;; </summary>
    public readwrite property FirstName, string

    ;;; <summary>
    ;;; Last name
    ;;; </summary>
    public readwrite property LastName, string

endclass
```

Demo 5: Exposing Complex Types

- Goals
  - More realistic example, involving complex data via a model class
  - Expose functionality relating to new model
  - Hard-coded data for now

- Activities
  - Define a model class
  - Create a Web API controller
  - Provide some sample data
  - Implement CRUD operations
  - Create Postman tests

Debugging Web API Actions

Building RESTful Services with Synergy .NET

Debugging Web API Actions

- Full Visual Studio debugging environment available

- Basic debug operations
  - F9 Set breakpoint(s)
  - F5 Start debugging

  - F10 Step over
  - F11 Step into
  - Shift + F11 Step out

  - Hover Examine

  - Locals window

Demo 6: Debugging Web API Actions

- Goals
  - Set breakpoints in action code
  - Debug actions

- Activities
  - Set various breakpoints in code
  - Start a debugging session
  - Use Postman to initiate actions
  - Stop debugging and delete breakpoints

Content Negotiation

Building RESTful Services with Synergy .NET

Content Negotiation

- Web API respects HTTP rules and features
- **Content-Type** header
  - Client ability to specify the TYPE of data being sent
- **Accept** header
  - Client ability to express preference for type of data received
- With web services, usually a choice between XML or JSON
  - Inherent support in Web API
  - Determines which serialization mechanisms used

Content Negotiation - Data Sent TO Server

- Client specifies type of data being sent
  - **Content-Type** request header
    - **application/json**
    - **text/json**
    - **application/xml**
    - **text/xml**
    - **application/x-www-form-urlencoded**

- Server will
  - Process the data ... if it's a supported format
  - Or … respond with a 406 Not Acceptable

Form Encoded Example

```
PUT http://localhost/api/person/1 HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

id=1&FirstName=John&LastName=Doe
```

Content Negotiation - Data Sent FROM Server

- Client expresses PREFERENCE for type to receive
  - **Accept** request header
  - Could specify several types

- Server will
  - Respond with the first requested type it supports
  - Or … respond with it's first supported format

  - And … indicate format via a Content-Type response header

Content Negotiation Example

- GET http://services.myapp.com/api/people/1
  - Requests information about a specific person
- **Accept: application/json**
  - Person data in JSON format (standard)
- **Accept: application/xml**
  - Person data in XML format (standard)
- **Accept: text/vcard**
  - Vcard data for the person (custom)
- **Accept: image/png**
  - Photograph of person (custom)

Demo 7: Content Negotiation

- Goals
  - Be able to send either XML or JSON to a service
  - Be able to receive either XML or JSON from the service
- Activities
  - Modify a request to explicitly specify Accept header
  - Modify a request to explicitly specify a Content-Type header

Exception Handling

Building RESTful Services with Synergy .NET

Exception Handling

- By default unhandled exceptions turned into HTTP 500 errors
  - Internal Server Error
- Another option is to throw an **HttpResponseException**
  - Web API turns this into a structured response
- Response type specified via parameter
  - System.Net.**HttpStatusCode** enum

```
data p = people.FirstOrDefault(lambda (p) { p.Id == id })
if (p==^null)
    throw new HttpResponseException(HttpStatusCode.NotFound)
mreturn p
```

- More exception handling options later

Common HTTP Status Codes in Web APIs

- **HttpStatusCode.OK**
  - Success, data in response body
- **HttpStatusCode.NoContent**
  - Success, no data in response body
- **HttpStatusCode.NotFound**
  - Requested item not found
- **HttpStatusCode.Created**
  - Resource created, data being returned
  - Otherwise use NoContent
- **HttpStatusCode.Unauthorized**
  - You don't have access to this
- **HttpStatusCode.BadRequest**
  - You're sending me bad data
- **HttpStatusCode.ServiceUnavailable**
  - Service currently unavailable

Exception Filters

- Customize how Web API responds to exceptions
- Custom class - extend ExceptionFilterAttribute
- Override OnException()
- Called when an unhandled exception occurs, *except* HttpResponseException

