# Web API 2

The term API stands for ‘Application Programming Interface’. ASP.NET Web API is a framework for building Web API’s

<https://searchmicroservices.techtarget.com/tip/REST-vs-SOAP-Choosing-the-best-web-service>

# WCF (Windows Communication Foundation)

**SOAP (Simple object access protocol)**

# What is Rest

Representational State Transfer

REST is an architectural style of distributed

A Restful system should follow the constrains so it can be called Restful.

* Uniform Interface
* StateLess
* Clien-Server
* Cacheable
* Layered system

# There are several key implementation details with HTTP that you should be aware of

* Resources – REST uses addressable resources to define the structure of the API.

These are the URLs you use to get to pages on the web

* Request Verbs – These describe what you want to do with the resource.
* Request Headers – These are additional instructions that are sent with the request.
* Request Body - Data that is sent with the request.
* Response Body – This is the main body of the response.
* Response Status codes – These codes are issues with the response and give the client details on the status of the request.

# Content Negotiation in Web API

Depending on the Accept header value in the request, the server sends the response. This is called Content Negotiation.

Accept: application/xml returns XML  
Accept: application/json returns JSON

# List of media types of HTTP

* Image/Png
* Text/HTML
* Application/Json

# What does API header consist

* User-Agent: Fiddler
* content-type: application/json
* accept: application/json
* Host: localhost:39999
* Content-Length: 1097
* X-Api-Version: 2
* custom: [ { "id": 2 }]

# What is the use of “IgnoreDataMember” in Web API?

By default if the properties are public then those can be serialized and deserialized,

if we does not want to serialize the property then decorate the property with this attribute.

# What is “Under-Posting” and “Over-Posting” in Web API?

* “Under-Posting” – When client leaves out some of the properties while binding then it’s called under – posting.
* “Over-Posting” – If the client sends more data than expected in binding then it’s called over-posting.

# Why to use “FromUri” in Web API?

In Web API to read complex types from URL we will use “FromUri” attribute to the parameter in action method. Eg:

public HttpResponseMessage Get([FromUri] MyCustomer c) { … }

# Why to use “FromBody” in Web API?

This attribute is used to force Web API to read the simple type from message body. “FromBody” attribute is along with parameter. Eg:

public HttpResponseMessage Post([FromBody] int customerid, [FromBody] string customername) { … }

# Filters

|  |  |  |  |
| --- | --- | --- | --- |
| Filter Type | Interface | Class | Description |
| Simple Filter | IFilter | - | Defines the methods that are used in a filter |
| Action Filter | IActionFilter | ActionFilterAttribute | Used to add extra logic before or after action methods execute. |
| Authentication Filter | IAuthenticationFilter | - | Used to force users or clients to be authenticated before action methods execute. |
| Authorization Filter | IAuthorizationFilter | AuthorizationFilterAttribute | Used to restrict access to action methods to specific users or groups. |
| Exception Filter | IExceptionFilter | ExceptionFilterAttribute | Used to handle all unhandled exception in Web API. |
| Override Filter | IOverrideFilter | - | Used to customize the behaviour of other filter for individual action method. |

Web API includes filters to add extra logic before or after action method executes

**ActionFilterAttribute**

public override void OnActionExecuting(HttpActionContext actionContext)

public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)

