NX.Node

Have you wondered how to have your C# code run inside NodeJS? You have probably tried Edge,JS or running your code as a command line process. I did. Found the first to be a bit unreliable and the second to be a bit cumbersome, so I said to myself, Isn't the probem that NodeJS, which is a great product, is JavaScript? So what would happen if instead I had a NodeJS plus ExpressJS in C# instead.

So here we are.

BTW: This README as a PDF can be found here.

The best way to learn

The best to learn what the code does is and always will be by looking at the code itself. *NX.Node* is a .NET Code 3.1 C# Visual Studio project, so download the code, open it with Visual Studio (use **Visual Studio Community 2019** or later, it's free) and get ready to view the code. using MS Windows 10 or later, you can easily install Docker Desktop on Windows.

Back to how it works

The route is the first *NX.Node* item that you need to understand. The following is a basic route definition in *NX.Node* :

```
using System.Collections.Generic;

using NX.Engine;
using NX.Shared;

namespace Route.System
{
    /// <summary>
    ///
    /// Echoes any URL values
    ///
    /// </summary>
    public class Echo : RouteClass
    {
        public override List<string> RouteTree => new List<string>() { RouteClass.GET, "echo", "?opt?" };
        public override void Call(HTTPCallClass call, StoreClass store)
        {
            call.RespondWithStore(store);
        }
    }
}
```

This route definition uses two C# modules NX.Engine and NX.Shared. Itwill not go into detail on what those do at this time, just keep in mind that they make this code much simpler.

The Echo class is based in the RouteClass. All of the built in classes are suffixed with the word 'Class' to minimize conflict with other code.

The class has two items that override the defaults in the RouteClass, **RouteTree** and **Call**:

Call

The Call code is what gets executed when the route is matched. In this example, the called gets passed a StoreClass, which can be thought as a JSON Object and responds with the contents of the same store. A simple echo.

RouteTree

The route tree defines the route. The first entry is the HTTP method GET/POST/PUT/DELETE/PATCH or anything you like, followed by the URL to be used. You can have four type of definitions:

Format	Meaning
text	The text entered must be seen in the URL.
:key	The URL can have any text, but it must be non-empty. The value entered will be passed to the route handler as a key/value pair in the store parameter.

Format	Meaning
?key	The URL can have any text which can be empty. Once an optional definition is seen, all following definitions must also be optional. If a value is entered, the rules will be passed back like the :key definition.
?key?	Same as the ?key definition, but must be the last entry in the tree. Behaves like if multiple ?key were entered.

Next I will run through some example of the above route. Let's start with the simplest:

```
http://localhost/echo
```

This is the smallest call that will be handled by the route, as the route only requires the word "echo" as the first piece of the URL. The call would produce a return of:

```
{"opt":[]}
```

which tells you that opt is an optional entry but no values were defined. This is similar to NodeJS but not quite the same. Lets try a more complete example:

```
http://localhost/echo/john/mary/sam/peter
```

which produces:

```
{"opt":["john","mary","sam","peter"]}
```

But we are not limited by the tree itself, we can make use of the URL parameters as well:

```
http://localhost/echo/john/paul?dept=sales&route=NY
```

Which return:

```
{"dept":"sales","route":"NY","opt":["john","paul"]}
```

as all entries are made part of the store, only one place needs to be checked for values.

Following the rule that when multiple values are seen for a given key, the passed values are turned into a JSON array, we get this:

```
http://localhost/echo/john/paul?age=34&age=47
```

which returns:

```
{"age":["34","47"],"opt":["john","paul"]}
```

Note that the parameters are return before the route entries, so calling:

```
http://localhost/echo/john/paul?opt=sam&age=22&age=34&age=47
```

returns:

```
{"opt":["sam","john","paul"],"age":["22","34","47"]}
```

Now to show the :key option, let's trun the route tree into:

```
public override List<string> RouteTree => new List<string>() { RouteClass.GET, "echo", ":dept", "?opt?" };
```

Now let's try the first example:

```
http://localhost/echo
```

the return is now:

```
{"code":500,"expl":"InternalServerError","error":"Internal error"}
```

This means that no route was found, as a "dept" entry is required. Let's try a valid call:

```
http://localhost/echo/sales/john/mike/alice
```

which returns:

```
{"opt":["john","mike","alice"],"dept":"sales"}
```

The route tree definitions, while similar to NodeJS, create a more powerful and consistent set of rules.

Securing routes

There is a built-in route that allows one bee to call another using an HTTP POST call. This is a back door into the system and back doors are trouble in the making. This is the code for that call:

```
using System.Collections.Generic;

using NX.Engine;
using NX.Shared;

namespace Route.System
{
    /// <summary>
    ///
    /// A way to reach the bee in a semi-opaque way
    ///
    /// </summary>
    public class FN : RouteClass
    {
        public override List<string> RouteTree => new List<string>() { RouteClass.POST_SECURE, "{id}", ":name"};
        public override void Call(HTTPCallClass call, StoreClass store)
        {
            // Call the function and respond
            call.RespondWithStore(call.FN(store["name"], call.BodyAsStore));
        }
    }
}
```

Note that the method in the **RouteTree** is a modified POST that tell the system that this route is only available if a secure code is passed.

