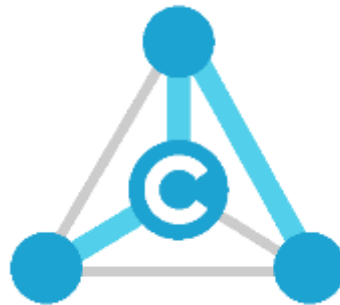


C-DEngine™ Concepts

Web-to-Mesh Connectivity



Last Revision: September 19, 2019

Chapter 1	C-DEngine Distributed Web Services	4
1.1	Overview	4
1.2	Setup and Configuration	4
1.3	Hosting a Static HTML Page	4
1.4	Add an Image	5
1.5	Distributed Web Browsing	5
1.6	Distributed Cloud Web Browsing	6
1.7	Advanced Topic: Caching	7
1.8	Advanced Topic: Synchronous vs Asynchronous content	7
1.9	Advanced Topic: Redundant content	8
1.10	Web Content Authoring	8
Chapter 2	Creating HTTP Interceptors in a C-DEngine Plugin	10
2.1	Overview	10
2.2	The C-DEngine Communication Core	10
2.3	Extending the C-DEngine with Plugins	10
2.4	Inbound HTTP Traffic	10
2.5	Static HTML Pages	11
2.6	Dynamic HTML Pages	11
2.7	Network Request and Response in <i>TheRequestData</i>	13
2.8	Unregistering an HTTP Interceptor	15
2.9	Detecting Conflicts between Interceptors	16
2.10	Interceptor Helper Function	16
2.11	Providing REST-Style Data	17
2.12	Other <i>IHttpRequest</i> Functions	18
2.13	Authenticating REST Users	19
Chapter 3	Accessing Mesh Resources	24
3.1	Overview	24
3.2	The C-DEngine Mesh	24
3.3	Messages	24
3.4	A REST Server and REST Client	25
3.5	Simplest REST Client to REST Server Operation	26
3.6	REST Server with C-DEngine Node Messages	28
3.7	Message Requests and Responses	30

Chapter 4	Compare and Contrast C-DEngine and REST API.....	34
4.1	Overview	34
4.2	Feature Checklist Comparison	34
4.3	Similarities.....	35
4.3.1	HTTP Protocol.....	35
4.3.2	JavaScript Friendly	35
4.3.3	Data Compression	35
4.3.4	Use of SSL / TLS for Security.....	36
4.4	Differences between C-DEngine and REST.....	36
4.4.1	Synchronous vs. Asynchronous	36
4.4.2	HTTP vs. Web Sockets.....	37
4.4.3	HTTP Dependencies.....	39
4.4.4	Location of Command and Parameters	41
4.4.5	Support for Queues	42
4.4.6	Batching in Message Delivery.....	42
4.4.7	Message Priorities Fine-Tune Message Delivery	43
4.5	Conclusion	44
Chapter 5	Shared Web Worker in C-DEngine JavaScript/TypeScript (cdeWorker.js).....	45
5.1	Installing C-DEngine Web Worker Support.....	45
5.2	A small Sample explained.....	45
	The sample shows:.....	45
5.3	Supported Web Browsers.....	45
5.4	Unsupported Web Browsers.....	46
5.5	Debugging.....	46
5.6	Required Classes.....	51
5.7	Important Notes	55
5.8	Future enhancements	56
5.9	Dependencies	56
Appendix A: The Fire Gate Plugin.....		58
A.1	About the Fire Gate Plugin.....	58
A.2.	Configuring Fire Gate Plugin	58

Chapter 1 C-DEngine Distributed Web Services

1.1 Overview

This document describes the usage of the Distributed Web Service (DWS) of the C-DEngine. It is meant for Web Designers, Web Developers and IT personal in charge of setting up distributed IoT Web Scenarios in Industrial Environments.

1.2 Setup and Configuration

DWS is part of the C-DEngine but in order for the DWS to be active, the C-DEngine must be hosted in a “relay”. An example for this Relay is the “Factory-Relay” or similar product. With the C-DEngine SDK, developers build custom relays and configure distributed web service environments.

There is no special installation necessary to activate the DWS functionality of a relay. Please follow the instructions to install your Factory-Relay or go through the “Getting Started” guide that accompanies the C-DEngine SDK.

A C-DEngine mesh is created when two or more C-DEngine nodes are connected together and share a common Scope ID. Connections between nodes are established by defining the `ServiceRoute` value of each node to reference the URL of one or more C-DEngine nodes. In order for each part of the DWS to work together, all participating C-DEngine nodes must use the same Scope ID value.

For security reasons, the DWS is meant for static pages only. It is provided to help distribute reference material (such as manuals, help text, device information) to all of the nodes in a mesh. When dealing with dynamic content, you must provide a plugin that has been created with an HTTP Interceptor (see Chapter 2).

1.3 Hosting a Static HTML Page

When a relay starts for the first time, it will have a `ClientBin` folder in the same folder as the relay's executable file. This folder is the location where shareable content should be placed. The easiest way to understand how this works is to try it out yourself.

1. Put the following lines of HTML in a file and name the file `relay.html`:

```
<html>
  <body>
    <h1>This is My Relay Server</h1>
  </body>
</html>
```

2. Copy the file `relay.html` to the `ClientBin` folder of one of your C-DEngine relay nodes.
3. Type the following URL into a browser address bar: <http://localhost:8706/relay.html>. You may need to modify this address for your relay's actual port number.

4. You now see the HTML file appear in your browser.

1.4 Add an Image

The HTML support provided by the DWS provides all the basic features that you expect to find in a web server. You can, for example, use HTML anchor ("") tags to navigate between pages. You can organize your HTML files into folders. And you can image ("") tags within your HTML to improve the visual appeal of your web pages.

Since the DWS is a "Distributed Web Service," you can create links to content on any relay in the mesh (remember: all nodes must have the same Scope ID) as if all the content is located on a single server. Here is something to try, so you can see that images are supported in DWS:

1. Within the `ClientBin` folder on any node in the mesh, create a folder and name it `Images`.
2. Find a nice royalty-free and license-free JPEG image on the web. Copy the file into the `ClientBin/Images` folder.
3. Rename the image file to `myimg.jpg`.
4. Modify the HTML file you created earlier

```
<html>
  <body>
    <h1>This is My Relay Server</h1>
    
  </body>
</html>
```

You can now load the `relay.html` file in your browser to see that the page and the embedded image appear as you expect them to. These

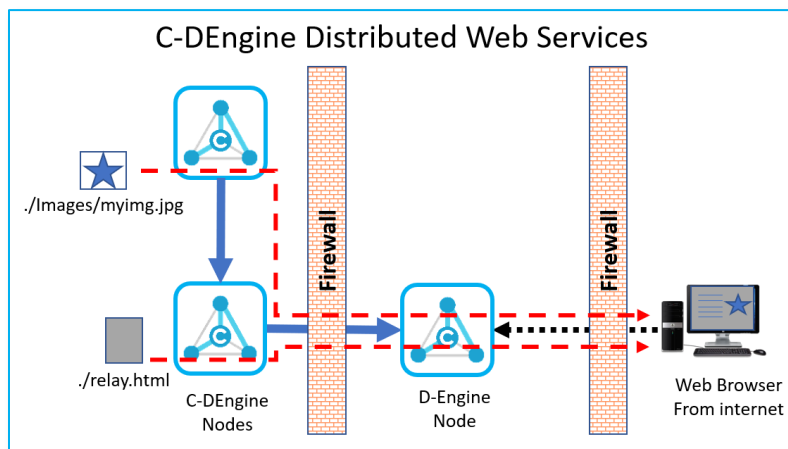


Figure 1.1. Distributed web services (DWS) enables seamless access to mesh-wide content.

1.5 Distributed Web Browsing

You can distribute the HTML files, images, scripts, and other web page resources between all of the relays in your C-DEngine mesh. You only need to maintain the same folder structure from your original node. When you open a browser and type in the URL for the web page, you will see the page along with any embedded images.

From time to time, when loading an HTML page into a browser, you may see a message that says:

Waiting for page...

The DWS shows this message to alert the user that it is searching for a page in the mesh. This may take a few seconds, depending on the number of nodes in the mesh, the speed of the connections between nodes, and how the node-to-node communication is configured (for example, whether nodes are communicating using the http protocol, or whether nodes are communicating using the newer and faster Web Sockets protocol).

Note: Node-to-node communication requires authentication. Web browsers are authenticated with a user-id and password. But sometimes you wish to allow unauthenticated browsers access to content, to display a help file, for example. Enable unauthenticated browsers to access distributed web browsing resources, by setting the `AllowDistributedResourceFetch` flag to `true`.

1.6 Distributed Cloud Web Browsing

If your mesh includes a cloud node relay, you can connect to the cloud node from anywhere on the internet and use this connection to view content from anywhere in the mesh. To prevent unauthorized access to your mesh, the connection is possible only after you have established a secure connection to your mesh. In addition to the security considerations, there is a practical one as well: when you connect through a cloud relay node, the cloud cannot know where to find the nodes in your mesh until you authenticate the connection.

Make sure your Factory-Relay node is connected to the cloud.

1. Go to the Factory-Relay plug-in screen and click on the “Cloud Setup” button. If it shows “Public Cloud” you are connected to the Factory-Relay cloud.
2. Open a browser and point at www.Factory-Relay.com/NMI
3. Enter your email and password for your Factory-Relay.
4. Once you see the main portal of the Factory-Relay go to another tab in the same browser.

Note: Make sure to use the same browser and not open a new browser as the cookie of the C-DEngine is only shared between tabs of the same instance of a browser. As soon as you close the browser or use another one, the session is no longer valid, and cookies are automatically deleted.

5. Now enter www.factory-relay.com/relay.html in the URL of the new tab

After a short wait you will see the page with the “This is My Relay Server” and the image on it.

1.7 Advanced Topic: Caching

Caching plays a very important role in improving a user experience of web site. Caching helps improve web browser performance and minimize the load on a network. Web servers and web browsers have a very complicated relationship when it comes to caching. A web server can offer a cache period, and a web browser might choose to accept that cache period or might ignore it and handle the caching itself. Web page developers place hints within HTTP headers as well as in HTML pages to influence caching behavior.

While user experience is, of course, an important concern, the C-DEngine was designed with a very high priority placed on security considerations. For that reason, the C-DEngine uses a caching policy that is certainly more stringent than that used by web browsers. Here are some of the elements of the C-DEngine caching policy:

- HTML Pages are cached for 30 seconds in the “first node”. (The first node is the node the browser is connected to).
- PDFs, Icons, Fonts, ZIP Files, JavaScript Files, Style Sheets (CSS) and Images are cached on a node until that node restarts.
- There is no caching for XML and JSON files. Instead, these are requested from mesh nodes every time there is a request for a file of this type.

Caching can be changed with two settings in the App.Config:

- `DisableCache=true` (turns off all caching)
- `CacheMaxAge =xx` (sets the time in second for the cache (default is 30))

All caching that is performed by the C-DEngine is done so in a way that is compatible with the safe and secure operation of the C-DEngine's multi-tenant support. Among other things, this means that the C-DEngine cache is aware of the important role that scope IDs use in enabling connections between nodes. All of this means that the C-DEngine enforces the rule that each customer can only see their own resources and not the resources of any other customer.

1.8 Advanced Topic: Synchronous vs Asynchronous content

The HTTP protocol was invented in the 1970s and has evolved very little. It is still inherently a synchronous protocol, meaning that it operates like someone with a checklist that must finish one task before starting any other task. For example, if a browser requests a HTML page, it sends a request to the web server and blocks the request thread until a page is returned. The server must return the page in the response to the incoming request.

In an asynchronous environment, a browser could request a page and then go work on other things instead of waiting for the request to be filled. Once a web server has gathered all the information needed for a page, it returns that information to the browser. This might be very fast (less than a second) or it might be very slow (many minutes).

To optimize network throughput and improve the user experience, an implementation of the C-DEngine is available in JavaScript (the `cdeNMI.js` JavaScript-Engine) that works asynchronously. The JavaScript C-DEngine sends pages and content asynchronously between the DWS and the web

browser. Unfortunately, this works only for data and dynamic updating content – not for static content such as image resources and html pages.

Sometimes, the DWS must gather synchronous content from multiple nodes before it can send content back to the browser. In such cases, the C-DEngine DWS must wait for all content to be delivered from other nodes before it can send a response to a web browser. Only after all required content for a request has been assembled can the DWS transmit the results to the web browser.

Although the DWS is a distributed system, that does not mean that the DWS does any caching of HTML Pages (or other content) in the web server node. If required content is not present on the C-DEngine node closest to the web browser (also known as the "first node"), the DWS holds the request until the resource has been gathered across the mesh. The DWS holds the request for a maximum of 50 seconds and if the resource could not be found by then, returns a `StatusCode=404` (Page Not Found). As of now, there is no configuration setting to change the wait time, however, this might be added in the future.

Although the C-DEngine has been optimized for asynchronous operation, the limits in the operation of HTTP and HTML means that most HTML content is provided synchronously. There are some important exceptions worth noting, including zip files, XAP (Silverlight Content) and Fonts. Some browsers do automatically retry these resources if they get a request timeout (HTTP status code 408) from a web server.

1.9 Advanced Topic: Redundant content

A C-DEngine mesh could have two nodes or it could have twenty. The benefit of having multiple nodes is that there is redundancy in case a node goes down or is unavailable for some reason. One disadvantage of having many nodes is that it can take time to copy content from one node to another, especially when the data requester and the data provider have many nodes between them.

We mentioned earlier that the DWS does not cache content between multiple nodes. If `relay.html` is served up from one node, that node is asked to provide that content. In other words, there is no automatic redundancy of content.

However, there is no reason why you cannot copy content between nodes to allow for the benefits of redundancy. We already mentioned how content availability is improved with redundancy. It is also likely that overall system performance and throughput is improved by redundancy. Certainly, if each node in a mesh has a copy of all of the content in that mesh, then there won't be any delays from waiting for content to be served up by another node in a mesh.

Note: Care must be taken to avoid file name conflicts. Consider two different versions of a file named `starfish.jpg`. If all images were in the `ClientBin\Images` folder on each node, but one version was on some nodes and another version was on other nodes. There would be no way to predict which version would appear a request was received that included that file name.

1.10 Web Content Authoring

C-Labs has implemented its own website using the Distributed Web Services support provided by the C-DEngine. We know that it works because we rely on it to provide content to customers and prospective customers who point their browsers at <http://www.c-labs.com>.

Our website is an example of a distributed web solution. Some of the resources on www.c-labs.com are provided from different nodes in the www.c-labs.com mesh. For example, the main static pages are on the www.c-labs.com cloud webserver while documents, whitepapers and other resources are stored on a different node in our secure network at C-Labs headquarters in Bellevue, Washington, USA. Developers can use the C-DEngine platform to build their own distributed web solutions as well.

