Building your First Web API with ASP.NET Core

# Getting Acquainted with ASP.NET Core

ASP.NET Core is an open source, cross platform framework for building cloud based web applications, API’s (Application Programming Interface) and even applications for the internet of things.

ASP.NET Core runs on Windows, Mac and Linux and has been rethought from the ground up as opposed to older versions of ASP.NET which were based on the system.web assembly. ASP.NET core is built on granular nuget packages. We can just use the packages we require which leads to a smaller application surface area. This has the following benefits.

* Tighter security
* Reduced servicing.
* Improved performance.

ASP.NET Core applications can run on either the full .Net framework or the .Net Core framework. The main differences between the two are.

* .Net Core is a modular version of the .Net Framework.
* .Net Core is portable across different platforms. The same code is used across different platforms providing identical functionality.

ASP.NET Core is also and implementation of the .Net standard, which defines a common base layer that all frameworks should support.

Therefore, building an ASP.NET application using the full .Net framework has some disadvantages.

* We lose the modularity of .Net Core
* We lose the small footprint of an app built on .Net Core
* We lose the performance Improvements gained from the modular design
* We lose any cross platform functionality.

For these reasons it is recommended to use .Net Core when building ASP.NET Core applications. The .Net framework should only be used if its specifically required.

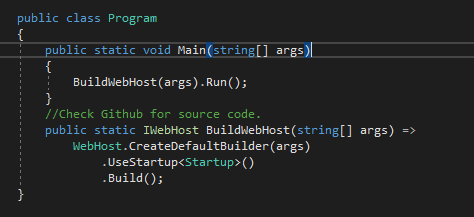
# Creating the API and Returning Resources

An ASP.NET Core application is actually a Console App that calls into specific ASP.NET libraries.

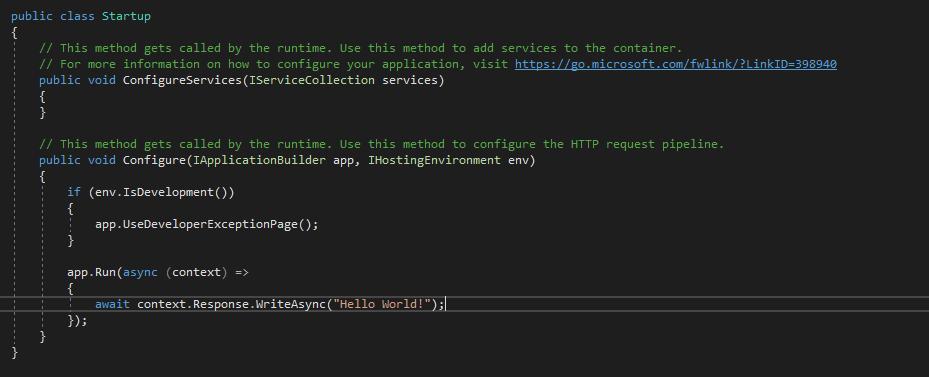
HTTP Request Pipeline



When a client makes a request to the API the request is sent through the ASP.NET Http request pipeline and a response is sent back. Middleware is added to the pipeline to add specific functionality. E.g. MVC is a middleware which is added. Each component chooses if the next is called so it is imported that the order is correct. E.g. Authentication middleware is above MVC middleware.

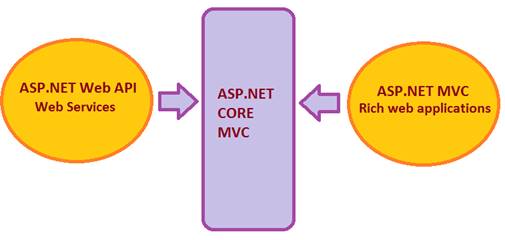


The program class is the starting point of the application. The main method configures and runs the application. Any changes to the way the application is run and built should be made here.



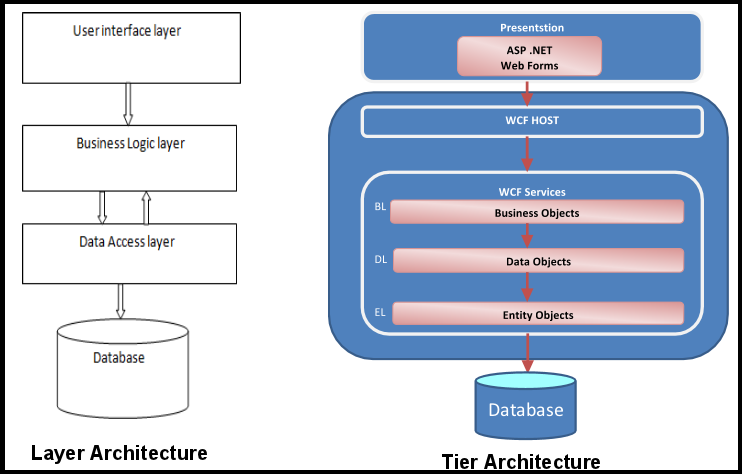
The Startup class is the entry point of the application. The services method is used to add framework services to the container. A service is a component that is intended for common consumption in an application. The Configure method is used to specify how an ASP.NET Core application will respond to individual HTTP requests. Middleware is added here.

The MVC Pattern



In ASP.NET core, the old Web API and MVC frameworks have been combined in ASP.NET Core MVC.

MVC is an Architectural pattern used for implementing user interfaces. It encourages loose coupling, separation of concerns, testability and reusability. It is not a full application architecture.



In a typical N-Tier architecture MVC is used in the presentation layer. Angular could be seen as a client side MVC application.