The **_SECURE** changes the call form this:

```
POST  /idofbee/nameoffn
```

to:

```
POST /securecode/idofbee/nameoffn
```

The first is semi-secure, as the caller would have to know the id of the bee, but the second becomes better secured, as the caller also has to know the secure code.

And the secure code can be changed on the fly:

```
NX.Node.exe --secure_code codeyouwant
```

This is the format to use when there no secure code already defined, the format is:

```
NX.Node.exe --secure_code oldcode=newcode
```

when there is a secure code already in place.

The change can also be done programatically by calling:

```
env["secure_code"] = newcode;
```

The old code is not needed as the code runs in a secure environment.

And to solve the issue of forgetting to se the secure code, the secure routes are disabled until one is set.

Note that you can change the secure code at any time without having to recycle. It may take a small amount of time until all nodes switch.

Note: The secure code must not match any route entry that the system uses, as the behavior is not guaranteed.

Multi-threading and handling of HTTP calls

While NodeJS is a single threaded process, *NX.Node* is built with the ability to handle multiple HTTP call processors at the same time, while letting the programmer handle each call as its own process.

The **Do** call above has two parameters, the first is the encapsulation of the HTTP Call via the **HTTPCallClass** and the second is the parameters in the call URL using the **StoreClass**.

Functions

A long time ago, someone taught me that what a computer does is move data from point A to point B and maybe do a bit of processing on the way. It was true then and it is true now. If you look at the **Echo** route above, it takes two parameters and returns something. And In this case, does nothing in the way there.

But there are times where you want to do something in the way. This is an example of a very simple function:

```
 using NX.Engine;
using NX.Shared;

namespace Fn.System
{
    /// <summary>
    ///
    /// Evaluates an expression and returns a value
    ///
    /// Uses from passed store:
    ///
    /// expr        - The expression to be evaluated
    /// new         - If true or non-zero, return a new store
    ///
    /// Returns in return store:
    ///
    /// value       - Result of the expression
    ///
    /// </summary>
    public class Eval : FNClass
    {
        public override StoreClass Do(HTTPCallClass call, StoreClass values)
        {
            // The return store
            StoreClass c_Ans = values;
            // If not the original, make a new one
            if (XCVT.ToBoolean(values["new"])) c_Ans = new StoreClass();

            // Eval
            c_Ans["value"] = call.Env.Eval(values["expr"], values).Value;

            // ANd return the store
            return c_Ans;
        }
    }
}
```

The function definition looks a lot like the route definition form **Echo**, as all that computers do is move data from point A to point B and maybe do a bit f processing on the way. But function do not have a route tree, as they are not called via an HTTP call.

Let's look at what this function does.

The first thing it does is to move the passed store value into a local variable. Next it checks to see if the value of the **new** entry in the passed store is true, converting to Boolean from whatever was passed. If this is so, it will create a new store instead of using the passed one.

Next it sets the value of **value** in the store with the evaluation of the string **expr** in the passed store and using the passed store as the values to be used when evaluating the expression.

Last it will return the result store back to the caller.

Note: I am not trying to explain you what each class and call does, I am explaining the structure of the system, or how I think.

Calling a function

The easiest way to make a function call is to use the HTTPCallClass **call** parameter, so to call the **Eval** function, you would do this:

```
var ans = call.FN("System.Eval", "expr", "1+2")["value"];
```

The **ans** variable would be a string whose value would be "3". If you had other values to be used as variables you could use:

```
var ans = call.FN("System.Eval", "expr", "1+a")["value", "a", "10"];
```

Which would result in **ans** being "11";

And you could use a store instead:

```
var passed = StoreClass.Make("a", "10", "b", "99", "expr", "a+b");

var returned = call.FN("System.Eval", passed);

var ans = returned["value"];
```

and in this case **value** would be "109".

Naming conventions for Routes and Functions

While we have not discussed functions, they are the second part of the *NX.Node* system. Both share the same naming convention, which start at the DLL level.

Route DLLs must have the name of "Route.xxx.dll" and the namespace whould be Route.xxx. The DLL can contain multiple classes based on the RouteClass. Each class will be mapped using the DLL name and the class name, for example the **Echo** class above comes from the Route.System.dll so the name would be System.Echo. While the class name is limited toa single word, the DLL name is not limited to two parts, so if the **Echo** call is located in the Route.Allen.Cute.dll, the name would be Allen.Cute.Echo.

Environment settings

The environment settings are system-wide settings, easily accessible via code:

```
env["settname"]
```

When the NX.Sever.exe starts, they are obtained from the command line:

```
NX.Sever.exe --http_threads 10 --uuid STATIONA
```

If the key is entered more than once, the setting will turn into a JSON array.

