# Contents

[Contents 1](#_Toc8763037)

[A Modular Approach 6](#_Toc8763038)

[Net Core Plugin Manager 9](#_Toc8763039)

[Sub Website 9](#_Toc8763040)

[API Controller 10](#_Toc8763041)

[Middleware Extensions 10](#_Toc8763042)

[Services 10](#_Toc8763043)

[Custom Interfaces 10](#_Toc8763044)

[Embedding Resources 11](#_Toc8763045)

[Installing the Plugin Manager 12](#_Toc8763046)

[Configuring the Plugin Manager 12](#_Toc8763047)

[PluginSettings 15](#_Toc8763048)

[PluginSetting 16](#_Toc8763049)

[Inter Plugin Event Notification 17](#_Toc8763050)

[INotificationService 17](#_Toc8763051)

[Methods 17](#_Toc8763052)

[RegisterListener 17](#_Toc8763053)

[UnregisterListener 17](#_Toc8763054)

[RaiseEvent 18](#_Toc8763055)

[Parameters 18](#_Toc8763056)

[INotificationListener 18](#_Toc8763057)

[Methods 18](#_Toc8763058)

[GetEvents 18](#_Toc8763059)

[EventRaised 19](#_Toc8763060)

[Bad Egg Plugin 20](#_Toc8763061)

[Request Limits 20](#_Toc8763062)

[Sql Injection and Hack Attacks 21](#_Toc8763063)

[Black/White Lists 21](#_Toc8763064)

[Firewalls 21](#_Toc8763065)

[Breadcrumb Plugin 23](#_Toc8763066)

[Cache Control Plugin 24](#_Toc8763067)

[Company Plugin 25](#_Toc8763068)

[Error Manager 26](#_Toc8763069)

[Geo Ip Plugin 27](#_Toc8763070)

[Helpdesk Plugin 28](#_Toc8763071)

[Login Plugin 29](#_Toc8763072)

[Memory Cache Plugin 30](#_Toc8763073)

[Plugin Middleware 31](#_Toc8763074)

[Products Plugin 32](#_Toc8763075)

[Restrict Ip Plugin 33](#_Toc8763076)

[Seo Plugin 34](#_Toc8763077)

[Shared Plugin Features 36](#_Toc8763078)

[IBreadcrumbService 37](#_Toc8763079)

[IConfigureApplicationBulder 38](#_Toc8763080)

[IConfigureMvcBuilder 39](#_Toc8763081)

[ICultureProvider 40](#_Toc8763082)

[IErrorManager 41](#_Toc8763083)

[IGeoIpDataService 42](#_Toc8763084)

[IGeoIpProvider 43](#_Toc8763085)

[IGeoIpStatisticsUpdate 44](#_Toc8763086)

[IInitialiseEvents 45](#_Toc8763087)

[IIpManagement 46](#_Toc8763088)

[IIpValidation 47](#_Toc8763089)

[ILoadSettingsService 48](#_Toc8763090)

[ILogFile 49](#_Toc8763091)

[IMemoryCache 50](#_Toc8763092)

[IPlugin 51](#_Toc8763093)

[IPluginClassesService 52](#_Toc8763094)

[IPluginHelperService 53](#_Toc8763095)

[Methods 53](#_Toc8763096)

[PluginLoaded 53](#_Toc8763097)

[AddAssembly 53](#_Toc8763098)

[IPluginTypesService 54](#_Toc8763099)

[IPluginVersion 55](#_Toc8763100)

[IRouteDataService 56](#_Toc8763101)

[Methods 56](#_Toc8763102)

[GetRouteFromClass 56](#_Toc8763103)

[GetRouteFromMethod 56](#_Toc8763104)

[ISeoProvider 57](#_Toc8763105)

[ISettingsProvider 58](#_Toc8763106)

[ISharedPluginHelper 59](#_Toc8763107)

[IShoppingCartSevice 60](#_Toc8763108)

[ISystemAdminHelperService 61](#_Toc8763109)

[IUserCultureChangeProvider 62](#_Toc8763110)

[IUserSessionService 63](#_Toc8763111)

[BaseController 64](#_Toc8763112)

[BaseCoreClass 65](#_Toc8763113)

[BaseMiddleware 66](#_Toc8763114)

[BaseModel 67](#_Toc8763115)

[Breadcrumb 68](#_Toc8763116)

