Wintellect
Know how.

Best Practices in Modern Web Development

Best Practices in Modern Web Development

PUBLISHED BY
Wintellect, LLC
990 Hammond Drive
Bldg. One, Suite 302
Atlanta, GA 30328

Table of Contents

Part 1 – ASP.NET Web API Hosting, Client, Async

Part 2 – Serialization and Model Binding

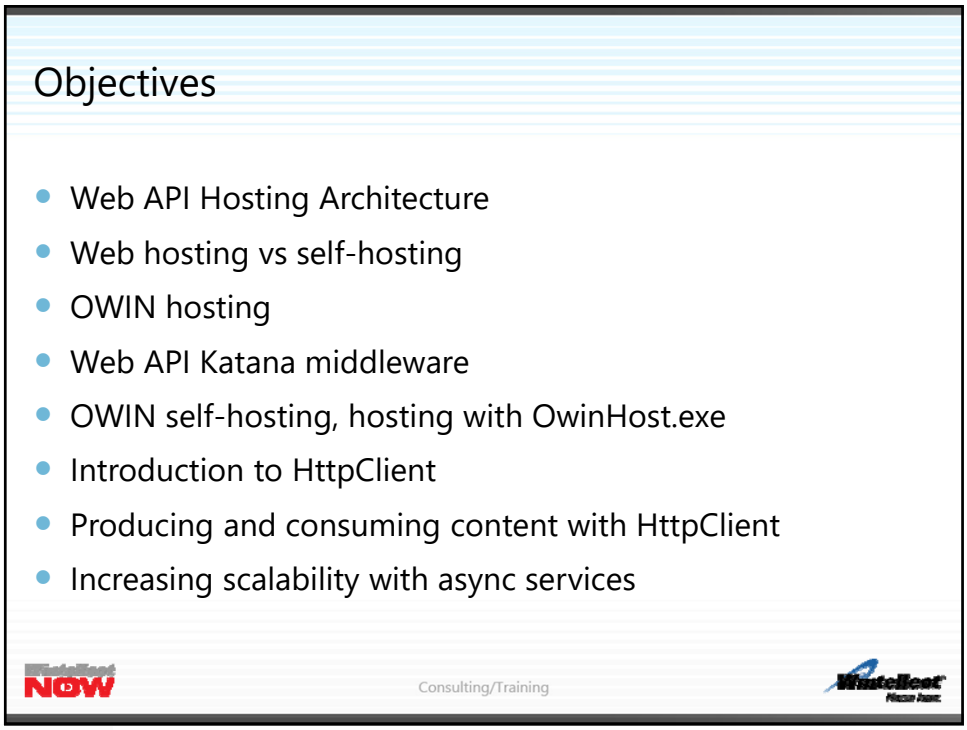
Part 3 – Validation and Testing with ASP.NET Web API

Part 4 – Entity Framework in N-Tier Applications

Part 5 – Repository and Unit of Work Patterns

Part 6 – Introduction to Web API in ASP.NET 5

Notes Pages



Wintellect
Know how.

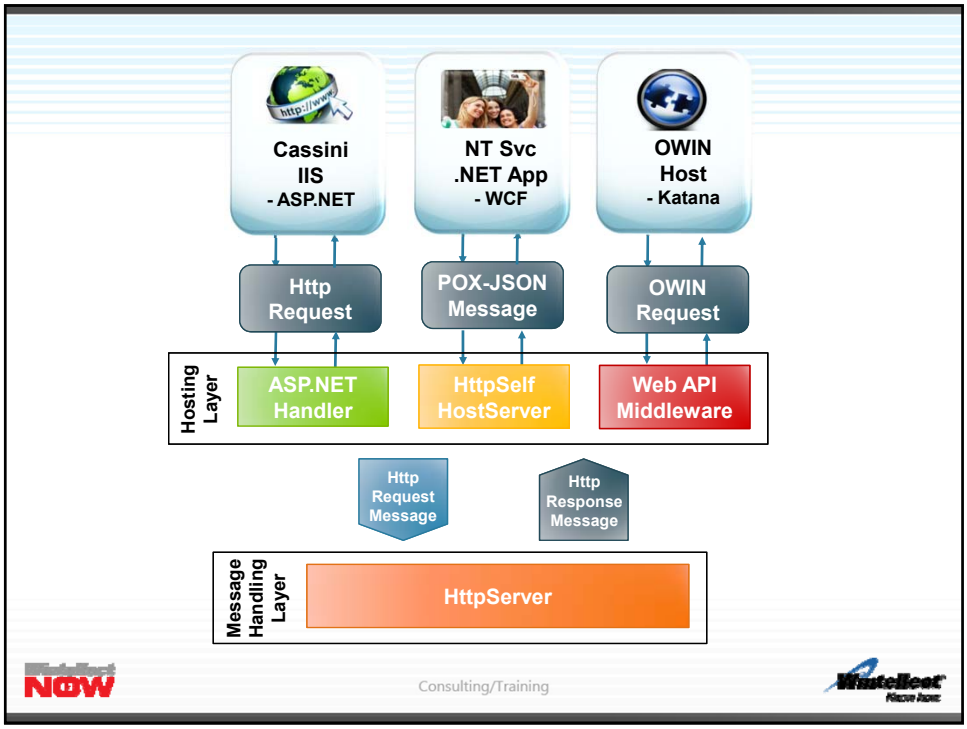
Web API Hosting Architecture

Consulting/Training

Web API Hosting Architecture

- Web API is design to be **host independent**
  - Hosting layer acts as a **bridge** to an external host
  - HTTP request is transformed from its *native representation* into an **HttpRequestMessage**
  - **HttpResponseMessage** is transformed back into the *native representation* of an HTTP response

Consulting/Training

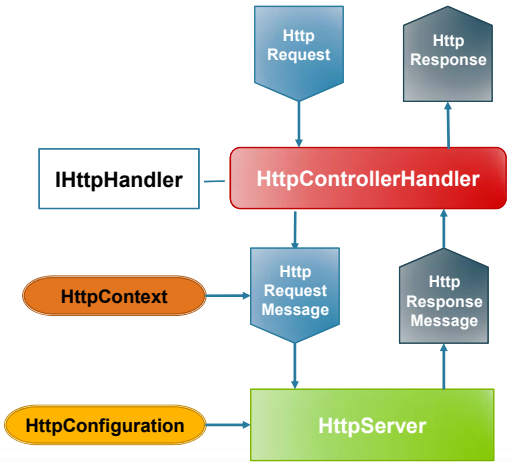


Web Hosting: ASP.NET Handler

- Handles incoming HttpRequests
  - Converts HttpRequest to **HttpRequestMessage**
  - Converts **HttpResponseMessage** to HttpResponse
  - Stores **HttpContext** and as request property
  - Creates **HttpServer** on first request
    - HttpConfiguration is passed to HttpServer

Consulting/Training

Web Hosting: ASP.NET Handler (cont.)



Web Hosting: HttpConfiguration

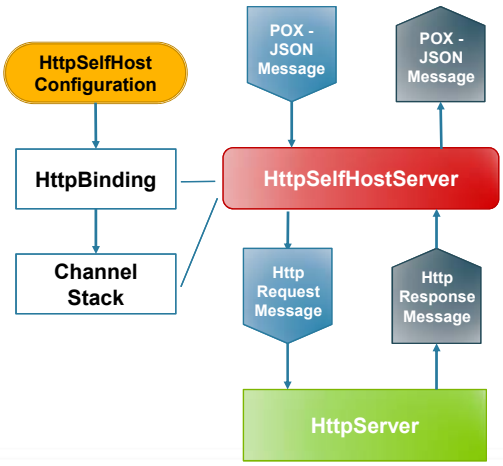
- Drives Web API runtime configuration
  - Available on **GlobalConfiguration.Configuration** and as a property on **HttpRequestMessage**

Property	Description
DependencyResolver	Resolution of dependencies by an IoC container
Filters	Collection of global filters
Formatters	Media type formatters used for content negotiation
IncludeErrorDetailPolicy	Sets policy on error reporting: Always, Never, LocalOnly
Initializer	Initialization method - usually WebApiConfig.Register
MessageHandlers	Ordered list of handlers for the message handling layer
Routes	List of Web API routes

Self Hosting: HttpSelfHostServer

- Builds the WCF channel stack
  - **HttpSelfHostConfiguration** drives configuration of **HttpBinding**
    - Determines components of the **WCF Channel Stack**
- Listens for incoming messages
  - Incoming POX-JSON message converted to **HttpRequestMessage**
  - **HttpResponseMessage** converted to POX-JSON message

Self Hosting: HttpSelfHostServer (cont.)



Self Hosting: HttpSelfHostConfiguration

- Drives configuration of **HttpBinding** and WCF channel stack
  - Adds WCF-specific properties to those in the **HttpConfiguration** base class

Property	Description
ClientCredentialType	Transport-level credentials which client must supply
MaxBufferSize	Number of bytes in the buffer – default is 64K
MaxConcurrentRequests	Maximum concurrent requests – default is 100
MaxReceivedMessageSize	Maximum size of incoming messages – default 64K
ReceiveTimeout	Message receive timeout – default is 10 minutes
SendTimeout	Message send timeout – default is 1 minute
TransferMode	Buffered, Streamed, Streamed Request / Response

Hosting with OWIN & Katana

Motivation: Host Decoupling

- After selecting a hosting option, you are forever **coupled** to it
  - Self hosting: configure security in the **WCF channel stack**
  - Web hosting: configure security in the **ASP.NET pipeline**
- Can make it difficult to **switch** hosting options
  - Cross-cutting concerns should be **completely decoupled** from the host (for example: security, diagnostics, etc)

Solution: The OWIN Specification

OWIN

Open Web Interface for .NET

OWIN defines a **standard interface** between .NET web servers and web applications.

The goal of the OWIN interface is to **decouple** server and application.

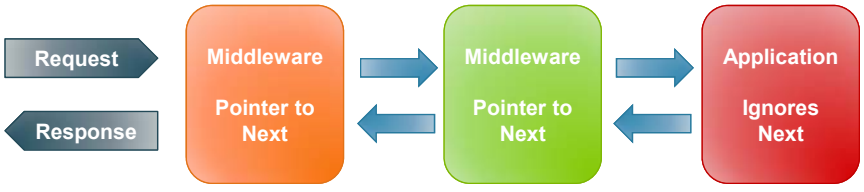
http://owin.org

Solution: Project Katana

- Microsoft's open-source implementation of the OWIN specification
  - Project home: **http://projectkatana.codeplex.com**
  - Released as a set of **NuGet** packages

OWIN Pipeline

- The pipeline is configured on app startup
  - Each component has a pointer to the **next component**
  - Components can **ignore** the next one and **return a response**
  - Requests and responses are **framework-agnostic**



Do-It-Yourself Middleware – Non-terminating

```
public class LoggingComponent {

    // Store pointer to next component's Invoke method
    Func<IDictionary<string, object>, Task> _next;
    public LoggingComponent(Func<IDictionary<string, object>, Task>
next) {
        _next = next; }

    public async Task Invoke(IDictionary<string, object>
environment) {

        // Log request and response info
        Console.WriteLine(environment["owin.RequestPath"]);
        await _next(environment);   // Invoke next component
        Console.WriteLine(environment["owin.ResponseStatusCode"]);
    }
}
```


Do-It-Yourself Middleware – Terminating

```
public class GreetingComponent {

    // Need this ctor or we'll get a MissingMethodException
    Func<IDictionary<string, object>, Task> _next;
    public GreetingComponent(Func<IDictionary<string, object>, Task>
next) {
        _next = next; }

    public Task Invoke(IDictionary<string, object> environment) {

        // Get response stream and write to it
        var response = environment["owin.ResponseBody"] as Stream;
        using (var writer = new StreamWriter(response))
            return writer.WriteAsync("Hello!");
    }
}
```