You can also place environment settings inn your OS environment values definition, but if you do so, prefix the name with **nx_** in the OS table, so **http_threads** would read like **nx_http_threads=10**

The settings are stored in different places according to the resources available, but they are shared among he bees. Any setting that you use when creating a bee replaces the values previously known.

You can load settings from a file, which you can then use by calling:

```
--config pathtofile
```

This setting is not stored in the shared settings. This file is a JSON object with each environment setting that you want to use as the key and the value can be a string, JSON array or JSON object. This is an example of a config file:

```
{
    "hive": "sales",
    "fields": [
        "10.0.192.68",
        "10.0.192.69:2788"
    ],
    "qd_bumble":["redis","percona"],
    "qd_worker": [ "Portal:2", "Chores:3" ]
}
```

The following settings are used at the present time, but not mentioned elsewhere:

Setting|Meaning -------|------ http_threads|The number of HTTP worker threads to run (Default: 4) http_port|The port that the server will listen for HTTP requests (Default: 80) proc|The Fn to run at startup. This value is not saved in the settings folder and must be set via command line or environment setting.

Extra modules

While the base system has a limited number of routes and functions, *NX.Node* itself has a few extra modules that can be made available by calling:

```
env.Use("module");
```

In this mode, all processes, routes and functions are brought into the system,

You an optionally select which combination of the above to load by calling:

```
env.Use("module", Profiles.Procs | Profiles.Routes | Profiles.Fns);
```

or any combination thereof.

These are the modules available:

SubSystem	Modules	Use
DEX	Route.DEX, Fn.DEX	Data exchange between sites
Dynamic	Route.Dynamic	Allows for runtime loadeing of routes and functions
Files	Route.File	Allows for the upload, download and merging of files
MongoDb	Fn.MongoDb	MongoDb support
USPS	Route.USPS Fn.USPS	USPS support

Note that modules must have a two part name and the first part must be **Fn**, **Proc** or **Route**. The contents do not have to math the name, you can put functions in a DLL named Routes. But it may help your sanity.

Dynamic code

Dynamic code is code loaded by the user. It is kept in a sub-folder, which defaults to **dyn**.

As code stored in the **dyn** folder is automatically loaded at start time and since you can get the contents of a Git repository loaded into the **dyn** folder, there is little to do to link in any code that you or others create. By copying the DLL into the folder before launching the *NX.Node* does the job.

Containers are bees

As the **NX.Node** is designed to run as a Docker container, a set of classes are part of the **NX.Engine** to supports containers. In the NX universe, a container is called a **bee**, which live ina **hive** and can transverse many **fields**. Let's dive into how they relate.

Term|Meaning ----|------- field|The IP address a Docker deamon. hive| logical entity, like a company or department. A hive can span multiple fields, which can be shared with other hives. roster|Each hive has a roster, which keeps track of all the bees in the hive. bee|A Docker container which is part of a hive. It can be born in any one of the fields. DNA|All bees have a DNA definition when they are being created. DNA looks like the information passed to the Docker CreateContainer call with a few extra values, but a shorthand definition is available. genome|A Dockerfile cv|Once born, the bee has a CV, which looks a lot like the Docker definition for ListContainers, with a few extra values. ticklearea|Each bee can have any number of tickle areas, which are exposed ports

I have given a fair amount of thought on how to explain the architectures of container based system that I have developed over time, and while my favorite animal is an ant, bees are a better description of containers, you have a lot of worker bees, HTTP servers and processors, and a few bumble bees, databases and like services.

The roster

The roster is the most important piece of the hive, as it can signal when bees of a certain DNA or with certain tickle areas come into or out of the hive.

Built-in DNA

The following DNA definitions are part of the base system:

DNA	Use
processor	The basic task in the system. It runs a copy of the NX.Node
consul	Consul 1.4.4
minio	Minio (latest)
mongodb	MoongoDb 3.4.10

DNA	Use
nginx	NginX 1.16.0-1~bionic
percona	Percona MongoDb 4.0.10-5.bionic
redis	Redis 4.0.2
traefik	Traefik 1.7.9

All of the above originated in a different container based project, and use a variant of Ubuntu as the OS.

You can add your own genomes and DNA by adding to the Nx.Engine.Hive DNA and Genome folders. You can use the same version of the OS by using in your Dckerfile:

```
FROM {repo_project}/base:{tier}
```

Note that you MUST include the following:

```
LABEL {proj_label}
```

which is used to track to which hive the bumble bee containers belong. Having this allows the Docker repository to have other entries that are not associated with NX.Node.

Note: Please use the **Docker Build CLI** to test that any Genome and DNA that you introduce can be both **built** and **started** before integrating them into the system. A genome and/or DNA that fails will most likely cause the queen to go into and endless loop. You have been warned.