[BreadcrumbItem 69](#_Toc8763117)

[CaptchaImage 70](#_Toc8763118)

[ErrorInformation 71](#_Toc8763119)

[Events 72](#_Toc8763120)

[MainMenuItem 73](#_Toc8763121)

[SystemAdminSubMenu 74](#_Toc8763122)

[ShoppingCartSummary 75](#_Toc8763123)

[StopWatchTimer 76](#_Toc8763124)

[SystemAdminMainMenu 77](#_Toc8763125)

[Timings 78](#_Toc8763126)

[Attributes 79](#_Toc8763127)

[BadEgg 79](#_Toc8763128)

[Breadcrumb 79](#_Toc8763129)

[DenySpider 79](#_Toc8763130)

[LoggedIn 79](#_Toc8763131)

[LoggedInOut 79](#_Toc8763132)

[LoggedOut 79](#_Toc8763133)

[RestrictedIpRoute 79](#_Toc8763134)

[Shopping Cart Plugin 80](#_Toc8763135)

[SieraDelta Geo Ip Plugin 81](#_Toc8763136)

[Spider Plugin 82](#_Toc8763137)

[System Admin Plugin 83](#_Toc8763138)

[User Account Plugin 84](#_Toc8763139)

[User Session Plugin 85](#_Toc8763140)

[References 86](#_Toc8763141)

# A Modular Approach

Many software projects suffer from a lack of initial design, while this approach allows developers to quickly create and release their software, it often creates a post initial release nightmare when it comes to updating, extending or fixing bugs. There are many reasons for throwing things together quickly, primarily I would argue that market and financial pressure are chief amongst them. However, there comes a time when the cost of updating or enhancing a product far outweighs the lack of initial design.

There are several principles in good software design-architecture which are critical to the success and longevity of a project, chief amongst these is the following:

N-Tier: A client–server architecture in which presentation, application processing, and data management functions are physically separated into layers.

SOLID: Single responsibility, open-closed principle, Liskov substitution principle, interface segregation and dependency inversion principle.

DRY: Don't repeat yourself. Software development principle aimed at preventing repetition of code.

IoC: Inversion of control is used to increase modularity and extensibility through dependency injection.

When software is badly designed some of the principles above are missing entirely, or in other cases all tier's are housed within the same executable or assembly. This is where things begin to fall down and maintenance becomes more of a headache and ultimately more expensive in the long run.

The problem is magnified when an entire product base is copied and pasted to form the basis of a new solution for a new client. Whilst this gives an initial leap in development time, it also means that the maintenance of the products base is doubled. Fixing a bug in one product ultimately means you have to copy and paste the fix to each derivative solution.

Whilst different client solutions can diversify and become more generic to the client, there are still many aspects which are jointly owned. For instance the loading of application settings will not generally differ, however the settings being loaded could differ. Likewise storing application behaviour data such as debug information or process flow will chiefly remain the same. The elements that may differ are when it comes to displaying specific product information or a user’s account could have different account options.

If we look at a single website built using C# with the model-view-controller (MVC) pattern there will typically be many modules that make up the software, for instance you could have:

* Product
* Shopping Cart
* Stock Control
* User account
* Searching
* Marketing
* Settings

Each of these modules makes up the first tier or layer; this is effectively known as the Presentation Layer (PL).

The next layer in the system is the Business Object Layer (BOL), within this layer we provide the business logic and interface with the data access layer to enable us to present the data. Each module within the PL has a corresponding module within the BOL.

The next layer is known as the Data Access Layer (DAL), this layer interacts via the BOL to ensure all data (CRUD) operations take place in a reliable way. Again, each BOL module has a corresponding DAL module.

Based on the above example a basic website would have 7 modules within each layer. Each module has responsibility for a specific area and could easily span multiple projects. If we take a single module and share it between multiple projects we have the benefit of reducing the code base, the cost of development and ultimately increasing the maintainability of the product.

If or when it is time to give our website a makeover we only have to focus on a single layer within the system, this being the PL. Likewise if a client has specific needs to use a specific data store then only the DAL would need to change.

Plugin technology is a natural proponent to the above principles, it does not replace any of them, instead it enhances them and allows developers to extend websites using SOLID and DRY principles. Each module within system becomes a plugin. The plugin manager is responsible for loading the modules and ensuring they are registered within the MVC architecture. Once registered, the module would behave like an internally housed MVC component.

