JSON

[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you’re ready to add your content, just click here and start typing.]

[Document subtitle]

Contents

[Problem statement 3](#_Toc419455319)

[Problem (and why)? 3](#_Toc419455320)

[Plan 3](#_Toc419455321)

[Research 4](#_Toc419455322)

[Methods 4](#_Toc419455323)

[REST API norms 4](#_Toc419455324)

[Most used variable types in programming 4](#_Toc419455325)

[Testing 5](#_Toc419455326)

[Compression 6](#_Toc419455327)

[Lossy compression 6](#_Toc419455328)

[Lossless compression 6](#_Toc419455329)

[Lossy vs lossless 7](#_Toc419455330)

[Compression algorithms 7](#_Toc419455331)

[Approaches 8](#_Toc419455332)

[Testing 9](#_Toc419455333)

[Encoding 12](#_Toc419455334)

[Encryption 13](#_Toc419455335)

[Symmetric 13](#_Toc419455336)

[Asymmetric 13](#_Toc419455337)

[Https 13](#_Toc419455338)

[Data consumption 14](#_Toc419455339)

[Encoding 15](#_Toc419455340)

[Types 15](#_Toc419455341)

[Data consumption 15](#_Toc419455342)

[Order (Encryption/compression/encoding) 16](#_Toc419455343)

[Other solutions 17](#_Toc419455344)

[JSON stream 18](#_Toc419455345)

[Research Findings 19](#_Toc419455346)

[Product 20](#_Toc419455347)

[Overview 20](#_Toc419455348)

[JSON format 20](#_Toc419455349)

[Compression 21](#_Toc419455350)

[Encryption 21](#_Toc419455351)

[Configuration 21](#_Toc419455352)

[Backend library (C#) 21](#_Toc419455353)

[Installation 21](#_Toc419455354)

[Configuration 21](#_Toc419455355)

[Usage 21](#_Toc419455356)

[How it works 22](#_Toc419455357)

[Internal dependencies 22](#_Toc419455358)

[External dependencies 22](#_Toc419455359)

[Frontend library (Swift) 23](#_Toc419455360)

[Installation 23](#_Toc419455361)

[Usage 23](#_Toc419455362)

[How it works 23](#_Toc419455363)

[Internal dependencies 23](#_Toc419455364)

[External dependencies 23](#_Toc419455365)

[Testing 23](#_Toc419455366)

[Environment 23](#_Toc419455367)

[MySql Database 23](#_Toc419455368)

[Conclusion 24](#_Toc419455369)

[References 25](#_Toc419455370)

# Problem statement

## Problem (and why)?

JSON is a widely used data transport mechanism, but there are issues with security and bandwidth consumption. JSON is usually carried over HTTP, which is easily intercepted by standard sniffing software.

Mobile devices don't always have high data allowances and often have slow connections. The JSON format is not as efficient as it could be, given that it sends the same data keys multiple times. As a result of this some mobile applications can seem slow and unresponsive.

It is also a cumbersome process to research and develop your own optimizations and security while developing a product as it requires extra time and effort. As a result of this, JSON requests are often left in the current format and open to misuse.

Problems:

* .NET JsonResult and Web Api result formats dates badly
* Implementing security requires lots of work
* Serializing and desterilizing between Swift objects to JSON results in lots of redundant code

## Plan

I plan to look into the best possible technologies, libraries, and methods to optimize the JSON format while considering both security and data consumption. I then plan to develop a C# and Swift framework to communicate with each other via this format.

I aim to make it as easy as possible for a future developer to install this library on both the server and the frontend, and make them as intuitive as possible. This may result in some compromises having to be made, which will be done so while prioritizing ease of use for a future developer.

Write problems:

E.g. parsing json in swift

Why im doing this project:

# Research

To complete this project I need to do some research to find out possible solutions to the outlined problems.