The NMI of the C-DEngine (for example the “Factory-Relay Portal”) is an example of distributed content viewing and a central part of the Factory-Relay. It is important to understand that there is a big difference between traditional web-sites with static and dynamic content and the C-DEngine NMI (Natural Machine Interface).

A traditional web site receives a request for a web page, assembles the page on the server, and then injects dynamic content into the page. Once a page has been assembled, it is sent to the web browser for display to a user. This type of web page content is static. Updating the page requires a refresh on the entire page.

By contrast, a C-DEngine NMI page is highly dynamic in the browser. Content is constantly being updated on-the-fly without a user having to manually refresh a page. In fact, when a user manually refreshes a page by clicking a browser's refresh button, we consider this an event with possibly security implications. To maintain the integrity of your data, the NMI responds by dropping a session and requiring users to log in again.

Note: this is true as of Version 4.206 of the C-DEngine. In a later version, we want to resume a session after F5 is pressed.

Chapter 2 Creating HTTP Interceptors in a C-DEngine Plugin

2.1 Overview

This article describes how HTTP Interceptors enable C-DEngine plugins to customize the actions of the C-DEngine's web server based on the value of the URL of the incoming resource request. An important aspect of this web server support is that it is limited to the specific C-DEngine nodes that are directly accessible to the web clients. In other words, other nodes in the mesh are not visible to the web client. This is illustrated in Figure 2.1.

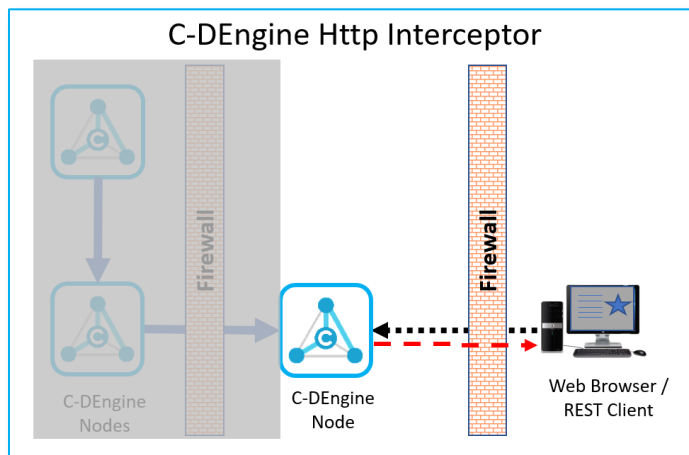


Figure 2.1. An HTTP Interceptor handles incoming web requests for a single C-DEngine node.

2.2 The C-DEngine Communication Core

The C-Labs C-DEngine supports a rich set of network communication features. It can run as a stand-alone web server, and can alternatively run within the Microsoft web server, Internet Information Service (IIS). In addition to the HTTP protocol, enhanced performance is available through support of web sockets. The primary implementation of C-DEngine is a .NET-based C# library, and a Javascript version enables operation from within all popular web browsers. A rich message passing mechanism enables node-to-node communication for nodes on a common mesh. In addition, secure communication is supported over the public internet by means of cloud nodes.

2.3 Extending the C-DEngine with Plugins

Plugins provide the primary means to extend the core features of C-DEngine. A Plugin is a .NET-compatible DLL created using the C-DEngine SDK. A C-DEngine plugin can register "interceptor" callback functions to listen for and respond to inbound HTTP traffic. In this way, a plugin can use the network to send or receive any data that fall within the support that the HTTP protocol provides.

2.4 Inbound HTTP Traffic

Among the standard set of HTTP methods, the C-DEngine itself makes use of these two: `GET` and `POST`. The `GET` method asks that a resource (such as a file) be retrieved and provided to the caller. The `POST` method moves data in the opposite direction, so that a caller can send a resource to a C-DEngine plugin. (The maximum size for an incoming resource size is 25 MB.)

When integrating an HTTP interceptor into a C-DEngine plugin, care should be taken to avoid conflicts with resource names that the C-DEngine reserves for its own uses. Table 2.1 shows a list of reserved resource names.

<pre>/ /BROWSERCONFIG.XML /CDE /CDECLEAN.ASPX /CDEUSERSTATUS /CSS /DEVICE.XML /DEVICEREG.JSON /EVTLOG.RSS /IMAGES /IPXIBOOT.PXI /JS /NONE /NMIEXT /SITEMAP.XML /SYSLOG.RSS</pre>
--

Table 2.1. Reserved C-DEngine URLs.

2.5 Static HTML Pages

The C-DEngine supports the ability to serve up static web pages from the C-DEngine's data folder, `ClientBin`. Any user, including anonymous users, can access these pages. If a file named `relay.html` was placed into the `ClientBin` folder with contents like the following:

```
<html>
  <body>
    <h1>This is My Relay Server</h1>
  </body>
</html>
```

That file would be available using a URL like this: <http://localhost:8700/relay.html>. The value for the port (8700) is defined in the startup configuration of the application that hosts the C-DEngine.

2.6 Dynamic HTML Pages

The C-DEngine also supports dynamically created HTML pages. The same `relay.html` page can be dynamically created using an **HTTP Interceptor**. An HTTP interceptor is a function that gets called when a specified path is received by the C-DEngine's http handler. To start an interceptor, a plugin connects a URL and a specified callback function by calling the `RegisterHttpInterceptorB4` function. Here is an example of an interceptor being registered from within the standard plugin initialization function, `Init()`:

```
public bool Init()
{
    if (!mIsInitStarted)
    {
```

```

    mIsInitStarted = true;

    TheCommCore.MyHttpService.RegisterHttpInterceptorB4(
        "/relay.html", sinkRelay);

    mIsInitCompleted = true;
    MyBaseEngine.ProcessInitialized();
}
return true;
}

```

The function `RegisterHttpInterceptorB4` is defined in the C-DEngine SDK as follows:

```

void RegisterHttpInterceptorB4(
    string pUrl,           // Target Url
    Action<TheRequestData> sink) // Intercept function

```

In the code sample, the first parameter, `pUrl`, is `"/relay.html"`. When a URL is requested that starts with this target URL, the following function is called to fulfill the resource request:

```

private void sinkRelay(TheRequestData pRequest)
{
    pRequest.ResponseMimeType = "text/html";
    pRequest.ResponseBufferStr =
        "<html>"
        + "    <body>"
        + "        <h1>This is My *Dynamic* Relay Server</h1>"
        + "    </body>"
        + "</html>";
    pRequest.ResponseBuffer =
        TheCommonUtils.CUTF8String2Array(
            pRequest.ResponseBufferStr);
    pRequest.StatusCode = (int)eHttpStatusCode.OK;
    pRequest.DontCompress = true;
    pRequest.AllowStatePush = false;
}

```

Although this example uses the same target file name as the previous example, note that we've changed the text slightly to make sure that we are, in fact, calling the dynamic html page instead of the static one. Figure 2.2 shows the resulting page in a web browser.

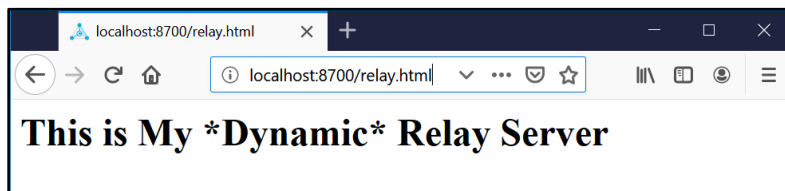


Figure 2.2. A dynamic HTML page as it appears in a browser window.

This example of a dynamic HTML page targets a single file name. While this certainly works, a more common case would be to serve up multiple pages for a given interceptor. To accomplish that, we could change the target URL to a folder name like `"/dynamic"` and use that value as the `pUrl`

parameter that we pass to `RegisterHttpInterceptorB4`. We might then access "relay.html" with a URL like this: <http://localhost:8700/dynamic/relay.html>.

2.7 Network Request and Response in `TheRequestData`

A key part of creating an HTTP interceptor involves understanding the incoming request and formulating an appropriate response. In other settings, these tasks are accomplished by reading a request from an incoming object and putting the return values in a response object. In a C-DEngine http interceptor, both incoming request fields and outgoing response data are packaged with one structure: `TheRequestData`. Details about the most important `TheRequestData` property fields are provided in Table 2.2.

Property	Description
<code>cdeRealPage</code>	A subset of the URL that identifies the requested page and local path, without a server name or any properties. For example, this URL http://localhost:10/my/page?num=12 , generates a value for <code>cdeRealPage</code> of <code>"/my/page"</code> .
<code>ClientCert</code>	An object with certificate information. The specific format is dependent on the type of web server. For example, in <code>AspNet</code> the certificate objects are of type <code>System.Web.HttpClientCertificate</code> . In an <code>HttpListener</code> type web server, the certificates are of type <code>System.Security.Cryptography.X509Certificates.X509Certificate2</code> .
<code>Header</code>	A dictionary field initialized to hold the headers of the incoming request.
<code>HttpMethod</code>	A string with the name of the HTTP request method. For example, GET, POST, OPTION or DELETE.
<code>HttpVersion</code>	Provides the HTTP version from an incoming request, formatted as a double value. For example, 1.0 or 1.1.
<code>HttpVersionString</code>	Provides the HTTP version from an incoming request, formatted as a string. For example, "1.0" or "1.1".
<code>PostData</code>	A byte array holding data sent with an incoming post request.
<code>PostDataLength</code>	An integer value indicating the length of the data in the <code>PostData</code> array.
<code>RequestUri</code>	A <code>Uri</code> type field containing the URL of the incoming request. For example, all of the details in a URL like this

Property	Description
	http://localhost:10/my/page?num=12 are available within various members of <code>RequestUri</code> .
<code>ResponseBuffer</code>	A byte array property for response being returned to the sender. When an interceptor sends return data to the client, this field holds the data to be returned.
<code>ResponseBufferStr</code>	The response being sent to the client by an http interceptor, formatted as a string.
<code>ResponseEncoding</code>	A string value used to populate the "Content-Encoding" field in the header of the response. Examples include "gzip", "compress", "deflate", "identity", and "br".
<code>ResponseMimeType</code>	A string indicating the format, or "media type", of the returned data. This is initially set to the value in the <code>ContentType</code> field of the incoming request. Examples include "text/html", "text/xml", "text/json", "image/png", "application/javascript", and "application/json". There is a helper function to provide a mime type based on a file's extension. That function is <code>TheCommonUtils.GetMimeTypeFromExtension</code> .
<code>StatusCode</code>	An integer for the HTTP Status Code. Initially set to zero, an interceptor function must set this value to indicate that a request has been handled. Possible values include: <ul style="list-style-type: none"> • Not Handled (0) • OK (200) • NotFound (404) • NotAcceptable (406) • ServerError (500) • RequestTimeout (408) These values are defined in the <code>eHttpStatusCode</code> enum.
<code>UserAgent</code>	String value holding the <code>UserAgent</code> field of an incoming http request. For example: "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:69.0) Gecko/20100101 Firefox/69.0"
<code>SessionState</code>	A <code>TheSessionState</code> object that enables the tracking of individual users when the users make multiple calls to an interceptor. The member properties of <code>SessionState</code> are provided in Table 2.3.

Table 2.2. Important property values in `TheRequestData`.

An http interceptor uses the following fields to determine what the incoming request is:

- `RequestUri` – provides the URL used to create the request, including the full path and all parameters.

And then an http interceptor fills in the following fields as part of the response:

- `ResponseBuffer` – the data being returned.
- `ResponseMimeType` – the type of data being returned in `ResponseBuffer`.
- `StatusCode` – set to 200 (OK) to indicate the request has been handled.

Property	Description
<code>CID</code>	A <code>Guid</code> value with the current user id, which can be validated using various user manager functions to check user permissions.
<code>cdeMID</code>	A <code>Guid</code> value that can be used to store a unique identifier for a session. This is similar to the <code>cdeMID</code> field used by the C-Engine as a unique ID for things. An http interceptor can use this field for a similar purpose, to help persist data between calls from a client.
<code>CustomData</code>	A <code>string</code> value for non-standard storage of persistent session state information in an exchange between clients and servers. For example, an access token or API key.
<code>CustomDataLong</code>	A <code>long</code> value for non-standard storage of persistent session state information in exchanges between clients and servers. For example, a counter value.
<code>EntryTime</code>	Time stamp indicating when the session was created.
<code>LastAccess</code>	Time stamp indicating when session state was last accessed.
<code>UserAgent</code>	A string with the User Agent of the initial request. This should have the same value as the <code>UserAgent</code> property in <code>TheRequestData</code> .
<code>WebPlatform</code>	A string indicating the platform from which the request was sent. Valid values include "Mobile", "Desktop", "XBox", "IoT", "HoloLens".

Table 2.3. Important property values in `TheSessionState`.

2.8 Unregistering an HTTP Interceptor

There might be the need to remove an HTTP Interceptor for one reason or another. The function that you call to do this is `UnregisterHttpInterceptorB4`. Here is an example of that function being called to uninstall the interceptor created earlier in this paper:

```
TheCommCore.MyHttpService.UnregisterHttpInterceptorB4("/relay.html");
```

2.9 Detecting Conflicts between Interceptors

There are a few types of conflicts to watch for when creating interceptors. The first thing to know is that interceptors are stored in a dictionary container, and that the URL is the dictionary keys. This means that there cannot be two URLs that are exactly the same. But it is possible to have URLs that are similar.

Consider the problem if there are two plugins that each register an http interceptor for the URL "/dynamic". This is possible, and when it happens the last one to register the URL will win. In other words, the callback for the plugin that registered the interceptor *last* will be called when the target URL is contained in an incoming http request. The callback for the first plugin to register is not called. If you suspect this is happening, you can look for the following message in the C-DEngine system log:

```
TXT: Warning: Replaced existing http intercept before.  
PLS: URL passed into call to register interceptor.  
Engine Name: HttpService  
Debug Level: eDEBUG_LEVELS.VERBOSE (2)
```

In addition, it is possible to have two interceptors with similar URLs. Consider, for example, these two URLs: /dynamic and /dynamic/relay.html. If two different plugins registered interceptors using these two URLs, it is possible that there would be a conflict if both plugins were looking for a request for the resource named /dynamic/relay.html. To help identify when this situation occurs, there is a system log message generated when a URL is registered that has an overlap with an existing interceptor. Here are the details of that system log message:

```
TXT: Warning: Possible conflict in URLs pass to http intercept before.  
PLS: InputUrl:{pUrl} ExistingUrl:{strKey}  
Engine Name: HttpService  
Debug Level: eDEBUG_LEVELS.ESSENTIALS (1)
```

2.10 Interceptor Helper Function

To help in the development, debugging, and unit testing of http interceptor functions, there is a helper function that calls into the http call stack at the exact place where interceptor functions are called. The name of the function is `TheCommonUtils.GetAnyFile`. `GetAnyFile` tries to load the requested resource first from Disk under /ClientBin then tries to load the file from the Plugins Internal Embedded Resources (any file marked as "EmbeddedResource" in Visual Studio).