Configuring the Pipeline

- Add a **Startup** class
  - Add a **Configuration** method that accepts an **IAppBuilder** parameter
  - Discovered either by convention, configuration, or in code

```
// Configure middleware components in the Katana pipeline
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.Use<LoggingComponent>();
        app.Use<GreetingComponent>();
    }
}
```

Consulting/Training

Inserting Web API Middleware

- Call IAppBuilder.**UseWebApi** extension method
  - Place terminating middleware (UseWebApi, MVC, SignalR, etc) after non-terminating middleware (security, logging, etc)

```
public void Configuration(IAppBuilder app) {
    app.Use<LoggingComponent>(); // Non-terminating middleware

    var config = new HttpConfiguration(); // Configure routing
    config.Routes.MapHttpRoute("DefaultApi",
        "api/{controller}/{id}",
        new { id = RouteParameter.Optional });

    app.UseWebApi(config); }  // Terminating middleware
```

Consulting/Training

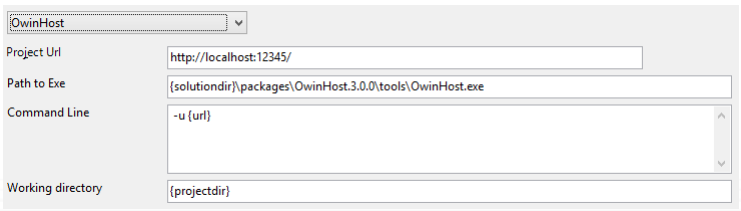
Self Hosting with Katana

- Create a .NET application – Console, WPF, NT Service, etc
  - Add **Microsoft.Owin.SelfHost** NuGet package
  - Add **Microsoft.AspNet.WebApi.OwinSelfHost** NuGet package
- In Main, call **WebApp.Start<Startup>**
  - Pass a **URI** for the application base address
  - Call **Dispose** to clean up when app shuts down (can place in **using** block)

```
static void Main(string[] args) {
    using (WebApp.Start<Startup>("http://localhost:12345/")) {
        Console.WriteLine("Service is running ...");
        Console.ReadLine();
    } }
```


Hosting with OwinHost.exe

- Create an "Empty" Web project in Visual Studio 2013
  - Add **Microsoft.AspNet.WebApi.Owin** NuGet package
  - Add startup and controller classes – there's an "**OWIN Startup class**" template
  - Add **OwinHost** NuGet package
    - On the **Web** tab of the project properties page, select **OwinHost** as the web server
  - **Press F5**, and you're hosting a Web API service *without IIS or ASP.NET!*



Using HttpClient

Introducing HttpClient

- Replaces HttpWebRequest and WebClient
  - More closely aligned with HTTP
- Portable across **all** .NET platforms
  - .NET 4.x (WPF, SL, WCF, ASP.NET MVC, Web API, etc)
  - Windows Store, Windows Phone, as well as iOS and Android (via Xamarin)
- Uses **Task-based Asynchronous Pattern** (TAP)
  - Supports C# async and await
- Adheres to HTTP Programming Model
  - Same **message handling pipeline** on both client and server

HttpClient Basic Usage: GET Request

- HttpClient has **GetXxxx** methods
  - Each accepts a **Uri** and returns a **Task**
  - Variants include **GetString**, GetStream, GetByteArray
    - Will throw an **Exception** if response StatusCode does not indicate success
  - **GetAsync** returns an **HttpResponseMessage**
    - Useful for checking response properties

```
// Basic usage: GET request
async Task<string> GetGreeting(int id) {
    var client = new HttpClient();
    return await client.GetStringAsync
        ("http://localhost:12345/api/Values/" + id);
}
```

Consulting/Training

HttpClient Basic Usage: POST Request

- HttpClient has **PostAsJsonAsync** and **PostAsXmlAsync**
  - Will format request content as either JSON or XML
  - Returns an **HttpResponseMessage**
    - Useful for checking response properties
    - Call **EnsureSuccesstatusCode** to throw an Exception in case of errors

```
// Basic usage: POST request
static async Task PostGreeting(string greeting) {
    var client = new HttpClient();
    HttpResponseMessage response = await
client.PostAsJsonAsync
        ("http://localhost:12345/api/Values", greeting);
    response.EnsureSuccessStatusCode();
}
```

Consulting/Training

HTTP Headers API

- Headers can be added to these:
  - HttpRequestMessage, HttpResponseMessage, HttpContent

HttpRequest Headers	HttpContent Headers	HttpResponse Headers
• Accept	• Allow	• AcceptRanges
• Authorization	• ContentDisposition	• Age
• CacheControl	• ContentEncoding	• CacheControl
• Connection	• ContentLanguage	• Date
• Expect	• ContentLength	• ETag
• Host	• ContentLocation	• Location
• Pragma	• ContentMD5	• Server
• Range	• ContentRange	• Trailer
• UserAgent	• ContentType	• TransferEncoding
• Via	• Expires	• Upgrade

Consulting/Training

Response Status Codes

- Status codes do not trigger exceptions
  - It's up to the application to handle response codes
    - For example, app may **re-try** a request in response to **503 Service Unavailable**
- Can check for status codes besides 2xx
  - **IsSuccessStatusCode** will return false
  - **EnsureSuccessStatusCode** will throw an exception

```
HttpResponseMessage response = await
client.SendAsync(request);
response.EnsureSuccessStatusCode(); // Throws if not 2xx
return await response.Content.ReadAsStringAsync();
```

Consulting/Training

Producing Message Content: HttpContent Classes

ObjectContent **ObjectContent<T>**	• Objects that are serialized using media type formatters
StreamContent **PushStreamContent**	• Content available as a stream (for ex, files) • Content produced by a stream writer
MultipartFormContent	• HTML form data with multi-part MIME content
StringContent **ByteArrayContent** **FormUrlEncodedContent**	• Decoded message content • Buffered copy of message content • Name / value pairs from HTML forms

Consuming Message Content: HttpContent

- Message content consumed using HttpContent **methods**
  - *Pulled* from a stream or *pushed* to a stream
  - Read into a **string** or **byte array**
  - Deserialized into **CLR objects**

HttpClient Method	Description
Task<Stream> **ReadAsStreamAsync**()	Returns a stream to pull content from
Task **CopyToAsync**(Stream)	Push message content into a stream
Task<byte[]> **ReadAsByteArrayAsync**()	Read message content into a byte array
Task<string> **ReadAsStringAsync**()	Read message content as plain text
Task<T> **ReadAsAsync<T>**()	Deserialize message content

Async Actions with Tasks

- Write **async** methods returning **Task<IHttpActionResult>**
  - Use "**await**" keyword with async I/O methods that return Task or Task<T>
  - Entity Framework v6 or greater provides a Task-based API for async

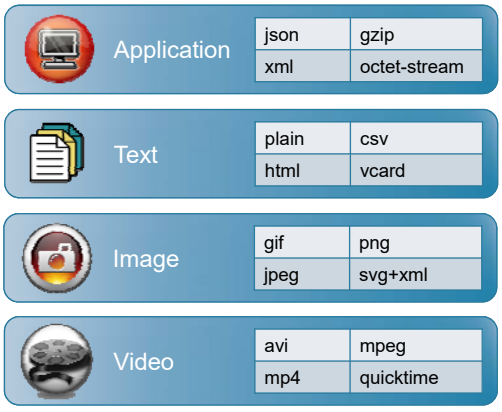
```
public class CustomerController : ApiController {
   private readonly Northwind _dbContext = new Northwind();

   [ResponseType(typeof(IEnumerable<Customer>))]
   public async Task<IHttpActionResult> GetCustomers() {
    IEnumerable<Customer> customers = await _dbContext.Customers
       .ToListAsync();
       return Ok(customers);
   }
```


Wintellect
Know how.

Serialization

Media Types Revisited

- Media type identifies **format** of data item
  - Consists of **two-part** identifier
- Media **top-level** type
  - General type information and type handling rules
  - For ex: **text, image, application, video, multipart**
- Media **sub-type**
  - Specific data format and handling
  - For example: **application/xml, text/html, image/jpeg, video/avi**
  - Can include variants with different formats
    For example: **application/atom+xml**



Media Type Formatters

- Handle **transformation** of a data format to .NET types
  - Usually depends on one or more **serializers**
- Abstract class **MediaTypeFormatter**
  - Properties: SupportedMediaTypes, SupportedEncodings, CanReadType, CanWriteType
  - Methods: ReadFromStreamAsync, WriteToStreamAsync
- Web API ships with some **default** formatters
  - Json, Bson – uses Json.Net
  - Xml – uses DataContract or XmlSerializer
  - Form Url Encoded – HTML form submission

HttpConfiguration Formatters Property

- HttpConfiguration has a **Formatters** property
  - Each of the **default formatters** is exposed as a strongly-typed property
    - **Json**Formatter, **Xml**Formatter, **FormUrlEncoded**Formatter
  - Can **add or remove** individual formatters

```
public static class WebApiConfig {
    public static void Register(HttpConfiguration config) {

        // Reference Json formatter
        JsonMediaTypeFormatter jsonFormatter = config.Formatters.JsonFormatter;

        // Add custom formatter
        config.Formatters.Add(new CustomMediaTypeFormatter());
```

Json.Net: Serializer Configuration

- JsonFormatter has a **SerializerSettings** property
  - Instructions for **date formatting**
  - How to handle **missing members** and **null values**
  - Casing of JSON property names
    - To specify camelCasing, supply a **ContractResolver**

```
jsonFormatter.SerializerSettings. DateTimeZoneHandling =
    DateTimeZoneHandling.Utc; // Dates will omit time zone offset

jsonFormatter.SerializerSettings.MissingMemberHandling = MissingMemberHandling.Error;

jsonFormatter.SerializerSettings.ContractResolver =
    new CamelCasePropertyNamesContractResolver(); // Json properties camel cased
```


Json.Net: Serialization of Properties

- All public properties are serialized
  - **Read-only** properties are serialized
  - Exclude specific properties by attaching **[JsonIgnore]** attribute

```
public class Product {
    // Will be serialized
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public Category Category { get { return _category; } }

    [JsonIgnore] // Will not be serialized
    public int CategoryId { get; set; }
}
```


Wintellect Know how.