```
public class NotFoundExceptionFilterAttribute extends ExceptionFilterAttribute

    public override method OnException, void
        context, @HttpActionExecutedContext
    proc
        if (context.Exception .is. KeyNotSameException) || (context.Exception .is. EndOfFileException)
            context.Response = new HttpResponseMessage(HttpStatusCode.NotFound)
    endmethod

endclass
```

Registering Exception Filters

- Several ways to register a Web API exception filter
  - Per action - apply attribute to a specific method

```
{NotFoundExceptionFilter}
public method Get, @Person
    required in id, int
```

  - Per controller - apply attribute to the controller class

```
{NotFoundExceptionFilter}
public class PeopleController extends ApiController
```

  - Globally, in the Web API host configuration code

```
config.Filters.Add(new NotFoundExceptionFilterAttribute());
```

Demo 8: Exception Handling

- Goals
  - Return appropriate HTTP status codes to the client
- Activities
  - Modify the Get, Put, and Delete methods to return “Not Found” as appropriate
  - Test in Postman

Action Results in Web API 2

Building RESTful Services with Synergy .NET

Action Results in Web API 2

- Prior to Web API 2 primary exception handling mechanisms
  - HttpResponseException
  - Exception filters

- Web API 2 controllers can return any of the following:
  - void
  - HttpResponseMessage
  - IHttpActionResult
  - Some other type

- Each handled differently when producing HTTP response

Action Returns Void

- Simplest case: action returns void

- Web API returns
  - Empty response body
  - HTTP 204 (no content)

```
public method  Delete, void
    required in personId, int
proc
    _repository.Delete(personId)
endmethod
```

Action Returns HttpResponseMessage

- Web API converts return value directly into an HTTP response
- Developer has a lot of control over the specific details of the response

```
public method Get, @HttpResponseMessage
    required in id, int
proc
    data p = people.FirstOrDefault(lambda (p) { p.Id == id })
    if (p == ^null) then
        mreturn Request.CreateResponse(HttpStatusCode.NotFound)
    data response = Request.CreateResponse(HttpStatusCode.OK, p)
    response.Headers.CacheControl = new CacheControlHeaderValue() { MaxAge = TimeSpan.FromMinutes(20) }
    mreturn response
endmethod
```

Action Returns IHttpActionResult

- Implement IActionResult interface to create your own custom HttpResponseMessage factory
  - Isolates common logic for creating HTTP responses
  - Simplifies code in controller actions
- ApiController provides several helper methods
  - Makes it really easy to implement this pattern
  - Ok()
  - NotFound()
  - Others

```
public method Get, @IHttpActionResult
    required in personId, int
proc
    data foundPerson = _repository.Get(personId)
    if (foundPerson == ^null) then
        mreturn NotFound()
    else
        mreturn Ok(foundPerson)
endmethod
```

Action Returns Some Other Type

- Web API uses a media formatter to serialize the return value
- Just return something serializable!
- Writes the serialized value into the response body
- Response status is 200 (OK)

```
public method Get, @IEnumerable<Person>
proc
    mreturn GetAllPeopleFromDB()
endmethod
```

Demo 9: Web API 2 Action Results

- Goals
  - Improve the quality of return status information issued to clients
  - Explicitly control return status

- Activities
  - Enhance the actions in PeopleController
  - Use HttpResponseMessage
  - Use IHttpActionResult
  - Test in Postman

Model Validation in ASP.NET Web API

Building RESTful Services with Synergy .NET

Model Validation in ASP.NET Web API

- Data sent from clients to the server needs validating
- Web API performs some validations
  - Simple types can be somewhat validated
    - Don't pass a string to an integer parameter
    - Make sure date values are valid, etc.
  - Complex types also can be somewhat validated
    - Does deserialization fail?
  - If there's a problem—HTTP 400, Bad Request
- Other things require additional validation
  - Required fields
  - Maximum string lengths (remember, we use alphas)
  - Referential integrity to data files

Data Annotations Framework

- System.ComponentModel.DataAnnotations
  - Contains attributes that may be applied to model properties
  - Express data validation rules
- Lots of them, some of the more useful ones:
  - Required
  - StringLength(n), MaxLength(n), and MinLength(n)
  - Range(min,max)
  - Phone(), EmailAddress()
  - CreditCard()
  - RegularExpression(regex)
- Attribute classes have various properties
  - E.g., ErrorMessage

Applying Data Annotations to Models

- Import the namespace
- Apply attributes to properties
- Enhance with error messages if required