Here is an example of the function being called to access a file identified by the URL "/dynamic/relay.html".


```
// using nsCDEngine.ViewModels;

TheRequestData p = new TheRequestData();
p.cdeRealPage = "/dynamic/relay.html";
TheCommonUtils.GetAnyFile(p);
If (p.StatusCode > 0) { . . . }
```

2.11 Providing REST-Style Data

An http interceptor can provide REST (Representational State Transfer) support as well. For a simple example, we have a database of record holders in track and field for the distance of one mile. To keep things extremely simple, we limit the size of our database and define the class `TheRecordHolder` has having the following fields:

```
public class TheRecordHolder
{
    public int id;
    public string strTime;
    public string strName;
    public string strNationality;
    public string strDate;
}
```

We create an http interceptor that is triggered when the URL starts with `/api/MileRecordHolder` as follows:

```
TheCommCore.MyHttpService.RegisterHttpInterceptorB4("/api/MileRecordHolder",
sinkMileApiInterceptor);
```

Our interceptor function is set up to handle requests like the following:

```
localhost:8700/api/MileRecordHolder?id=6
```

That is, the interceptor handles requests where the record id is provided. The interceptor, `sinkMileApiInterceptor`, is as follows:

```
/// <summary>
/// sinkThingApiInterceptor - Interceptor Function
/// </summary>
/// <param name="pRequest"></param>
public void sinkMileApiInterceptor(TheRequestData pRequest)
{
    string strQuery = pRequest.RequestUri.Query;
    if (strQuery.StartsWith("?"))
        strQuery = strQuery.Substring(1);
    if (!strQuery.StartsWith("id"))
        return;

    int iValue = strQuery.IndexOf("=");
```

```

if (iValue > -1 && iValue+1 < strQuery.Length)
{
    int id = -1;
    if (int.TryParse(strQuery.Substring(iValue+1), out id))
    {
        if (id > 0 && id <= TheRecordHolder.aData.Length)
        {
            ProvideResponseData(pRequest, id);
        }
    }
}
}

```

If the client's request can get met, the `ProvideResponseData` function is called. This function fills in the required values in `TheRequestData` object so that the data can be provided to the client:

```

public void ProvideResponseData(TheRequestData p, int id)
{
    p.ResponseMimeType = "application/json";
    TheRecordHolder trh = TheRecordHolder.aData[id - 1]
    string strJ =
TheCommonUtils.SerializeObjectToJSONString<TheRecordHolder>(trh);
    p.ResponseBuffer = TheCommonUtils.CUTF8String2Array(strJ);
    p.StatusCode = (int)eHttpStatusCode.OK;
    p.DontCompress = true;
    p.AllowStatePush = false;
}

```

2.12 Other IHttpInterceptor Functions

Table 2.4 summarizes other functions in the `IHttpInterceptor` interface that may also be of interest. You might notice that there are a few other types of interceptors. In addition to the "before" interceptor that we discussed earlier, there is also an "after" interceptor. The "B4" in the function we introduced earlier means "before" because `RegisterHttpInterceptorB4` is called relatively early in the page rendering process. By contrast, `RegisterHttpInterceptorAfter`, is called quite late in the page rendering process, when a request that matches the URL pattern has not been processed by any other handler.

Function	Description
<code>RegisterHttpInterceptorB4</code>	This function registers an interceptor callback function, and a target URL path, to be called BEFORE the C-DEngine processes the request further. If this function is called from inside an isolated plugin, the request is forwarded to the master node.
<code>UnregisterHttpInterceptorB4</code>	Removes a registered interceptor function from the list of before interceptors.
<code>RegisterHttpInterceptorAfter</code>	This function registers an interceptor callback function, and an associated target

Function	Description
	URL, to be called for unhandled http processing requests. An unhandled request is one where the returned status code is either zero (unhandled) or 408 (request timeout).
UnregisterHttpInterceptorAfter	Removes a registered interceptor function from the list of after interceptors.
RegisterGlobalScriptInterceptor	This function registers a callback function to provide one or more global Javascript file. If this function is called from inside an isolated plugin, the request is forwarded to the master node.
RegisterStatusRequest	This function allows a plugin to register a callback function that adds HTML to the web page that is displayed when the user summons the <code>cdeStatus.aspx</code> status page with either ALL or DIAG specified as a parameter.
CreateHttpHeader	A helper function to create a header from the provided <code>TheRequestData</code> .
cdeProcessPost	This is the main http processing function of the C-DEngine. You can call this function to inject an HTTP request into the C-DEngine. Among other uses, it could be used for unit-testing of an http interceptor.
GetGlobalScripts	Returns a list of all globally registered Javascript scripts.

Table 2.4. Summary of available functions in the `IHttpInterface`.

2.13 Authenticating REST Users

By default, incoming requests to an HTTP Interceptor are not authenticated. The C-DEngine NMI component supports user accounts and passwords in a manner that addresses the inherently insecure nature of a web browser. That component is not accessible to a plugin running in an application host, which by its very nature is quite a bit more secure than that of a browser.

Adding user authentication support to a REST server is as straightforward as defining the structure of the income URL and the names of the required fields. Here are the key portions of an implementation that C-Labs is happy to share as a working sample. Here is the HTTP Interceptor function that is the doorway to the REST server support:

```

/// <summary>
/// sinkThingApiInterceptor - Called when our URL is referenced.
/// </summary>
/// <param name="pRequest">HTTP request and response fields.</param>
public void sinkThingApiInterceptor(TheRequestData pRequest)
{

```

```

string strLocalPathLower = pRequest.RequestUri.LocalPath.ToLower();
string strQuery = pRequest.RequestUri.Query;
Dictionary<string, string> aParameters =
utils.ParseQueryParameters(strQuery);

if (strLocalPathLower == "/api/milerecordholder/logon")
{
    utils.ValidateUserCredentials(pRequest, aParameters);
}
else if (strLocalPathLower == "/api/milerecordholder/query")
{
    if (utils.IsTokenValid(aParameters))
    {
        if (aParameters.ContainsKey("id"))
        {
            string strValue = aParameters["id"];
            int id;
            if (int.TryParse(strValue, out id))
            {
                if (id > 0 && id <= TheRecordHolder.aData.Length)
                {
                    ProvideResponseData(pRequest, id);
                }
            }
        }
    }
}
}

```

To parsing parameters from the incoming URL, there is the `ParseQueryParameters()` function:

```

public static Dictionary<string, string> ParseQueryParameters(string
pQuery)
{
    Dictionary<string, string> pOut = new Dictionary<string, string>();
    string[] astrParameters = pQuery.Split(new char[2] { '?', '&' },
        StringSplitOptions.RemoveEmptyEntries);

    foreach (string str in astrParameters)
    {
        string[] astrResult = str.Split(new char[1] { '=' },
            StringSplitOptions.RemoveEmptyEntries);
        if (astrResult.Length == 2)
        {
            string key = astrResult[0].ToLower();
            string value = astrResult[1].ToLower();
            pOut[key] = value;
        }
    }

    return pOut;
}

```

We check whether the user and password value are valid in the `ValidateUserCredentials()` function:

```

/// <summary>
/// Called using a URL like this: /xxxx/xxxxx/Logon?user=xxxx&pwd=yyyyy
/// </summary>
/// <param name="pRequest">TheRequestData receive in http
interceptor.</param>
/// <param name="aParameters">Parameter values parsed into a
dictionary.</param>
///
public static bool
ValidateUserCredentials(TheRequestData pRequest,
                        Dictionary<string, string> aParameters)
{
    bool bSuccess = false;

    if (aParameters != null & aParameters.Count > 1)
    {
        string strUser = aParameters["user"].ToString();
        string strPwd = aParameters["pwd"].ToString();

        if (strUser != null && strPwd != null)
        {
            if (strUser == "myuser" && strPwd == "asterisks")
            {
                Guid pToken = CreateToken(pRequest);
                SaveToValidTokenTable(pToken);

                // Return access token to caller.
                SetResponseValue(pRequest, pToken);
                bSuccess = true;
            }
        }
    }

    if (bSuccess == false)
    {
        SetEmptyResponse(pRequest);
    }

    return bSuccess;
}

```

We return a `Guid` as access token if the user has entered valid credentials.

```

public static void SetResponseValue(TheRequestData pRequest, Guid
guidValue)
{
    pRequest.ResponseMimeType = "application/json";
    string strJson =
TheCommonUtils.SerializeObjectToJSONString<Guid>(guidValue);
    pRequest.ResponseBuffer =
TheCommonUtils.CUTF8String2Array(strJson);
    pRequest.StatusCode = (int)eHttpStatusCode.OK;
    pRequest.DontCompress = true;
}

```

```
pRequest.AllowStatePush = false;
}
```

We return an empty response for invalid credentials.

```
public static void SetEmptyResponse(TheRequestData pRequest)
{
    pRequest.ResponseMimeType = "text/html";
    pRequest.ResponseBuffer = new byte[1];
    pRequest.ResponseBuffer[0] = 0;
    pRequest.StatusCode = (int)eHttpStatusCode.OK;
    pRequest.DontCompress = true;
    pRequest.AllowStatePush = false;
}
```

The various aspects of creating, storing, and validating access tokens is provided in this set of functions. In this implementation, tokens have a limited life span. A user must authenticate again once an access token has expired.

```
private static Dictionary<Guid, DateTime> dictValidAccessTokens = null;

private static void InitAccessTokenTable()
{
    dictValidAccessTokens = new Dictionary<Guid, DateTime>();
}

private static Guid CreateToken(TheRequestData pRequest)
{
    Guid pNewToken = Guid.NewGuid();
    return pNewToken;
}

private static bool SaveToValidTokenTable(Guid pToken)
{
    bool bSuccess = false;

    try
    {
        if (dictValidAccessTokens == null)
            InitAccessTokenTable();

        DateTime pNow = DateTime.Now;
        dictValidAccessTokens[pToken] = pNow;

        bSuccess = true;
    }
    catch { }

    return bSuccess;
}

public static bool IsTokenValid(Dictionary<string, string> aParameters)
{
    bool bSuccess = false;
```

```

CleanupOldTokens();

if (aParameters.TryGetValue("key", out string strToken))
{
    if (!String.IsNullOrEmpty(strToken))
    {
        Guid guidToken = TheCommonUtils.CGuid(strToken);
        DateTime dt;

        if (dictValidAccessTokens != null)
        {
            if (dictValidAccessTokens.TryGetValue(guidToken, out
dt))
                bSuccess = true;
        }
    }
}

return bSuccess;
}

private static void CleanupOldTokens()
{
    if (dictValidAccessTokens != null)
    {
        List<Guid> keysToRemove = new List<Guid>();

        foreach (KeyValuePair<Guid, DateTime> kvp in
dictValidAccessTokens)
        {
            DateTime dt = DateTime.Now;
            TimeSpan ts = (dt - kvp.Value);
            if (ts.TotalHours > 4)
                keysToRemove.Add(kvp.Key);
        }

        foreach (Guid g in keysToRemove)
        {
            dictValidAccessTokens.Remove(g);
        }
    }
}

```

A later version of the C-Engine may offer this a token-based authentication feature as part of the core APIs.

Chapter 3 Accessing Mesh Resources

3.1 Overview

This chapter builds on the material of earlier chapters. In Chapter 1, we introduced Distributed Web Services (DWS) as a feature that allows static content in one mesh node to be shared with other mesh nodes. In Chapter 2, we covered the creation of HTTP Interceptors, which allow incoming web requests to serve up dynamic content: for example, an html page or REST API.

In this chapter, we bring together the handling of incoming web requests with the ability to access any resource from anywhere in the mesh. This ability already exists for standard static browser content such as html pages or graphic images. In this chapter, we extend the reach of incoming web requests to enable access to any type of dynamic content. In particular, we show how one node can receive an incoming REST API request and use other mesh node to assist in fulfilling those requests (see Figure 3.1).

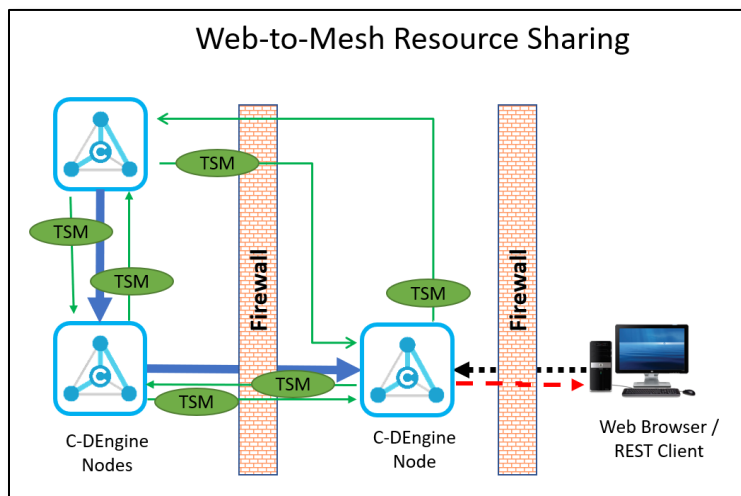


Figure 3.1. Messages (TSM = "The System Message") enable communication between C-DEngine nodes within the same mesh.

3.2 The C-DEngine Mesh

A C-DEngine mesh consists of two or more connected C-DEngine nodes. As has been mentioned earlier, the `ServiceRoute` setting allows one C-DEngine node to "point-to" one (or more) other C-DEngine node(s).¹ A C-DEngine mesh consists of the collection of all C-DEngine nodes that (a) share a common Scope ID and (b) are "pointed to" by other C-DEngine nodes, and (c) for which every node is reachable by at least one route. Note that a route may involve traversing one or more intermediate nodes.

3.3 Messages

¹ To be able to access the http service from the Windows 10 desktop, run the application host as an administrator.

Nodes communicate with each other by means of messages. To send a message, the TSM data structure is populated and transmitted by one of these C-DEngine API calls:

- `TheCommCore.PublishCentral` – transmit a message to every node in a mesh. This is somewhat expensive, especially when a mesh is very large. It provides a means to advertise a service to all mesh nodes.
- `TheCommCore.PublishToNode` – transmit a message to a specific node. This requires the sender to know the node ID for the target node (of type `Guid`).
- `TheCommCore.PublishToOriginator` – transmit a message to a specific node to provide a response to a previously received message.