Net Core Plugin manager also provides its own interfaces which allow the host application to query plugin modules, this allows for a joined up approach when the system fit's together as a final solution.

A further advantage of using plugins is that each plugin becomes self-registering, when a plugin is loaded it can register it's capabilities and interfaces within the IoC container, this further removes essential tasks from the developer, freeing them up to concentrate on creating enhancements or fixing bugs.

Rewriting an entire application for a new architecture is not always a good approach, there are quite a few case studies and articles on why it could be a bad idea. Chief amongst these arguments would be that you already have a code base that is widely used, tested and bug fixed. This however should not prevent you from refactoring your existing code base to make it modular, enforce the use of the SOLID and DRY principles and begin a journey into true modular design that could then benefit from plugin technology.

It is never too late to expand a little time and energy on refactoring your existing code, in many cases it involves nothing more than decoupling different modules from the main application and housing them within their own assemblies. In the first instance these can be statically linked to the application with the benefit that you can begin to ensure your application is more modular, more extensible and fits within the aforementioned development principles.

The benefits of refactoring existing code into a modular design has to be a good thing, too often an application is so tightly coupled and intertwined that maintenance becomes an issue. A good example of this tight coupling is where IoC is advocated and used when creating object instances using a DI container, however the implementation is flawed when both the interface and interface implementations are housed within the same module. The disadvantage of this approach being that the intended loose coupling is now tightly coupled within the application.

Replacing the interface implementation with another implementation is nye on impossible without further tightly coupling the application, and preventing any type of code reuse. In this circumstance the interfaces should reside within their own assembly and the implementation should reside within another assembly.

The benefits of decoupling an application are many, in computing new technologies are continually being developed, whilst upgrade paths may not always be easy, if you have a well-designed, modular application, you can easily rewrite aspects to use a different technology without having to sacrifice the entire existing code base.

There is also a huge downside when developing non modular applications, if an application is too tightly coupled and relies on a specific technology, as the technology progresses it becomes more and more difficult to upgrade. The decision to upgrade is postponed as the amount of development and testing time increases, market and business pressures come into effect and the once dazzling application suddenly has a limited life span and becomes outdated rather quickly.

In a continually evolving sector, developers will inevitably upgrade their own skill base. Knowledge of older technology can eventually be lost over time resulting in a dwindling number of people with the required knowledge and expertise to maintain the older technology. Employers will find it more difficult to attract developers in order to maintain their application and the cost of development will rise exponentially.

With a well-designed modular application the task of upgrading a single layer of the application is no more difficult than replacing a single module or layer, much of the existing code can be reused and reutilised.

# Net Core Plugin Manager

Net Core Plugin Manager has been developed to augment a modular design, it's primary purpose is to load modules from multiple assemblies and ensure they are registered within the MVC pipeline. This approach ensures that websites can easily expand as new features are bought online whilst allowing shared modules to be reused across multiple implementations. The primary benefit being that well tested code can be reused within multiple websites.

There are many types of plugin modules that can be created for use within a Net Core MVC application, they include

* Sub website. Individual Controller, models and views for a specific module such as product, company services, shopping cart, login or user account.
* API controller. Add individual or multiple APIs to a website, this could include individual versions of a specific API.
* Middleware extensions. Easily include middleware classes that can control or affect the request pipeline.
* Services. Add services for IoC.
* Custom interfaces.

## Sub Website

A good example of a sub website plugin module would be a login plugin which allows people to log into a website. The act of logging in is quite simple, you provide a user with two inputs, a username and password within the UI, upon entering and clicking submit the application will verify these details and allow the user to proceed or not. The implementation for logging in is handled by the BOL, which in turn talks to the DAL and a decision is returned.

In most circumstances it is fairly simple, however this could easily be upgraded within the UI to accept a captcha code, there is no BOL changes for this as it is handled within the UI alone. The UI can verify the code and if successful pass the request on to the BOL. It could be that security needs strengthening further to accept a unique code sent to the users via mobile or email after logging in on a new terminal or device. This task would occur within the BOL, upon successful login the BOL sends the required message via email or SMS, the user enters the code within the UI and proceeds. None of the required upgrades have necessitated any changes to the DAL, who's only responsibility is to validate the username and password.

The required UI login module can contain the necessary code for captcha, it is enabled if the settings mandate it. The UI code for displaying a verification code is always present and will only be shown if the correct response is returned from the BOL. This login module can easily be shared across multiple applications, the options for how it operates can be based on settings and configuration.