```
;;; <summary>
;;; Represents a person.
;;; </summary>
public class Person

    ;;; <summary>
    ;;; Person ID.
    ;;; </summary>
    public readwrite property ID, int

    ;;; <summary>
    ;;; First name.
    ;;; </summary>
    {Required}
    {StringLength(15)}
    public readwrite property FirstName, string

    ;;; <summary>
    ;;; Last name.
    ;;; </summary>
    {Required}
    {StringLength(15)}
    public readwrite property LastName, string

endclass
```

```
{Required}
{StringLength(15, ErrorMessage="First can't be longer than 15 characters!")}
public readwrite property FirstName, string
```

Custom Validation

- If the built-in validation attributes are not enough, you can write your own

- {CustomValidation}
  - Names a method to call to perform custom validation
  - Applied at the property level, or for an entire model!

Implementing Validation

- If data annotations are present in a model, Web API will validate against them during deserialization
  - If there's a problem, your controller's **ModelState.IsValid** property is set to false
  - Actions can check this property and respond with "bad request"

```
public method Post, @HttpResponseMessage
    {FromBody}
    required in newPerson, @Person
proc
    if (!ModelState.IsValid)
        mreturn Request.CreateErrorResponse(HttpStatusCode.BadRequest,ModelState)
```

- Model state validation can be applied in a global filter

Returning Model State to the Client

- Make validation failure information available to the client by returning ModelState in the HttpResponseMessage

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Date: Tue, 16 Jul 2013 21:02:29 GMT
Content-Length: 331

{
  "Message": "The request is invalid.",
  "ModelState": {
    "product": [
      "Required property 'Name' not found in JSON. Path '', line 1, position 17."
    ],
    "product.Name": [
      "The Name field is required."
    ],
    "product.Weight": [
      "The field Weight must be between 0 and 999."
    ]
  }
}
```

Demo 10: Model Validation

- Goals
  - Implement simple server-side validation of inbound data
  - Improve the quality of exception info returned to clients

- Activities
  - Reference the data annotations framework
  - Decorate the Person model with some data validation attributes
  - Implement model validation in an action
  - Implement model validation as a global filter

Using the Repository Pattern

Building RESTful Services with Synergy .NET

Direct Data Access

- In many applications, business logic accesses data directly from data stores
  - Files, databases, web services, etc.

- Directly accessing the data can result in
  - Duplicated code
  - A higher potential for programming errors
  - Weak typing of the business data
  - Difficulty centralizing data-related policies such as caching
  - An inability to easily test the business logic in isolation from external dependencies

Using the Repository Pattern

- Separate the logic that retrieves data and maps it to an entity model from the business logic that acts on the model

- Repository class mediates between the data source and the business logic layers of an application
  - Queries the data source for the data
  - Maps the data from the data source to a business entity
  - Persists changes in the business entity to the data source

- Separate business logic from the interactions with the underlying data source or web service

Repository Pattern Benefits

- Centralizes data logic or web service access logic
- Provides a substitution point for the unit tests
- Provides a flexible architecture that can adapt as the overall design evolves

Demo 11: Using the Repository Pattern

- Goals
  - Separate business logic from data access logic
  - Make data access logic easily reusable across multiple controllers
  - Interact with data in ISAM files