I need to explore:

* Different methods of securing and reducing data during internet transport
* The norms for REST Web Api’s
* Best library installation practices
* Whether or not any other similar solutions exist that tackle similar problems

## Methods

For the research I plan to use a combination of online research and manual testing. The testing will include installing and setting up testing environment in a web page, as well as testing different libraries and analysing their performance.

## REST API[[1]](#footnote-1)

I aim to create a class to handle communication between an object and a Web API on the frontend. For this to function properly it must follow the accepted REST API standards. I am going to find out what standards exist and how to best implement them. I also need to make sure that the server side implementation will not interfere with a normal REST API implementation.

API stands for “Application Programming Interface”. It is a web resource, providing the ability to read and write data to an online database. It is controlled by the server, and is setup to only allow certain tasks to be performed from an external source.

REST is short for Representational State Transfer. It sends a data representation for part of the server state as a response to a request made via the HTTP protocol. REST itself is an architecture with some set rules to how it should be setup.

Together, a REST API is a controlled architecture for creating a controlled web interface for allowing external sources to gain access to a data source. Mobile apps often draw from multiple API’s and can display the results inside the native UI.

An HTTP request contains:

* **Resource:** The web address (URL)
* **Request Verbs:** Themethod of sending data to the same URL which can be interpreted by the server (GET/POST/PUT/DELETE/PATCH)
* **Request headers:** Additional information needed, could be authentication.
* **Request body:** The container for request data.
* **Response body:** The container for response data.
* **Response status codes:** All requests have a status code which represents whether or not the request was successful or not. An incorrect URL will always provide status code 404 (Not Found) for example, but you can also set the status code from the backend.

### URL Format

A typical URL has three main components. The base, resource identifier and ID.

The server domain is usually the base for all requests, while including the path to the Web API resource (e.g. “api” folder/route) e.g. “<http://www.mysite.com/api>”. The resource identifier is used to identify which resource is going to be read or written to. Typically this would be a table in a database e.g. **“Users”.** The ID usually points to the primary key for a database which is a unique identifier for a database record. This ensures the API is will not update the wrong record.

These components are then formatted: *domain* / *resource Identifier* / *ID*

e.g. “<http://www.mysite.com/api/Users/5>”

### Request Verbs

The HTTP Methods (Verbs) are to be used like this:

|  |  |
| --- | --- |
| **Method** | **Action** |
| GET | Get |
| POST | Insert |
| PUT | Update |
| DELETE | Delete |
| PATCH | Update from included fields only |

Examples of how to format REST URL’s:

|  |  |  |
| --- | --- | --- |
| **Method** | **Resource** | **Response** |
| GET | /Users | A list of users |
| GET | /Users/4829 | User with id of 4829 |
| POST | /Users | Confirmation of creation |
| PUT | /Users/3 | Confirmation of update |
| DELETE | /Users/194 | Confirmation of deletion |

## Most used variable types in programming

(for object mapper)

## Testing

#### Tools

##### Wireshark

##### Google Chrome

## Compression[[2]](#footnote-2)

I am going to research which types of compression exist, which is the most effective and how much data it will save on different JSON requests. I will also look for existing libraries for both C# and Swift to achieve this.

Compression is a method used to reduce the size of files, to allow for faster transfer over the internet as well as taking up less storage space.

I am going to use a combination of desktop research and manual testing using some network tools to measure the data size of a request using different compression techniques.

### Lossy compression

Lossy compression removes information from the original data to reduce its size. A good example of this is an MP3 file. It gradually starts removing information for sounds in the high frequencies, starting with those outside the frequency range for human hearing,

It can however be compressed further, and the more you compress it, the “muddier” it starts to sound as the compression starts to remove some of the higher frequencies which are audible to humans.

As data is removed during the compression process, the original cannot be decompressed again.

### Lossless compression

Lossless compression reduces the size of the data without while allowing it to be restored to its full original state. Lossless compression removes redundancy by looking for patterns, using various mathematical tricks before applying a “code” to represent the original information. This representation takes up less space than the original data.