Messages provide the primary means for communicating between nodes. The most important TSM fields are summarized in Table 3.1.

Field	Description
ENG	A string field for the engine (plugin service class) that owns the message.
TXT	A string field for the message command. Examples include CDE_INITIALIZED, CDE_PING. One or more strings can be added as parameters. By convention, a colon (":") is used to separate a command from parameters. For example: "REPLY_NODEID:039BEECB-0130-4EA3-BBAA-1A3CBF4C8B27.
PLB	(Payload binary) A byte array for binary parameters.
PLS	(Payload string) A string field for longer parameters. When PLB is empty, the PLS can be compressed and placed into the PLB to reduce the network overhead for the message.

Table 3.1. Summary of important fields in a TSM object.

In the context of using messages to transmit REST requests, one approach would involve copying the request from the incoming URL and including that string in the `TXT` field of a message. A reply message could similarly use the `TXT` field to hold whatever reply is being provided. Alternatively, when longer replies are expected then the `PLS` field could be used to hold the details of the reply.

3.4 A REST Server and REST Client

The best way to see how this can operate is to look at a sample implementation. Figure 3.2 shows a sample host application that is set up to configure several REST servers. There are configuration settings that enable the nodes to connect with one another, and also settings to enable different parts of the REST server operation.

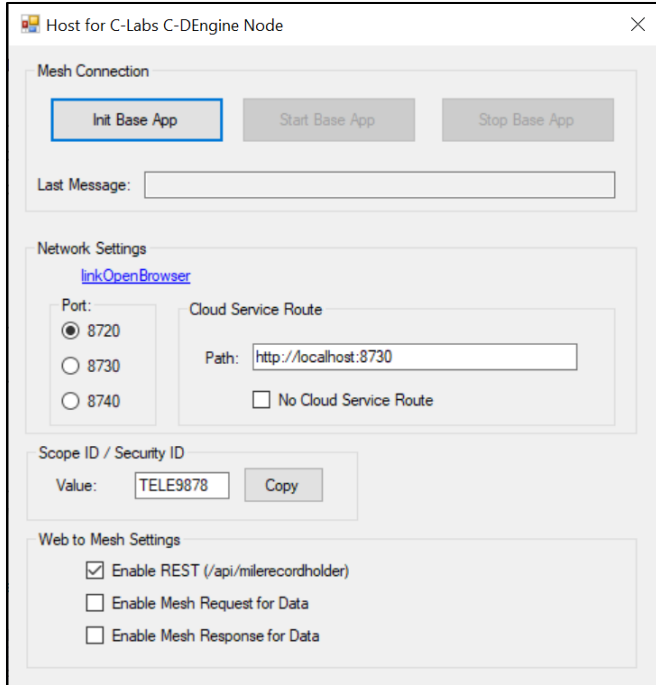


Figure 3.2. The configuration settings for a sample REST Server.

Figure 3.3 shows a sample REST Client that is set up to call into the REST Server. The REST Client is a Windows Forms application without any support from the C-Engine API.

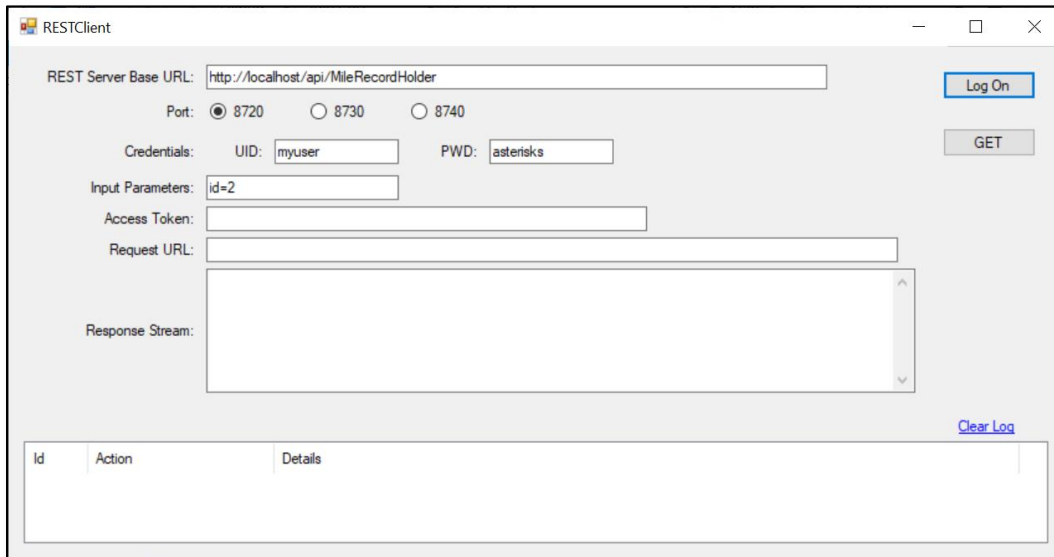


Figure 3.3. A sample REST Client for calling into our REST Server.

3.5 Simplest REST Client to REST Server Operation

You can start the two sample programs in any order that you'd like. It doesn't make a difference because neither one performs any automatic network actions. As you may expect, the REST server

needs to be operational before we expect to get any reasonable results from it. The default configuration is set up to be operational without requiring any changes.

To start the REST server, click the "Init Base App" button. You see a message that says "Success initializing Base App." Next, click the button labeled "Start Base App." You see a message that says, "Success starting Base App. Click URL to launch browser." The REST server is now operational.

In the REST Client, click the button labeled "Log On". This sends a request to the REST server for authentication and, if that is successful, an access token is returned. Figure 3.4 shows the REST client immediately after successful authentication. Notice that the field labeled "Access Token" has been populated with a Guid value that gets included in the URL to prove that we have the right to use the available services.

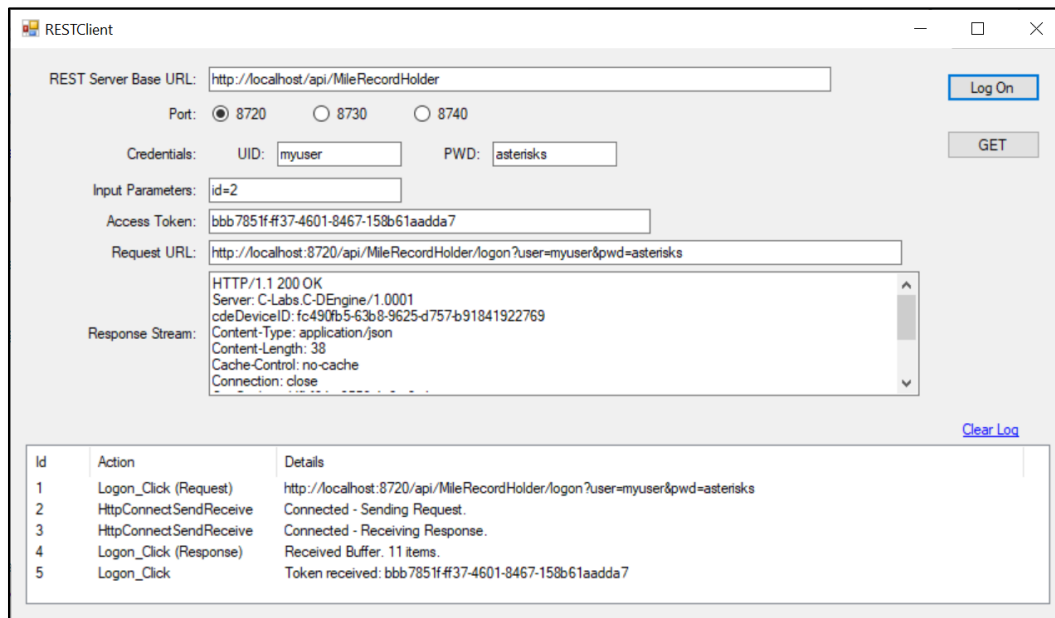


Figure 3.4. The REST client after successful log on authentication.

You may also note that the box labeled "Request URL" shows the complete URL that was sent to the REST server. You can copy this to the address line of a browser to experiment with the REST server directly. Notice also that the complete response stream is shown in the "Response Stream" field, including the header information. You need to scroll down to find the body of the response.

This REST server provides the same information as the sample REST server in the earlier chapter, namely information about people who have held the record for running a mile in the fastest time. The "Input Parameters" field is set up to request the record with an id value of 2. You can modify this field (the sample only has 5 records) then click the "GET" button.

The results of our query appear in two places: (1) At the end of the response stream, and (2) at the end of the transaction log. Figure 3.5 and 3.6 show the results from each of these two locations.

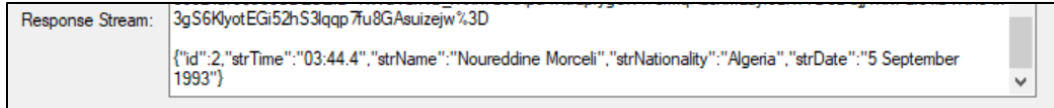


Figure 3.5. Results of the query from the response stream.

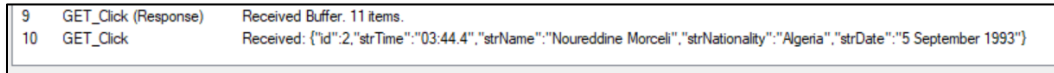


Figure 3.6. Results of the query in the transaction log.

3.6 REST Server with C-Engine Node Messages

To configure the REST server for demonstrating node-to-node message sending, start two more copies of the REST server sample. In one of them, set the port value to "8730". And in the second one, set the port value to "8740". The sample has been set up so that when these port values are selected, the other configuration settings are automatically configured to demonstrate the node-to-node support for the REST server. Figures 3.7 and 3.8 shows two copies of the REST server set up for this demonstration.

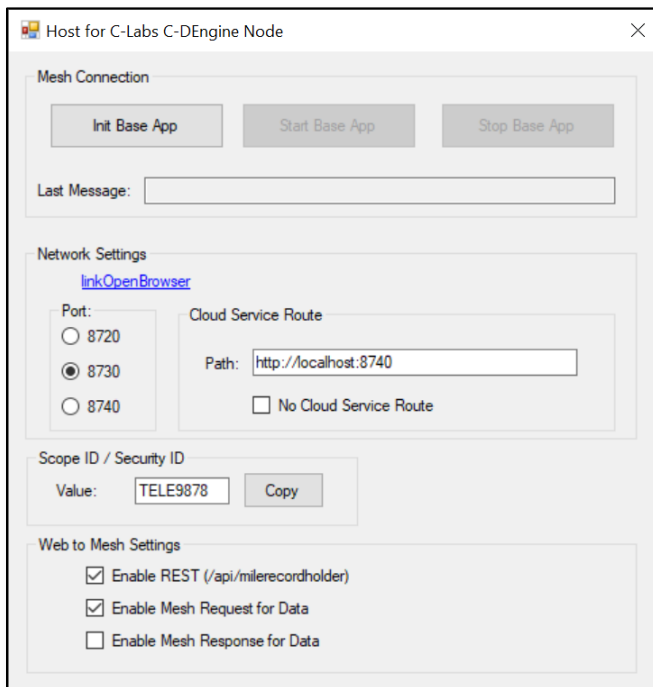


Figure 3.7. The REST server set up to receive REST requests and forward them to another node for handling.

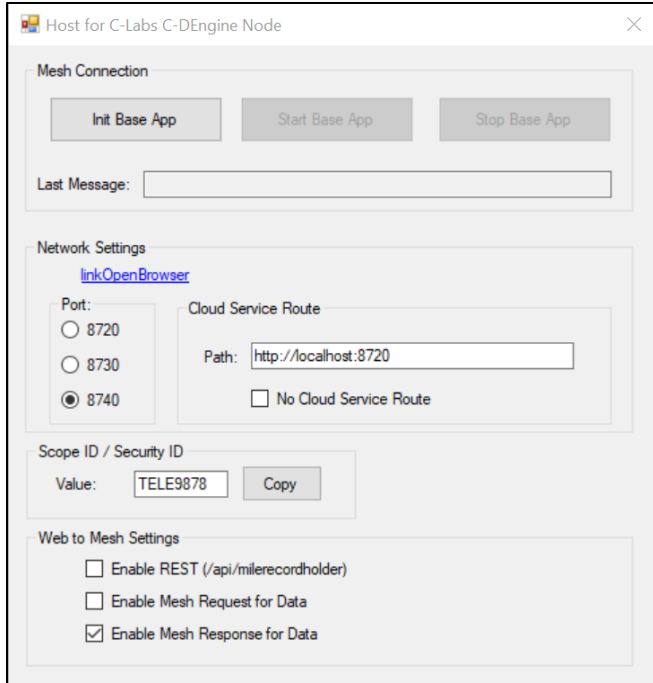


Figure 3.8. The REST server set up to receive mesh requests for data from another node.

Notice that the REST server that is using port 8730 has a cloud service route configured for <http://localhost:8740>. That is the key piece of information that is needed to tie the two nodes together. Notice also that the other REST server is set up to use port 8740.

Initialize and start each of the REST servers by clicking the "Init Base App" and "Start Base App" buttons. Make sure you see the "Success starting" message for each of them.

In the REST client, set the port to "8730" by clicking the appropriate radio button. Although we received an access token when we access the REST server at port 8720, that access token is not recognized by the two new REST servers. After setting the port, click the "Log On" button to receive an access token. Then click the "GET" button to retrieve the data from the REST server. To the REST client, there is nothing to indicate that the data came from anywhere other than the REST server that was referenced in the base URL.

Figure 3.9 and 3.10 show the C-DEngine Event Viewer application, with details on messages that were received by each of the nodes. Figure 3.9 shows the messages sent to the node with port 8730, which is the node that was the REST server.

```
CDMyWebToMeshSample.cdePluginService1: IncomingMessage 466 bytes [MsgMileRecordHolder]
    TXT = MsgMileRecordHolder:C835E8F271974BB7BAFD9ECD5B41A5DF:
    PLS = {"id":2}
```

Figure 3.9. Event Viewer output showing a message sent to the port 8740 node.

```

CDMyWebToMeshSample.cdePluginService1: IncomingMessage 429 bytes [MsgMileRecordHolder]
    TXT = MsgMileRecordHolder:C835E8F271974BB7BAFD9ECD5B41A5DF:
    PLS = {"id":2}
ContentService: IncomingMessage 430 bytes [MsgMileRecordHolder_RESPONSE]
    TXT = MsgMileRecordHolder_RESPONSE:C835E8F271974BB7BAFD9ECD5B41A5DF:
    PLS = {"id":0,"data":{"id":2,"strTime":"03:44.4","strName":"Noureddine Morce

```

Figure 3.10. Event Viewer output showing messages sent to the port 8730 node.

