CompresJSON

http://alex.bechmann.co.uk/compresjson

Contents

Introduction

Problem statement

Are there viable optimizations to be made on the transportation of JSON data in terms of reducing the data size, and can it be combined with adding security to provide an easy, low cost way to improve the way data is sent between a web server and client?

JSON is a widely used data transport mechanism throughout the internet, 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 doesn't appear to be as efficient as it could be, and as a result some mobile applications can seem slow and unresponsive.

It is also a cumbersome process to research and develop your own optimizations and security at the same time as developing your own 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.

Initial thoughts

I had initial ideas about tackling the problem statement. I split issues into two main areas:

Reduce data size

- Sending keys once and applying to subsequent requests
- Specifying your own delimiter characters (Limiting use of URL encoding)
- Caching of static data
- Compression algorithms

Security

- HTTPS
- Encryption without HTTPS
- Not sending keys makes data less meaningful, if intercepted.
- Masking URL's

Plan / Scope

I plan to look into the best possible technologies, libraries, and methods to optimize the transportation of object data while considering both security and data consumption. I will explore the possibilities of securing HTTP requests, to see if compression can be combined with security.

I aim to develop an open source C# and Swift framework to enable communication between the two technologies via an optimized data format. Time constraints will not allow for every feature to be fully implemented so I would like to deploy this project publically on GitHub, opening it up to contributions from other developers online. This means anyone can download the full source code, make amendments and submit a pull request, requesting their code be merged into the project.

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 the ease of use and functionality.

I will base the product on a set of problem assumptions, which are deduced from personal experience and will require validation.

JSON

JSON stands for JavaScript Object Notation. It is a "human-readable" format to transport objects which consist of key/value attribute pairs. It often referred to as a "fat free alternative to XML". XML (Extensible Mark-up Language) is a preceding data format which looks very similar to HTML, consisting of various tags. It is a very widely used format and is very easy to extract values from it in most programming environments.

In this project I have decided to focus on JSON as it is more modern than XML and can send the same data using less data. It does this by requiring less characters to define each key/value.

Assumptions

The problems to be tackled in this project are based on a set of assumptions. An assumption is an idea based on personal experience that has not undergone further examination.

Cost of security

JSON response and HTTP requests made between a server and a frontend send all their data in plain text, and can be easily intercepted by an unwanted third party.

It might send some parameters to the server for user verification:

Username: *Alex*

Password: secretpassword

Implementing security often requires a lot of setup time and costs.

If this was intercepted, a hacker could very easily get your credentials. Security, while very important, often requires a lot of setup time and costs and sometimes results in it being forgotten or deprioritized due to the required investment.

Redundant code in common tasks

Serializing and desterilizing between Swift objects to JSON results in lots of redundant code. iOS programming is very object orientated. When you make a request for external data via the internet, you will most likely want to extract the values, and map them to a Swift class instance. Unfortunately there is quite a tedious set of steps to implement this.

When you receive an HTTP request, anything inside the HTTP body is read into swift as "NSData" which is an integrated class in Swift containing the raw data.

This data has to then be converted into a dictionary, checked for errors, and then each key needs to be manually unpacked and bound to an instance of the target class. Date formatting requires date formatters to convert strings into dates. This example shows mapping two properties from JSON to an instance of "User", the end result is quite messy:

```swift
let data: NSData = …some data loaded…
let jsonError: NSError?
let decodedJson = NSJSONSerialization.JSONObjectWithData(data, options: nil, error: &jsonError!)
    as Dictionary<String, AnyObject?>

if !jsonError {

  var user = User()

  if let name = decodedJson["Name"] as? String {

    user.Name = name
  }
  if let dateOfBirth = decodedJson["DateOfBirth"] as? String {

    let dateFormatter = NSDateFormatter()
    dateFormatter.dateFormat = "dd/MM/yyyy"
    user.DateOfBirth = dateFormatter.dateFromString(dateString)!
  }
}
```

This process has to be done on every JSON request and is different for every class. A lot of this seems to be unnecessary work as the key names are very often exactly the same as the property names in your class, and you rarely expect date strings to be formatted differently in different requests. As a result of this it seems like this processes can be improved.

Web API URL Redundancy

Setting up Web API URL's in the front end for your different models results in a lot of redundant code. The URL's seem to be very similar to each other, and follow a specific format. When you make data requests, you are often retyping the resource domain and the database table key for each request.