Note: Make sure that any script files have Linux line endings. How to set this up in Visual Studio can be found here

Environment Settings

The following settings are used by the Hive sub-system:

Setting	Meaning
field	The name and IP address of the Docker API. The syntax is name=ip, for example sales=http://cr.myco.com:8087 or test=10.0.192.99:2375. You can enter any number of entries by repeating the --field name=ip setting. If none are entered a=localhost:2375 is used.
repo_name	The name of the repository. Typically this is an URL in the form of https://cr.myco.com:8888 and is used only if there is an external repository
repo_project	The project name. Defaults to **nxproject**
repo_username	user name to log into the repository
repo_userpwd	Password used to log into the repository
repo_useremail	E-Mail address where your company may be contacted if any issues arise

NB: The limitation in this structure is that the sample project name is used for your local and any external repository.

Hives via Routes

The following routes can be made available by:

```
env.Use("Route.Hive")
```

Making the processor genome

The NX.Node has a built-in mechanism to create an genome of the base code, plus any code that you wish to make into base code.

There are two basic rules you need to follow:

- Your code must be .NET Core or .NET Standard
- You need to copy the DLLs you want included into a single folder structure

Once those are done, execute the following in the command line:

```
NX.Node.exe --make_genome y --genome_source foldername --field name=iptodocker
```

The **gnome_source** is optional but the folder should contain all of our DLL's to be included in the genome. It may me used multiple times to include multiple sources.

If you leave out the field, the program will assume that is the local Docker instance.

Note that the process uses folder **/build/container**, so any contents in that folder will be automatically deleted.

DNA

You can think a DNA being the definition used to create a running container. It is a JSON object with the values used in the Docker CreateContainer call. You can optionally use the following additions:

Key	Meaning
$$From	The name of the DNA that defines the basis for this DNA. The from task is used as the template and any keys defined in the DNA definition replace those in the from DNA. Can be recursive.
$$Unique	Set to "1" if only one bee of this type can be in the system at any one time.
$$Ports	A JSON array of port numbers to expose. Any port defined before a zero is treated as a private port and exposed with a dynamic port number. Ports defined after the zero are exposed with the same port number.
$$Map	A JSON array of volumes to use. They are the source and target, separated by a semi-colon. Note that the order is the reverse of the way Docker defines volumes, but IMHO is easier.
$$Requires	A JSON array of DNA that must be running before this DNA can be used. The system automatically creates a bee if necessary for any missing DMA.

The above plus an **Image** and **Cmd** entries are all that is needed to define a task.

DNA and the environment settings

While most of the settings in a DNA is static, you can use the environment settings to modify the following keys at the time that the task is launched: **Image**, **Cmd**, **Env**, **Ports** and **Map**.

Here is the DNA for Redis:

```
{
    "@Unique": "*",
    "@SkipRecycle": "1",
    "@Ports": [
        "6379"
    ],
    "@Map": [
        "{shared_folder}/redis:/etc/wd"
    ],
    "create": {
        "Image": "{repo_project}/redis:{tier}",
        "WorkingDir": "/etc/wd"
    }
}
```

Note how environment settings are shown in the fields surrounded by curly braces, so:

```
"Image": "{repo_project}/redis:{tier}"
```

may be converted to: `JavaScript "Image": "classic/redis:latest"` when the environment settings are set as:

Var	Value
repo_project	classic
tier	latest

@Unique and *

The @Unique entry is the field where the bee will be created, or in Docker speak, which computer will host the container. Since the DNA for the built-in services is hardcoded, the field name used in it may not be one that you want to use. For this reason, the value of * is used.

When the system sees the wildcard, it will look into the environment settings for a an entry in the form of **field_xxx**, where xxx is the genome name, so for redis it would be:

```
--field_redis jack
```

which would create Redis in field **jack**.

If no environment setting is defined, the system will use the first field.

Hive hierarchy and its health

While you are familiar with bee keepers, a natural hive has no such person as relying on a beekeeper brings the danger of single point of failure. It is the same with this hive. All worker bees have at most one leader and one follower, where the queen bee has no leader and the lowest worker bee has no follower.

The selection of each worker leader and follower is done by the hive roster, as it keeps track of all bees. There is no single hive, the hive is duplicated in all worker bees. Maintenance of the roster is done via polling each field until a redis bumble bee is known, which is then used to synchronize bee's birth and deaths.

Checking the health of a bee

Each bee runs a Docker **healthcheck** call 30 seconds and sets itself to be unhealthy if it cannot be reached. Each bee checks its follower state every five minutes and if it not running, it will kill the bee and replace it with a clone. The last drone in the chain checks on the Queen's health, closing the loop.

Queen's selection

Each bee is given a unique identifier at birth. The bee with the largest identifier becomes queen. There are moments when there are more than one queen, especially before Redis is available, and there could be moments when there is no queen, as the bee may die unexpectedly.

As soon as any condition where there is no queen is detected, the hive will select a new queen from the worker bees.