E.g.

“mmmmmmmmmmYYYY” could be represented as: “m10Y4”

### Lossy vs lossless

Lossy compression is not appropriate for JSON optimization as it implies the loss of information which will make the JSON requests incomplete. Lossless compression however could be used.

### Compression algorithms

There are two main methods of compression which are accepted by HTTP standards.

#### Deflate

The deflate algorithm is a combination of LZ77 and Huffman coding. LZ77 is an algorithm which was published in the 1970’s and was a starting point for many other compression algorithms. Describe Huffman coding

#### GZip[[3]](#footnote-3)

GZip is a file format based on the deflate algorithm but contains extra headers such as timestamp, original file name and a CRC-32[[4]](#footnote-4) checksum. The checksum is used to verify the accuracy of the decompressed output. It also allows the compressed result to be saved to a file and can be opened in common compression software such as WinZip.

As a result of this extra data, the output is larger than a standard deflate compression.

### Approaches

#### Content-encoding

HTTP has a standard which allows you to define your compression preference via an HTTP header: “Content-Encoding”. It accepts one of two compression types: “gzip” and “deflate”.

It allows you to read the headers from an MVC action filter, and then compress the entire request. The request can then be decompressed on the other client (a web browser does this automatically), and the data can then be used normally.

This screenshot shows the same request made with different headers:

1. No extra headers
2. “Content-Encoding”: “gzip”
3. “Content-Encoding”: “deflate”
4. No extra headers – response compressed using deflate manually



I found that this method of compression was extremely efficient, reducing a 38.4kb request to 2.9kb. It even produced smaller requests than manually compressing using the same compression technique. (request 4). I believe that this is due to compression of the entire request, including headers etc.

It was also a very simple process to implement, requiring only two lines of code. The first of which is simply a header declaration for the benefit of letting the client know which compression algorithm was used. A modern web browser is able to recognise the header “content-encoding” and de-compress it automatically without having to code anything.

The compression itself is achieved by adding a compression filter to the response:



Although providing some implementation advantages, there is a problem with this using content encoding due a lack of control for the ordering of encryption and compression to maximize efficiency. This compression is the final wrapper for the request, and cannot be tampered with afterwards.

As my findings showed that encryption should be done after compression, this method is not appropriate for this project.

#### Manual compression/decompression

Compressing manually basically implies the compression of data inside the HTTP Body of the request. This means that the data is compressed, but the request headers aren’t. The compression is done in the backend before being sent across the network. The compression has to be compressed and decompressed manually using C# methods.

### Testing

For testing, I decided to run some requests from a web browser, use Google Chrome’s networking tools to read the size of each request, and then look for trends with the resulting data.

I tested a number of different JSON strings of varying lengths by compressing them with both compression algorithms and comparing them to the original string data length (in bytes). The results of this are shown in the graphs below:

The best compression method I found was LZ77. This outperformed the GZip algorithm every single time. Compression seems to be more effective, for the more data you have that needs to compression on what appears to be a direct correlation.

There is also a minimum string length needed for compression to be useful at all, because below this threshold it actually increases the data size. I found that any response below 103 bytes will be better off not compressed.

Overall I found that compression was extremely effective on any decent sized JSON response.

## Encoding

After testing, I found that after a string has been compressed, it needs to then be encoded afterwards. This is because the compressed string doesn’t transport properly through POST forms, and the decompressor throws an error.

**e.g.** "-??\t?0\fW???QZ?h???F??\r\"???\"{????n???8|b0?0??z???G???oJ\b??~?s??^”

## Encryption

There are different types of encryption with varying data overhead as well as security. I am going to find out which methods are available, and appropriate for a JSON response.

### Symmetric

Symmetric encryption requires a secret key to be exchanged between the sender and receiver. This key is the same for both parties, and is used to both encrypt and decrypt the message.