- Activities
  - Configure Synergy repository
  - Code-generate repository classes and new controllers
  - Improve API help
  - Test in Postman

Configuring Routes in Controllers

Building RESTful Services with Synergy .NET

Configuring Routes in Controllers

- So far we have relied on the Web API's default routing

```
// Default Web API routing rules
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

- Alternatives are available
  - To use INSTEAD of default routine
  - To use ALONGSIDE default routing

Routing Requests to a Controller

```
{RoutePrefix("api/employee")}
;;; <summary>
;;; A Web API Controller exposing CRUD functionality for the Employee master.
;;; </summary>
public partial class EmployeeController extends ApiController
```

- {RoutePrefix} attribute adds an entry to the routing table
  - Here the default route would do the same thing!

Routing Requests to an Action

```
{Route("department/{aDepartment}")}
public method GetByDEPARTMENT, @HttpResponseMessage
    required in aDepartment, String

{Route("department/{aDepartment}/state/{aState}")}
public method GetByDEPARTMENT, @HttpResponseMessage
    required in aDepartment, String
    required in aState, String
```

- {Route} attribute determines how requests are routed to actions within a controller
- http://my.site.com/api/employee/**department/QA**
  - All employees in the QA department
- http://my.site.com/api/employee/**department/QA/state/CA**
  - All employees in the QA department who live in California

Demo 12: Configuring Routes in Controllers

- Goals
  - Make use of custom routes to identify specific functionality in our controllers
- Activities
  - Examine the code already present in EmployeeController

Dependency Injection

Building RESTful Services with Synergy .NET

Dependency Injection

- Pattern implementing **Inversion of Control** (IoC) for resolving dependencies
  - **Dependency** is an object that can be used (a **service**)
  - **Injection** is passing of a dependency to an object that requires it (client)
  - **Injector** is a class which constructs services and injects them into clients
- Goals and benefits
  - Pass dependencies to clients rather than allowing clients to build or find them
  - Decouple objects so no client must be changed because an object it depends on needs to be changed to a different one

IoC Containers

- Frameworks that facilitate DI
  - Declared as a service within the environment (e.g., Web API)
  - Manage creation, allocation, and injection of dependencies

- Some .NET IoC Containers
  - Castle Windsor http://www.castleproject.org/projects/Windsor
  - Spring.NET http://www.springframework.net
  - StructureMap http://structuremap.github.io
  - Ninject http://www.ninject.org
  - Unity https://github.com/unitycontainer/unity

IoC Containers and Dependency Lifetime

Unity supports three lifetime managers:

- **TransientLifetimeManager** (default)
  - New service instance for each request
- **ContainerControlledLifetimeManager**
  - Singleton service instance scoped to container lifetime
  - One instance application wide
- **HierarchicalLifetimeManager**
  - Singleton service instance scoped to container performing resolution (not necessarily the container where service registered)
  - In environments with session state (not Web API), a mechanism for one instance per session

Implementing Dependency Injection

- Revolves around interfaces, allowing specific implementations to be swapped out
  - Implement and configure an IoC container
  - Describe dependencies (services) via interfaces
  - Implement services in concrete classes
  - Declare services to IoC container
  - Code clients to declare dependencies

Demo 13: Dependency Injection

- Goals
  - Implement dependency injection for data repository classes

- Activities
  - Create interfaces for repository classes (CodeGen)
  - Implement interfaces
  - Create a Unity dependency resolver (CodeGen)
  - Configure the resolver in the web host app
  - Alter controllers to use DI pattern

Extending Generated Code

Building RESTful Services with Synergy .NET

Extending Generated Code

- We've used CodeGen to generate:
  - Model classes