Queen's duties

The duty of the Queen is to ensure that the required bumble bees are alive. You can set what the queen is required as a bumble bee by:

```
--qd_bumble minio
```

and you can have multiple requirements by:

```
--qd_bumble minio --qd_bumble percona
```

By default, the only bumble bee needed is of DNA redis which is automatically done, but you can change that behavior via: `--qd_bumble !redis`

Similarly, you can also add to the Queen duties the orchestration of worker bees by calling:

```
--qd_worker 10
```

which will ensure that ten bees are running. You can specify which type of bee to keep alive by:

```
--qd_worker Run:2
```

where the prefix is the process name **proc** to pass each bee.

The above checks are done every minute, unless you override it by:

```
--qd_every numberofminutes
```

You can add code tasks to the Queen's duties by calling:

```
var id = env.Hive.Roster.AddQueenToDo(delegate() {
  ...
});
```

Which returns an ID for the todo.

You can delete the to by calling

```
env.Hisv.Roster.RemoveQueenToDo(id);
```

Note that the Queen's duties do not start for two minutes after her ascension. This is to handle moments where a number of bees have come into play and things are getting sorted out.

Mason bee

There is a special type of bee, which along with its normal duties also accepts requests via a message queue. For this bee to exist, the **redis** bumble bee must also exist.

To turn any bee into a mason, use the following call:

```
env.Hive.Mason.Listen("queuename", delegate(WorkMessageClass msg){
  ....
});
```

and to respond to a request:

```
envHive..Mason.Respond(msg);
```

Any bee can call on a mason bee by:

```
var id = env.Hive.Mason.Send(msg);
```

which will open a work item for the mason bees handling the queue and continue on. The ID is a unique value so the requesting bee can keep track of requests.

This is how to set a handler for the request returns:

```
env.Hive.Mason.Handle(delegate(WorkMessageClass msg){
  ....
});
```

As the mason processing is asynchronous, and may take a bit of time, it is best not to base any processing in a given time period. If the requesting bee dies, any requests and responses are removed from the system. The Handle call must be made prior to sending any messages, otherwise the messages will be sent as **DoNotRespond**.

Mason work message

The mason work message has the following properties:

Property	Type	Meaning
RequestorBee	string	ID of bee that sent the message
MasonBee	string	ID of the bee that processed the message
Request	Store	Data sent by the sender bee
Response	Store	Date return by the mason bee
DoNotRespond	bool	If true, no response is allowed
RequestTTL	TimeSpan	Amount of time in milliseconds that the request is valid. Zero means no expiration
ResponseTTL	TimeSpan	Amount of time in milliseconds that the response is valid. Zero means no expiration

Sharing bumble bees

While the structure of the hive is designed to be self-contained, sometimes is necessary to share resources, like databases, across multiple hives. To accomplish this, set an environment setting like:

```
--hive_percona marketing
```

which tells the system that this hive and the marketing hive will use the same percona bumble bee. This is assuming that the bee has:

```
--hive sales
```

You can include any number of hives to share in the bumble bee:

```
--hive_percona marketing --hive_percona frontoffice
```

Now the bumble bee will be shared among the tree hives, sales, marketing and frontoffice.

In order for this to work correctly, all hives entries must be the **same** in **all the hives used**. Changes therefore require a full recycle of all the hives.

Tiers

Tiers are used for versioning, the default value being **latest**. If you want a hive to use a different version, set its **tier** environment setting to the value that you want.

Rolling your own DNA

You can upload your own DNA and genomes by calling:

```
POST /hive/genome?name=myname
```

with the Dockerfile as the body of the call. In code the same is accomplished via:

```
env.Hive.SetGenome("myname", "dockerfilecontents");
```

You can get any genome by calling:

```
GET /hive/genome?name=myname
```

or in code via:

```
env.Hive.GetGenome("myname");
```

Also, You can upload your own DNA by calling:

```
POST /hive/dna?name=myname
```

with the JSON definition as the body of the call. In code the same is accomplished via:

```
env.Hive.SetDNA("myname", "dnacontents");
```

You can get any DNA by calling:

```
GET /hive/dna?name=myname
```

or in code via:

```
env.Hive.GetDNA("myname");
```

Bees lifecycle

A bee can be created from outside the hive by running:

```
pathtofile\NX.Node.exe
```

Or running the project from inside Visual Studio. This creates what I call a **ghost** bee, one that has access to everything, but is known to no one. You can also create a hive by calling:

```
NX.Node.exe --field iptodeockerdeamon
```

Now the bee has access not only to the code, but to all hive functions, but it is not part of the hive and does not take part in the check of the hive's health. This is the **one field** model, which is the most common one.

Until you run into hardware limitations, where you can add other fields by calling:

```
NX.Node.exe --field iptodeockerdeamon1 --field iptodockerdeamon2
```

And still work with a single hive.

A bee can also be created by another bee by calling:

```
var bee = env.Hive.MakeWorkerBee();
```

which returns the new bee.

When a bee is born and has one or more hives, it will check for other bees in the hive and make a roster of all the bees. It will repeat this process every minute until a redis bumble bee is seen.

When a redis bumble bee is seen, the bee will alert the hive (itself) and switch over to using the bumble bee's publish/subscribe to keep track of bees. This is done only after the first build of the hive's roster. Now when any bee creates or kills a bee, all of the other bees will be alerted without having to scan the hive.