Json.Net: Cyclical References

- By default Json.Net serializer writes all objects as **values**
  - Multiple object references result in **multiple instances**
  - Serializer will throw an **exception** when it detects **cycles** in an object graph
    - For example, Product has a Category property, and Category has a Products property
- Object references can be preserved
  - Set SerializerSettings.PreserveReferencesHandling = **PreserveReferencesHandling.All**
  - Attach **[JsonObject(IsReference = true)]** attribute to specific classes

```
// Object references preserved – no exception if cycles are detected
[JsonObject(IsReference = true)]
public class Product { ... }
```


Xml Formatter Options

- By default the **DataContractSerializer** is used
  - Is generally *faster* than XmlSerializer
  - Can handle POCO classes (without any attributes)
- Can elect to use **XmlSerializer** instead
  - May desire *greater control* over XML format (for example, using XML attributes), or working with *legacy model classes* that rely on XmlSerializer
  - Set **XmlFormatter.UseXmlSerializer** = **true**

```
// Use XmlSerializer instead of DataContractSerializer
config.Formatters.XmlFormatter.UseXmlSerializer = true;
```

DataContract: Without Attributes

- If [DataContract] attribute is **not applied**, POCO classes may be used
  - All **read-write** properties are serialized
  - **Read-only** properties are not serialized

```
// Without [DataContract] – all public properties are serialized
public class Product {

    // Read-write properties will be serialized
    public int ProductId { get; set; }
    public string ProductName { get; set; }

    // Read-only properties will not be serialized
    public Category Category { get { return _category; } }
```


DataContract: With Attributes

- If **[DataContract]** attribute applied to class, property serialization is **opt-in**
  - Only properties adorned with **[DataMember]** are serialized
  - **Private fields** with [DataMember] are also serialized

```
[DataContract] // Only properties and fields with [DataMember] are serialized
public class Product {

    [DataMember] // Will be serialized
    public int ProductId { get; set; }

    [DataMember] // Will be serialized
    private string _productName;

    // Will not be serialized
    public decimal UnitPrice { get; set; }
```

DataContract: Cyclical References

- By default DataContract serializer writes objects as **values**
  - Multiple object references result in **multiple instances**
  - Serializer will throw an **exception** when it detects **cycles** in an object graph
- Object references can be preserved
  - Set XmlFormatter = **new** DataContractSerializer(**preserveObjectReferences : true**)
  - Attach **[DataContract(IsReference = true)]** attribute to classes

```
// Object references preserved – no exception if cycles are detected
[DataContract(IsReference = true)]
public class Product { ... }
```


Binary Encoded JSON – BSON

- Can enable BSON on the server
  - Less compact than JSON for text, but more efficient for **binary formats** (they're not base64 encoded)
  - Supports handling cyclical references – programmatically or with attributes

```
config.Formatters.Add(new BsonMediaTypeFormatter());
```

- Use BSON formatter on the client
  - Set **Accept** and/or **ContentType** headers to "application/bson"

```
client.DefaultRequestHeaders.Add("Accept", "application/bson"); // Request
var persons = await response.Content.ReadAsAsync<IEnumerable<Person>>
    (new MediaTypeFormatter[] { new BsonMediaTypeFormatter() }); // Content
```

Protocol Buffers – Protobuf

- Protobuf is Google's fast, compact serializer – outperforms Json.Net
  - Install **WebApiContrib.Formatting.ProtoBuf** NuGet package
  - Use either DataContract or ProtoContract and add property or field attributes (opt-in)
  - Handle cyclical references with ProtoContract attribute or in code: **AsReferenceDefault = true**

```
config.Formatters.Add(new ProtoBufFormatter());
```

  - To avoid decorating classes with attributes (POCO), configure types **in code**

```
MetaType personMeta = ProtoBufFormatter.Model.Add(typeof(Person), false);
personMeta.Add(1, "Name").Add(2, "Age"); // Include properties
personMeta.AsReferenceDefault = true; // Handle cyclical references
```

  - On the *client* use **formatter**, set Accept and/or ContentType headers to "**application/x-protobuf**"

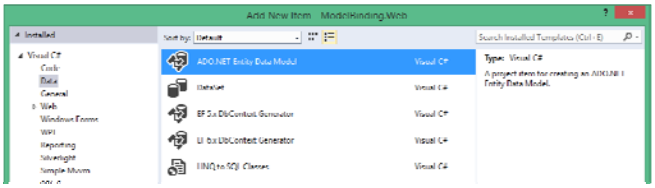
Custom Media Type Formatters

- Derive from MediaTypeFormatter (async) or BufferedMediaTypeFormatter (sync)
  - In constructor add **supported media types**
    - Optionally add support for different character encodings (for ex, UTF-8, ISO 8859-1, etc)
  - Override CanReadType, CanWriteType (return true)
  - Override **WriteToStream**(Async), **ReadFromStream**(Async) for serialization and deserialization

```
public class CsvMediaTypeFormatter: MediaTypeFormatter {
    public CsvMediaTypeFormatter() { // Constructor
        SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv")); }
  // Override read and write methods
  public override Task WriteToStreamAsync(Type type, object val, Stream stream ...
  public override Task<object> ReadFromStream(Type type, object val, Stream stream ...
```

Code Generation Tools

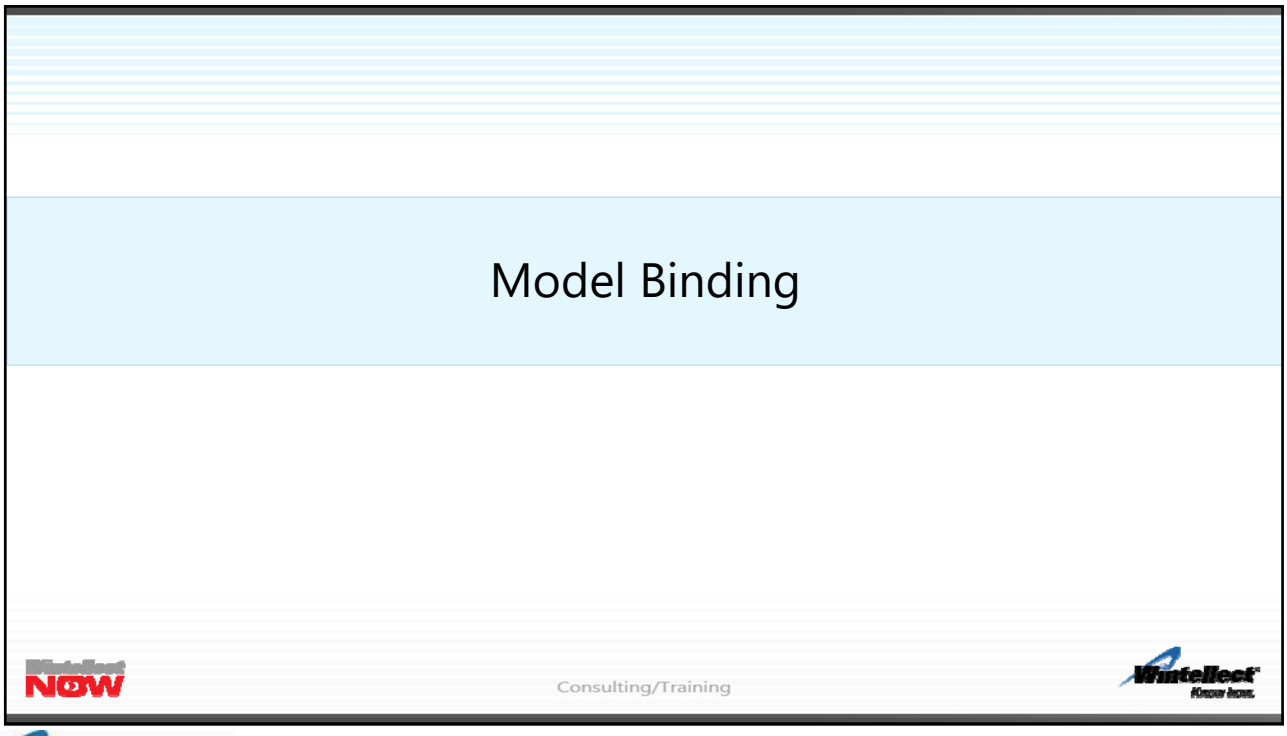
- What happened to "Add Service Reference"?
  - Web API's do not expose **metadata** (WSDL) for code-generation tools to use
  - RESTful approach favors embedded hyperlinks over strict contracts
- Entity Framework tooling offers another approach
  - Generate model classes based on an **Entity Data Model**, which is a *conceptual view* of the database
  - Add an EDM, then select either "EF Designer from Database" or "Code First from Database"
  - Code generation can be customized with **T4 templates**
    - Install **EntityFramework.CodeTemplates.CSharp** NuGet package



Entity Framework Gotchas

- Entity Framework generates **dynamic runtime proxies**
  - Used to support features such as Lazy Loading
    - Usually not required for n-tier scenarios
  - Enabled when all properties are defined as **virtual**
    - Should be explicitly **disabled** because runtime proxies are not serializable

```
public partial class Northwind : DbContext {
   public Northwind() : base("name=Northwind") {

       // Disable dynamic proxies, which are not serializable
       Configuration.ProxyCreationEnabled = false;
   }
```



A World without Model Binding

- Http requests are composed of many **different parts**
  - URI, Headers, Cookies, Body
    - How do you map message parts to **method parameters**?

```
public async void Post(HttpRequestMessage request) {
    // Get Id from query string
    int id = int.Parse(request.RequestUri.ParseQueryString().Get("id"));

    // Get Name and Age from url-encoded body
    NameValueCollection values = await request.Content.ReadAsFormDataAsync();
    var person = new Person { Id = id, Name = values["Name"],
        Age = int.Parse(values["Age"]) }; }
```

  - Code is *tightly coupled* to the URI and message format!

Model Binding to the Rescue

- **Model Binder** maps components of an HTTP message to **method parameters**
  - **Value Provider** exposes parts of the message to the Model Binder
    - Includes **key/value pairs**, such as headers, url segments, query strings, form url-encoded body

```
Request Uri: http://.../Person?id=1
Method: POST
Accept: application/x-www-form-urlencoded
Body: Name=Peter&Age=20
```

Value Provider

Uri: Id | Body: Name | Body: Age

```
public void Post(int id, Person person)
```


Model Binder

Default Model Binding

- By default **URI** segments and query strings are mapped to "simple" types
  - Include .NET primitive types (**string, int, bool, double**, etc), plus **TimeSpan, DateTime, Guid**
- Message **bodies** are serialized to .NET types using **media type formatters**

```
Request Uri: http://.../Person/1
Method: POST
Content-Type: application/json
Body: {"Name":"Peter","Age":20}
```

```
public void Post(int id, Person value) { // Multiple complex types not allowed
    var person = new Person { Id = id,    // id set by model binder
        Name = value.Name,                // value set by media type formatter
        Age = value.Age }; }
```

Model Binding Attributes: FromBody

- Use **[FromBody]** on a parameter to map **simple type** to a request **body**
  - Without [FromBody] default model binding would map simple type to **URI** segment or query string
  - **Media type formatter** is selected based on content negotiation using **Content-Type** header

```
Request Uri: http://.../Greeting
Method: POST
Content-Type: application/xml
Body: <string xmlns=
"http://schemas.microsoft.com/2003/10/
Serialization/">Hello</string>
```

```
Request Uri: http://.../Greeting
Method: POST
Content-Type: application/json
Body: "Hello"
```