Notice the value in the TXT fields for the various messages. The `MsgMileRecordHolder` message is a request to provide the data. There is also a set of messages that have the same name with the value `"_REPNSE"` at the end.

3.7 Message Requests and Responses

There is a very common design pattern within C-Engine plugins. That pattern involves sending a message to another node, and then waiting for the other node to reply. To support this design pattern, the C-Engine provides a support class named `TheCommRequestResponse`. This class was used in the REST server sample. Here are the elements which are part of this sample:

- 1) A message request class (`MsgMileRecordHolder`).
- 2) A message response class (`MsgMileRecordHolderResponse`).
- 3) A case statement in the `HandleMessage()` function for receiving the request.
- 4) A call to one of the support functions within `TheCommRequestResponse`. In the sample, the function called is `PublishRequestJSONAsync`.

Each of these elements appears below:

The request data structure, `MsgMileRecordHolder`, appears here:

```

public class MsgMileRecordHolder
{
    public int id;
}

```

The response data structure, `MsgMileRecordHolderResponse`, is shown here:

```

public class MsgMileRecordHolderResponse
{
    public int id;
    public TheRecordHolder data;
}

```

The `HandleMessage` function is shown here:

```

public void HandleMessage(ICDEThing sender, object pIncoming)
{
    TheProcessMessage pMsg = pIncoming as TheProcessMessage;
    if (pMsg == null) return;

    string[] cmd = pMsg.Message.TXT.Split(':');

```

```

switch (cmd[0])
{
    case "CDE_INITIALIZED":
        MyBaseEngine.SetInitialized(pMsg.Message);
        break;
    case nameof(MsgMileRecordHolder):
        if (g_EnableMeshDataResponse)
        {
            // Request from another node for mile record holder
information.
            var request =
TheCommRequestResponse.ParseRequestMessageJSON<MsgMileRecordHolder>(pMs
g.Message);
            var MsgResponse = new MsgMileRecordHolderResponse();
            if (request != null)
            {
                MsgResponse.data =
TheRecordHolder.QueryRecordHolder(request.id);
            }

TheCommRequestResponse.PublishResponseMessageJson(pMsg.Message,
MsgResponse);

            MsgResponse = null; // Prevent legacy response
handler.
        }
        break;
    default:
        break;
}
}
}

```

The function from the REST server sample that brings all of this together is named MeshQueryRecordHolder. Here is the complete function:

```

public static TheRecordHolder MeshQueryRecordHolder(int idRecord, Guid
node, string strEngineName, Guid cdeMIdThing)
{
    TheRecordHolder trh = null;

    // Package up request info.
    MsgMileRecordHolder msgRequest = new MsgMileRecordHolder()
    {
        id = idRecord
    };

    // Start asynchronous task to send a message and wait for a reply.
    // Sends a message named nameof(MsgMileRecordHolder)
    // Receives a reply named nameof(MsgMileRecordHolderResponse)
    // See function "HandleMessage" for actual handling.

    Task<MsgMileRecordHolderResponse> t = null;
    try
    {
        TheMessageAddress tma = new TheMessageAddress()

```

```

        {
            Node = Guid.Empty,
            EngineName = strEngineName,
            ThingMID = cdeMIDThing,
            SendToProvisioningService = false,
        };
        t = TheCommRequestResponse.PublishRequestJsonAsync<
            MsgMileRecordHolder, MsgMileRecordHolderResponse>
            (tma, msgRequest);
    }
    catch (Exception ex)
    {
        string strMessage = ex.Message;
    }

    // Wait for a bit
    t.Wait(20000);
    bool bTaskCompleted = t.IsCompleted;

    // Check for success.
    if (bTaskCompleted)
    {
        MsgMileRecordHolderResponse msgResponse = t.Result;
        trh = msgResponse.data;
    }

    return trh;
}

```

Here is the definition for the class `TheRecordHolder`, which is the central data providing class in this example and in another example from an earlier chapter.

```

public class TheRecordHolder
{
    public int id;
    public string strTime;
    public string strName;
    public string strNationality;
    public string strDate;

    public TheRecordHolder(int i, string t, string name, string nat,
string d)
    {
        id = i;
        strTime = t;
        strName = name;
        strNationality = nat;
        strDate = d;
    }

    public static TheRecordHolder[] aData = new TheRecordHolder[]
    {
        new TheRecordHolder( 1, "03:43.1", "Hicham El Guerrouj",
"Morocco", "7 July 1999" ),
    }
}

```



```
        new TheRecordHolder( 2, "03:44.4", "Noureddine Morceli",
"Algeria", "5 September 1993" ),
        new TheRecordHolder( 3, "03:46.3", "Steve Cram", "United
Kingdom", "27 July 1985" ),
        new TheRecordHolder( 4, "03:47.3", "Sebastian Coe", "United
Kingdom", "28 August 1981" ),
        new TheRecordHolder( 5, "03:48.4", "Steve Ovett", "United
Kingdom", "26 August 1981" ),
    };

    public static TheRecordHolder QueryRecordHolder(int id)
    {
        if (id > 0 && id <= TheRecordHolder.aData.Length)
            return TheRecordHolder.aData[id - 1];
        else
            return null;
    }
}
```

Chapter 4 Compare and Contrast C-Engine and REST API

This chapter compares C-Engine to REST. It provides a close look at the two, with details on specific similarities and specific differences between the C-Labs C-Engine and a typical REST server.


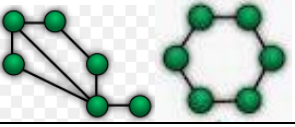
4.1 Overview

The C-Engine enables asynchronous communication, while REST APIs are built on the assumption of synchronous communication. Both C-Engine and REST support the HTTP protocol, while only C-Engine supports the more efficient web sockets. In addition to its support for asynchronous message delivery, C-Engine also supports the batching of multiple messages for great network throughput than a comparable set of REST calls.

There can be a great deal of variation between how various REST servers are implemented, and various REST servers use the HTTP protocol and URL in a wide range of styles. With C-Engine, by contrast, data is sent and delivered in JSON format. The C-Engine uses the message body for API commands and parameters, which allows C-Engine messages to be routable. By contrast, REST APIs require commands and parameters on the URL which prevents REST requests from being routed.

4.2 Feature Checklist Comparison

Table 4.1 provides a feature-by-feature comparison of key features of both C-Engine and the REST API.

	C-Engine	REST API
Network Transports		
TCP / IP supported?	Yes	Yes
HTTP / HTTPS supported?	Yes	Yes
Web Sockets supported?	Yes	No
Synchronous / Asynchronous Models		
Supports synchronous transactions?	Yes	Yes
Supports asynchronous transactions?	Yes	No
Persistent Connections (w/ HTTP 1.1 or web sockets)	Yes	Yes (No web socket support)
Network Topologies		
Star? 	Yes	Yes
Mesh? Ring? 	Yes	No

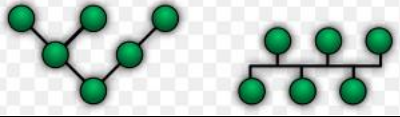
	C-DEngine	REST API
Tree? Bus? 	Yes	No
Network Models		
Client / Server	Yes	Yes
Peer to Peer	Yes	No
Data Request Models		
Request / Response	Yes (asynchronous queued content)	Yes (synchronous)
Publish / Subscribe	Yes	No
Broadcast	Yes	No
Data Security		
White List Trusted Nodes	Yes	No
Automatic Retry on Delivery Failure	No	No
Multi-Tenant Support Available	Yes	Yes

Table 4.1 Feature by feature comparison of C-DEngine to REST

4.3 Similarities

This section covers some of the similarities between the C-DEngine and REST. For example, both the C-DEngine and REST can be used to send a request / response interaction over a network between two entities. The two entities might be two processes running on a single computer, or two processes running on two different computers. What are other ways in which the two are similar?

4.3.1 HTTP Protocol

Both C-DEngine and REST both support the HTTP protocol, a foundation of the world-wide web. And, in fact, C-DEngine itself uses the REST protocol in some limited circumstances. The C-DEngine has a wide range of additional communication features not possible with REST, which will be the focus of the next section in this chapter.

4.3.2 JavaScript Friendly

Both C-DEngine and REST operate in a JavaScript-friendly manner. By this, we mean that C-DEngine and REST can be accessed from JavaScript code. Both C-DEngine and REST support the use of JSON (JavaScript Object Notation) formatted data objects.

4.3.3 Data Compression

Both C-DEngine and REST enable data compression to save network bandwidth when large blocks of data need to be moved. One potential sticking point involves choice of compression algorithm. When both client and server support a common algorithm, then that algorithm can be used to gain the various benefits of compression. But when a given client and a given server cannot find a common compression algorithm supported by both, then compression cannot be used. This can become an issue because the developer of a REST server is likely to be different from the developer of REST clients that use that server. This is less likely to be a problem with C-DEngine compression,

since the C-DEngine is likely to be both supplier and consumer of compressed data as it is moved from one node to another.

4.3.4 Use of SSL / TLS for Security

Connections made by HTTP and Web Sockets will have their data security encrypted when SSL / TLS transport encryption is enabled.

4.4 Differences between C-DEngine and REST

This section covers differences between C-DEngine and REST.

4.4.1 Synchronous vs. Asynchronous

One important difference between the C-DEngine and REST is that C-DEngine is asynchronous while REST is synchronous. Here are some examples to clarify what this means.

When a REST client makes a request of a REST server, the REST client must wait for the REST server to provide a response. To recover from situations when a REST server never responds, there is a timeout value (typically around 60 seconds) after which the REST client receives an error rather than a valid response. This synchronous behavior is one factor that keeps the REST interaction simple. There is no need to remember earlier transactions, nor any need to check back for forgotten requests. That simplicity comes at a cost, however, and that cost is paid in terms of delays in a REST server that cause delays in REST clients.

The C-DEngine, on the other hand, is inherently an asynchronous platform. When a request is made, in the form of a message being sent, the sender does not have to wait for the recipient to respond to the request. Instead, the sender can work on other things. There are, of course, situations in which a message sender does not wish to do anything but to wait for the response. There is a commonly required design pattern to send a message and wait for a reply, and there are several support classes within C-DEngine that make it very easy to do this. The overall benefit of the asynchronous nature of C-DEngine is that less time is spent waiting and more actual work gets done.

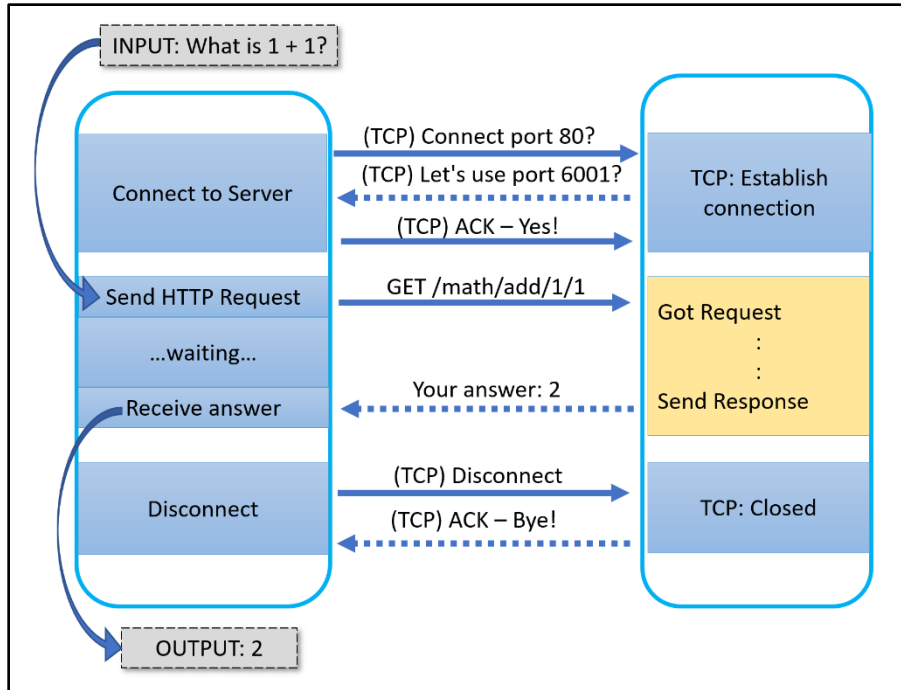


Figure 4.1. Details of HTTP Connect / Request / Close Model

4.4.2 HTTP vs. Web Sockets

In the section on similarities, it was mentioned that both C-DEngine and REST support the HTTP protocol. While on the subject of communication protocols, it's worth pointing out an important difference between C-DEngine support and REST support: the C-DEngine supports the Web Sockets protocol while REST does not. The differences between HTTP and Web Sockets are enough to be worth exploring in more detail.

- Opening and Closing of TCP Connections** – An HTTP connection involves opening a connection, sending a request, receiving a response, then closing the connection. This is illustrated in Figure 4.1, where the request “Add 1 to 1” causes the server to generate the response of “2”. In this figure, each arrow represents a transmission of some data either from the client to the server, or from the server to the client (dotted lines). There is a total of seven arrows, with four from the client and three from the server.

The actual “work” of this REST-like interaction is handled by two of these arrows: the “GET” from the client and the “Your answer” from the server. The other five represent overhead. There are ways to keep a connection open, and in fact the HTTP 1.1 protocol has this “keep alive” feature enabled by default. Nonetheless, there is no escaping the fact that using just the HTTP protocol requires many more open and close operations than use of the web sockets protocol.

To be fair, a web socket connection starts out as an HTTP connection which is then upgraded to a web socket connection. This means that the startup and shutdown costs of

both types of connections are the same. But a web socket connection is typically kept open for as long as the client and the server need to communicate with each other, which might be minutes, hours or days. And while an HTTP connection can be kept alive for a few minutes, the inevitable timeout (usually within a matter of minutes) means that additional HTTP requests require the overhead of establishing a new connection. For this reason, the overhead of starting and stopping a web socket connection can be amortized over many dozen (or even many thousands) of requests and responses. By contrast, the overhead of opening and closing an HTTP connection can only be amortized over a few (less than ten) requests.

This item alone suggests that the REST protocol can be recommended when just a few requests are required. With few requests from a given source, the overhead of each protocol is about the same. But an increase in the number of requests starts to tilt the balance in favor of web socket as more efficient in terms of network resources.

- **Overhead of Headers** – The HTTP protocol requires a header for requests and responses. Here is an example of such a header:

```
GET / HTTP/1.1
Host: jsonplaceholder.typicode.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90
Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: __cfduid=dd64f69aef736b2129e50f7fcc1379d291569875784;
_ga=GA1.2.218285718.1569875784
If-Modified-Since: Mon, 05 Aug 2019 03:07:14 GMT
```

This header, which is 547 bytes long, was generated by the Chrome browser. (The header shows that the request was sent to this URL: <http://jsonplaceholder.typicode.com>.)