In either case, redis bumble bee or not, after the hive's roster is built, the roster will select a queen. The Queens job is to make sure that the required bumble bees are found, including a redis bumble bee, so you ae pretty much guaranteed that if a redis bumble bee was not already part of the hive, one would be made in short order.

Each bee does an internal health check every 30 seconds and if the check fails, it will flag itself as unhealthy. Every minute or so, each bee will check the health status of the bee that follows it, and if it is found unhealthy or exited, the follower will be killed and a clone brought into life. This guarantees that the hive population remains at an even level.

Customizing a bee

When a bee is created, it will run the function defined by the **proc** environment setting, which by default is empty. When a value is passed by calling:

```
NX.Node.exe --proc Chores
```

or programmatically:

```
var bee = env.Hive.MakeWorkerBee("Chores");
```

where you replace "Chores" with your own value or set of values, the bee will do a:

```
env.Use("Proc.Chore");
```

which can customize a bee by loading routes and other code.

You can assure that a number of bees that run a given **proc** are running by calling:

```
env.Hive.AssureWorkerBee("Chore", min, max);
```

where min and max are numbers. This is something that the queen bee can do as part of her chores **qd_worker** parameter.

Creating a bee from the command line

All the above code details how to make a bee once the hive is up and running, but to have a new hive, we need to have a bee:

```
NX.Node.exe --make_bee y --hive myhive --field roses=10.0.192.168
```

This creates a single bee in a hive in a field, which automatically becomes the queen.

From there on out, either by code or command line, you can add more bees to make a working hive.

The default proc, which is an empty string, can also be customized. At start, it will call:

```
env.Use("Proc.Default");
```

to mimic a customized bee.

External bees

While the hive can easily manage bees, sometimes some bumble bees are outside the hive. An example would be a shared database that is managed from the outside, while allowing access to the hives' bees.

You setup these bees using the following:

```
--external mongodb=10.0.199.9:9999
```

which tells the system that the DNA is **mongodb**, and it can be reached at the IP given. Like many other calls, you can define multiple external resources.

Process routes

while you will later learn how to use **NginX** or **Traefik** to route HTTP calls to customized worker bees, you can also change the route trees themselves to work with the load balancers.

You can make routes use the **proc** environment setting by modifying the first parameter, which is the HTTP method in the route tree, for example:

```
public override List<string> RouteTree => new List<string>() { RouteClass.GET, "echo", "?opt?" };
```

is a non-proc based route as:

```
GET /echo/...
```

but:

```
public override List<string> RouteTree => new List<string>() { RouteClass.GET_PROC, "echo", "?opt?" };
```

changes the call to

```
GET /chores/echo/...
```

You can also use the same mechanism for secured routes:

```
public override List<string> RouteTree => new List<string>() { RouteClass.POST_PROC_SECURE, "{id}", ":name"};
```

which changes the call to:

```
POST /chore/securecode/idofbee/nameoffn
```

In the above examples, **chore** is replaced by the value of **proc**.

Folders

These are the environment settings used by the folder mapping:

Setting	Meaning	Default
root_folder	Starting point of the folder structure	Working directory
dyn_folder	Folder where loaded DLLs are kept	#root_folder#/dyn
shared_folder	Folder where the shared items are kept	#root_folder#/shared
doc_folder	Folder where documents are kept	#shared_folder#/files
ui_folder	Folder where UI files are kept	#shared_folder#/ui

You can change the subfolder by setting the environment setting as follows:

```
--ui_folder web
```

which would produce the folder to be the value of **share_folder** with **web** appended as a sub folder.

If you want to set the entire path, simply enter the path as follows:

```
--ui_folder @/etc/web
```

The one setting that you should (nust?) change is the **shared_folder** as by default points to the bee's file structure, making it a private to the bee, where it is intended to be shared among all the bees. Typically this is set to a shared volume in your network.

A static (maybe) web site

Including the **Route.UI** DLL, makes the bee into a "static" web server. Let's look at the code:

```
using System.Collections.Generic;

using NX.Engine;
using NX.Shared;

namespace Route.System
{
    /// <summary>
    ///
    /// A route that allows a "regular" website support
    ///
    /// Make sure that all of the files to be served are in the #rootfolder#/ui
    /// folder and that none of the subdirectories match a defined route
    ///
    /// </summary>
    public class UI : RouteClass
    {
        public override List<string> RouteTree => new List<string>() { RouteClass.GET, "?path?" };
        public override void Call(HTTPCallClass call, StoreClass store)
        {
            // Assure folder
            call.Env.UIFolder.AssurePath();

            // Get the full path
            string sPath = store.PathFromEntry(call.Env.UIFolder, "path");

            // If not a file, then try using index.html
            if (!sPath.FileExists()) sPath = sPath.CombinePath("index.html");

            // And deliver
            call.RespondWithUIFile(sPath);
        }
    }
}
```

The key is the **RouteTree**, which has no text to match, just optional. What this causes is to create a route where if a GET request finds no matches elsewhere, it will match whatever the requestor entered.