This requires the secret key to be hardcoded on both server and client. This may have security implications as it only takes one phone to be hacked, gaining access to the source code to potentially compromise every user of the app’s security.

### Asymmetric

Asymmetric encryption works slightly differently. It uses a combination of private and public keys.

The sender can encrypt a message using the receiver’s public key. The receiver can then decrypt this using his own private key. It is also possible for the sender to sign the message using their private key. The sender is then able to verify the sender.

The receiver's public key cannot be used to decrypt the message so does not need to be kept secret and can be sent un-securely to the sender.

The benefit of Asymmetric encryption as it would be possible to verify the sender on both the server and frontend. This means that it will be much harder for a hacker to fake requests.

salt/iv ?

### Https

Https is a protocol which enhances the security of web communication. It encrypts and decrypts HTTP requests between the web server and client.

The security works by initializing a connection with a handshake using very secure symmetric encryption. Once each party has verified the integrity of each other, a less secure symmetric key is negotiated for subsequent requests. This allows for faster performance for all requests in that session, after the initial handshake.

HTTPS requires a digital certificate to be installed on the web server. This certificate has be bought form a certificate authority, and renewed annually. It also requires another port (443) to be opened to allow the traffic through. It adds cost to the running costs as well as requiring time to setup. This is often a big dis-incentive, especially for some smaller businesses.

If security is your main priority, you can’t really get any better than HTTPS. It is a very effective security method which accounts for most common hack types. It encrypts the whole request, masking the target URL aswell as all its headers and content.

It is often neglected however, due to extra cost and setup processes, often resulting in leaving all communication including passwords, credit card information etc. open for anyone to hack easily.

### Data consumption

Encrypting data increases the size of the data.

#### Reduce encryption overhead

It might be a good idea to not encrypt the entire response, but maybe only some of it. If the keys to encrypt were specified as part of the request, only these fields would be encrypted, reducing the encryption overhead.

Findings

While HTTPS is definitely the most secure option, it has its drawbacks and …

For the purpose of this project I am going to use Symmetric encryption. Although there are more secure encryption methods, it is simpler to implement and requires less data traffic. Asymmetric encryption could be used to simulate an HTTPS handshake to agree on a symmetric encryption key, but it didn’t make sense to try to emulate HTTPS, as it will never be as well tested as HTTPS itself. If more security is needed, this library may not be suitable.

Some of the features from HTTPS however can be emulated. The ability to mask the target URL co

## Encoding

### Types

### Data consumption

## Order (Encryption/compression/encoding)

The final output will have been affected by compression, encryption and encoding. I am going to try to find out whether or not the order of when these methods are done will affect the data size of the output.

## Other solutions

## JSON stream

## Data formats

Json/xml

## Research Findings

* Best compression/encryption/encoding methods from research
* plan for product

# Product

The product consists of a server side and a client side library. Its purpose is two provide a framework for communicating to and from a Web API while ensuring an optimized transport of data which takes handles both security and compression.

The main aim is for the product is to be easy to install, and as easy as possible for a future developer to use. This means considering the installation and updating of the frameworks as well as the number of lines of code and DRY principles.

It also tries to be as unobtrusive as possible. This means that you can apply this framework to an existing .NET Web API and Swift project without having to make any big changes.

## Overview

### JSON format

The JSON format for a response is normal valid JSON which can would pass any JSON validator.

A sample request looks like this (while in transport):

{    
   **"data"**:"U2FsdGVkX1/Q0adIWkeHud7buJ797pcaO4V1Accxe0/Ox+e4votw8"  
}

The string inside the data parameter is its own JSON string (also valid JSON) which has been compressed and encrypted using the settings defined by the user (explained later on). This can be processed on the front end and the normal JSON extracted.

The processed JSON is completely valid JSON although has been “percentage escaped” with a backslash to allow it to sit inside a string:

{ \”Name\” : \”Alex\” }

In a request to the server, the data parameter is parsed to the server as a normal HTTP body parameter. This allows these values to be easily retrieved again by the server.