## API Controller

If you provide an API via a website you could easily support multiple versions in individual plugin modules. Versioning can be implemented using specific routes, as an API is deprecated it can be removed or disabled using configuration settings.

## Middleware Extensions

Middleware extensions are used to affect or control the pipeline used for requests in asp.net core. Each middleware plugin can process part or all of the request, and then either choose to return the result or pass on down to the next piece of middleware. Middleware extensions can and are often used across multiple websites. Some examples of potential middleware extensions include:

* Adding security headers.
* User session management.
* Extracting marketing data.

Note: Plugin middleware extensions can be used without including the plugin manager, manual registration of their services should take place. There may or may not be dependencies on other assemblies which also need to be registered.

## Services

Plugin modules could be created with the specific purpose of implementing individual or multiple interface for use within an IoC container. For instance you could provide the BOL for system settings using a specific module, this registers itself and is available for use within the application.

## Custom Interfaces

Custom interfaces offer the greatest flexibility when developing an application, there are many possible uses. There are a multitude of uses for extending an interface, for instance you could have a menu system which is dynamically populated depending on the plugins that are loaded. Another use would be to display carousel items depending on active marketing campaigns, the list is literally as open as the website being developed.

Plugin Resources

Each plugin module can contain all the resources that it needs in order that it can operate as expected. Examples of resources include:

* .cshtml files. The individual full or partial view files required to view a web page.
* .css files. The css files used for displaying a web page within a browser.
* .js files. Javascript files used for client side scripting when viewing pages within a browser.
* Image Files. Any image file required by the plugin.

Resources are an important aspect of any web page, they determine how a page is viewed, the layout, how it operates in the client browser and compliment the overall design.

It is important to point out at this point that although individual webpages can be viewed in their default state, the entire layout could be altered by a web designer by substituting one or more of the default resource files required. This approach allows for greater flexibility as developers can develop a website with a default look or view, designers could then alter the layout to provide for many different views. To prevent custom files being overridden please view the PluginSetting section to show how this can be achieved.

## Embedding Resources

To embed resources, these being .cshtml files, images, css or script etc, right click the item in the solution explorer, select properties and choose Embedded Resource as the build action.

\*IMAGE\*

An important caveat on embedding resources is in their naming. Due to how C# embeds files they cannot contain a full stop character (.) in their name. When compiling the compiler will use the path and file name as the name of the embedded resource. For instance if you had an image that was displayed on the login screen and this was located at:

[ProjectPath]\wwwroot\images\login\loginimage.png

When compiled as an embedded resource the name would become:

wwwroot.images.login.loginimage.png

When extracted it would be in the same folder location as it was when it was compiled as an embedded resource. If the image was called:

[ProjectPath]\wwwroot\images\login\login.image.png

When extracted it would reside in the following folder:

[ProjectPath]\wwwroot\images\login\login\image.png

The result would be a 404 error when viewed within a browser.

# Installing the Plugin Manager

AspNet Core Plugin Manager is available from nuget and can be installed from:

PM> Install-Package AspNetCore.PluginManager -Version 2.1.0

Where version 2.1.0 represents the latest current version. It is recommended that you upgrade to the latest available version for your platform. Currently the plugin manager supports:

* Net Core 2.1
* Net Core 2.2
* Net Core 3.0
* Net Framework 4.6.1

# Configuring the Plugin Manager

In order to begin using the plugin manager it must be initialised. By default the settings for the Plugin Manager are contained within appsettings.json, this however can be overridden by using a custom ILoadSettingsService implementation. A custom ILoadSettingsService implementation could be used to gather configuration settings from a different file, a database, environment variables or whatever type of storage you use.

As well as the ILoadSettingsService configuration can include a custom ILogger implementation, this simple interface is passed to all plugin modules and allows for a simple and unified method of logging messages, warning and errors.

Default implementations of both ILoadSettingsService and ILogger exist and will be used should no custom implementations be required.

PluginManagerService is a static class contained within Module: AspNetCore.PluginManager.dll; You can add a using directive in the header of your code or directly reference within code.