The request, of course, is only half the story. Or, in the context of our earlier example, the GET /math/add/1/1 portion of the connection). There is still the response to consider, which has a header like the following:

```
HTTP/1.1 304 Not Modified
Date: Fri, 04 Oct 2019 18:45:21 GMT
Connection: keep-alive
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Cache-Control: public, max-age=14400
Last-Modified: Mon, 05 Aug 2019 03:07:14 GMT
Etag: W/"278a-16c5fbec6d0"
```

```
Via: 1.1 vegur
CF-Cache-Status: HIT
Age: 2490
Expires: Fri, 04 Oct 2019 22:45:21 GMT
Server: cloudflare
CF-RAY: 52094ff84874967e-SJC
```

Taken together, the overhead of this HTTP request/response pair is 982 bytes (547 bytes + 435 bytes). This (approximately) 1,000-byte overhead is incurred per actual HTTP request and response. In the case of the 1+1 request and response of "2", the 1,000-byte cost is the total header cost, since the HTTP-defined headers are only sent with the request and the response transmissions. The opening and closing transmissions have no such header associated with them.

In the context of completed REST calls, the overhead of 1,000-bytes per HTTP request/response applies. For example, the total overhead for the following 3 REST calls is about 3,000 bytes:

- <http://jsonplaceholder.typicode.com/users?id=1>
- <http://jsonplaceholder.typicode.com/users?id=3>
- <http://jsonplaceholder.typicode.com/users?id=10>

The overhead scales linearly as more REST calls are made, so that 100 REST calls incur an overhead of 100,000 bytes of network traffic.

The overhead for web socket calls, by contrast, is much lower. **While REST calls incur a 1000-byte overhead per call / response pair, web sockets incur only a four bytes overhead per comparable request / response.** In per-transaction overhead, web sockets are 99.6% more efficient than REST transactions.

4.4.3 HTTP Dependencies

Both C-DEngine and REST support HTTP, as was mentioned in an earlier section. The way each uses HTTP is quite different. The C-DEngine use of HTTP is minimal, to allow C-DEngine to run on other protocols.

REST, on the other hand, is extensively interwoven with HTTP². We say "can be" here because it is possible to create REST servers with a minimal reliance on HTTP. While there are some REST servers that will exhibit some (or all) of the dependencies mentioned here, there are likely other REST servers that have few of the dependencies mentioned here. These variations exist because there are no REST standards that dictate the details of how a server must operate in order to be called a REST server.

² This should, perhaps, be expected since Roy Fielding, who is credited with developing the REST architectural style, had participated extensively in the HTTP/1.1 standardization process.

- **HTTP Methods** – HTTP/1.1 defines seven HTTP methods: `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, and `TRACE`³. Of these methods, the C-DEngine uses just one: `POST`.

By contrast, REST server developers may use up to four different HTTP methods. The familiar “CRUD” model from SQL programming can be mapped to these methods as follows⁴:

SQL “CRUD”	HTTP Method
Create	POST
Read	GET
Update	PUT
Delete	DELETE

- **Mime Types** – It’s important to get the data you requested. It’s also important to get the data in the format that is most useful to you. The `Accept` tag of an HTTP request header allows a client to let a server know what data formats would be acceptable. For example, here is the `Accept` tag from the HTTP request header example shown earlier in this chapter:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
```

On the response end of things, an HTTP response header can use the `Content-Type` tag to notify the client about the data format.

To keep things simple, the C-DEngine uses `application/json` as its standard data format. There are numerous helper functions within C-DEngine to simplify the process of converting between .NET types (which C# prefers) to JSON (which JavaScript prefers).

There are no official standards for REST servers, and so developers of REST clients need to pay attention to the types of data returned by their favorite REST servers and adapt accordingly.

About Variations among REST Servers

The observation that “...developers of REST clients need to adapt to differences between REST servers...” is an important one. The variations between different REST servers causes trouble for developers.

The best that can be hoped for is to have a solid set of examples for a given REST server to figure out what works and what does not work. All REST servers use the command line for command input. But how that is done varies from one REST server

³ See IETF RFC 2068: <https://tools.ietf.org/html/rfc2068>.

⁴ Adapted from <https://www.restapitutorial.com/lessons/httpmethods.html>.

to another. In the JSON Placeholder test server, the following two URLs produce identical results:

-- <http://jsonplaceholder.typicode.com/users/5>
-- <http://jsonplaceholder.typicode.com/users?id=5>

The theme of "variations are everywhere" applies to almost everything related to REST, including how the URL is structured for REST access, how (and whether) URL parameters are used, which HTTP methods (POST / GET / PUT / PATCH) are supported as well as which HTTP status codes are supported and which mime types are supported.

These variations are sometimes mentioned as a security benefit, with logic that suggests that a confusing API is safer (more secure) because it is more inaccessible to those who wish to misuse and exploit a server. That logic is flawed, since it is just another way of saying that "security by obscurity works." No professional computer security expert would ever suggest such a thing.

An obscure REST API confuses developers. A confused developer is a less productive developer.

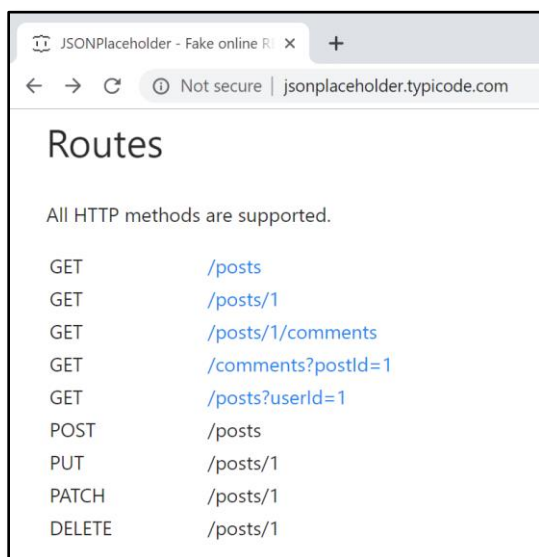


Figure 4.2 The supported HTTP methods and some of the URLs available for one REST server.

4.4.4 Location of Command and Parameters

Another important difference between REST transaction and C-DEngine transactions has to do with the location of commands and parameters. Although there is a lot of variation between different REST servers, in general there are two places that command information can be found:

- In the body of the URL itself.
- In the HTTP method.

The REST test site JSON Placeholder illustrates this nicely. Figure 4.2 shows the supported pairings of HTTP methods (GET, POST, etc.) with supported URLs⁵.

All C-Engine commands and parameters are located in the body of a message. Within the TSM data structure, the TXT field holds the command and parameters. When longer parameter data is passed with a message, there is a PLS field for strings and PLB for binary data.

4.4.5 Support for Queues

It was mentioned earlier in this chapter that an important difference between C-Engine and REST is that C-Engine is inherently asynchronous. Expanding on that theme, C-Engine supports a feature that is not supported in REST: message queues. Figure 4.3 illustrates how each C-Engine node as a queue, known as the "sender queue", of messages being sent out.

The sender queue is created and managed by the C-Engine, with no intervention required by application developers or plugin developers. There are some configuration settings for modifying how the sender queue operates, but otherwise application and plugin developers do not need to know anything more than that this support exists.

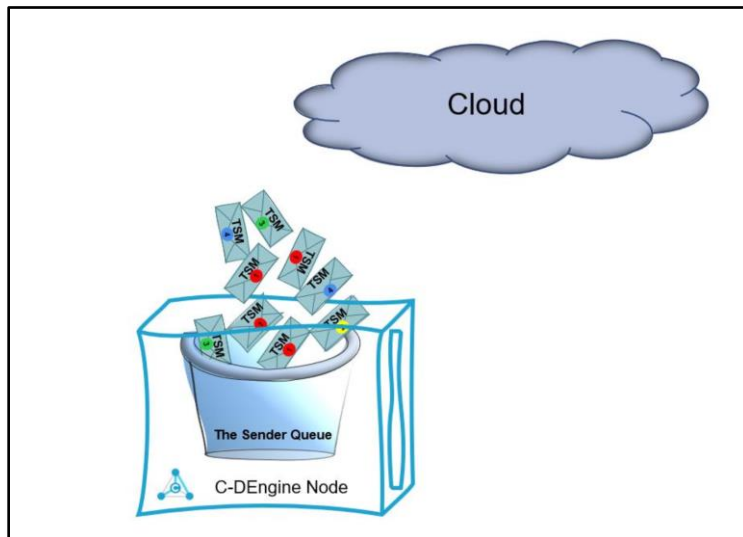


Figure 4.3. Asynchronous Operation is Enabled through smart message queues.

4.4.6 Batching in Message Delivery

In terms of comparing C-Engine to REST, it's worth mentioning that C-Engine batches messages. While each REST call delivers one and only one API requests, a single C-Engine network transmission might deliver ten or twenty (or more!) messages.

This means that even when the C-Engine must use the HTTP protocol to communicate between nodes, it does so in a manner that is more efficient than the approach taken by REST. It was mentioned earlier in this chapter that each HTTP request / response pair incurs an overhead of about 1,000 bytes. When ten REST calls are made, there is an inescapable overhead of 10,000 bytes just for the text headers alone. By comparison, when ten C-Engine messages are sent over HTTP,

⁵ From <http://jsonplaceholder.typicode.com/>.

the ability to batch messages means that the overhead of transmitting these messages might only incur 1,000 bytes for the request, and perhaps another 1,000 bytes to deliver responses. (Web sockets improve upon this story even more, with just 4 bytes to deliver a batched set of requests and perhaps another 4 bytes to deliver a batched set of responses.)

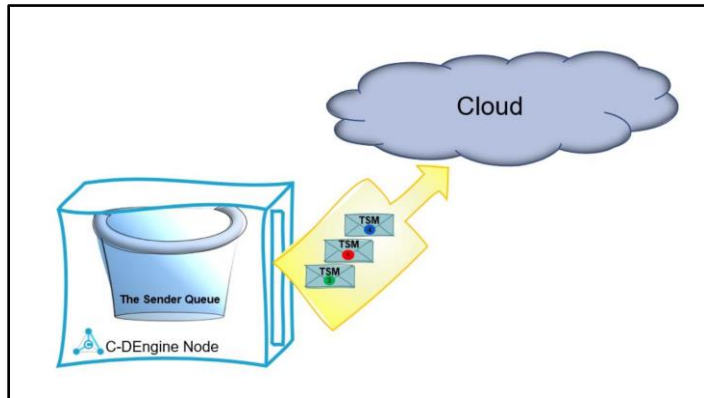


Figure 4.4. Batching of Message Delivery helps optimize overall mesh throughput.

4.4.7 Message Priorities Fine-Tune Message Delivery

Among the many ways that message delivery can be fine-tuned, perhaps the most important involves the ability to set priorities for different types of C-DEngine messages. Figure 4.5 illustrates how the ability to set message priorities enable control over how messages are delivered. Each message has a priority value that varies from 255 (the lowest priority) to 0 (the highest priority). When a message is created, as an instance of a TSM, the default priority that is set is 5.

It's worth mentioning that messages with a priority of 0 (zero) are never sent to the cloud, but rather are only delivered locally. Application developers can use the range from 1 – 4 to define various levels of "high priority" message traffic. Messages with a priority from 6 to 255 are less urgent and are handled as such. To prevent complete starvation of lower priority messages, the C-DEngine has a priority-inversion feature that helps ensure that at least a few lower priority messages get delivered even during a storm of higher priority messages. (A configuration switch allows an application developer to disable priority inversion.)

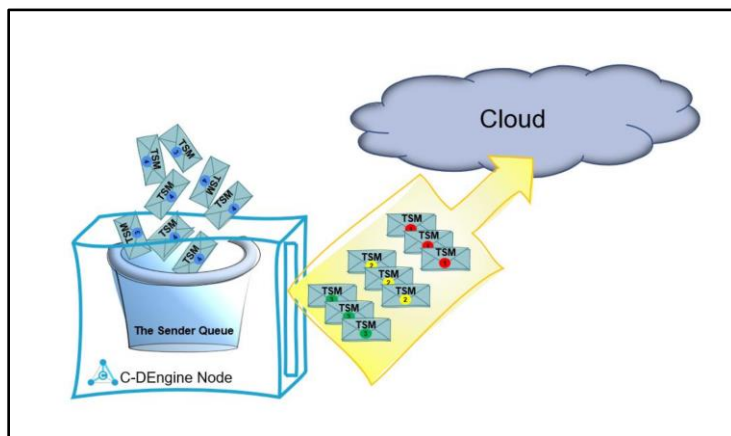


Figure 4.5. Message priorities allows applications to fine-tune message delivery.

4.5 Conclusion

There is no question that there is a lot of momentum behind REST in the industry today. Where, then, does it make sense to use REST versus other solutions? A benefit of REST that is often mentioned is the "loose coupling" between client and server. As described in this chapter, it does that at the cost of a higher overhead and less flexibility in terms of request routing. These factors suggest that REST does best when occasional requests are required. In such cases, the overhead of those few calls are unlikely to overwhelm network resources.

Where solutions like the C-Labs C-DEngine thrive, on the other hand, are those that combine one or more of the following factors:

- Tight coupling between nodes.
- The need for peer-to-peer interactions. In other words, where there is the need for any node to initiate requests.
- Medium to high number of transactions between nodes.
- Medium to high data volume to move between nodes.
- Where efficient usage of scarce (or expensive) network resources is required.

Chapter 5 Shared Web Worker in C-DEngine JavaScript/TypeScript (cdeWorker.js)

Starting with version 4.209.1, the C-DEngine supports the HTML5 Web Worker as part of the browser-based NMI. This feature is available to JavaScript and TypeScript developer, enabling a background thread on browsers that support this feature. As of September 2019, approximately 36% of web browsers (worldwide) have Web Worker support (see usage data, below).

5.1 Installing C-DEngine Web Worker Support

1. Copy the file `workertest.html` into the ClientBin folder of the C-DEngine application host. Note that the ClientBin folder is the root folder for C-DEngine web pages.
2. Copy the file `cdeWorker.js` to the ClientBin/CDE folder of the C-DEngine application host.
3. To access C-DEngine (NMI) features, copy the file `cde.js` to the ClientBin/CDE folder.
4. Start the C-DEngine application host.
5. Load the page `workertest.html` into the browser, using an http URL.

Note: File access (`file://`) to an HTML page does not cause a Web Workers to be active.

The "Shared" part of "Shared Web Worker" comes from the fact that the Web Worker is shared between browser tabs when multiple browser tabs are open. To observe this, load the file `index.html` from two or more tabs. All tabs in the browser share the Web Worker.

5.2 A small Sample explained

The following sample shows a small sample how to use the Shared Web Worker.

The sample shows:

1. Creating a `SharedWorker` (`new SharedWorker("ClientBin/CDE/cdeWorker.js")`)
2. Starting communication with the C-DEngine (`StartCommunication()`)
3. Handling Login, which means waiting for the `CDE_CONN_CHANGED` event. (This is not needed when Auto-Login is enabled and valid credentials are available).
4. Waiting for the Login Success (on receipt of the message `CDE_LOGIN_EVENT`).
5. An example of sending a custom subscription to the `FirstNode` (`Subscribe()`), as well as other message sending example.
6. Handling incoming messages from the mesh (on receipt of the message `CDE_INCOMING_MSG`).

It also shows how to asynchronously access a resource (using either `fetch` or `XMLHttpRequest`) from the mesh (`GetResourceString()`). In the sample, the web worker selects the appropriate access method, using `Fetch` if the browser supports it, otherwise, `XMLHttpRequest` is used.

5.3 Supported Web Browsers

Browsers that support Web Workers make up 36% of browsers in use today (*).

- Chrome on desktop (all versions) - 23%
- Firefox (version 29 and later) - 4%

- Safari (version 5-6) - < 1%
- Opera (version 11.5 and after) - < 1%
- Microsoft Edge (Chromium-based), beta release August 2019 - 0%

5.4 Unsupported Web Browsers

Browsers that do not support Web Workers make up 64% of browsers in use today(*). - Chrome for Android - 35% - Safari 6.1 - 12.1) - 2% - Safari iOS - 11% - Microsoft Internet Explorer - 2% - Microsoft Edge - 2%

(*) Supported and Unsupported Web Worker data from caniuse.com, with worldwide browser market share from StatCounter GlobalStats for September 2019.

5.5 Debugging

The following instructions were specifically created for Google Chrome. 1. Open a new browser tab. 2. Type chrome://inspect in the address bar. 3. On the left side of the page (under the "DevTools" heading) click Shared workers. (If there is an empty screen, then no shared workers are active.) 4. On the left side of the page, click Pages. Click on inspect under the source file cdeWorker.js. 5. You should see an F12 Developer Tool window show up with the worker.js in the Sources Tab.