# Global asax events

* **Application\_Init**: Fired when an application initializes or is first called. It's invoked for all HttpApplication object instances.
* **Application\_Disposed**: Fired just before an application is destroyed. This is the ideal location for cleaning up previously used resources.
* **Application\_Error**: Fired when an unhandled exception is encountered within the application.
* **Application\_Start**: Fired when the first instance of the HttpApplication class is created. It allows you to create objects that are accessible by all HttpApplication instances.
* **Application\_End**: Fired when the last instance of an HttpApplication class is destroyed. It's fired only once during an application's lifetime.
* **Application\_BeginRequest**: Fired when an application request is received. It's the first event fired for a request, which is often a page request (URL) that a user enters.
* **Application\_EndRequest**: The last event fired for an application request.
* **Application\_PreRequestHandlerExecute**: Fired before the ASP.NET page framework begins executing an event handler like a page or Web service.
* **Application\_PostRequestHandlerExecute**: Fired when the ASP.NET page framework is finished executing an event handler.
* **Applcation\_PreSendRequestHeaders**: Fired before the ASP.NET page framework sends HTTP headers to a requesting client (browser).
* **Application\_PreSendContent**: Fired before the ASP.NET page framework sends content to a requesting client (browser).
* **Application\_AcquireRequestState**: Fired when the ASP.NET page framework gets the current state (Session state) related to the current request.
* **Application\_ReleaseRequestState**: Fired when the ASP.NET page framework completes execution of all event handlers. This results in all state modules to save their current state data.
* **Application\_ResolveRequestCache**: Fired when the ASP.NET page framework completes an authorization request. It allows caching modules to serve the request from the cache, thus bypassing handler execution.
* **Application\_UpdateRequestCache**: Fired when the ASP.NET page framework completes handler execution to allow caching modules to store responses to be used to handle subsequent requests.
* **Application\_AuthenticateRequest**: Fired when the security module has established the current user's identity as valid. At this point, the user's credentials have been validated.
* **Application\_AuthorizeRequest**: Fired when the security module has verified that a user can access resources.
* **Session\_Start**: Fired when a new user visits the application Web site.
* **Session\_End**: Fired when a user's session times out, ends, or they leave the application Web site.

# Authentication

It’s about validating the identity of a user who is accessing our system,

that he is authenticated enough to use our resources or not.

# Authorization

what all permissions the authenticated user has to access web resources.

# Basic Authentication

* where an end user gets authenticated through our service with the help of plain credentials such as user name and password.
* easy to implement
* it is nearly supported by all modern browsers
* has become an authentication standard in RESTful / Web APIs
* sending user credentials in plain text, sending user credentials inside request header, i.e. prone to hack.
* No session is maintained

# Token Based Authorization

* once authenticated a service can send a token to an end user through which user can access other resources.
* The token could be any encrypted key, which only server/service understands and
* when it fetches the token from the request made by end user,
* it validates the token and authorizes user into the system.
* Derive that class from AuthorizationFilterAttribute, this is a class under System.Web.Http.Filters.

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false)]

public class GenericAuthenticationFilter : AuthorizationFilterAttribute

{

public override void OnAuthorization(HttpActionContext filterContext)

{

if (!\_isActive) return;

var identity = FetchAuthHeader(filterContext);

if (identity == null)

{

ChallengeAuthRequest(filterContext);

return;

}

var genericPrincipal = new GenericPrincipal(identity, null);

Thread.CurrentPrincipal = genericPrincipal;

if (!OnAuthorizeUser(identity.Name, identity.Password, filterContext))

{

ChallengeAuthRequest(filterContext);

return;

}

base.OnAuthorization(filterContext);

}

protected virtual BasicAuthenticationIdentity FetchAuthHeader(HttpActionContext filterContext)

{

string authHeaderValue = null;

var authRequest = filterContext.Request.Headers.Authorization;

if (authRequest != null && !String.IsNullOrEmpty(authRequest.Scheme) && authRequest.Scheme == "Basic")

authHeaderValue = authRequest.Parameter;

if (string.IsNullOrEmpty(authHeaderValue))

return null;

authHeaderValue = Encoding.Default.GetString(Convert.FromBase64String(authHeaderValue));

var credentials = authHeaderValue.Split(':');

return credentials.Length < 2 ? null : new BasicAuthenticationIdentity(credentials[0], credentials[1]);

}

protected virtual bool OnAuthorizeUser(string user, string pass, HttpActionContext filterContext)

{

if (string.IsNullOrEmpty(user) || string.IsNullOrEmpty(pass))

return false;

return true;