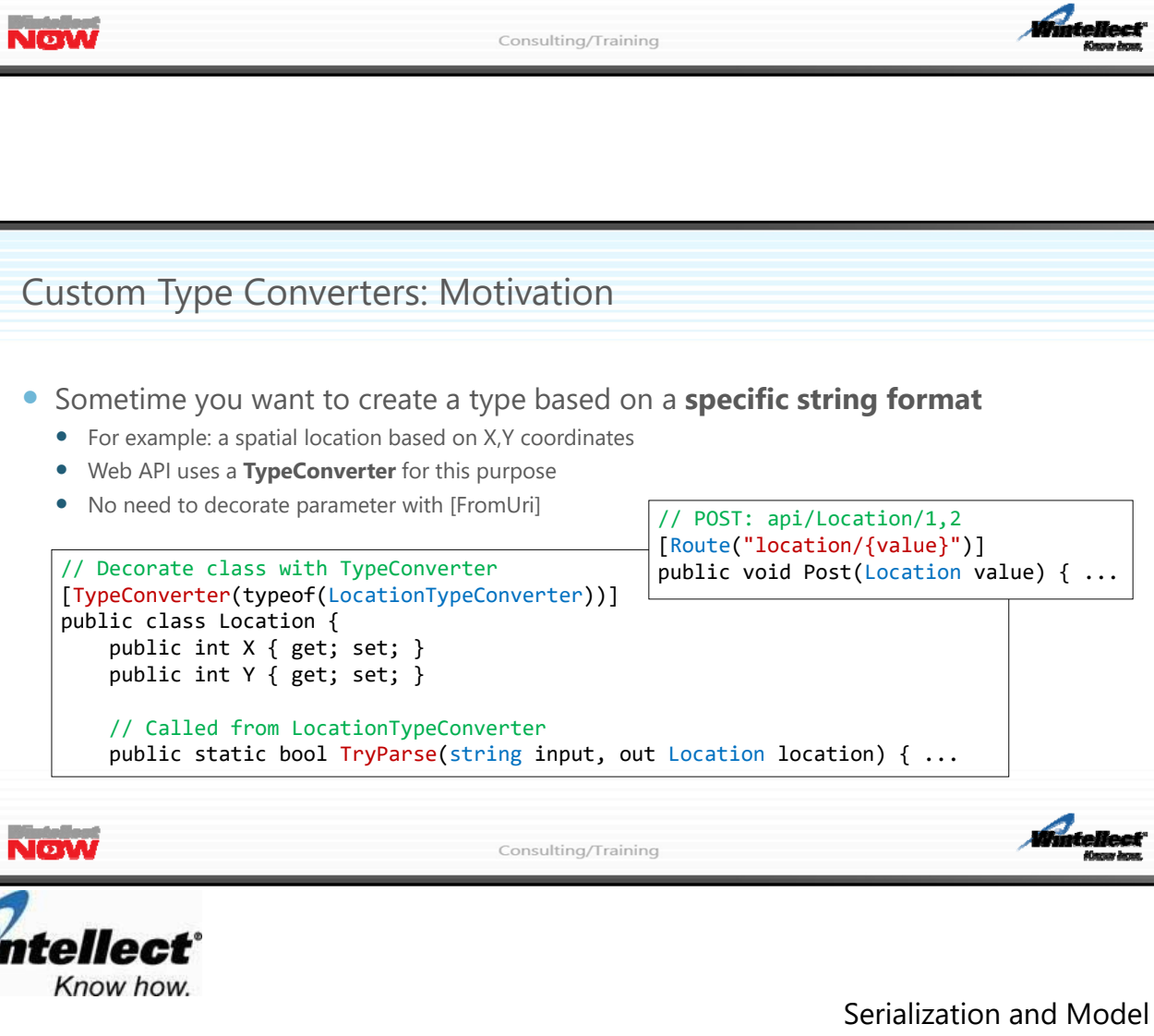
```
public void Post([FromBody]string greeting) { ...
```

Model Binding Attributes: FromUri

- Use **[FromUri]** to map a **complex type** to a **URI** segment or query string
  - Without [FromUri] default model binding would map complex type to a **request body**
  - **Model binding** is used rather than a media type formatter
  - Can implement custom **Type Converter** to perform conversion from string to complex type

```
Request Uri: http://.../Person/1?Name=Peter&Age=20
Method: POST
```

```
public void Post([FromUri]Person value) { ...
```



Custom Type Converters: Motivation

- Sometime you want to create a type based on a **specific string format**
  - For example: a spatial location based on X,Y coordinates
  - Web API uses a **TypeConverter** for this purpose
  - No need to decorate parameter with [FromUri]

```
// POST: api/Location/1,2
[Route("location/{value}")]
public void Post(Location value) { ...
```

```
// Decorate class with TypeConverter
[TypeConverter(typeof(LocationTypeConverter))]
public class Location {
    public int X { get; set; }
    public int Y { get; set; }

    // Called from LocationTypeConverter
    public static bool TryParse(string input, out Location location) { ...
```

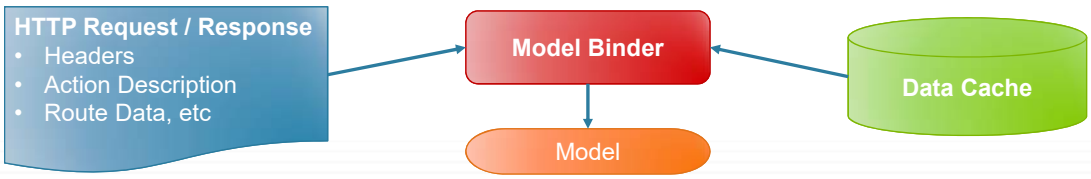
Custom Type Converters: Implementation

- Inherit from TypeConverter in System.ComponentModel
  - Override **CanConvertFrom**, **ConvertFrom** methods

```
public class LocationTypeConverter : TypeConverter {

    public override bool CanConvertFrom(ITypeDescriptorContext context,
        Type sourceType) { // Return true if converting from a string
        if (sourceType == typeof(string)) return true; return false; }

    public override object ConvertFrom(ITypeDescriptorContext context,
        CultureInfo culture, object value) { var input = value as string;
        if (input != null) { Location location; // Parse string to create Location
            if (Location.TryParse(input, out location)) return location; }
        return base.ConvertFrom(context, culture, value); } }
```


Custom Model Binders: Motivation

- Default model binding and type converters only provide values from the **URI**
  - What if you wanted to perform model binding based on *other parts* of an **HTTP request**?
  - What if you wanted to *look up items* in a **cache**?
- Creating a custom model binder offers a **more flexible** approach
  - You will have access to all the **details** of the current HTTP request or response
  - Allows you to go *beyond basic type conversion*



Custom Model Binders: Implementation

- Implement **IModelBinder** interface with **BindModel** method
  - HttpActionContext provides **HTTP request / response** information
  - ModelBindingContext exposes a **value provider** and model binding information

```
public class LocationModelBinder : IModelBinder {
    public bool BindModel(HttpActionContext actionContext,
        ModelBindingContext bindingContext) {

        // Get input from URI segment or query string
        var input = bindingContext.ValueProvider.GetValue(bindingContext.ModelName);

        // Look up default locations in a data cache
        if (LocationsCache.TryGetValue(input.RawValue, out location)) {
            bindingContext.Model = location; return true; } ...
```

Custom Model Binders: Usage

- Decorate model class with **[ModelBinder]** attribute
  - Can also apply attribute to **specific parameter** in a controller action
  - Or add a model binder provider to **HttpConfiguration**
  - Also possible to replace **default value provider** with a custom **IValueProvider**
    - Expose other parts of HTTP message in a *reusable manner*

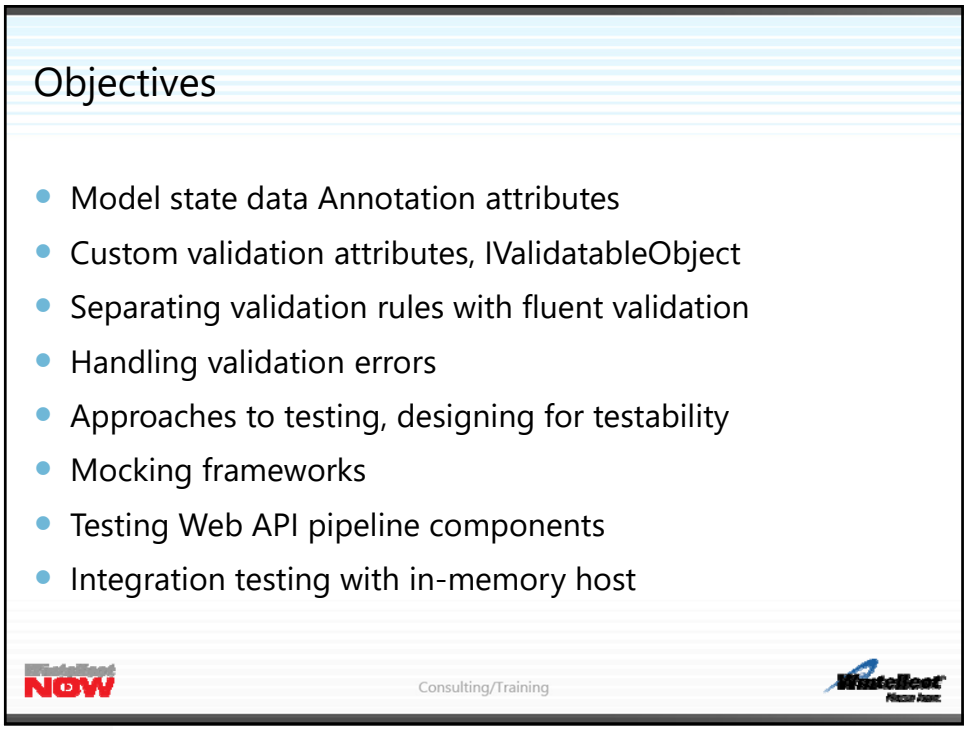
```
// POST: api/Location?value=top-right
public void Post(Location value) { ...
```

```
// Decorate class with ModelBinder
[ModelBinder(typeof(LocationModelBinder))]
public class Location {
    public int X { get; set; }
    public int Y { get; set; }
}
```

Conclusion

- Default JSON and XML **serializers** can be customized
  - **SerializerSettings** property used to configure Json.Net
  - XmlFormatter can be configured to use **XmlSerializer** instead of DataContractSerializer
  - **Cyclical references** can be handled either with attributes or in code (preferred)
  - For best performance, **Protobuf** formatter can be added
- Generate **model classes** from an Entity Data Model
  - Code generation can be customized via **T4 templates**
- Create **model binders** to set model class properties from HTTP request
  - Can map URI segments, query strings, headers, etc, and look up entries from a data cache

Wintellect Know how.



Data Validation

Validating Input

- Always **validate** client input
  - Invalid input can pose a **security risk**
- Validation is performed using **Data Annotations**
  - Rules defined as **attributes** placed on properties
- Custom validation performed by implementing **IValidatableObject**
  - Supports entity-level validation
- Can **separate** validation rules from model classes
  - FluentValidation open-source library enables this

Model State

- After validation, the **ModelState** is set on ApiController
  - Has a dictionary of **model errors**
  - If there are errors, the **IsValid** property is set to false
- If ModelState is invalid, return 400 Bad Request status code
  - Pass ModelState to the **BadRequest method**
  - Model errors are serialized to the **message body**

```
public IHttpActionResult PostProduct(Product product) {

    // Return bad request status, model errors in message body
    if (!ModelState.IsValid) return BadRequest(ModelState);
}
```

Consulting/Training

Data Annotation Attributes

- Apply Data Annotation attributes to **model properties**
  - Can set an optional **ErrorMessage**, which can be parameterized and stored in a resource file

Attribute	Parameters	Example / Notes
Required	AllowEmptyStrings	Required
StringLength	Minimum, Maximum	StringLength(40)
Range	Min, Max, Type	Type implementing IComparable
EmailAddress	None	EmailAddress
MaxLength, MinLength	Length	Constrain length of an array
RegularExpression	Pattern	RegularExpression ("^(\+91[\-\s]?)\d{10}$")

Consulting/Training

Under and Over Posting

- Under-Posting is when client **leaves out** fields in a request
  - Value types are set to their default value
  - Prevent by using **[Required]** attribute with **nullable value types**
- Over-Posting is when client sends **more data** than expected
  - Json and Xml formatters *ignore* properties **not present** on a model
  - Don't include properties not intended to be set by client input

```
public class BlogComment {
    public string Body { get; set; }
    [Required]
    public int? Rating { get; set; } // Required, can be zero
    public bool Approved { get; set; } }  // Could be over-posted
```

Consulting/Training

Custom Validation Attributes

- Apply custom validation logic
  - Extend **ValidationAttribute** and override **IsValid** method to return **ValidationResult**
  - Set **ErrorMessageString** and override **FormatErrorMessage** to format error message
    - **ValidationContext** includes relevant info such as object instance, object type and member name

```
public class BlogComment {
    public string Body { get; set; }
    [Required]
    public int? Rating { get; set; } // Required, can be zero
    public bool Approved { get; set; } }  // Could be over-posted
```