.NET Date formatting inconsistencies

A JsonResult, if executed from a MVC5, or Web API 2 controller format dates differently and produce different results. You might want to mix API controllers with separate methods from MVC5 controllers in your application and the inconsistent date formatting is an inconvenience.

Web API 2 Controller: IHttpActionResult	MVC5 Controller: ActionResult
| ```
return Json(new User() {
    DateOfBirth = DateTime.Now
});
``` | ```
return Json(new User() {
    DateOfBirth = DateTime.Now
},
JsonRequestBehavior.AllowGet);
``` |
| `"DateOfBirth":"2015-05-25T13:28:06.9616293+02:00"` | `"DateOfBirth":"\/Date(1432553423578)\/"` |

External libraries management

There is a whole wealth of third party libraries on the internet, meaning a developer doesn't have to do everything himself which wouldn't be practical. Developing your own code base also has its own problems with keeping it up to date in all your applications unless you use some tool to do this for you.

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
- Code principles to provide implementation guidelines

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 functions, and analysing their performance.

I will consider blog posts and articles containing people's experiences to be valid if they coincide with preconceptions I have of a subject from my own experiences, and they don't contradict factual information sourced elsewhere.

Principles

As this project will be open source, I will ensure that my code follows a set of principles to create a logical structure optimize its legibility, and that it conforms with guidelines specific to its programming environment. Having a pre-defined set of accepted principles helps make your code readable and understandable, making it possible for multiple developers to work on the same solution. If done correctly, it should be impossible to see who wrote what code in an application with multiple developers.

Separation of concerns

The separation of concerns principle states that different aspects of functionality should be managed by separate, clearly distinguished sections of code. This means separating business logic, UI and data

requests etc. Mixing up too these can create very messy and hard-to-read code. For example, changes to the user interface should not require changes to the business logic.

MVC (Model, View, Controller) is a common example of separation of concerns. It is a pattern used in a number of frameworks including .NET C# and Swift which will be used in this project. It separates the UI logic (View) from the main functionality of the app (Controller) and the data source (Model).

HTML5 in web design is another example, separating HTML from CSS and JavaScript. HTML is used to define the elements on a web page, CSS defines the styling and positioning, while JavaScript handles how it interacts with user input. It also demonstrates that you can have separation of concerns within other separation of concerns.

Number of lines of code

It is very important to consider the lines of code something takes. It helps with legibility, allowing future developers to interpret what is happening in the code as well as reducing the application size. I am going to consider this in my comparisons of different libraries and code snippets for this project, prioritizing a smaller code base where possible.

DRY[1]

DRY stands for Don't Repeat Yourself. It states that you shouldn't repeat code where you it can be avoided. The reasoning for this is that redundant code requires changes in the code to be mirrored in multiple places. This is both time consuming, and increases the possibility for error.

There are many constructs in most programming languages to allow for this, including loops, classes, functions etc. A practical implementation of DRY programming would make use of these different constructs and side off code into functions or base classes. By using inheritance from base classes, you put as much code as you can into the base class, meaning that all classes which inherit from the base class have access to its methods and properties. I am also going to consider the use of utility classes.

---

[1] http://joelabrahamsson.com/the-dry-obsession/
http://www.artima.com/weblogs/viewpost.jsp?thread=331531

DAMP[2]

DAMP stands for "Descriptive and Meaningful Phrases". This is a complimentary principle to DRY to try and prevent you from being too "DRY". The idea is that sometimes it can be preferred to have some redundant code, instead of having one method which tries to cover multiple scenarios for example. It prefers legibility over pure effectiveness. Also, if you side off too much logic into separate functions, a developer trying to read your code could end up having to follow a long chain of functions and loose the overview of the context.

Clarity [3]

This principle doesn't dictate which conventions you choose to follow, but that you choose one and stick to it. This applies all throughout the code, in respect to spacing, indentation and naming conventions etc. The naming conventions should also be descriptive as well as consistent. It is preferred that a variable or function name is longer, and more descriptive than short.

By abiding to this, you make it easier to read the code and spot errors. A lot of software development is maintaining existing code so it is very important.

REST API[4]

I aim to create a class to handle communication between an object on the frontend and a Web API. 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 library 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.

---

[2] http://simplythetest.tumblr.com/post/93114654620/keep-your-test-code-damp
[3] (Taylor, n.d.)
[4] (Microsoft, 2015)

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:** The method 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

Compression[5]

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 requests using different compression techniques.

---

[5] (Pot, 2012)

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.