}

private static void ChallengeAuthRequest(HttpActionContext filterContext)

{

var dnsHost = filterContext.Request.RequestUri.DnsSafeHost;

filterContext.Response = filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);

filterContext.Response.Headers.Add("WWW-Authenticate", string.Format("Basic realm=\"{0}\"", dnsHost));

}

}

public class ApiAuthenticationFilter : GenericAuthenticationFilter

{

protected override bool OnAuthorizeUser(string username, string password, HttpActionContext actionContext)

{

var provider = actionContext.ControllerContext.Configuration

.DependencyResolver.GetService(typeof(IUserServices)) as IUserServices;

if (provider != null)

{

var userId = provider.Authenticate(username, password);

if (userId > 0)

{

var basicAuthenticationIdentity = Thread.CurrentPrincipal.Identity as BasicAuthenticationIdentity;

if (basicAuthenticationIdentity != null)

basicAuthenticationIdentity.UserId = userId;

return true;

}

}

return false;

}

}

}

# There are three ways in which you can use this authentication filter.

* Just apply this filer to ProductController.

[ApiAuthenticationFilter]

[RoutePrefix("v1/Products/Product")]

public class ProductController : ApiController

* You can also globally add this in Web API configuration file , so that filter applies to all the controllers and all the actions associated to it,

GlobalConfiguration.Configuration.Filters.Add(new ApiAuthenticationFilter());

* You can also apply it to Action level too by your wish to apply or not apply authentication to that action,

[ApiAuthenticationFilter(true)]

[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()

{ }

[ApiAuthenticationFilter(false)]

[GET("productid/{id?}")]

[GET("particularproduct/{id?}")]

[GET("myproduct/{id:range(1, 3)}")]

public HttpResponseMessage Get(int id)

{

var product = \_productServices.GetProductById(id);

if (product != null)

return Request.CreateResponse(HttpStatusCode.OK, product);

return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for this id");

}

* If a method return type is void, by default status code 204 No Content is returned.

# **What is same origin policy**

Browsers allow a web page to make AJAX requests only with in the same domain. Browser security prevents a web page from making AJAX requests to another domain. This is called same origin policy.   
  
  
  
**The following 2 URLs have the same origin**  
http://localhost:1234/api/employees  
http://localhost:1234/Employees.html  
  
**The following 2 URLs have different origins, because they have different port numbers (1234 v/s 5678)**  
http://localhost:1234/api/employees  
http://localhost:5678/Employees.html  
  
**The following 2 URLs have different origins, because they have different domains (.com v/s .net)**  
http://pragimtech.com/api/employees  
http://pragimtech.net/Employees.html  
  
**The following 2 URLs have different origins, because they have different schemes (http v/s https)**  
https://pragimtech.com/api/employees  
http://pragimtech.net/Employees.html  
  
To prove browsers does not allow cross domain ajax requests

There are 2 ways to get around this problem

# Using JSONP (JSON with Padding)

* + - **JSONP will wrap the data in a function as shown below**  
      CallbackFunction({  
          "FirstName" : "Mark",  
          "LastName"  : "Hastings",  
          "Gender"    : "Male",  
      })

**Step 1 :** To support JSONP format, execute the following command using NuGet Package Manager Console which installs WebApiContrib.Formatting.Jsonp package.  
Install-Package WebApiContrib.Formatting.Jsonp  
  
**Step 2 :** Include the following 2 lines of code in Register() method of WebApiConfig class in WebApiConfig.cs file in App\_Start folder

var jsonpFormatter = newJsonpMediaTypeFormatter(config.Formatters.JsonFormatter);

config.Formatters.Insert(0, jsonpFormatter);

**Step 3 :** In the ClientApplication, set the dataType option of the jQuery ajax function to jsonp   
dataType: 'jsonp'

# Enabling CORS (Cross Origin Resource Sharing)

**Step1:** Install Microsoft.AspNet.WebApi.Cors package.

**Step 2 :** Include the following 2 lines of code in **Register()**method of **WebApiConfig**class in **WebApiConfig.cs** file in **App\_Start**folder