To initialise the plugin manager you need to call the initialise method, this is done when the application first starts in the static Main method:

public class Program

{

public static void Main(string[] args)

{

// Initialise the plugin manager service

PluginManagerService.Initialise();

CreateWebHostBuilder(args).Build().Run();

}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>();

}

Plugin modules can be self configuring, in that they can register any services they provide and configure the MVC application, in order to facilitate this the next step to integrating the plugin manger is to add the required steps within the ConfigureServices and Configure methods:

public void ConfigureServices(IServiceCollection services)

{

// Allow plugin manager to configure all services in each plugin

PluginManagerService.ConfigureServices(services);

// other intialisation code here

services.AddMvc()

.SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1)

.ConfigurePluginManager();

}

The first step above makes a call to the plugin managers ConfigureServices method, this in turn will complete the following tasks:

* Find all plugin modules that implement the IInitialiseEvents interface and call the before initialise method.
* Call the IPlugin configure services method to ensure that each plugin registers the services they provide.
* Call the after initialise method on all plugins which implement the IInitialiseEvents interface.

The call to ConfigurePluginManger() ensures that three important steps are completed, they are:

* Ensuring that the plugin manager integrates correctly within the MVC configuration by replacing the MetadataReferenceFeatureProvider with a custom implementation.
* Load any views that have been precompiled into their own assemblies.
* Call each plugin module which implements IConfigureMvcBuilder interface thereby allowing the plugin to also configure the MVC Services.

The next and final step in configuring the plugin manager is to integrate with the MVC application, this is completed in a similar fashion to configuring MVC services:

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

// Allow plugin manager to configure options for all plugins

PluginManagerService.Configure(app, env);

// other MVC application options

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

}).UsePluginManager();

}

The first step is to call the plugin service Configure method, this method performs three actions, they are:

* Find any plugins which implement the IInitialiseEvents interface and call the before initialise method.
* Call the Configure method for all plugin modules, allowing them to configure the services they provide.
* Call the after initialise method for all plugins which implement the IInitialiseEvents interface.

The next and final step in configuring the MVC application is to call the UsePluginManager method, this will call the configure method for all plugin modules that implement the IConfigureApplicationBuilder interface.

As can be seen only five lines of code are required to integrate the plugin manager into any Net Core application. The next step for integration involves configuration using the PluginSettings class.

## PluginSettings

PluginSettings contains properties that can be used to configure how it is used and what plugins are loaded.

Module: AspNetCore.PluginManager.dll

Disabled. boolean - If set to true the plugin manager is disabled and will not operate.

PluginPath. string - The path where plugin modules are contained. All assemblies within the path will be loaded during initialisation, if no IPlugin interface is found within the assembly it will not be added to the list of available plugins.

PluginSearchPath. string - Root path to be searched when looking for plugin modules. This is typically used in conjuction with the PluginFiles property. All sub directories will also be searched until a match is found or the file does not exist.

SystemFiles. boolean - TBC

CSSLocation. string - Root path where css files extracted from plugin modules will be saved.

JScriptLocation. string - Root path where css files extracted from plugin modules will be saved.

PreventAreas. boolean - Not currently used.

DisableRouteDataServices. boolean - TBC.

PluginFiles. string[] - Array of plugin modules. If the string does not include a path to where the plugin module can be found, PluginPath is used to search for the module. Generally you would only need to specify the names of plugins which required a specific load order. This could include those plugin modules which contained middleware extensions which were required to run before other middleware extensions, or similar circumstances.

Plugins. List<PluginSetting> - Individual plugin configuration settings. See PluginSetting.

## PluginSetting

The PluginSetting class defines properties that affect how an individual plugin module is loaded.

Module: AspNetCore.PluginManager.dll

Name. string - Name of the plugin module assembly, e.g. MyPlugin.dll

Disabled. boolean - Determines whether the plugin is disabled or not, if true the plugin module will not be loaded within the plugin manager.

PreventExtractResources. boolean - If true then resource files will not be extracted from the plugin module.

ReplaceExistingResources. boolean - If true resources that have been previously extracted from a plugin module will be replaced each time the plugin module is loaded.

Version. string - TBC.

# Inter Plugin Event Notification

The plugin manager contains an interface which enables inter plugin notifications. Individual plugins can register one or multiple classes which can be used to obtain notifications throughout the application lifecycle. This can be useful when you need to receive updates or notify listeners that a specific event has occurred.

Notifications are completely user defined, the event id used is a unique string.