"mmmmmmmmmmmYYYY" 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. Huffman coding tackles the issue of the data size of characters being the same for all characters regardless of the frequency of its usage. E.g. both an "e" and "x" occupy one byte of data. It then builds a tree prioritized by the frequency of usage and uses this to compress the data.

GZip[6]

GZip is a file format based on the deflate algorithm but contains extra headers such as timestamp, original file name and a CRC-32[7] 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.

Testing

Google Chrome

I will use Google chromes network tab in the developer tools to monitor the HTTP request times and data sizes. It is more accurate than the values that can be extracted from the response via JavaScript. It provides a space to monitor all traffic going in and out of a web page, providing access to all headers/content/status codes etc.

To compare the different compression algorithms I decided to run some requests from a web browser, using 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 time. Compression seems to become more effective with the more data you send.

There is also a minimum string length needed for compression to be useful at all, as 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 compression to be extremely effective on any decent sized JSON response.

Approaches

I am going to find out if compression, applied using different methods makes a difference to the data size of the final output. This will be done using manual testing.

The test is a JSON response totalling 38.4kb

☐ compressWithFilter	POST	200	application/json	j...	38.4 KB	10 ms

Content-encoding

HTTP has a feature which allows you to define your compression preference via an HTTP header: "Content-Encoding". It accepts a number of different compression types, including "gzip" and "deflate".

It is possible 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) "Content-Encoding": "gzip"
2) "Content-Encoding": "deflate"

☐ compressWithFilter	POST	200	application/json	j...	2.9 KB	6 ms
☐ compressWithFilter	POST	200	application/json	j...	2.9 KB	5 ms

I found that this method of compression was extremely efficient, reducing a 38.4kb request to 2.9kb. Interestingly, the difference between GZip and Deflate were un-noticable, contradicting findings from the compression method tests.

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 add code.

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

```
response.AppendHeader("Content-encoding", "deflate");
response.Filter = new DeflateStream(response.Filter, CompressionMode.Compress);
```

Compressing HTTP body

This method takes the data from the response, compresses it using the same DeflateStream (or GZipStream) and putting into back into the HTTP body. This method was used in the compression comparison tests. This technique is limited to compressing the contents of the HTTP body and provides no ability to compress the headers or any other part of the response. It also requires the compressed data to be encoded to be able to transport properly. This increases the data size to 3.4kb. It also disproportionately increased the response time to 36ms which was an interesting result although could be explained by server compiling.

| compressManually | POST | 200 | application/json | j... | 3.4 KB | 36 ms |

Findings

I found that the way the compression was implemented made a difference to the final data size. By using Content-encoding headers, widely supported by many browsers and HTTP protocols for many different languages, you can achieve a higher compression.

Although being easier to implement and more efficient, there is a problem due a lack of control for the order in which encryption and compression is applied to maximize efficiency (See Order Encryption/compression). This compression is the final wrapper for the request, and cannot be tampered with afterwards (except using HTTPS).

Although applying compression to the HTTP body is not as efficient as content encoding, it does however give you more control over the actual data, allowing you to further process it if needed.

As my findings showed that encryption should be done after compression, content encoding is not appropriate for this project and I will aim to use compression techniques which can be applied with the same level of control.

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.

It is possible to increase the security from a single key symmetric algorithm via the use of a SALT or IV. This adds a randomly generated aspect to the encryption. It generates a random key, separate from the encryption key, which is used as part of the encryption process. This key is then appended somewhere inside the finished product. The receiver knows where to find the salt, extracts it and uses that value in the decryption. This is not absolutely secure, but does make it harder, as the encrypted value for the same value will be different each time.

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 is the possibility to verify the sender on both the server and frontend. This means that it will be much harder for a hacker to create fake requests. Asymmetric encryption is much more secure than symmetric, but does require more processing power.

HTTPS [8]

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 Asymmetric 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 "expensive" handshake. It requires information of the symmetric connection to be stored in memory on the server.

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 on the server to allow the traffic through. It adds 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 as well as all its headers and content.

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

The key benefits to HTTPS are:

- Very secure initial handshake
- Entire request is encrypted (including headers and URL)

Data consumption & Latency [9]

Encrypting data increases the size of the data, the main problem being in the initial request where the initial handshake takes place. This is a comparatively slow process often requiring extra requests between the client and to relevant certificate authorities.

---

[8] (Gilbertson, 2011)

[9] http://www.semicomplete.com/blog/geekery/ssl-latency.html

It might be a good idea to not encrypt the entire response, but 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.