Consulting/Training

Implementing IValidatableObject

- Implement IValidatableObject for validating specific types
  - Validate method accepts a **ValidationContext** and returns **IEnumerable<ValidationResult>**
    - Use C# **yield return** to return one or more validation errors
    - Useful for *cross-property validation* – for ex, one property greater than another

```
public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext) {

    // Perform validation using multiple properties
    if (CategoryId.GetValueOrDefault() == 1
        && UnitPrice.GetValueOrDefault() > 100)
        // Repeat yield return for multiple validation errors
        yield return new ValidationResult
            ("Beverages cannot exceed Price of 100");
    yield return ValidationResult.Success; }
```


Handling Validation Errors

- Checking ModelState in each controller action is a "code smell"
  - Violates the **DRY principle** *– do not repeat yourself!*
  - Cleaner to handle validation errors centrally
- Instead create an **action filter** to check ModelState
  - Processed before controller action is invoked

Validation Action Filter

- Inherit from ActionFilterAttribute
  - Override **OnActionExecuting** and set **HttpActionContext.Response**
  - Can place attribute on specific controllers or actions
  - Or add it to the **HttpConfiguration.Filters** collection during configuration

```
public class ValidateModelAttribute : ActionFilterAttribute {
    public override void OnActionExecuting
        (HttpActionContext actionContext) {

        if (!actionContext.ModelState.IsValid)
          // Set action context response to Bad Request
          actionContext.Response = actionContext.Request
            .CreateErrorResponse(HttpStatusCode.BadRequest,
                actionContext.ModelState);
    } }
```

Consulting/Training

Separating Validation Rules

- FluentValidation is an open-source lib for validation rules
  - Uses a fluent API placed in a **separate class**
  - Supports per-property and **cross-property** validation and **async** validation
  - Install the **FluentValidation.WebApi** NuGet package

```
public class ProductValidator : AbstractValidator<Product> {
    public ProductValidator() {

        // ProductName required, max length of 40 characters
        RuleFor(x => x.ProductName).NotEmpty().Length(0, 40);

        // Unit price must be between zero and 200
        RuleFor(x => x.UnitPrice).
            GreaterThanOrEqualTo(0).LessThanOrEqualTo(200);} } }
```

Consulting/Training

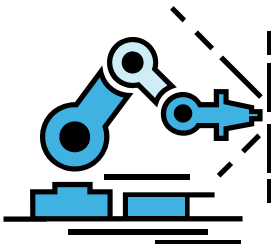
Testing

Problem: Manual Testing

- Manual testing tools
  - **Browser**
    - Web API Test Client
    - Google Advanced REST Client
  - **Fiddler** HTTP Debugging Tool
- Problems with manual testing
  - Generally **time-consuming**
  - Difficult to **reproduce**
  - Can be **error-prone**

Solution: Automated Testing

- Popular unit testing frameworks
  - MSTest – Visual Studio Unit Testing Framework
  - NUnit – originally ported from JUnit
  - xUnit.Net – proposed as successor to NUnit
- Benefits of automated testing
  - Document and validate **expected behaviors**
  - Verify fixing a defect doesn't break something else
  - Can be run **automatically**
    - Source control check-ins
    - Build servers, continuous integration



Testing Approaches

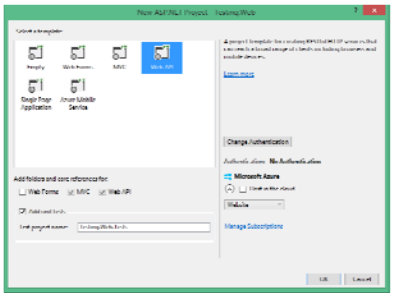
- Plain Old Unit Testing (POUT-ing)
  - Write tests **after** writing code
  - Focus is on defect discovery
- Defect Driven Testing (DDT)
  - Fix a defect by writing a **failing test**
  - Normal part of both POUT and TDD
- Test Driven Development (TDD)
  - Define how piece of code is **expected to behave**
  - **Refactoring** is an integral part of the process
- Behavior Driven Development (BDD)
  - Define **acceptance tests** for features: Given-When-Then

Unit Tests versus Integration Tests

- Unit Tests
  - Serve as **specification** of required functionality
  - Aim to demonstrate just **one behavior**
  - Decoupled from **external dependencies**
  - No **interdependencies** with other tests
  - Can be run **in parallel**
- Integration Tests
  - **End-to-end** from client to server
  - Includes all **components** in the stack
    - Controllers, filters, message handlers, etc
  - Can include **external dependencies**

Visual Studio Unit Test Template

- Add tests to a Web API project
  - Just a **checkbox** in New ASP.NET Project dialog
  - Uses **Visual Studio Unit Testing Framework**
  - Provides unit tests for MVC and API controllers



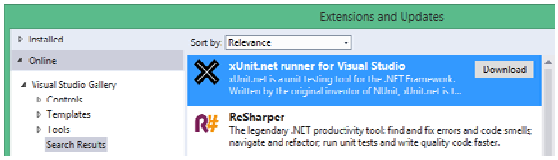
Hello World Unit Tests

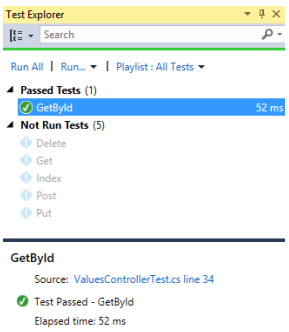
- Create the **controller** and invoke the **action**
  - Write **Asserts** to verify expected results

```
[TestClass] // MSTest attributes
public class ValuesControllerTest {

    [TestMethod]
    public void GetById() {

        var controller = new ValuesController();  // Arrange
        string result = controller.Get(5);        // Act
        Assert.AreEqual("value", result);         // Assert
    } }
```


Visual Studio Test Runner

- Also works with xUnit.net, NUnit
  - Install extensions: Tools, Extensions and Updates
- Other test runners also available
  - ReSharper, TestDriven.net, etc





Wintellect Know how.

Design for Testability

- Controllers should be designed with **testability** in mind
  - External dependencies defined as **interfaces,** declared as **ctor parameters**
  - Tests implement interfaces with **stubs** - test without external dependencies
    - Stubs are **fake objects** which provide **predefined results** for method calls

```
public class ProductsController : ApiController {

    // Dependencies defined as interfaces
    private readonly IProductRepository _productRepository;

    // Dependencies declared as constructor parameters
    public ProductsController
        (IProductRepository productRepository) {
        _productRepository = productRepository; }
```

Consulting/Training

Problem: Testability of HttpResponseMessage

- Actions returning HttpResponseMessage not easily testable
  - If controller calls **Request.CreateResponse** or **Url.Link**, controller must be configured with *route data* - tests require setup code

```
public async Task<HttpResponseMessage> Post(Product product) {
  Product result = await _productRepository.CreateAsync(product);

    // Test must initialize Request and set route data
    var response = Request.CreateResponse
        (HttpStatusCode.Created, result);
    string uri = Url.Link("DefaultApi",
        new {id = result.ProductId}) ?? string.Empty;

    response.Headers.Location = new Uri(uri);
    return response; }
```

Consulting/Training

Problem: Testability of HttpResponseMessage

```
[Fact] // xUnit.net attribute
public async void Post_Returns_Message_With_Product() {

    const int prodId = 42; const string uri =
        "http://localhost/api/products";
    var controller = new ProductsController
        (new FakeProductRepository(prodId));

    // Configuring request and route data can be a nightmare!
    controller.Request = new HttpRequestMessage
        { RequestUri = new Uri(uri) };
    controller.Configuration = new HttpConfiguration();
    controller.Configuration.Routes.MapHttpRoute(
      name: "DefaultApi", routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional });
    controller.RequestContext.RouteData = new HttpRouteData(
route: new HttpRoute(), values: new HttpRouteValueDictionary { {
"controller", "products" } });
```


Solution: IHttpActionResult

- Actions should return IHttpActionResult
  - **Helper methods** return implementations of IHttpActionResult
  - Alleviates the need for tests to include unnecessary set up code

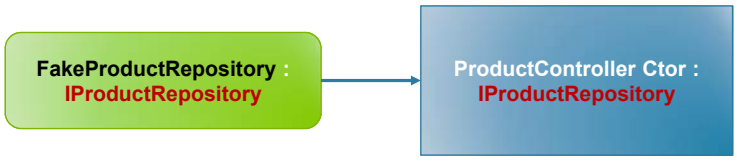
```
public class ProductsController : ApiController {

    [ResponseType(typeof(Product))]
    public async Task<IHttpActionResult> Post(Product product)
    {
        // No need to call Request.CreateResponse or Url.Link
        Product result = await _productRepository
            .CreateAsync(product);
        return CreatedAtRoute("DefaultApi",
            new { id = result.ProductId }, result);
    } }
```

Solution: IHttpActionResult (cont.)

```
[Fact] // Arrange - omitted for brevity
public async void Post_Returns_Result_With_Product() {

    // Act
    IHttpActionResult response = await controller
        .Post(new Product());

    // Assert – No need to inspect response headers
    var result = response as
        CreatedAtRouteNegotiatedContentResult<Product>;
    Assert.Equal(prodId, result.RouteValues["id"]); }
```


Using a Mocking Framework

- What we need is a **fake** IProductRepository
  - Eliminates **external dependencies** – for ex, databases, file system, web service
  - Mocking frameworks let us implement *only members used*
    - Popular open-source mocking frameworks include **Moq** or **RhinoMocks**



Mocking Controller Dependencies

```
[Fact] // The happy path
public async void Get_Returns_Product () {
    var product = new Product { ProductId = 5,
        ProductName = "Chai", UnitPrice = 5 };
    var mockProductRepo = new Mock<IProductRepository>();

    // Arrange - Set up mock IProductRepository with FindAsync
    mockProductRepo.Setup
       (p => p.FindAsync(It.IsAny<int>())).ReturnsAsync(product);
 var controller = new ProductsController(mockProductRepo.Object);

    // Act
 IHttpActionResult actionResult = await controller.GetProduct(5);

    // Assert
    var contentResult =
        (OkNegotiatedContentResult<Product>)actionResult;
    Assert.Equal(5, contentResult.Content.ProductId); }
```


Mocking Controller Dependencies (cont.)

```
[Fact] // The unhappy path
public async void GetProduct_Returns_NotFound() {

    // Arrange – Set up FindAsync to return null
    var mockProductRepo = new Mock<IProductRepository>();
    mockProductRepo.Setup(p => p.FindAsync(It.IsAny<int>()))
        .ReturnsAsync(null);
    var controller =
        new ProductsController(mockProductRepo.Object);

    // Act
 IHttpActionResult actionResult = await controller.GetProduct(1);

    // Assert
    Assert.IsType<NotFoundResult>(actionResult); }
```

Unit Testing HttpMessageHandler

- HttpMessageHandler: Web API pipeline **extensibility point**
  - Allows for message interception and processing on both the client and server
  - Can provide a **mock implementation** of a service to validate processing

```
[Fact] public void Logging_Handler_Logs_Headers() { // Arrange

    // Create storage for mock logger output
    string message = null;
    var mockLogger = new Mock<ILogger>();

    // Set up LogMessage on Ilogger for formatted string
    mockLogger.Setup(m => m.LogMessage
        (It.IsAny<string>(), It.IsAny<object[]>()))
        .Callback<string, object[]>((f, a) =>
        message = string.Format(f, a));
```


Unit Testing HttpMessageHandler (cont.)