Being able to send and receive notifications can enables different plugins to communicate in a generic manner, whilst allowing for custom data to also be communicated. This could be useful in many circumstances, for instance: A plugin module caches certain information for speed of retrieval, another plugin could update this static information however, without any notifications the cache would end up with stale data. The plugin module which caches the data can register an instance of INotificationListener which listens for an event called “CacheUpdated”, when the data is updated the updating plugin module obtains an instance of INotificationService and calls the RaiseEvent method. The listening plugin module receives the notification and empties the cache.

## INotificationService

This interface is designed to allow callers to register and unregister INotificationListener objects and raise events throughout the system.

Acquisition Method: Dependency Injection

### Methods

#### RegisterListener

Result: boolean – True if the listener completes registration, otherwise false.

Registers an INotificationListener for receiving notifications.

##### Parameters

listener: in INotificationListener - The class being registered.

#### UnregisterListener

Result: boolean – True if the listener is unregistered, otherwise false.

Unregisters an INotificationListener class from receiving notifications.

##### Parameters

listener: in INotificationListener – The class being registered.

### RaiseEvent

Result: void. This method has no return value.

Raises an event that will be broadcast to all listeners.

#### Parameters

eventId : in string – Name of the event.

param1: in object – user defined parameter value.

param2: in object – user defined parameter value.

## INotificationListener

This interface allows plugin modules to register their interest in specific events that could occur within the system and respond accordingly. Events are completely user defined and allow, through the use of generic parameters to pass event specific data for the event being raised.

Classes which implement INotificationListener need to obtain an instance of INotificationService in order to register themselves so as events can be passed.

### Methods

#### GetEvents

Result: List<string> - user defined list of events which the inteface instance is interested in receiving.

This method is called after a call to RegisterListener, if the function returns null, an empty list or a list which contains an empty or null string then an Invalid Operation exception will be raised.

An instance of this interface can register multiple events.

##### Parameters

This function does not contain any parameters.

#### EventRaised

Result: bool – return true to allow other INotificationListeners to continue to receive notifications or false to prevent other listeners from receiving notifications.

This method is used to notify all registered listeners that an event has been raised.

##### Parameters

eventId: in string – the name of the event being raised.

Param1: in object – user defined parameter for the event being raised.

Param2: in object – user defined parameter for the event being raised.

# Bad Egg Plugin

Nobody likes it when people don't play fair; the bad egg plugin is designed to complete several functions:

* Limit number of requests per minute.
* Determine the probability that a Sql injection attack is taking place.
* Determine the probability that a hack attempt is taking place.
* Allow for individual Ip addresses to be white/black listed.

The important element to take into consideration is that Bad Egg does not make decisions on its own, in conjunction with IIpValidation which is implemented by the host application, reports are made through the interface which include:

* Ip Address.
* Query/Form data being submitted.
* Result of validation.

Given the information the host implementation can decide to ban the Ip address, if it wishes.

To prevent unnecessary delays in the request pipeline, Bad Egg middleware employs a background thread which is used to communicate with the IIpValidation implementation, this means there could be a slight delay in a ban being requested and actioned or a report being made for a request so as the host application can make decisions.

The module is available from nuget:

PM> Install-Package BadEgg.Plugin -Version 2.1.0

## Request Limits

A normal web user would not complete too many web requests within a one minute period, excluding static files like images, css files and java script files, the number of requests would remain relatively low. Bad Egg middleware monitors the number of requests being sent, excluding static files, and if the average amount exceeds the value set by the host application then a http response of too many requests (429) is returned. As soon as the average hit rate (default of 100) falls below the rate set then pages will be served once more.

This will help prevent the server becoming overloaded by hacking, sql injection attacks, denial of service or rogue spider bots that are badly designed.

Extensive tests have been conducted and on average a google bot will submit less than 4 requests per minute, on average. Not all bots are this well behaved and the request limits can help combat this cyber nuisance.

The default number of requests is an average of one hundred a minute, the minimum value allowed is five and goes up to the maximum value for an unsigned int.

## Sql Injection and Hack Attacks

Despite better design within web applications, Sql injection attacks are still quite prevalent within the web, hacking attempts continue to be attempted as exploits are realised in standard off the shelf solutions. Whether you utilise standard web based solutions or not, these types of attacks can become annoying, they utilise valuable server resources serving requests and can ultimately degrade the performance of your web application.