If what was entered does not exist as a file, **index.html** is appended.

The root path is **ui_folder** and the file must be in that folder or a child folder. If the file is not found a 4004 Not found error is returned.

And I say maybe a static web system, as I can see where with a bit of code, you can make this route into a processor and modify the files as they are being returned.

Support

Redis

It is difficult to undervalue the use of Redis in the system. All inter-be communications and handling of settings are done via Redis and it is done in many places in the system.

The Redis services are sharable with any party, as the keys used by the system are prefixed with **$$**. If this conflicts with any of the keys in use, it cal be changed by setting the **redis_prefix** environment setting.

Synchronized store

NX.Node has a special class **NX.Engine.SynchronizedStoreClass** that makes use of Redis to keep each copy in each bee synchronized. It does this by keeping mirror copies of the data in memory and also in Redis.

The store has built-in behaviors to make it work:

Creation of the Redis store

When the store is first created, the data already kept in memory by the bee is copied to Redis. This also happens when a ghost bee opens the store, allowing for updates to already established stores.

When the Redis store become available

When the store becomes available, when a Redis bumble bee is seen, any data already saved in the store will be read into memory. If the data is not known, it will be copied to Redis.

Changes to the store

Changes are always kept in memory. If the Redis store is available, it is also copied to the Redis store and propagated to all the other bees.

Something about key names

The synchronized store does not allow it's keys to have a leading plus **+** or minus **-** sign. These two leading characters mean that the key being set is an array and that the values are to be added or removed from the array.

Making changes to a running hive from the outside

Remember that the environment settings are a synchronized store. If you run NX.Node.exe from outside the hive as a normal program, any parameters in the command line are copied to the hive's environment settings store, thereby updating the hive.

When the change entails adding or removing fields, you should expect a fair amount of mayhem as bees are killed in some places and added in others. Once the changes stabilize, the queen will take over and reconstruct the hive.

Minio

Minio is the document storage system in the NXProject. Its use is automatic when you use either the **Proc.File** or **Route.File** modules. As all AWS S3-compliant system use a bucket structure and not a folder structure, the NX Project file manager maps a pseudo-file tree into buckets. Each folder is a bucket which holds all files in the folder as well as pointers to sub-folders.

Document manager

You can instatiate a document manager, which is the interface to Minio by calling:

```
var mgr = env.Env.Globals.Get<Proc.File.ManagerClass>();
```

Folders

The **Proc.File.FolderClass** is the equivalent of a folder. It requires a dcument manager,

Documents

The **Proc.File.DocumentClass** is the equivalent of a file. It requires a dcument manager,

At startup

You can write to the folder structure while the minio bumble bee is being launched, and any writes will be integrated into Minio as soon as the bumble bee is available. Reads cannot be queued, so they return empty strings for any file.

making the bumble bee an external and AWS

You can make the minio bumble be be external by calling:

```
--external minio=url
```

where the url is the locaton where your Minio instance is located. If you wish to use AWS call:

```
--external minio=s3.amazonaws.com --minio_acces youraccesskey --minio_secret yoursecretkey
```

Note that to run the minio bumble bee locally, the access and secret keys are optional, except that if you destroy the environment store in the redis bumble bee, you will lose access to the minio bumble bee documents, so use of your own keys is strongly suggested.```

Git

NX.Node has built-in support for any Git-like product. Let's first describe the environment settings used:

Setting	Meaning
git_url	he URL for the Git repository (Default: https://github.com/)
git_product	The Git product ID (Default: NX.Node)
git_token	The Git access token
git_repo	The Git repository name. This repository will be loaded at start time and any changes will be reloaded. (Example: MyRepo/SuperProject)

When a bee starts, the repository will be checked against any code loaded into the **modules** folder and if there is a newer version, obtained, compiled and loaded into

the folder. C# and Visual Basic code is supported. When compiling both NX.Shared.dll and NX.Engine.dll are available as using or Import.

Format	Meaning
token@owner/project/module.ext	Full path. The token is optional, as the git_token environment setting will be used
project/module.ext	The same owner as the git_repo environment setting, but a different project
/module.ext	The same owner and project as git_repo. Not needed as module is loaded automatically
token@owner//module.ext	The same project as git_repo, but a different owner. The token is optional, as the git_token environment setting will be used

NoSQL

// TBD

NginX

// TBD

Traefik

// TBD

Kubernetes

While NX.Node has its own method for inter-bee connectivity and hive health checks, it also provides lite-weight support for Kubeernetes livelinesss and readiness probes.