  - Repository interfaces and classes
  - Controller classes
- Generated code supports basic CRUD operations
  - Create, read, update, and delete (by primary key)
  - Read all
  - Read by alternate key
- But what about custom, hand-crafted code?
- Templates generate PARTIAL interfaces and classes
  - Add custom functionality alongside generated code

Demo 14: Extending Generated Code

- Goals
  - Extend the generated Employee code with a custom operation
  - Get all employees born in a specific year

- Activities
  - Customize the employee repository interface
  - Customize the employee repository class
  - Customize the employee controller
  - Test in Postman

Using Custom HTTP Headers

Building RESTful Services with Synergy .NET

Using Custom HTTP Headers

- Most operations return information to clients via
  - Method return value
  - Which becomes the HTTP response body

- Additional option is to use HTTP headers
  - Simple name / value pairs
  - Use for metadata, not actual data
  - Supporting information

- Example:
  - MatchingResults: 133

Demo 15: Using Custom HTTP Headers

- Goals
  - Modify the GetBornBeforeYear operation
  - Return matching record count in an HTTP header

- Activities
  - Modify EmployeeController to return a custom HTTP header
  - Test in Postman

Self Hosting with OWIN

Building RESTful Services with Synergy .NET

Self Hosting with OWIN

- Most web services are hosted in websites, by a web server
  - IIS on Windows Server (on-premises or hosted)
  - Azure website (cloud)
- Alternate use case is “self hosting”
  - Host the web service in some other process
    - Windows service
    - Some other application
- OWIN—Open Web Interface for .NET
  - Defines standard interface between web servers and applications
- NuGet Microsoft.AspNet.WebApi.OwinSelfHost (aka Katana)
  - Self-hosting mechanism for ASP.NET Web API

Demo 16: Self Hosting with OWIN

- Goals
  - Host an instance of our Web API services outside of IIS
  - Console app

- Activities
  - Create a console application
  - Add Web API and OWIN
  - Configure Web API for self hosting
  - Start the hosting app
  - Test in Postman

Calling RESTful Services

Building RESTful Services with Synergy .NET

Calling RESTful Services

- RESTful services
  - Interact via HTTP
  - Receive and send JSON (or XML)
- Use your environment's native tools
- Traditional Synergy
  - HTTP API
  - XML API
- Synergy .NET
  - HttpClient (System.Net.Http)
  - Newtonsoft Json (from NuGet)

Demo 17: Calling RESTful Services

- Goals
  - Call a RESTful service from Synergy .NET
  - Call a RESTful service from traditional Synergy

- Activities
  - Create a Synergy .NET Console Application
  - Use HttpClient to call a service
  - Create a traditional Synergy DBR project
  - Use %HTTP_GET to call a service

Wrapping Up

Building RESTful Services with Synergy .NET

Wrapping Up

- RESTful web APIs are the way to go
  - Simple, open, and accessible
- ASP.NET Web API
  - State of the art framework for building .NET RESTful APIs
  - Easy to implement
  - Fast and scales well
- .NET Core support
  - ASP.NET Core Web API—multi-platform
  - Even leaner and meaner than the “full fat” version
  - Not identical, but high level of compatibility

We Only Dealt with the Basics

- ASP.NET Web API is an extensive framework
  - Entity framework support
  - OData support
  - BSON support (Binary JSON)
  - Customizable parameter binding
  - Authentication and authorization support
    - Identity framework
  - And more…

- Getting started

Education Resources

- FREE
  - ASP.NET Web API Overview and Getting Started Videos
    - https://docs.microsoft.com/en-us/aspnet/web-api/videos/getting-started
  - Introduction to the ASP.NET Web API
    - https://channel9.msdn.com/Events/aspConf/aspConf/Introduction-to-the-ASP-NET-Web-API

- PLURALSIGHT
  - Implementing an API in ASP.NET Web API
    - https://app.pluralsight.com/library/courses/implementing-restful-aspdotnet-web-api
  - Building and Securing a RESTful API for Multiple Clients in ASP.NET
    - https://app.pluralsight.com/library/courses/building-securing-restful-api-aspdotnet

Questions?

Building RESTful Services with Synergy .NET