Bad Egg employs a highly optimised, unique algorithm, which automatically validates query strings and form input on specific routes within your web application. It validates form and query string input data for known keywords that can be used to hack websites. Given the scale of words being submitted this is weighed against the probability that an attack could be underway. Again, it is down to the host application to act upon the information provided by the Bad Egg middleware.

To include automatic form and query string validation, the attribute BadEgg needs to be included on the public action method receiving the input. The following simplified code example, demonstrates the BadEgg attribute being used on the Login method

[HttpPost]

[BadEgg]

public IActionResult Index(LoginViewModel model)

{

// login validation code...

return View(model);

}

## Black/White Lists

Bad Egg middleware does not maintain a black/white list of Ip addresses, instead it holds a list of ip addresses supplied by the host application. The host application can request an instance of IIpManagement via the IoC container, this interface has methods for adding black/white list of addresses.

The start of the validation process is used to determine whether the ip address is black/white listed, if it is then no further validation is completed. White listed ip addresses are always excluded from validation, no matter how badly they behave whilst requests from black listed addresses are always prevented from continuing down the request pipeline by providing an http error response (default value of 400, invalid request).

## Firewalls

Whilst the Bad Egg middleware provides a good software based solution to banning certain ip addresses, there is no substitution to banning an ip address using your firewall. Using a firewall prevents the server from wasting valuable resources and in turn will help ensure you web application runs at an optimal rate.

Using integrated services, could be a way of ensuring that badly behaved ip addresses identified by Bad Egg middleware are passed to the firewall for server blocking.

# Breadcrumb Plugin

# Cache Control Plugin

# Company Plugin

# Error Manager

# Geo Ip Plugin

# Helpdesk Plugin

# Login Plugin

# Memory Cache Plugin

# Plugin Middleware

# Products Plugin

# Restrict Ip Plugin

# Seo Plugin

# Shared Plugin Features

The shared plugin features module contains the interfaces, base classes and abstract classes that can be shared between all plugins.

The module is available from nuget:

PM> Install-Package SharedPluginFeatures -Version 2.1.0

## IBreadcrumbService

## IConfigureApplicationBulder

## IConfigureMvcBuilder

## ICultureProvider

## IErrorManager

## IGeoIpDataService

## IGeoIpProvider

## IGeoIpStatisticsUpdate

## IInitialiseEvents

The IInitialiseEvents interface allows plugins to optionally receive notification for configuring applications and services when loading.

A plugin module should only implement this interface if it is

## IIpManagement

IIpManagement interface is implemented using the Bad Egg plugin and allows host applications to provide a list of black or white listed Ip addresses, which are queried when connections are initially created during the request pipeline. The host application is responsible for maintaining the list, and loading the initial data when the application is started.

Acquisition Method: Dependency Injection

### Methods

#### AddBlackListedIp

Result: void.

Adds an individual ip address to the list of ip addresses which are banned.

Parameters

ipAddress: in string – Ip address to be black listed.

#### AddWhiteListedIp

Result: void.

Adds an individual ip address to the list of ip addresses which are never validated.

##### Parameters

ipAddress: in string – Ip address to be white listed.

#### RemoveIpAddress

Result: void.

Removes an individual ip address from the list of ip addresses which are black/white listed.

##### Parameters

ipAddress: in string – Ip address to be removed.

#### ClearAllIpAddresses

Result: void

Removes all ip addresses from the list.

##### Parameters

none.

## IIpValidation

IIpValidation is implemented by the host application and is called by the Bad Egg middleware in response to events taking place. The host is responsible for ensuring that the implementation is added to the IoC container for use when required.

Implementing this interface allows a host to provide a rich set of data for Ip address reporting, as well as obtaining data on requests, query strings and form data being submitted through the web application.

Acquisition Method: Dependency Injection

### Methods

#### ConnectionAdd

Result: void.

Notifies the host that a new Ip address has connected and is being monitored.

Parameters

ipAddress: in string – Ip address that is being monitored.

#### ConnectionRemove

Result: void.

Notifies the host that an existing Ip address has not made a request for several minutes and that it will be removed from the list of Ip addresses being monitored.

Parameters

ipAddress: in string – Ip address that is being monitored.

hits: in double – The average requests per minute made from the Ip address.

Requests: in ulong – The total number of requests made.

Duration: in TimeSpan – Total time the ip address was active from the first request to the last request made.