- Cannot invoke SendAsync directly - marked as **internal**
  - Use **HttpMessageInvoker** to test the message handler by calling **SendAsync**
  - Can set **InnerHandler** on message handler to a manual mock handler

```
[Fact] public void Logging_Handler_Logs_Headers() { // Arrange

    // Create a message invoker to test the handler
    var handler = new LoggingHandler(mockLogger.Object);
    var invoker = new HttpMessageInvoker(handler);
    var request = new HttpRequestMessage();
        request.Headers.Add("x-header", "hello");

    invoker.SendAsync(request, new CancellationToken()); // Act
    Assert.Equal("x-header : hello ", message); // Assert
```

Unit Testing ActionFilterAttribute

- To test **OnActionExecuting**, initialize an **HttpActionContext**
  - HttpActionContext requires an **HttpControllerContext** with a **HttpRequestMessage**

```
[Fact] public void AddHeaders_Filter_adds_request_header() {
   // Arrange
   var filter = new AddHeadersAttribute();
   var request = new HttpRequestMessage();
   var actionContext = new HttpActionContext
       { ControllerContext = new HttpControllerContext
       { Request = request } };
   filter.OnActionExecuting(actionContext); // Act, Assert
   Assert.True(request.Headers.Contains("x-request-header"));
   Assert.Contains("hello request", request.Headers
       .GetValues("x-request-header")); }
```


Unit Testing ActionFilterAttribute (cont.)

- To test **OnActionExecuted**, initialize an **HttpActionExecutedContext**
  - Requires **HttpActionContext** with an **HttpResponseMessage**

```
[Fact] public void AddHeaders_Filter_adds_response_header() {
   // Arrange
   var filter = new AddHeadersAttribute();
   var response = new HttpResponseMessage();
   var actionExecutedContext = new HttpActionExecutedContext
       { ActionContext = new HttpActionContext(),
         Response = response };
   filter.OnActionExecuted(actionExecutedContext); // Act, Assert
   Assert.True(response.Headers.Contains("x-response-header"));
   Assert.Contains("hello response",
       response.Headers.GetValues("x-response-header"));}
```

Unit Testing Routes

- Useful to test **multiple** routes with a single test method
  - You can use xUnit.net to execute *parameterized* tests by using **[Theory]** attribute instead of **[Fact]**

```
[Theory] // Run test three times using different inputs
[InlineData("http://test.com/bogus/route", "GET",
    typeof(ProductsController), "Get", "id", "1", false)]
[InlineData("http://test.com/api/products/1", "GET",
    typeof(ProductsController), "Get", "id", "1", true)]
[InlineData("http://test.com/api/products", "POST",
    typeof(ProductsController), "Post", null, null, true)]
public void Default_route_returns_correct_route_info(string url,
    string method, Type expectedController,
    string expectedAction, string expectedParameter,
    object expectedParameterValue, bool shouldFind) {
```


Unit Testing Routes (cont.)

```
var config = new HttpConfiguration(); // Arrange - config
   config.Routes.MapHttpRoute
       ( name: "DefaultApi", routeTemplate:
       "api/{controller}/{id}", defaults: new
       { id = RouteParameter.Optional });
   var request = new HttpRequestMessage(new HttpMethod
       (method), url);

   // Act - Helper extension method
   RouteInfo routeInfo = request.GetRouteInfo(config);

   // Assert - Actual route information matches what is expected
   Assert.Equal(expectedController, routeInfo.ControllerType);
   Assert.Equal(expectedAction, routeInfo.ActionName);
   Assert.Equal(expectedParameterValue,
       routeInfo.Parameters[expectedParameter]); }
```

Unit Testing Routes : Helper Method

```
public static RouteInfo GetRouteInfo(this HttpRequestMessage
request, HttpConfiguration config) {

    //  Get route data from request
    var routeData = config.Routes.GetRouteData(request);
    if (routeData == null) return null;
    request.Properties[HttpPropertyKeys.HttpRouteDataKey]
        = routeData;

    // Get controller descriptor
    var controllerSelector =
        new DefaultHttpControllerSelector(config);
    var controllerDescriptor =
        controllerSelector.SelectController(request);
```


Unit Testing Routes : Helper Method (cont.)

```
    // Get action descriptor
    var actionSelector = new ApiControllerActionSelector();
    var controllerContext = new HttpControllerContext
        (config, routeData, request)
        { ControllerDescriptor = controllerDescriptor };

    var actionDescriptor = actionSelector
        .SelectAction(controllerContext);

    // Return route information - parameters omitted for brevity
    return new RouteInfo
    { ControllerType = controllerDescriptor.ControllerType,
      ActionName = actionDescriptor.ActionName };}
```

Integration Testing with In-Memory Host

- Use **integration testing** to verify component behavior in the **Web API pipeline**
  - Pass **HttpServer** to **HttpClient** constructor, accepts an HttpMessageHandler
  - Allows you to test the entire stack in memory *without opening ports*
- **Good**: simpler, faster to execute.
  Not so good: cannot trace with Fiddler

Integration Testing with In-Memory Host (cont.)

```
[Fact] public async void
ProductsController_get_should_return_product_1() {
   var config = new HttpConfiguration(); // Arrange - config
   config.Routes.MapHttpRoute(name: "DefaultApi", routeTemplate:
      "api/{controller}/{id}",
       defaults: new { id = RouteParameter.Optional });

   // Arrange - Setup IoC container
   var container = new ServiceContainer();
   container.Register<ProductsController>
       (new PerRequestLifeTime());
   container.Register<IProductRepository, ProductRepository>
       (new PerRequestLifeTime()
   container.EnableWebApi(config);
```

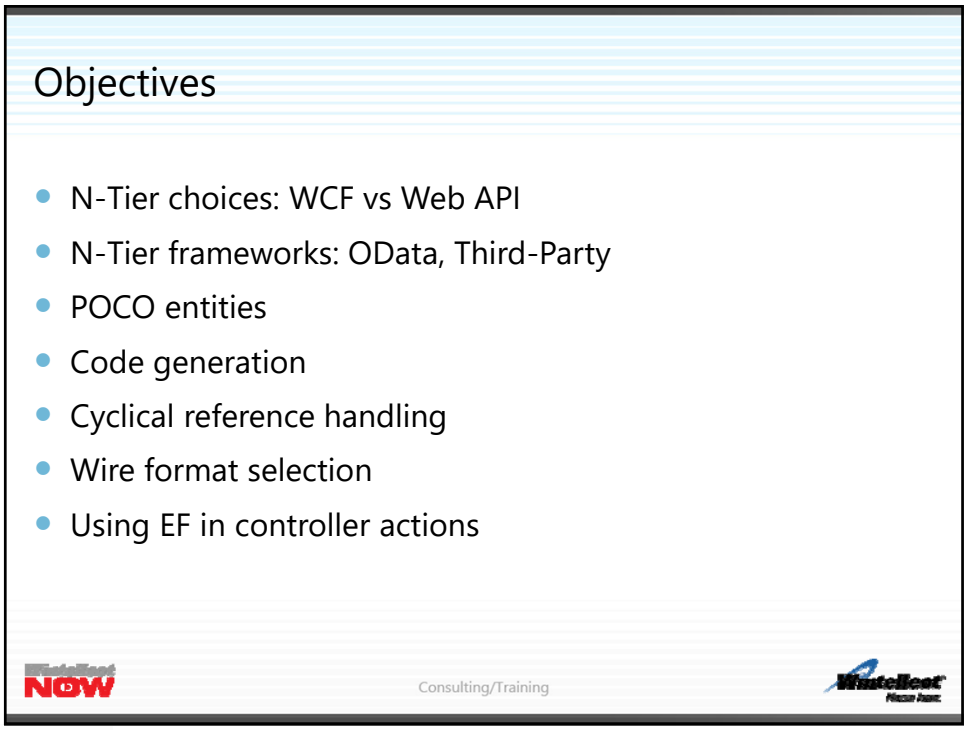
Integration Testing with In-Memory Host (cont.)

```
// Arrange - Create server and client
var server = new HttpServer(config);
var client = new HttpClient(server);

// Act
var response =
    await client.GetAsync("http://test.com/api/Products/1");
var product = await response.Content.ReadAsAsync<Product>();

// Assert
Assert.NotNull(product);
Assert.Equal(1, product.ProductId);
```

Consulting/Training

QUESTIONS

Consulting/Training

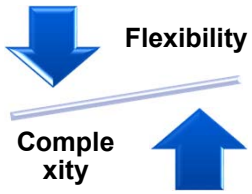


N-Tier Motivation

- Clients should **not** connect to the database directly
  - Installation of database drivers on the client requires *admin rights*
  - Direct client connections create *security* and *performance* problems
- Database queries and updates should take place from **within** a service layer
  - Clients don't know *anything* about the backend database
  - No need to install database drivers
  - More *flexible* architecture – not coupled to database vendor or API
  - Business logic and security *encapsulated* within the service

N-Tier Trade Offs

- N-Tier architectures involve trade-offs
  - Flexibility, scalability, maintainability
  - Requires a lot more work!



Web Services: WCF vs Web API

- **Windows Communication Foundation**
  - Built for **SOAP**
  - Transport-independent
  - Well-suited for Remote Procedure Calls (RPC)
  - Largely *deprecated* in favor of RESTful services with Web API
  - Still recommended for *inter/intra process* or *message queuing*
- **ASP.NET Web API**
  - Embraces **HTTP** and web programming model
  - Built for *RESTful* services
  - Supports XML, JSON and binary wire formats
  - Supports *dependency injection* (DI)
  - Designed for *testability* and test-driven development (TDD)

Consulting/Training

N-Tier Frameworks

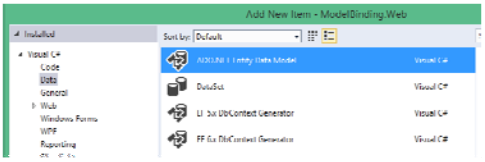
- **OData (Open Data Protocol)**
  - Expose an entity data model as a *REST service*
  - Hypermedia driven using **AtomPub** syndication format
  - Supports change-tracking and batch updates
  - Implemented for WCF Data Services and ASP.NET Web API
- **Trackable Entities** (open source framework)
  - Replacement for now defunct "Self-Tracking Entities"
  - Supports *change-tracking* and batch updates
  - Deployed as NuGet packages and a Visual Studio Extension
  - Includes *both* WCF and Web API templates
  - Supports both *model-first* and *code-first* approaches
  - Enables **domain driven design** (DDD) with repository and unit of work patterns

Consulting/Training

Entity Framework and Web API

Separate Entities from EF-Specific Code

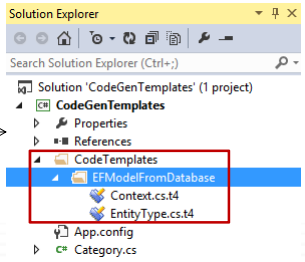
- Entities should be **ignorant** of persistence concerns
  - Place entities in a separate (portable) class library project
- May *reverse engineer* entities from an existing database
  - EF 6.x VS Tools: Add **ADO.NET Entity Data Model** to **.Data** project
    - EF Designer from Database
    - Code First from Database
  - Copy or "Add As Link" from **.Data** to **.Entities** project
  - Or use *EF Power Tools* to reverse engineer Code First classes



Customize Code Generation

- Control code generation for EF 6.x Tools with **T4 templates**
  - For CodeFirst from Database, install the NuGet package **EntityFramework.CodeTemplates.CSharp**
  - EF Power Tools also provide customizable T4 templates
  - Third-party T4 editors make editing T4 templates much easier

T4 templates control entity and context code generation



Disable Proxy Creation

- Entity Framework generates **dynamic runtime proxies**
  - Used to support features such as Lazy Loading
    - Usually not required for n-tier scenarios
  - Enabled when all properties are defined as **virtual**
    - Should be explicitly **disabled** because runtime proxies are not serializable

```
public partial class Northwind : DbContext {
    public Northwind() : base("name=Northwind")
    {
        // Disable dynamic proxies, which are not serializable
        Configuration.ProxyCreationEnabled = false;
    }
}
```