In an API the returned JSON/XML is the view in this model.

Metapackages and the Runtime Store

A new feature of ASP.NET Core 2.0 is the addition of meta packages. When a .Net Core 2 .0 application is created by default it references the AspNetCore.All meta package which is a collection. This meta packages includes references to all ASP.NET Core packages and all Entity Framework Core packages. This packages are stored in the runtime store which is a location on the disk of the machine. These packages are used by all applications installed on the machine which reference the meta package

The advantages of this are

* Faster deployments due to less packages in the route of the application
* Lower disk use.

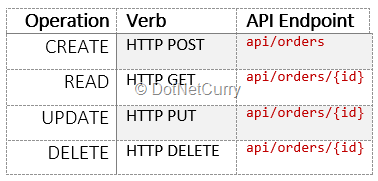
Because ASP.NET Core 2.0 apps are framework dependent this means the correct version of the runtime store must be installed on the machine we are publishing too. This means these apps are not self-contained. This is an optional feature and we can still reference each package individually.

Routing

There are two methods of routing requests in an ASP.Net Core application.

* Convention based routing
* Attribute based routing

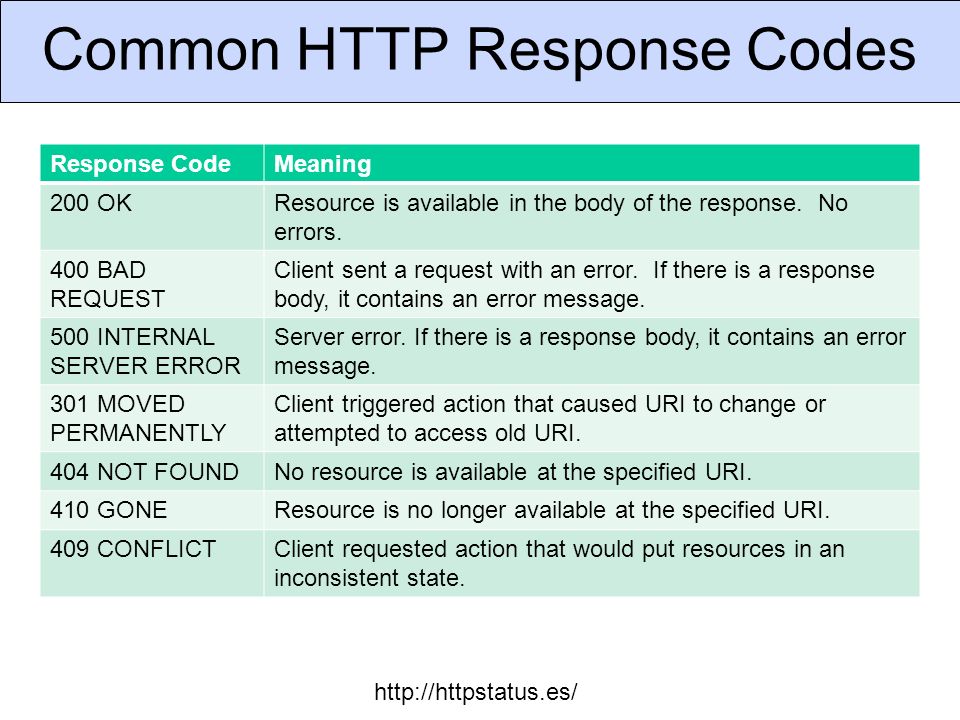
Convention based routing would be defined in the app.useMcv() defined in the Configure method of the startup class and would be more applicable in a web application. For APIs, Microsoft recommends attribute routing.



Above is the list of attributes for CRUD operations with examples of their endpoints.

Status Codes

Because the status code is the only way for the client to know if their request was successfully handled it is important that the correct status codes are returned. E.g. when a resources is not found.



Asp.net provides the option to display status code pages. This is useful when calling an API in a browser when testing because if this option is not enabled, a developer must use the developer tools provided by the browser to inspect the status code.

Serializer settings

Older Asp.Net APIs would usually return results based on class names which represented the returned resource. E.g. Id rather the id. The results returned by a .Net Core API are camel case. This is configurable when referencing the MVC service in the Startup Class.

Formatters and Content Negotiation

Clients can specify different format types using the Accept key in the Header of the request. If the application is not configured to return the specified type it will return a default type, usually JSON. Clients can also pass an output formatter in the header of the request using the content-type key. The output formatter specifies the posted media type.

# Manipulating Resources

* Creating – Post, returns 201 Created
* Updating - Put/Patch, returns 204 No Content or 200 Ok
* Deleting – Delete, returns 204 No Content

Validating Input

Check out [FluentValidation](https://github.com/JeremySkinner/FluentValidation) for building validation errors. It allows you to define validation in one place as opposed to using annotations in DTO’s and checks in code controllers.

See PointsOfInteresrtControllor in sample solution for examples of creating and deleting a Resource.

Updating a Resource

The HttpPut method is used for fully updating a resource. It is expected that every attribute will be updated. Returns a 204 No Content if successful.

Partially Updating a Resource

The HttpPatch method is used for partial updates. It requires a field to patch, a value and an operation.

The standard format for this is JSON patch RFC 6902. It defines a JSON document structure for expressing a sequence of operations to apply to a JSON document. The consumer creates a JSON patch document containing a list of operations as the body of the patch request adhering to the standard.

Examples of operations are at, replace, copy. An API should accept the patch document and apply operations to the resource.

Sample Patch Document

[

{

"op": "replace",

"path": "/name",

"value": "patched name"

}

]