## ILoadSettingsService

## ILogFile

## IMemoryCache

## IPlugin

The IPlugin interface is used to identify a plugin module, each assembly which wants to act as a plugin module is required to provide a single instance of this interface. There are four methods within the interface which can be used to initialise the plugin, they are:

* Initialise. This method is called when the module is first loaded, an ILogger instance is passed via the method which allows for basic logging facilities.
* Finalise. This method is called prior to the plugin being unloaded, a plugin should use this method to free up any resources that it may have allocated.
* Configure. This method is used to allow the plugin to configure itself within the MVC application architecture.
* ConfigureServices. This method is used to allow the plugin to configure any MVC services it may contain.

## IPluginClassesService

## IPluginHelperService

This interface is implemented by the plugin manager and allows plugin modules to retrieve information or set status of plugin modules.

Acquisition Method: Dependency Injection

### Methods

#### PluginLoaded

Result: bool - True if plugin module loaded, otherwise false.

Determines whether a specific plugin is loaded or not. If a plugin requires the services of another plugin it can query this function to obtain load status and version.

Parameters

PluginLibraryName: in string - Name of the assembly dll, i.e. myplugin.dll.

Version: out int - The version of the plugin if found.

#### AddAssembly

Result: void

Allows a plugin module to dynamically make any assembly a plugin. This can be useful should the need arise to discover specific class types. See TBC

Parameters

assembly: in Assembly - The assembly to be loaded as a plugin.

## IPluginTypesService

## IPluginVersion

The IPluginVersion interface allows a plugin to identify which version of the plugin manager it supports. Currently the only version is 1. If an instance of this interface is not provided then the plugin manger will assume version 1.

## IRouteDataService

This interface is implemented by the plugin manager and allows plugin modules to retrieve route data for methods and classes by querying the Route attribute or standard MVC routing table.

Acquisition Method: Dependency Injection

### Methods

#### GetRouteFromClass

Result: string - The route used by the controller or api class, e.g. /products or /api/version/2/products.

Given a class type of controller or api controller, this method will obtain the MVC route used by the class.

##### Parameters

type: in Type - The class type to be queried.

routeProvider: IActionDescriptorCollectionProvider - An instance of the ActionDescriptorCollectionProvider. This class can be obtained using dependency injections.

#### GetRouteFromMethod

Result: string - The route used by the controller method, e.g. /products/get/

Given a method from a controller class, provides the MVC route that is used.

method: in MethodInfo - MethodInfo class for the method being queried.

routeProvider: in IActionDescriptorCollectionProvider - an instance of the ActionDescriptorCollectionProvider. This class can be obtained using dependency injection.

## ISeoProvider

## ISettingsProvider

## ISharedPluginHelper

## IShoppingCartSevice

## ISystemAdminHelperService

## IUserCultureChangeProvider

## IUserSessionService

## BaseController

## BaseCoreClass

## BaseMiddleware

## BaseModel

## Breadcrumb

## BreadcrumbItem

## CaptchaImage

## ErrorInformation

## Events

## MainMenuItem

## SystemAdminSubMenu

## ShoppingCartSummary

## StopWatchTimer

## SystemAdminMainMenu

## Timings

## Attributes

### BadEgg

### Breadcrumb

### DenySpider

### LoggedIn

### LoggedInOut

### LoggedOut

### RestrictedIpRoute

# Shopping Cart Plugin

# SieraDelta Geo Ip Plugin

# Spider Plugin

# System Admin Plugin

# User Account Plugin

# User Session Plugin

# References

<https://apiumhub.com/tech-blog-barcelona/benefits-of-software-architecture/>

<https://en.wikipedia.org/wiki/Plug-in_(computing>)

<https://stackoverflow.com/questions/2818415/what-are-the-advantages-and-disadvantages-of-plug-in-based-architecture>

<http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html>

<http://jpf.sourceforge.net/index.html>

<https://en.wikipedia.org/wiki/Second-system_effect>

<https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/>

<http://notes-on-haskell.blogspot.com/2007/08/rewriting-software.html>

<http://www.neilgunton.com/doc/?o=1mr&doc_id=8583>

<https://www.slideshare.net/chinmoymohanty85/plugin-architecture>

<https://www.goetas.com/blog/modular-application-architecture-intro/>

<https://softwareengineering.stackexchange.com/questions/298600/struggling-about-the-benefits-of-using-a-plugin-approach-in-a-industrial-softwar>