Handling Cyclical References: Attributes

- Generated entities usually contain **cyclical references** (Product <-> Category)
- Serializers must be **configured** to handle cycles
  - By default referenced objects serialized *as values*
  - Configure with **attributes** or in **code** (preferred for POCO's)

```
// Json.Net preserves object reference to handle cycles
[JsonObject(IsReference = true)]
public class Product { ... }

// Data Contract preserves object reference to handle cycles
[DataContract(IsReference = true)]
public class Product { ... }
```

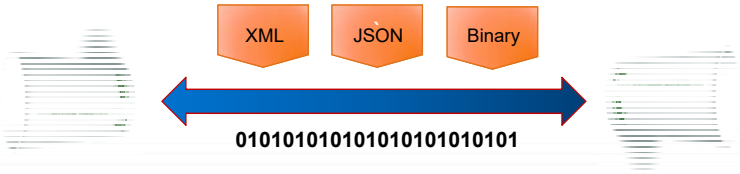

Handling Cyclical References: Code

- Configure Json and Xml serializers
  - HttpConfiguration.**Formatters**

```
public static class WebApiConfig {
    public static void Register(HttpConfiguration config) {

        // Configure Json formatter to handle cycles
        config.Formatters.JsonFormatter.SerializerSettings
            .PreserveReferencesHandling =
            PreserveReferencesHandling.All;

      // Configure Xml formatter for each entity type
      var dcs = new DataContractSerializer(typeof(Product), null,
int.MaxValue, false, /* preserveObjectReferences: */ true, null);
        config.Formatters.XmlFormatter
            .SetSerializer<Product>(dcs); } }
```

Wire Format Selection

- Wire format selection can impact **performance**
  - Generally *fewer bytes* on the wire will improve performance
  - Computational overhead of encoding is also important
  - XML is usually the most verbose and *least performant*
- JSON is better, but text encoding still not very efficient
  - BSON format intended for MIME encoded data
  - What is needed is an efficient JSON **binary encoder**



Protocol Buffers – Protobuf

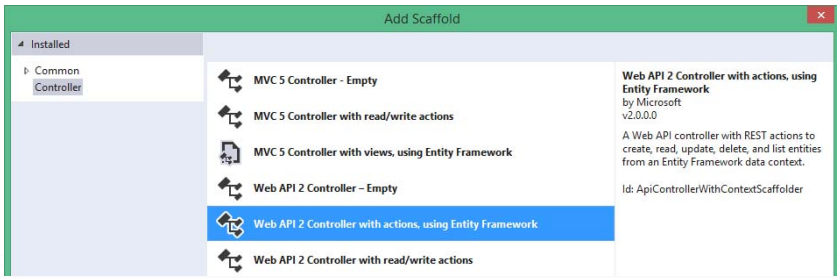
- Protobuf is Google's fast serializer, *outperforms* Json.Net
  - Install **WebApiContrib.Formatting.ProtoBuf** NuGet package
  - Handle cyclical references with **ProtoContract** attribute, or in code (preferred): **AsReferenceDefault = true**
  - On *client* use **ProtoBufFormatter**, set Accept and/or ContentType headers: **application/x-protobuf**

```
// Add protobuf formatter to HttpConfiguration
config.Formatters.Add(new ProtoBufFormatter());

// Configure types in code to handle cyclical references
MetaType personMeta = ProtoBufFormatter.Model.Add
    (typeof(Person), false);
personMeta.Add(1, "Id").Add(2, "Name"); // Properties
personMeta.AsReferenceDefault = true;   // Reference handling
```


Wintellect Know how.

Add Controller with Actions using EF

- There is **tooling support** in VS for using EF with Web API
  - Don't take as gospel but use as a starting point
  - For example, first Get method is not async, returns IQueryable
  - Modify queries to *eager load* related entities via **Include** operator



Service Operations: Retrieve (Get)

- Get everything you need in **one trip**
  - Prefer *chunky* over *chatty* database calls
  - Use **async** API for greater scalability
  - **Include** operator has overloads accepting lambda expression (single property) and string (nested properties)

```
[ResponseType(typeof(IEnumerable<Order>))]  // GET: api/Orders
public async Task<IHttpActionResult> GetOrders() {

    var orders = await (from o in db.Orders
       .Include(o => o.Customer)        // Load Customer property
       .Include("OrderDetails.Product") // Details, Products
       select o).ToListAsync();
    return Ok(orders); }
```

Service Operations: Insert (Post)

- Do *not* **retrieve entities** in order to update them
  - Explicitly set entity **State** to perform disconnected updates
  - For inserts, simply *adding* the entity will set the State (will set child entities to Added too)
  - Call **CreatedAtRoute** to return entity and set Location header

```
[ResponseType(typeof(Order))]     // POST: api/Orders
public async Task<IHttpActionResult> PostOrder(Order order) {

    db.Orders.Add(order);         // Sets State to Added
    await db.SaveChangesAsync();  // Saves changes in tx

    return CreatedAtRoute("DefaultApi",
        new { id = order.OrderId }, order); }  // Location header
```

Consulting/Training

Service Operations: Update (Put)

- To update entity set its State to **Modified**
  - Will update *all columns* (not a partial update)
  - State of child entities needs to be set *individually*
  - Return **NotFound** if entity with id does not exist
  - Return entity to include **db-generated** values (concurrency, etc)

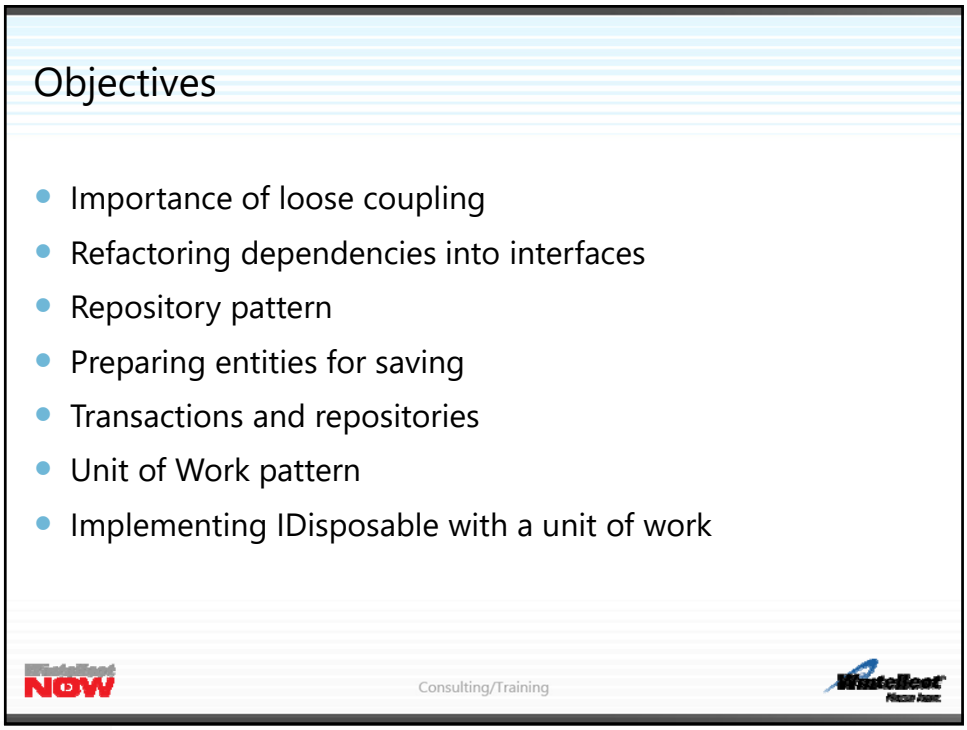
```
[ResponseType(typeof(Order))]  // PUT: api/Orders/5
public async Task<IHttpActionResult> PutOrder(Order order) {

    db.Entry(order).State = EntityState.Modified; // Set state

    try { await db.SaveChangesAsync(); } // Save changes in tx
    catch (DbUpdateConcurrencyException) {  // Order deleted
        if (!db.Orders.Any(e => e.OrderId == order.OrderId))
            return NotFound();
        throw; } return Ok(order); }
```

Consulting/Training

Service Operations: Delete

- Must **retrieve** entity by key in order to delete it
  - Include *child entities* if cascade deletes not specified in model

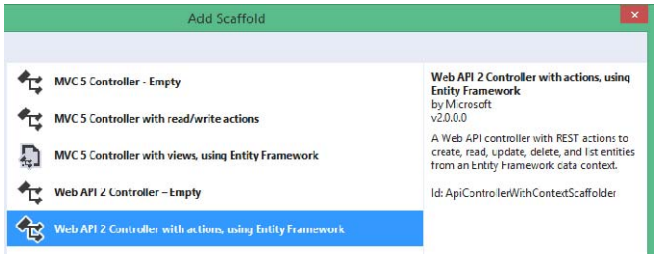
```
public async Task<IHttpActionResult> DeleteOrder(int id) {

    var order = await db.Orders           // Retrieve existing
        .Include(o => o.OrderDetails)     // Include children
        .SingleOrDefaultAsync(o => o.OrderId == id);
    if (order == null) return Conflict(); // Return conflict

    for (int i = order.OrderDetails.Count - 1; i > -1; i--) {
        var detail = order.OrderDetails.ElementAt(i);
        db.OrderDetails.Remove(detail); } // Remove children
    db.Orders.Remove(order);              // Remove entity
    await db.SaveChangesAsync(); return Ok(); }
```

Wintellect
Know how.

Problem: Data Access Coupling

- Default Web API scaffolding creates **direct dependency** on Entity Framework



Repository Pattern

Repository Pattern

- Repository interfaces **decouple** controllers from data access API

```
public interface IProductRepository {
    Task<Product> FindAsync(int id);
}
```

```
public class ProductsController : ApiController {

  // Dependencies declared as constructor parameters
  private readonly IProductRepository _productRepository;
  public ProductsController(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    } }
```

Consulting/Training

Repository Implementation

- Framework-specific implementation
  - Dependency injected by IoC container

```
public class ProductRepository : IProductRepository {

    // Uses Entity Framework for persistence
    private readonly NorthwindContext _dbContext;

    public ProductRepository(NorthwindContext dbContext) {
        _dbContext = dbContext; }

    public async Task<Product> FindAsync(int id) {
        return await _dbContext.Products.FindAsync(id);
    }
}
```

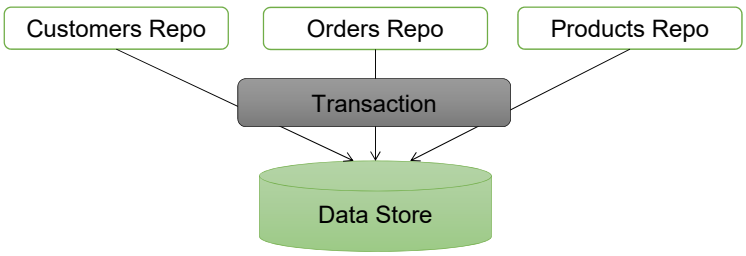
Consulting/Training

Preparing Entities for Saving

```
public class ProductRepository : IProductRepository {

    // Mark entity as Added
    public void Insert(Product product) {
        _dbContext.Products.Add(product);
    }

    // Mark entity as Modified
    public void Update(Product product) {
        _dbContext.Entry(product).State = EntityState.Modified;
    }

    // Mark entity as Deleted
    public async Task Delete(int id) {
        var product = await _dbContext.Products.FindAsync(id)
        _dbContext.Products.Remove(product);
    } }
```


Problem: Transactions

- Entities from multiple repositories should be saved within **same transaction**
  - In case of error, all changes *rolled back*



Unit of Work Pattern

Consulting/Training

Solution: Unit of Work Pattern

- Work spans one or more repositories
  - Expose repos as UoW properties

```
public interface IUnitOfWork
{
    // Repositories
    ICustomerRepository CustomerRepository { get; }
    IOrderRepository OrderRepository { get; }
    IProductRepository ProductRepository { get; }

    // Persistence
    Task<int> SaveChangesAsync();
}
```