Hashing[10]

Hashing data completely transforms data into something else. It is an irreversible process and does not contain the original data. You can hash an entire program to a small hash signature. This is often done on software to allow the possibility of checking if the piece of software is the original copy, and not a modified version.

When hashing a string for example, even with one character different, the end result is completely different:

```
hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542
```

Hashing the same data however, always produces the same result. This is very commonly used in password security. Instead of storing the user's actual password, they store a hashed version of it, and try to re-create the hash when they enter their login credentials. That way the password isn't transported over the web, or stored in a database.

Findings

While HTTPS is definitely the most secure option, it has its cost and setup time drawbacks. Symmetric and Asymmetric encryption both have their uses in different scenarios. For the purpose of this project I am going to concentrate on Symmetric encryption. Although there are more secure encryption methods available, 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 doesn't make sense to try to emulate HTTPS as it will never be as well tested

---

[10] https://crackstation.net/hashing-security.htm

as HTTPS itself. If more security is needed, this library may not be suitable and HTTPS should be used instead.

Some of the benefits from HTTPS communication however could be adopted however including masking target URL and encrypting response data. I am going to try and consider as many of the benefits of HTTPS as possible.

Hashing is not appropriate for this project at all, because it is a requirement that the data can be retrieved after transport, of which is impossible with a hashed value. Hashing is instead a completely separate topic which should be tackled "per-project".

AES [11]

AES is an accepted encryption standard used by the US Government and very widely adopted in internet security. I am going to base my use of this algorithm on its popularity and abide by the standard. AES encrypts data with a 128, 192 or 256 bit keys while utilizing other techniques like a SALT and IV. It is also well supported on multiple platforms which is important to the ability to decrypt the data in different programming environments.

Encoding[12]

After testing, I found that after a string has been compressed or encrypted, 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??^"

There are many types of encoding methods but I am going to use Base64[13] because it is a well-established and widely used encoding algorithm. This is important because it needs to be easily and accurately decoded on multiple platforms. Both C# and Swift have built-in Base64 encoding and decoding methods so it seems appropriate to utilize this.

---

[11] (Wikipedia, n.d.) AES | (Youse, n.d.) AES
[12] http://en.wikipedia.org/wiki/Basic_access_authentication
[13] http://en.wikipedia.org/wiki/Base64

Base64 encoding is used to transform it into a transportable string that looks something like:
"LT8/XHQ/MFxmVz8/P1FaP2g/Pz9GPz9cclwifz8/P1wiez8/Pz9uPz8/OHxiMD8wPz96Pz8/Rz8/P29KXGI
/P34/cz8/XuKAnQ=="

It is much easier to transport as it only uses a small set of characters[14], all of which can be transported through HTTP requests safely, including:

`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`

Base64 encoded strings while unreadable to a human, are easily decoded again. The purpose is not to secure data but simply to ease the transportation of data. Using this encoding does however increase the data consumption by 33%. While this seems like a drawback, it is a necessary cost.

---

[14] http://www.codeitive.com/7NHJUjPVVj/regex-for-invalid-base64-characters.html

Order (Encryption/compression)

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

The following tests compare two sets of data of varying size, with two methods. The first method is encryption before compression and the second is compression before encryption. Both show the data size after each step.

Test1: (array of 2 JSON objects)

Original: 725 bytes

Method 1	Method 2
AES 256 Encrypted output: 1024	Compression output: 182
Compression output: 1064	AES 256 Encryption output: 344
Output: 1064 bytes	Output: 344 bytes

Test2: (array of 1000 JSON objects)

Original: 79673 bytes

Method 1	Method 2
AES 256 Encrypted output: 106240	LZ77 Compression output: 7264
LZ77 Compression output: 89912	AES 256 Encryption output: 9688
Output: 89912 bytes	Output: 9688 bytes

I found that the ordering makes a huge difference. This is because the compressor has a much easier time finding patterns in the original string, than an encrypted string. Once a string has been encrypted, it is essentially nonsense and looks like a completely random set of characters.

By compressing first, you are able to make the most of the compression algorithms, before encrypting the final result which is a must smaller string. This reduces data size as well as the processing time it takes to encrypt the larger response.

Other solutions

Object mapping

I am going do some online research to see if there other object mapping (from JSON to Swift objects) solutions out there, and analyse them.