### Compression

### Encryption

## Backend library (C#)

### Features

The backend library includes the following features:

* Automatic compression and encryption of data via action filters.
* Mask URL routes with encryption
* The ability to continue using built in MVC and Web API 2 methods as normal

### Installation

The library is installed by dragging and dropping the folder into your project. You must then ensure you have a reference to the namespace in any files which you wish to access features from the library.

using CompresJSON;

### Usage

The processing is applied to a normal MVC or Web API 2 Controller via action filters. The action filters can be applied to a single “JsonResult”, “ActionResult” or “IHttpActionResult” providing it returns a “JsonResult” instance.



It can also be applied to the controller itself. This method reduces lines of code, but all actions from the controller are required to return a JsonRequest object to avoid this interfering with any other existing action results.

The developer has control over whether they want to process an incoming request, outgoing request or both. For the purpose of communicating between client and server, it is necessary to process both the incoming and outgoing data.



If the method does not receive any data, you can run it with just one action filter:



The action filters can be applied to a “JsonResult”, “ActionResult” or

By using action filters, the library doesn’t require any large changes to the existing code, and is very easy to implement:

### How it works

### Internal dependencies

### External dependencies

* Jint
* CrytoJS (CS library)

## Frontend library (Swift)

### Features

The frontend library includes the following features:

* Automatic compression and encryption of data
* Automatic generation of standard Web API URL’s
* Automatic JSON to object mapping
* Automatic object to JSON mapping
* Methods to execute all CRUD functions.

### Installation

This library can be installed via CocoaPods[[5]](#footnote-5). CocoaPods is a dependency manager for Swift and Objective-C libraries. The “pod” can be installed by adding this line to your Podfile:

pod "ABToolKit", :git => 'https://github.com/a1exb1/ABToolKit-pod.git'

The pod contains the code files for all internal dependencies, as well as references to external dependencies. After running “pod install” in the command line, the library is downloaded alongside all its dependencies and integrated into the project as a framework.

To use a framework you must import it to any code files where you wish to use it:



### Usage

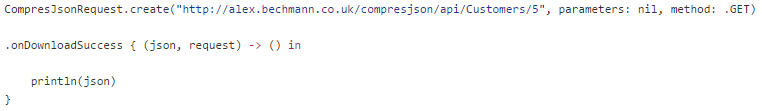
#### Making a request

A request can be made using the create method on the **CompresJsonRequest** class.

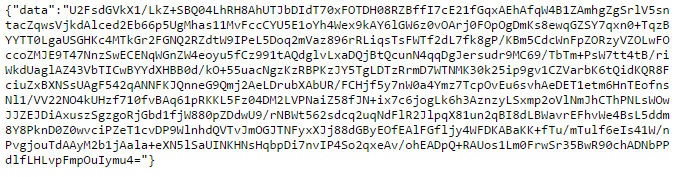


This will execute but will not run any code on completion.

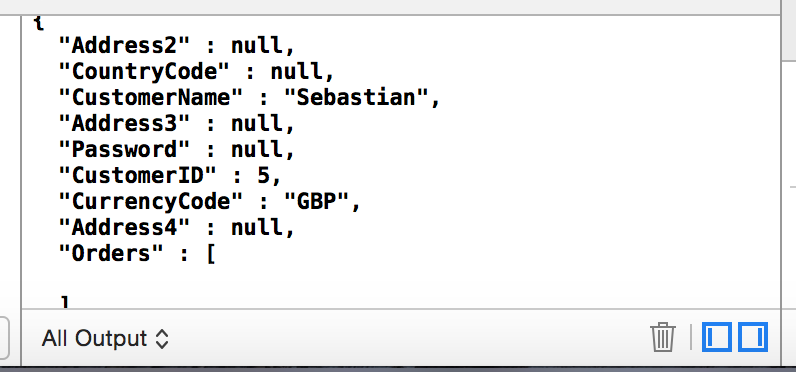
You can add code in the callback by adding a handler:



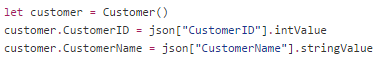
This will convert the processed JSON from the response:



Into normal JSON and print it to the console:



The values could then be unpacked using SwiftyJSON syntax:

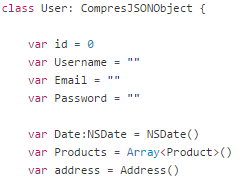


You can chain the handlers together:

#### JSON Mapping

Mapping JSON parameters to objects manually (as seen above) involves a lot of redundant code and can be time consuming if you have many complex models. Your models can be setup to handle this automatically.

Your model must inherit from “CompresJSONObject” instead of “NSObject”. You can use a pre-existing object class, and just replace the inheritance class. The class properties are setup in the usual way:



You can now use a class method on your object to instantiate a new instance and map the properties directly from a JSON object (providing your class properties match the JSON keys):



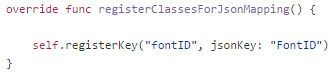
In swift a “class” method is the same as a “static” method in C#, meaning it can be executed without requiring an instance of that class.

If the properties do not match, you can use “registerKey” function to map a json key to a property key. The delegate method “registerClassesForJsonMapping()” is called just before it does the mapping. You can set your mapping preferences here and add any logic if you require it.

If you had the property:



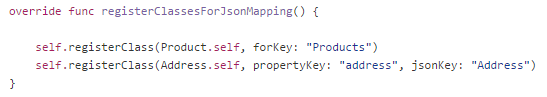
You can map the JSON paramter “FontID” to this property. Once this is defined it will do everything for you automatically:



You might also have embedded types within your JSON response. This could be an address object or a list of products. You might also have a date, formatted as a string.

|  |  |
| --- | --- |
| If your model looked like this: | And the JSON you were expecting looked like this: |
|  |  |

You can register a class to a parameter. It will detect automatically if the response is an array or single object and automatically map all the properties.



Automating the REST URL’s

The REST URL’s are managed within the “WebApiManager” class. A set of RESTful routing URLs can be setup using:

### How it works

Jsonrequst

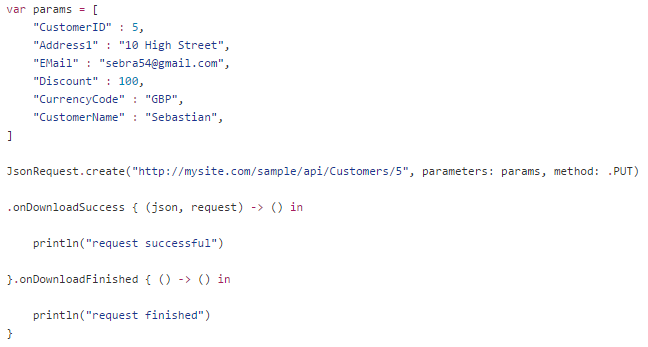
compresJSon request

javascriptanalyzer

jsonobject

### Internal dependencies

* JavaScriptAnalyzer
* JSONObject
* JsonRequest



* WebApiManager

### External dependencies

* CryptoJS

## Configuration

Both libraries feature ways to configure the settings. These settings include the symmetric encryption key, as well as Boolean variables for whether or not to encrypt, compress, or do both. This settings step is important because the server settings have to match the client side settings in order to correctly process the data it sends and receives. Both libraries have settings and are implemented similarly on each.

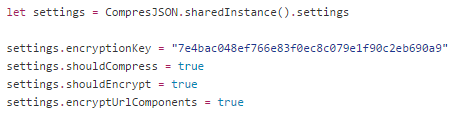
### Backend configuration

The configuration settings are static variables which mean you cannot change these from inside code, but by editing the source of the page via an IDE.



### Frontend configuration

In the frontend, the settings instance is a global variable which is configured in the “AppDelegate” file. This file is the first thing to run whenever the app is launched.