```
document.addEventListener("DOMContentLoaded", function () { // Wait
until the DOM has loaded

    // Load The SharedWorker. The Worker is inside the CDMyNMIHtml5
Plugin and will be
    // served automatically by its WebServer on this "FirstNode" if the
plugin is present
    // If the plugin is not present, the "cdeWorker.js" must be hosted
on the same Web
    // Server as the scripts that want to use it.
    let worker = new SharedWorker("ClientBin/CDE/cdeWorker.js");

    // First register the event handler for messages coming from the
Worker
    worker.port.onmessage = (ev) => {
        let message = ev.data;
        switch (message[0]) { // All Events have "MyWHSI" (Comm
Status) in message[1]
            case "CDE_INCOMING_MSG": // Incoming mesh messages from
the FistNode
                MyHSI = message[1];
                // message[2] contains "TheProcessMessage" (see below).
The ".Message"
                // field is of Type "TSM" (see below)
                break;
            case "CDE_LOGIN_EVENT": // Success or failure
notification on Login Event.
                MyHSI = message[1];
                // message[2]==true reflects login success /
false=login failed
                // and browser is NOT in the mes scope of the FirstNode
                if (message[2] === true) {
                    // essage[3] contains the "TheUserPreferences"
object (see below)
```

```

// DO NOT SEND MESSAGES TO THE MESH UNTIL YOU HAVE
RECEIVED THIS EVENT!
// Any message you send before will not be
handled/forewarded by the
// FirstNode because the browser will not have a
valid scope until this message

// Send Your subscriptions after login was
successful
worker.port.postMessage(["Subscribe",
"MyCustomTopic"]);
// The following call reverses the topci subscribe
// worker.port.postMessage(["Unsubscribe",
"MyCustomTopic"]);

// The following line sends a message using
individual parameters
// this.port.PostToWorker(["SendQueued", pOwner,
pTopic, pEngineName, pTXT,
// pPLS, pFLG, pQDX, pLVL, tTargetNodeID, pGRO,
pSender]);

// This line sends a TSM to a target Node
(bIncludeLocalNode not supported yet)
// this.port.PostToWorker(["SendToNode",
tTargetNodeID, targetTSM,
// bIncludeLocalNode]);

// Replies a TSM to a sourceTSM.ORG
(bIncludeLocalNode not supported yet)
// this.port.PostToWorker(["SendToOriginator",
sourceTSM, targetTSM,
// bIncludeLocalNode]);

// Sends a TSM with a custom Topic to a custom
NodeID setting the Sender Object
// this.port.PostToWorker(["SendTSM", tTSM, pTopic,
tTargetNodeID, pSender]);
// Sends a TSM do the FirstNode only
// this.port.PostToWorker(["SendToFirstNode",
targetTSM]);
}
break;
case "CDE_SELECT_MESH": // If a user has access to
multiple meshes
MyHSI = message[1];
// message[2] will be a list of TheMeshPicker (see
below). The login
// will not be complete until a "SelectMesh" message is
sent back to the FirstNode

// Picking the first node as the desired mesh.
// Please do proper error handling and selection!!
worker.port.postMessage(["SelectMesh",
message[1][0].cdeMID]);
break;

```

```

        case "CDE_SETSTATUSMSG": // Contains information for the
UI on // current Worker Activities
        MyHSI = message[1];
        // message[1] contains user friendly text;
        // message[2] contains importance (see eMsgLevels in
C#)
        break;
        case "CDE_NEW_LOGENTRY": // Informational messages from
the worker
        MyHSI = message[1];
        // message[1]=location
        // message[2]=error message
        // message[3]=serverity (0-3:
same as eMessageLevels in C#)
        break;
        case "CDE_COMM_STARTED": // Communication was started
successfully // but might not be connected
        MyHSI = message[1];
        break;
        case "CDE_ENGINE_GONE": // An Engine disconnected. This
will be // fired if the communication
was lost // message[2] contains Engine
Name
        break;
        case "CDE_CONN_CHANGED": // Event if connection has
changed
        MyHSI = message[1];
        // message[2]
        // =true=connection established
        // =false=connection lost
        if (message[2]==true && !MyHSI.IsUserLoggedIn)
        {
            // If called multiple times only the frst call will
be used
            worker.port.postMessage(["Login", { QUID:
"a@a.com", QPWD: "aaaaaaaa" }]);
        }
        break;
        case "CDE_UPDATE_HSI": // Event if the Global Settings
have changed by the // node and delivered to the
Worker
        MyHSI = message[1];
        break;
        case "GRS_/ClientBin/Lang/NMILang1041.json": //
GRS_<ResourceName used in
        // postMessage([GetResourceString]...
        // if GSR is used /ClientBin/ is added automatically
        MyHSI = message[1];
        var tMyLanguageFile = JSON.parse(message[2]);
        break;
        default:

```



```

        break;
    }
};

// Start the worker Port
worker.port.start();

// Otherwise get the WHSI from the worker first to find out its
state:
worker.port.postMessage(["GetWHSI",null]); // Return is delivered
above in CDE_UPDATE_HSI // (note that every
call to the WebWorker must // have at least one
parameter)

// Full "TheCommunicationConfig" see below.
// If called multiple times only the first call will be used
worker.port.postMessage(["StartCommunication", { port: 8704, host:
"localhost" }]);

// REST and Async Resource Fetch commands:

// Requests a Resource from the first node via HTTP "Fetch" or
XMLHttpRequest.
// Resource is fetched async and result will be delivered in:
// "onMessage("GRS_/ClientBin/<ResourceName>)" (see above)
// This can also be used for REST calls using "GET" (POST and other
// REST Methods are not yet supported)
worker.port.postMessage(["GetResourceString",
"Lang/NMILang1041.json"]);
});

// Full List of WebWorker postMessages:
worker.port.postMessage(["SetConfig", { port: 8704, host:
"localhost" }]);

// Starts the communication. If Config is null, SetConfig must be
called before
worker.port.postMessage(["StartCommunication", { port: 8704, host:
"localhost" }]);

// Adds an array of Name Value keys pairs to the IndexedDb of the
current session
worker.port.postMessage(["UpdateCustomSettings", [{ Name: "MyKey",
Value: "MyValue" }]]);

// Subscribes to a new PubSub topic(s) separated by ;
worker.port.postMessage(["Subscribe", "topic1;..."]);

// Unsubscribes from PubSub topic(s) separated by ;
worker.port.postMessage(["Unsubscribe", "topic1;..."]);

// Sends a message using an individual parameter
worker.port.PostToWorker(["SendQueued", pOwner, pTopic,
pEngineName, pTXT, pPLS,

```

```

        pFLG, pQDX, pLVL, tTargetNodeID, pGRO,
pSender]);

    // Send a TSM to a target Node (bIncludeLocalNode not supported
yet)
    worker.port.PostToWorker(["SendToNode", tTargetNodeID, targetTSM,
bIncludeLocalNode]);

    // Replies a TSM to a sourceTSM.ORG (bIncludeLocalNode not
supported yet)
    worker.port.PostToWorker(["SendToOriginator", sourceTSM, targetTSM,
bIncludeLocalNode]);

    // Sends a TSM with a custom Topic to a custom NodeID setting the
Sender Object
    worker.port.PostToWorker(["SendTSM", tTSM, pTopic, tTargetNodeID,
pSender]);

    // Sends a TSM do the FirstNode only
    worker.port.PostToWorker(["SendToFirstNode", targetTSM]);

    // Ends the current session and records the reason for ending the
session.
    worker.port.postMessage(["Logout", "reason"]);

    // Logs a user in with either UID/PWD or the Refresh Token given by
the CDE Host
    worker.port.postMessage(["Login", { QUID: "username", QPWD:
"password",
                                QToken:"RefreshToken" }]);

    // If a user has access to multiple meshes, SelectMesh must be
called after an initial
    // Login. The Worker will fires an event
    // (postMessage("CDE_SELECT_MESH", Array<cde.TheMeshPicker>) to the
caller.

    // The caller must call this method with one of the cdeMIDs in the
Array
    worker.port.postMessage(["SelectMesh", "GUID-Token of the Mesh"]);

    // Gets the current content of the WHSI (WebWorker Status Info)
    worker.port.postMessage(["GetWHSI", null]);

    // Fetches a JSON object from the CDE Host - return see below
    worker.port.postMessage(["GetJSON", "ResourceToFetch",
"AddHeaderIfRequired" ]);
    // Fetches a string Resource from the CDE Host. This works even
before login.
    // It is for FirstNode Resources only.
    worker.port.postMessage(["GetResourceString", "ResourceToFetch",
"AddHeaderIfRequired"]);

    // Fetches a string resource from the CDE Mesh (only works AFTER
login)
    worker.port.postMessage(["GetGlobalResource", "ResourceToFetch",
"AddHeaderIfRequired"]);

```

The three methods GetJSON, GetResourceString and GetGlobalResource are all asynchronous and each fires a postMessage back to the caller:

- GJ_resourceName for GetJSON
- GRS_resourceName for GetResourceString
- GGR_resourceName for GetGlobalResource

If the fetch (XMLHttpRequest for older browser that do not support fetch) fails, the worker will postMessage:

- GJ_ERROR_resourceName for GetJSON
- GRS_ERROR_resourceName for GetResourceString
- GGR_ERROR_resourceName for GetGlobalResource

The caller should remember the resourceName and then take the appropriate action. The TypeScript implementation of the WebWorker class handles this automatically.

Here a small sample using TypeScript with the Cde.js

```
var MyCommChannel = new cdeWEB.cdeWebWorkerComm();
    MyCommChannel.RegisterEvent("CDE_LOGIN_EVENT", (sender, success,
succesText) => {
        if (success) {
            MyCommChannel.Subscribe("testTopicSub");
            var tTSM: cde.TSM = new cde.TSM("ContentService");
            tTSM.TXT = "CDE_GET_SERVICEINFO";
            MyCommChannel.SendTSM(tTSM);
        }
    });
    MyCommChannel.RegisterEvent("CDE_INCOMING_MSG", (sender,
pProcessMsg:cde.TheProcessMessage) => {
        cde.MyEventLogger.FireEvent(true,
"CDE_NEW_LOGENTRY","incoming", pProcessMsg.Topic + ":" +
pProcessMsg.Message.TXT,1);
    });
    var tConfig: cde.TheCommConfig = new cde.TheCommConfig(0);
    tConfig.uri = "http://localhost:8704;;;a@a.com;;;aaaaaaaa";
    MyCommChannel.StartCommunication(tConfig);
```

You will need an include statement for the file cde.js in your HTML start page and reference the cde.d.ts TypeScript definition file in your TypeScript source files.

5.6 Required Classes

The following C-DEngine classes are used by the Shared Web Worker:

```
export class TheProcessMessage {    // Envelope of incoming Messages
from the Worker
    Topic: string;                // and sent to "CDE_INCOMING_MSG"
                                // Incoming Topic
```

```

    CurrentUser: any;           // UserID attached to this message.
Not supported in Browser
    Message: cde.TSM;         // The Message content
}

export class TheWHSI {        //The Worker Status Information
    CurrentRSA: string = null; // CurrentRSA Key used for RSA
Encryption
    IsConnected: boolean = false; // True if the Communication was
established
    CallerCount: number = 0;     // Amount of SharedWorker ports

    HasAutoLogin: boolean = false; // True if the credentials have
been set before
                                // the Login Dialog appeared
    (AutoLogin)
    FirstNodeID: string = '';    // NodeID of FirstNode after
connect
    AdminPWWasSet: boolean = false; // True if FirstNode requires
AdminPWToBe Set (browser
                                // is unscoped and cannot
send any telegrams
                                // except "SET_ADMIN_PWD")
    AdminRole: string = '';     // Role of the Current User
    (currently not used)

    UserPref: cde.TheUserPreferences; // User Preferences (see
below) coming in
                                // with the CDE_LOGIN_EVENT

    MyServiceUrl: string = '';   // Http URL of the FirstNode
- can be used for DeepLinks
    MyWSServiceUrl: string = ''; // WebSockets URL of the
FirstNode - can be empty
                                // if websockets are disabled
    IsUserLoggedIn: boolean = false; // True if a user is
currently logged in
}

export class TheCDECredentials { // Credentials for Login
    QUID: string = "";           // Username used to login
    QPWD: string = "";           // Password used to login
    QToken: string=null;         // A token that allows login in
    (coming soon)
}

export class TheTimeouts {      // Timeout class
    HeartBeat: number = 30;      // Send HB every xx seconds
    PickupRate: number = 250;    // Polling Cycle for http requests
- not used with websockets
    InitRate: number = 100;      // Not used in browser
    HeartBeatMissed: number = 4; // Amount of HB missed to consider
connection closed
    PickupRateDelay: number = 1; // Delay of pickup for Http (not
used with WebSockets)
    WsTimeOut: number = 5000;    // If >0 a connection will be
closed if the FirstNode

```