EnableCorsAttribute cors = new EnableCorsAttribute("\*", "\*", "\*");

config.EnableCors();

**Step 3 :** In the ClientApplication, set the dataType option of the jQuery ajax function to json  
dataType: 'json'  
  
**Parameters of EnableCorsAttribute**

| **Parameter** | **Description** |
| --- | --- |
| origins | Comma-separated list of origins that are allowed to access the resource. For example "http://www.pragimtech.com,http://www.mywebsite.com" will only allow ajax calls from these 2 websites. All the others will be blocked. Use "\*" to allow all |
| headers | Comma-separated list of headers that are supported by the resource. For example "accept,content-type,origin" will only allow these 3 headers. Use "\*" to allow all. Use null or empty string to allow none |
| methods | Comma-separated list of methods that are supported by the resource. For example "GET,POST" only allows Get and Post and blocks the rest of the methods. Use "\*" to allow all. Use null or empty string to allow none |

The following 2 lines of code in Register() method of WebApiConfig.cs file in App\_Start folder, enables CORS globally for the entire application i.e for all controllers and action methods 

EnableCorsAttribute cors = new EnableCorsAttribute("\*", "\*", "\*");

config.EnableCors();

EnableCors attribute can be applied on a **specific controller** or **controller method.**  
If applied at a controller level then it is applicable for all methods in the controller. To apply it at the controller level  
1. There is no need to create an instance of EnableCorsAttribute in Register() method of WebApiConfig.cs file. Call the EnableCors() method without any parameter values.

**config.EnableCors();**  
  
2. Apply the  EnableCorsAttribute on the controller class

[EnableCorsAttribute("\*", "\*", "\*")]

public class EmployeesController : ApiController

{

}

In the same manner, you can also apply it at a method level if you wish to do so.  
To disable CORS for a specific action apply [DisableCors] on that specific action

# Attribute routing

* **What is Attribute Routing**  
  Using the [Route] attribute to define routes is called Attribute Routing
* **What are the advantages of using Attribute Routing**  
  Attribute routing gives us more control over the URIs than convention-based routing. Creating URI patterns like hierarchies of resources (For example, students have courses, Departments have employees) is very difficult with convention-based routing. With attribute routing all you have to do is use the [Route] attribute as shown below.
* **What is the use of RoutePrefix attribute**  
  RoutePrefix attribute is used to specify the common route prefix at the controller level to eliminate the need to repeat that common route prefix on every controller action method
* **How to override the route prefix**  
  Use ~ character to override the route prefix
* [Route("api/students/{id}/courses")]
* [RoutePrefix("api/students")]

public class StudentsController : ApiController

{

    [Route("api/students")]

    public IEnumerable<Student> Get()

    [Route("api/students/{id}")]

    public Student Get(int id)

    [Route("api/students/{id}/courses")]

    public IEnumerable<string> GetStudentCourses(int id)

[Route("~/api/teachers")]

public IEnumerable<Teacher> GetTeachers()

[Route("{id:int}")]

public Student Get(int id)

[Route("{name:alpha}")]

public Student Get(string name)

[Route("{id:int:min(1)}")]

public Student Get(int id)

[Route("{id:int:min(1):max(3)}")]

public Student Get(int id)

[Route("{id:int:range(1,3)}")]

public Student Get(int id)

public IHttpActionResult Get()

    {

        return Ok(students);

    }

}

* **In addition to Ok() and NotFound() helper methods**, we have the following methods that we can use depending on what we want to return from our controller action method. All these methods return a type, that implements IHttpActionResult interface.
* BadRequest()
* Conflict()
* Created()
* InternalServerError()
* Redirect()
* Unauthorized()

# Versioning of Web API

<http://csharp-video-tutorials.blogspot.com/2017/02/web-api-versioning-using-querystring.html>