As the frontend isn’t stateless like the server, it can save global variables in memory and allow access to them from anywhere in the code.

The “encryptUrlComponents” variable toggles the encrypted routing option. This setting does not need to be on the backend, although if it is set to true, the routes must have been setup in the global.asax file.

## Testing

### Environment

To test the traffic and effectively measure whether data consumption with and without the use of this library I have setup an environment which is using real data and hosted on a web server.

both



Compres only



### MySql Database

The database is a copy of a real world database from another project. The database contains user and order data.

Diagram?

# Conclusion

Perspective:

What u could have done differently

Future plans

Analysis

* Is it worth it? Does it work?
* Encryption key stored on server + client device

# References

Pot, J., 2012. *How Does File Compression Work?.* [Online]   
Available at: http://www.makeuseof.com/tag/how-does-file-compression-work/  
[Accessed 26 April 2015].

Wikipedia (Cyclic redundancy check), n.d. *Wikipedia (Cyclic redundancy check).* [Online]   
Available at: http://en.wikipedia.org/wiki/Cyclic\_redundancy\_check  
[Accessed 04 May 2015].

Wikipedia (GZip), n.d. *Wikipedia (GZip).* [Online]   
Available at: http://en.wikipedia.org/wiki/Gzip  
[Accessed 04 May 2015].

<http://en.wikipedia.org/wiki/LZ77_and_LZ78>

<http://en.wikipedia.org/wiki/Huffman_coding>

<http://weblog.west-wind.com/posts/2009/Nov/13/Capturing-and-Transforming-ASPNET-Output-with-ResponseFilter>

<http://codebetter.com/johnvpetersen/2012/04/02/making-your-asp-net-web-apis-secure/>

<https://onallthingsweb.wordpress.com/tag/actionfilterattribute/>

<http://security.stackexchange.com/questions/82314/is-encrypted-body-http-secure>

<http://security.stackexchange.com/questions/35210/encrypting-using-aes-256-do-i-need-iv>

<http://stackoverflow.com/questions/3604582/which-attacks-are-possible-concerning-my-security-layer-concept/3604619#3604619>

<https://msdn.microsoft.com/en-us/magazine/dn781361.aspx>

<http://xss.tkch.dk/aes.php>

deflate, gzip, x-gzip, identity

compress – UNIX "compress" program method (historic; deprecated in most applications and replaced by gzip or deflate)

deflate – compression based on the deflate algorithm (described in RFC 1951), wrapped inside the zlib data format (RFC 1950);

exi – W3C Efficient XML Interchange

gzip – GNU zip format (described in RFC 1952). This method is the most broadly supported as of March 2011.[5]

identity – No transformation is used. This is the default value for content coding.

pack200-gzip – Network Transfer Format for Java Archives[6]

encrypt analysis:

<http://airccse.org/journal/nsa/6414nsa04.pdf>

<https://www.simple-talk.com/dotnet/asp.net/attribute-routing-in-web-api-v2/>

<http://automagical.rationalmind.net/2009/02/12/aes-interoperability-between-net-and-iphone/>

!!

<http://automagical.rationalmind.net/2009/02/12/aes-interoperability-between-net-and-iphone/>

compression dependency:

<https://github.com/dankogai/js-base64>

todo product:

clean c# lib

c# handle dates

decrypt/encrypt – add base64

shouldDecrypt (capital s)

web api verbs

<http://micheltriana.com/2013/09/30/http-verbs-in-a-rest-web-api/>

1. (Microsoft, 2015) [↑](#footnote-ref-1)
2. (Pot, 2012) [↑](#footnote-ref-2)
3. (Wikipedia (GZip), n.d.) [↑](#footnote-ref-3)
4. (Wikipedia (Cyclic redundancy check), n.d.) [↑](#footnote-ref-4)
5. <https://cocoapods.org/> [↑](#footnote-ref-5)