```

// websockets do not respond in the
given time
}

export class TheCommConfig { // Communication Settings class
    TO?: TheTimeouts; // TheTimeouts Class see above
    port?: number; // Port of the node to connect to
    uri?: string; // if port and host are not used,
the uri to connect

    wsuri?: string; // to can be set here
port than the http // in case WS uses a different

    host?: string; // requests, put WS URI Here
just the IP, DNS // Host of the node. Should be

    // Name or "localhost". Comm will
assemble "uri" // from

<http|ws><useTLS?"S":"">://<host>:<port>
    useTLS?: boolean; // Requires TLS (https/wss) to
connect to node

    Creds?: TheCDECredentials; // Login credentials (see above)
    IsWSHBDIsabled?: boolean; // if true, the Browser will not
send WebSocket

    // heartbeats (not recommended)
    cdeTIM: Date; // Time of last write to
IndexedDB

    DisableRSA: boolean = false; // Incoming Only. If true, RSA
will not be used

    RequestPath: string = null; // Incoming ISBPath (NPA) - only
for CDE Integrated calls

    KeepSessionAlive: boolean = false; // If false, MyConfig will be
deleted on logout

    constructor(pWSTimeOut:number) { // Defaults for above settings
        this.port = 80;
        this.host = null;
        this.Creds = null;
        this.useTLS = false;
        this.IsWSHBDIsabled = false;
        this.TO = new TheTimeouts();
        if (pWSTimeOut > 0)
            this.TO.WsTimeOut = pWSTimeOut; // number of ms to wait
until WS // connection is
considered dead

    }
}

export class TheNV {
    public Name: string;
    public Value: string;
}

export class TheDataBase { // Base Class of all Data related
classes

```

```

    cdeMID: string; // Unique ID of the message
    cdeCTIM: Date; // Timestamp of the message
    cdeEXP: number; // Expiration in seconds of
the message
    cdePRI: number; // Priority of the message
    cdeAVA: number; // Availability of the
message
    cdeN: string; // NodeID where the message
was created
}

export class TheUserPreferences extends cde.TheDataBase {
    ShowClassic: boolean; // User Wants to see the old
"Classic" NMI Frame
    ScreenParts: string[]; // [0]=StartScreen [1]=PortalScreen
[2]=hideheader (true/false)
    ThemeName: string; // If the user has a custom Style
Sheet as preference
// it will be here.
    LCID: number; // The users LCID (1031=us, 1033=de
etc)
    ShowToolTipsInTable: boolean; // The users wants to see tooltips in
tables
    SpeakToasts: boolean; // The user wants to use text-to-
speech for all toasts
// (user messages that briefly appear
in the browser window).
// This is an accessibility feature.
    Transforms: string; // The user has custom transforms
enabled
    CurrentUserName: string = ''; // Current User Name
    CurrentLCID: number = 0; // Current LCID of the Connection
(before user has logged in)
    PortalScreen: string = ""; // PortalScreen of the NMI (Must be a
dashboard)
    StartScreen: string = ""; // StartScreen of the NMI (a screen
in the
// PortalScreen Dashboard)
    HideHeader: boolean = false; // True if the FirstNode wants to see
no Header in the browser
}

export class TheMeshPicker extends cde.TheDataBase { // Current mesh.
An array if user has
// access to
multiple meshes
    MeshHash: string; // 4 Digits Mesh Hash
    NodeNames: Array<string>; // Names or IDs of nodes in the mesh
    HomeNode: string; // Name or ID of the node this user
is registered
}

export class TSM { // The System Message class - inter-node message
envelope
    TIM: Date; // TimeStamp of message creation

```

```

    ORG: string;           // Originator(s) of the message. For incoming
messages the
                           // ORG contains all nodes separated by ; the
message was
                           // relayed over. Dont use this to determin the
creator of
                           // the message. use TSM.GetOriginator()
    SID: string;         // Scrambled ScopeID of the message. For
security reasons this
                           // is no longer set starting 4.209
    FID: number;         // Serial Number of the message
    UID: string;         // UserID associated with this message. Not
set for
                           // browser messages

    FLG: number;         // Flags of the message
    QDX: number;         // Message Priority. Default=5, lower number
means
                           // higher priority. QDX=0 is not relayed via
the cloud
    CST: string;         // Per message costing
    OWN: string;         // Owner of the message
    LVL: number;         // Level of the Message (eMessageLevels)
Default:4 "Message"
    ENG: string;         // Engine/Subscription Topic of the message
    TXT: string;         // Command/Text of the message
    PLS: string;         // Payload string of the message
    PLB: any;           // Payload binary of the message
    GRO: string;         // GRO can be set for route optimization

    constructor() {     // Defaults for the Message:
        this.TIM = new Date();
        this.FID = MsgSendCounter++;
        this.QDX = 5;
        this.LVL = 4;
    }
    public static GetOriginator(pTSM: TSM): string { // Returns the
NodeID of the Message creator
        if (!pTSM.ORG) return "";
        var t: string[] = pTSM.ORG.split(';');
        return t[0];
    }
}

```

5.7 Important Notes

If you are accessing the `cdeWorker.js` from an `iFrame` or other Tab on the browser while the NMI is running, the communication might already be established. The small sample above handles this correctly.

If you "host" the `webworkertest.html` and the `cdeworker.js` on a different node than the C-DEngine NMI, you must set the following to flags in the `App.Config` of your node/relay:

Key	Value	Explanation
AllowRemoteISBConnect	true	Allows an ISB connect to the Site
Access-Control-Allow-Origin	*	Allows CORS access from another host:port

5.8 Future enhancements

By request only.

5.9 Dependencies

None.

Appendix A: The Fire Gate Plugin

A.1 About the Fire Gate Plugin

The Fire Gate plugin is a REST relay. It enables REST connections to one or more REST servers, and logs each REST request and the corresponding REST response.

A.2. Configuring Fire Gate Plugin

To configure the Fire Gate plugin, log on to Factory Relay (or a C-DEngine-powered application of your choice), and navigate to the Fire Gate Plugin's dashboard (see Figure A.1).



Figure A.1. The dashboard of the Fire-Gate plugin.

To create a new Fire-Gate, click on the Fire-Gates plugin. You see a table with all currently available fire-gates (see Figure A.2).

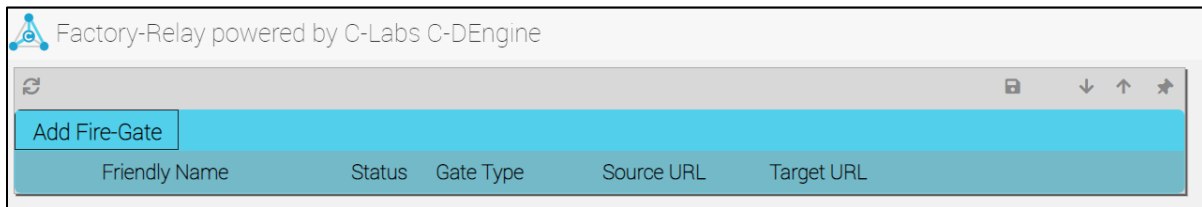


Figure A.2. Table of currently available fire-gates.

Click the **Add Fire-Gate** button. An empty row appears (Figure A.3).

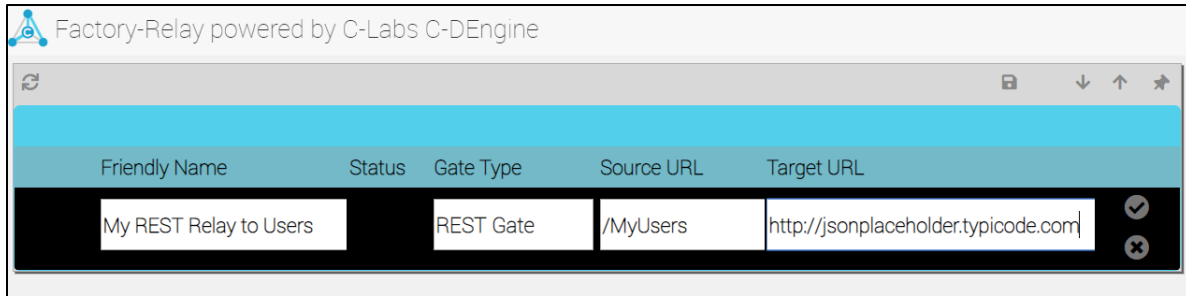


Figure A.3. An empty row for a new fire gate.

Fill in with your desired values, then click the checkmark icon to save.

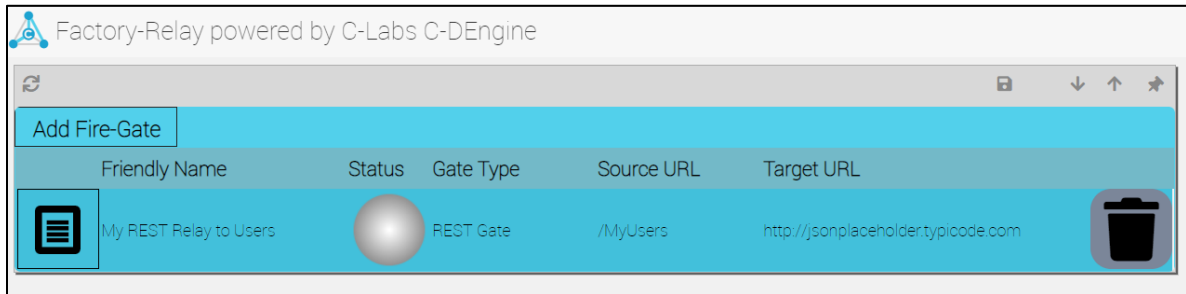


Figure A.4. A sample fire-gate.

An example appears in Figure A.4. In the example shown here, we use a public test REST site, <http://jsonplaceholder.typicode.com/>. Here are some of the URLs that are defined in that site:

- <http://jsonplaceholder.typicode.com/db>
- <http://jsonplaceholder.typicode.com/posts>
- <http://jsonplaceholder.typicode.com/users>
- <http://jsonplaceholder.typicode.com/comments>

Details of one way to fill in the fields appear in Figure A.5.

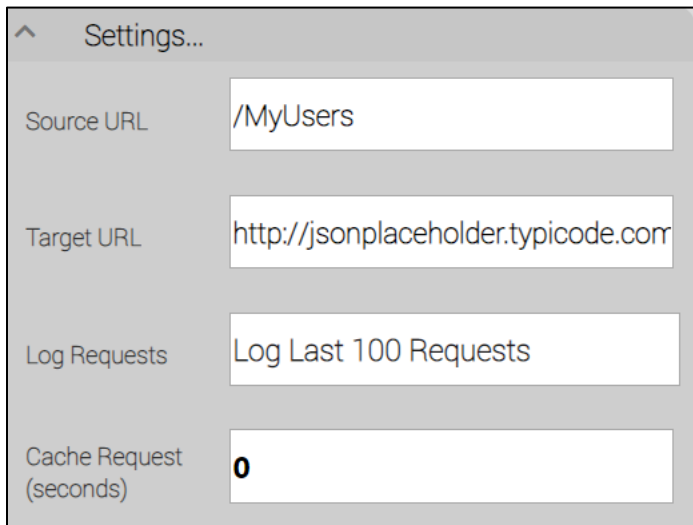


Figure A.5. Details of settings for example fire get setting.

Figure A.6 shows the results when a test REST client is used to call into the test REST server.

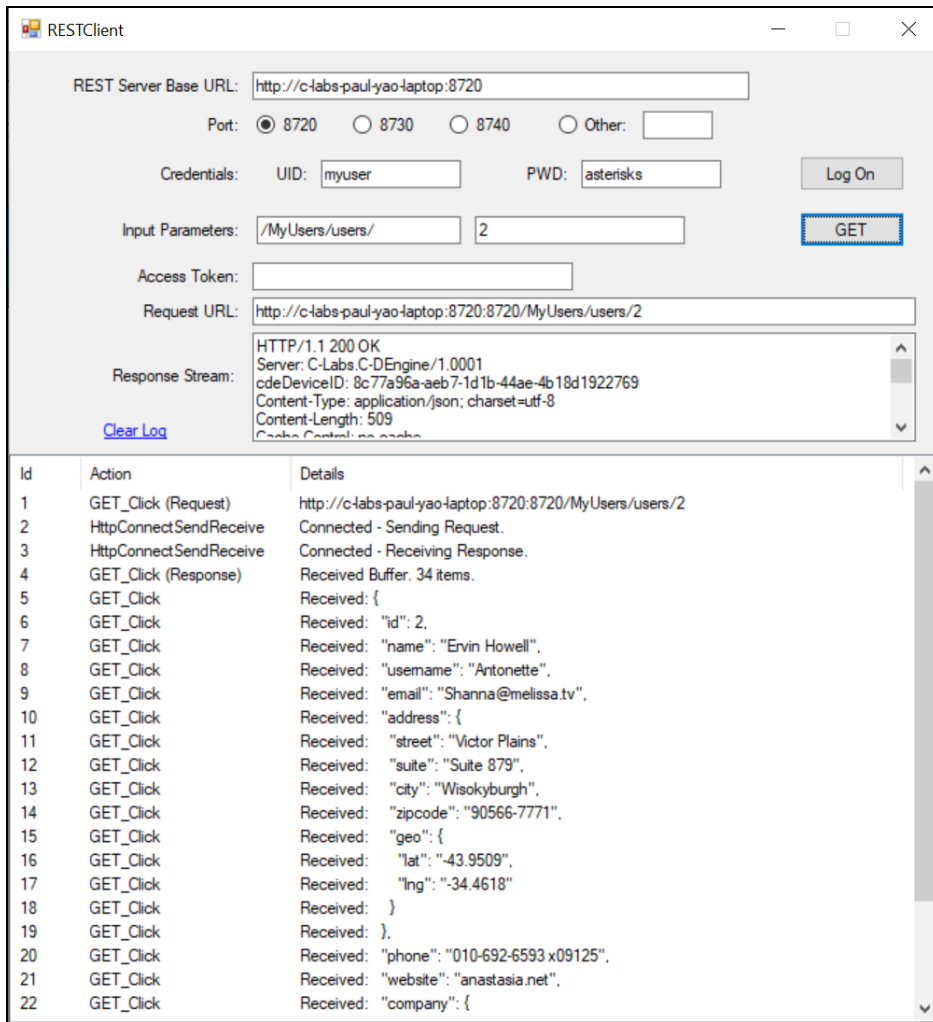


Figure A.6. REST Test Client calling through Fire Gate to REST Test Server.