This is entry in the deployment for the liveliness probe:

```
livenessProbe:
 # an http probe
 httpGet:
   path: /liveliness
   port: 80
 initialDelaySeconds: 15
 timeoutSeconds: 1
```

and for the readiness probe:

```
 readinessProbe:
   # an http probe
   httpGet:
     path: /readiness
     port: 8080
   initialDelaySeconds: 20
   timeoutSeconds: 5
```

The is extra code that needs to be setup for the readiness probe, as each bee's requirement will differ. This is the code to support the probe:

```
env.ReadinessCallback = delegate()
{
  ...

  return "ERROR MESSAGE or NULL IF OK";
}
```

XML

While the default POST data and returns are JSON, the system can easily work with XML data instead. This is done by the use of the **response_format** environment setting.

Value	Meaning

Value	Meaning
auto	Response format mirrors the POST data format. This is the default behavior
json	Force all responses to use JSON
xml	Force all responses to use XML

Note that the POST body is always treated as **auto**.

XML formatting rules

There is a restriction on how XML is formatted, it must always have a single tag labeled **data** as the root. For example:

```
<root>
  <key>size</key>
  <value>large</size>
</root>
```

is the equivalent of:

```
{
  "key": "size",
  "value": "large"
}
```

Note that some of the JSON used in the system may have keys that are prefixed with the **$$** prefix, which is not a valid XML tag.

HTTP authentication

You can define which authentication method to use by calling:

```
env.HTTP.SetAuthenticationScheme("scheme");
```

where scheme is one of the following:

Scheme	Meaning
None	No authentication is allowed. A client requesting an HTTPCallClass object with this flag set will always receive a 403 Forbidden status. Use this flag when a resource should never be served to a client.
Digest	Specifies digest authentication.
Negotiate	Negotiates with the client to determine the authentication scheme. If both client and server support Kerberos, it is used; otherwise, NTLM is used.
Ntlm	Specifies NTLM authentication.
Basic	Specifies basic authentication.
Anonymous	Specifies anonymous authentication.
IntegratedWindowsAuthentication	Specifies Windows authentication.

When an HTTP request is received, the system will place the user name and password in the HTTPCallClass UserInfo object. It will then check to see if the request has been authenticated and if it has not, the env.ValidationCallback() will be called. The function should return true if the validation was successful, otherwise the request will be rejected.

The above scheme table was taken from AuthenticationSchemes Enum

NB: The authentication method applies to all calls, so if you are setting up callbacks from other services, make sure that they are capable of handling the authentication method that you decide upon.

Linux v. MS Windows

The bees themselves all live in Linux, and you must think in Linux. Folder paths are in the format of **/folder/subfolder1/subfoldeer2**.... It applies to the following:

- config
- genome_source
- xxx_folder

The config and genome_source are not stored in the shared environment settings.

Prerequisites

This project was generated as a C# .NET Core 3.1 project, soo you will need Visual Studio Community 2019 or later.

If you do not wish to provide field IP definitions and having just one field, your field must be running Docker V18.03 or later.

Make sure that any firewall that you are using allows for full access to the ports being used by Docker. I just turn off protection for my private network.

Author

- **Jose E. Gonzalez jr.** - *All that you see here*

The reason for this is that the **make_genome** command packages the config.json as part of the genome (image) allowing for the DNA (CreateContainer) to have a Cmd that refers to the config.json, making each created worker bee use the same settings.

A trivia question

The DNA for a worker bee is **processor**. Why?

License

This project is licensed under the MIT License - see the LICENSE file for details

Packages used

- Docker.DotNet 3.125.2
- DocX 1.7.0
- HtmlAgilityPack 1.11.24
- ICSharpCode.SharpZipLib 0.85.4.369
- iTextSharp 5.5.13.1
- KubernetesClient 2.0.26
- Minio 3.1.13
- MongoDB.Bson 2.10.4
- MongoDB.Driver 2.10.4
- Newtonsoft.Json 12.0.3
- Octokit 0.48.0
- StackExchange.Redis 2.1.58
- TimeZoneConverter 3.2.0

Why

This code represents many years of coding in many languages, operating systems computer types and places. It is intended to be an example of open source as what I wish Open Source to be, with an MIT license.

Take from the code what you wish.

And to avoid any "style" wars, the code follows my own coding style, which does not mean that your coding style is any worse, it just means that we differ. The same goes for language, operating system and whatever else you care about. I am glad that you care about it, but I do not care to get into any arguments. Like a colleague once said **Let this be a learning moment for you, not a teaching moment for me** .

Acknowledgments

- To all of those that came before me
- To my eldest son **Joemar**, for whom I wish this to be the starting place of something good. **Alice** is two years older than you are. Thank you for your contribution of all the Genomes and DNA.
- To Lewis Carroll and his wonderful **Alice's Adventure in Wonderland**, after which the original project that ended here was named, **Alice**
- To the people at **Microsoft** for creating the often painful, but overall useful Visual Studio .NET, which has been my home since it was first released, and the creation of .NET to which I was introduced early in 2001 and have been toiling at ever since.
- To **Herre Kuijpers** for his a Tiny Parser Generator v1.2 which has been hacked to death in so many incarnations. It is the only non-package in the system.
- To the people at **Atom** https://atom.io/ which created the nifty editor, the results of which you just read