RestKit

https://github.com/RestKit/RestKit/wiki/Object-mapping

This is an Objective-C library which allows you to map JSON responses to classes. It requires however, a mapping attribute to be set for every single key, even if the property key matches the JSON key.

```
[mapping addAttributeMappingsFromDictionary:@{
    @"user.name":   @"username",
    @"user.id":     @"userID",
    @"text":        @"text"
}];
```

Also, according to the documentation the library contains up to 25 different classes. This definitely goes against the DAMP and lines of code principle by over complicating the class structure and seemingly more code than necessary.

DCKeyValueObjectMapping

DCKeyValueObjectMapping is another Objective-C library which doesn't require manually mapping of the properties as default. It handles nested classes, as well as automatically reading a dictionary using a class's properties as keys. There were some issues however, due to bad error handling of null values in the dictionary, resulting in crashes and hard-to-read mark-up. The mark-up goes against the clarity principle, with undescriptive naming e.g. "addAggrevator". It also seems a bit too DRY with a large amount of separate classes of which their purpose was hard to decipher. This made it very difficult for me to edit the library to fix the null value issue.

```
DCPropertyAggregator *aggregteLatLong = [DCPropertyAggregator aggregateKeys:[NSSet setWithObject
DCPropertyAggregator *aggregatePointDist = [DCPropertyAggregator aggregateKeys:[NSSet setWithObj

DCParserConfiguration *configuration = [DCParserConfiguration configuration];
[configuration addAggregator:aggregteLatLong];
[configuration addAggregator:aggregatePointDist];

DCKeyValueObjectMapping *parser = [DCKeyValueObjectMapping mapperForClass:[Tweet class] andConf
Tweet *tweet = [parser parseDictionary: json];
```

Findings

Of all the libraries I could find online, they didn't seem to coincide with my coding principles for this project. I was also unable to find a pure swift library and determined that it would be necessary to create my own solution to this problem.

JSON format

The compression algorithms simply compresses the result, looking for patterns. But it might be possible to optimize the JSON format completely, reducing the amount of data before sending it to the compressor.

CJSON[15]

In an array of objects, the JSON keys are repeated multiple times. The basic principle with CJSON is to remove the keys from the JSON responses, and use template maps to know which values belong to which keys.

```
[
  { // This is a point
    "x": 100,
    "y": 100
  }, { // This is a rectangle
    "x": 100,
    "y": 100,
    "width": 200,
    "height": 150
  },
  {}, // an empty object
]
```

---

[15] (Anon., n.d.)

Using the CJSON algorithm, this could be represented as:

```
{
  "templates": [
    [0, "x", "y"], [1, "width", "height"]
  ],
  "values": [
    { "values": [ 1,  100, 100 ] },
    { "values": [2, 100, 100, 200, 150 ] },
    {}
  ]
}
```

Object definitions are transported as arrays of values instead of key/value dictionaries. The templates are part of the response and include an "id" which is the first number of the array. E.g. [1, "width", "height"] – has template id of 1. The first value in the values array points to the template id. This allows it to be unpacked on the other side.

According to the source, it can drastically reduce data size:

CJSON seemed like a very promising algorithm for further compression, although there didn't seem to be a .NET library available but could potentially be ported to C#.

HPack is a data compressor which claims to be able to reduce the data by up to 70%. It takes the compression a step further than CJSON by removing an obvious split between the objects. It is similar to CJSON as it also uses templates for each object.

```
["name","age","gender","skilled"]
```

It can also setup enum values for repeating values for further optimizations

```
[["name","age","gender",["Male","Female"],"skilled"],["Andrea",31,0,true],["Eva",
27,1,true],["Daniele",26,0,false]]
```

In this case, gender is setup two accept one of two values "Male" or "Female", set inside an array. They can then be referenced in notation as 0 or 1, representing their position in the array.

Taking it further

I think that the HPack algorithm could be utilized in this project as it seems to have all the benefits from CJSON (reducing keys from the response as templates) as well as other optimizations. It could however be improved.

Each object doesn't actually need to be encased in curly or square brackets because the beginning and end of each object can be deduced by the number of properties in the each object definition.

Using the template:

```
["name","age","gender","skilled"]
```

["Alex", "23", "Male", 1], ["Tom", "25", "Male", 5] could be represented as:

["Alex", "23", "Male", 1, "Tom", "25", "Male", 5]

HPack also sends the templates in every request. I suggest the possibility of sending the template once, storing it in memory on the frontend device and using that to decode the data.