Consulting/Training

Unit of Work Implementation

```
public class UnitOfWork : IUnitOfWork, IDisposable
{
    public UnitOfWork(ICustomerRepository custRepo,
        IOrderRepository orderRepo,
        NorthwindContext dbContext) { // Code elided ...

    // Repository properties
    public ICustomerRepository CustomerRepository {
       get { return _customerRepository; } }

    public IOrderRepository OrderRepository {
        get { return _orderRepository; } }

    // Persistence
    public async Task<int> SaveChangesAsync() {
        return await _dbContext.SaveChangesAsync(); }
}
```

Consulting/Training

Don't Forget to Clean Up

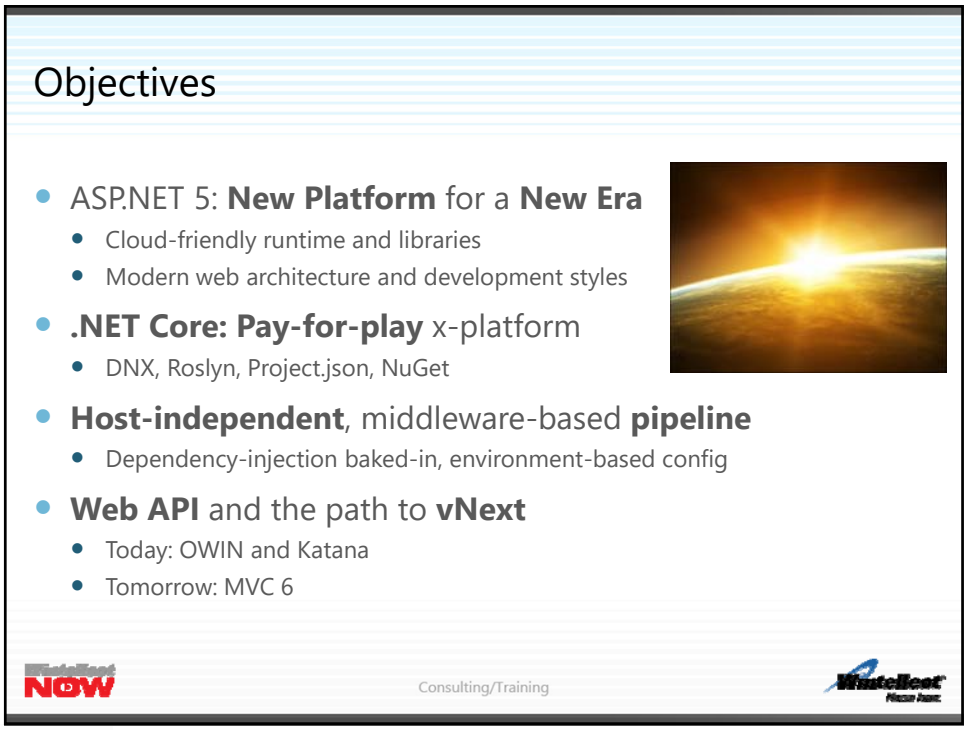
- UoW should dispose of DbContext
  - Dispose called at end of each request

```
public class UnitOfWork : IDisposable {

    public void Dispose()
    {
        // Safely cast to IDisposable, then call Dispose
        if (_disposed) return;
        var disposable = _dbContext as IDisposable;
        if (disposable != null) disposable.Dispose();
    }
}
```

Consulting/Training

Controllers and UoW

```
public class ProductController : ApiController {
    // Inject Unit of Work
    private readonly IUnitOfWork _unitOfWork;
    public ProductController(IUnitOfWork unitOfWork) {
        _unitOfWork = unitOfWork; }

    // GET api/Product/5
    [ResponseType(typeof(Product))]
    public async Task<IHttpActionResult> Get(int id) {
       return await _unitOfWork.ProductRepository.FindAsync(id); }

    // POST api/Product
    [ResponseType(typeof(Product))]
    public async Task<IHttpActionResult> Post(Product product) {
        _unitOfWork.ProductRepository.Insert(product);
        await _unitOfWork.SaveChangesAsync();
        return product; } }
```

What is the Cloud? Why Should I Care?

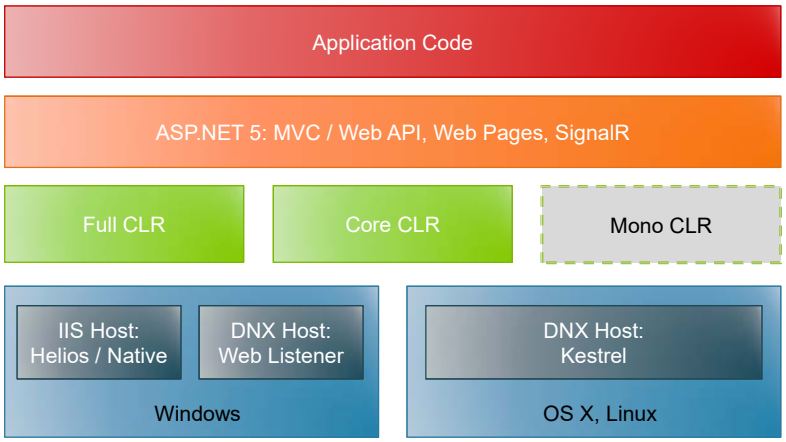
- "Cloud computing relies on **shared resources** to achieve **economies of scale**."
  - Wikipedia
  - Need greater isolation between apps
- Computing resources are allocated on a **pay-as-you-go** basis.
  - Resource-hungry apps are more expensive to scale

.NET Core 5: New Runtime, New Libraries

- **Cloud-optimized** version of the .NET Framework
  - Small footprint, high throughput, modular
- Bin-deployable
  - Each app gets its own private copy of the Core CLR
  - Runs **side-by-side** with other versions on the same machine
- Delivered via **NuGet**
  - Load only .NET components *used by your app*
- Cross-platform, open source
  - Runs on **Windows**, **Mac OS X**, and **Linux** (including Docker)
  - Accepting pull requests on **GitHub**!

Wintellect Know how.

ASP.NET 5: Platform and Host Independent



DNX: .NET Execution Environment

- **Consistent execution environment** across platforms
  - Execute project entry point via commands defined in **project.json** file
  - Supports self-hosting with a lightweight process: **dnx.exe**
- Set of **tools** for developing and running ASP.NET 5 apps
  - Environment management: **dnvm.exe**
  - Package management (NuGet): **dnu.exe**

```
dnvm list
Active Version     Runtime Architecture Location                    Alias
------ -------     ------- ------------ --------                    -----
       1.0.0-beta4 clr     x64          C:\Users\Tony\.dnx\runtimes
       1.0.0-beta4 clr     x86          C:\Users\Tony\.dnx\runtimes default
       1.0.0-beta4 coreclr x64          C:\Users\Tony\.dnx\runtimes
       1.0.0-beta4 coreclr x86          C:\Users\Tony\.dnx\runtimes
```


Wintellect Know how.

Project.json File

- Where you define **project information**
  - Target frameworks, dependencies, commands, etc

```
{ "webroot": "wwwroot", "version": "1.0.0-*",

  "dependencies": { "Microsoft.AspNet.Mvc": "6.0.0-beta4",
      "Microsoft.AspNet.Server.IIS": "1.0.0-beta4",
      "Microsoft.AspNet.Server.WebListener": "1.0.0-beta4" },

  "commands": {
      "web": "Microsoft.AspNet.Hosting --server
          Microsoft.AspNet.Server.WebListener
          --server.urls http://localhost:5000",
      "kestrel": "Microsoft.AspNet.Hosting --server
          Kestrel --server.urls http://localhost:5004" },
  "frameworks": { "dnx451": { }, "dnxcore50": { } } }
```


Global.json File

- Where you define **solution structure**
  - Project folder structure, minimum DNX version

```
{
  "projects": [ "src", "test" ],
  "sdk": {
    "version": "1.0.0-beta4"
  }
}
```

Roslyn: Compiler as a Service

- ASP.NET 5 apps are compiled **dynamically**
  - No need for a separate "Build" step
  - Compiled code is not written to disk – no "dll" generated
  - Can deploy **source code** files instead of binaries
  - Still possible to pre-compile web apps and deploy packages

NuGet: Improved Package Manager

- Only necessary to include "**top-level**" packages in project.json file
  - Downstream dependencies *resolved automatically*
- Packages are stored in a **central location**
  - Packages are no longer stored at the solution level
  - Instead they are stored in *one location* under the user's profile

New HTTP Request Pipeline

- Dependence on **System.Web** removed
  - Reduced memory footprint
  - Uses an "opt-in" model for only what you need
- Middleware is configured in code from a **Startup** class
  - Includes MVC / Web API, static pages, security, and custom components
  - Usually via **.UseXxx** extension methods to IApplicationBuilder

Startup Class

```
public class Startup {

    // Optional ctor
    public Startup(IHostingEnvironment env) { }

    // Add services to the DI container
    public void ConfigureServices(IServiceCollection services) {
        services.AddMvc(); }

    // Add middleware components
    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env) {
        app.UseStaticFiles();
        app.UseMvc(); }
}
```

Flexible Configuration

- New configuration system replaces **web.config**
  - Supports **multiple sources**, for example:
    json, xml or ini files; command-line args; environment variables
  - Complex structures supported – not just key/value pairs

```
public class Startup {

    public IConfiguration Configuration { get; set; }

    // Set up configuration sources
    public Startup(IHostingEnvironment env) {
        Configuration = new Configuration()
            .AddJsonFile("config.json")
            .AddCommandLine(args)
            .AddEnvironmentVariables(); } }
```

Consulting/Training

Dependency Injection Baked-In

- Unified **dependency injection** system
  - Register services in **Startup.ConfigureServices**
  - Specify **lifetime**: singleton, transient, scoped to request
  - Services available throughout *entire web stack* (middleware, filters, controllers, model binding, etc)
  - Easily *replace* default DI container with one of your own choosing

```
public class Startup {

    public void ConfigureServices(IServiceCollection services) {
        // Register services with the DI container
        services.AddScoped<IProductRepository,
        ProductRepository>();
    }
}
```

Consulting/Training

What's New in Web API for vNext?

- **Unified** programming model
  - Together at last: **MVC** and **Web API**
  - Single web app can contain *both UI and services*
- No more ApiController base class
  - Controllers can extend **Controller base class**
  - Controllers can simply be names with **Controller suffix**
- Shared core components
  - Routing engine
  - Dependency injection
  - Configuration framework

Migration with Web API Compatibility Shim

- Not all Web API 2 constructs carry forward to MVC 6
  - HttpRequestMessage and HttpResponseMessage no longer exist
  - Helper methods return **ObjectResult** instead of IHttpActionResult
- Compatibility Shim can bridge the gap
  - NuGet package: **Microsoft.AspNet.Mvc.WebApiCompatShim**
  - Derive from **ApiController** base class
  - Return **IHttpActionResult** using helper methods
  - Apply Web API **routing** configuration and conventions
  - Create HttpRequestMessage and HttpResponseMessage instances

Hosting for Today and Tomorrow

- ASP.NET 5 pipeline built on concepts from **OWIN** and Katana
  - Startup class with Configuration method
  - Chain together middleware components
  - Enables *decoupling* from web host
- Use **OWIN** today for easier migration to ASP.NET 5 tomorrow
  - **Web hosting**: Microsoft.Owin.Host.SystemWeb
  - **Self hosting**: Microsoft.Owin.SelfHost, Microsoft.AspNet.WebApi.OwinSelfHost
  - **Hosting with OwinHost.exe**: OwinHost
  - **Web API middleware**: Microsoft.AspNet.WebApi.Owin

QUESTIONS

NOTES

NOTES

NOTES

NOTES