---

[16] (hpack, n.d.)
  (JSONH, n.d.)

You could send a header in the HTTP request, asking for the template if needed. The response would then be smaller for all but the first one. I would suggest a method to specify an expiry date for the cached template or a way to manually refresh the template. This would have to be carefully thought out because you might end up in the situation where your response doesn't match the cached template if you have updated your database model, after release for example.

A new format could also be created, combining CJSON with HPack to further decrease unnecessary characters using templates as well as merging responses into a single array of values.

Library management

Swift

Cocoapods [17]

CocoaPods is the most popular library manager for iOS and OSX projects which originated with Objective-C. It has only recently been updated to support Swift frameworks, and now allows you package your library as a "pod" and integrate it easily into any project.

By using CocoaPods you gain some advantages over simply dragging and dropping the files into your project.

- **Maintenance:** The code has a central base. If you have multiple projects all using different versions of your library, it can be very difficult to maintain which files have the latest code. It also dissuades you from editing the library code from inside your project as they are added as read-only.
- **Version control**: You can specify "latest", or use a specific version. In some apps you might want to lock the version number, after it has been tested so you can ensure that the app performs as it should.
- **Updates**: Updates can be downloaded very easily by simply typing "pod update" in the terminal, from your project folder.
- **Dependencies**: Often, your code has its own dependencies. You can specify those in your CocoaPod config file. When your pod is installed, it will then automatically download its dependencies.

---

[17] https://cocoapods.org/
http://nshipster.com/cocoapods/
http://en.wikipedia.org/wiki/CocoaPods

CocoaPods are a very effective way to manage external libraries. They are well documented[18] and widely used throughout iOS and OSX development.

.NET

Class libraries

In Visual studio, you can setup a class library. You can then package all your classes into a useable DLL which can be referenced from another project. This will also make it harder to make edits to the class library code base which will provide the same benefits as for the Swift CocoaPods.

NuGet

A feature of the Visual Studio IDE are the NuGet packages. You are able to browse and install third party libraries from inside Visual Studio. It handles all the downloading of the files, updating and organization of the references.

You can publish your DLL as a NuGet package and make it searchable from the IDE.

Research Findings

As a result of the research I was able to compose a list of the optimal configuration, and a base for starting the build phase of the product.

Optimal settings

Compression	DEFLATE (LZ77)
Encryption	AES
Encoding	Base64
JSON format	Standard JSON
Swift Library	CocoaPod
C# Library	Class library

---

[18] https://guides.cocoapods.org/

I plan to implement as many as the optimal settings, decided from my research into a product which will satisfy the product statement and allow the transport of encrypted data between the front and backend.

To achieve the best results, a number of things will need to be considered and tested, including data size as well as the processing time of implemented compression and encryption algorithms. The benefits of compression become obsolete if the total time of the request doesn't get reduced due to a processing overhead.

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 handles both security and compression.

The product goal is to be easy to install, and use for a future developer. This means considering the installation and updating of third party libraries and abiding the coding 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 it breaking anything, and allow you to choose where to apply functionality from this library.

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 (See Configuration). 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 and Encryption

Both libraries are using a compression library called LZString[19], and AES encryption with a 128 bit key.

LZString is the only working compression library which successfully produced the exact same results on both front and backend.

The encryption key was generated using a GUID (Globally Unique Identifier) generator. A GUID contains **2^1225,316,911,983,139,663,491,615,228,241,121,400,000** combinations. The encryption also applies a random SALT to the process which ensures the encryption looks different every time, even with the same data.

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
- Easy installation to an existing project.

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.

---

[19] ((GitHub), n.d.)

```
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.

```
[EncryptAndCompressAsNecessaryWebApi]
[DecryptAndDecompressAsNecessaryWebApi]
public IHttpActionResult PutCustomer(int id, Customer customer)
```

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 JsonResult object to avoid errors.

```
[EncryptAndCompressAsNecessaryWebApi]
[DecryptAndDecompressAsNecessaryWebApi]
public class CustomersController : ApiController
```

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 is not setup to receive any data, you can just add one action filter to encrypt the output:

```
[EncryptAndCompressAsNecessary]
public JsonResult GetOneUserEncrypted()
```

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

```
[EncryptAndCompressAsNecessary]
public JsonResult GetItemsEncrypted(int take)
{
    return Json(AlexDbEntities.JsonDB().CardDesignItems.Take(take).ToList());
}
```

Masked route

A masked route allows you to encrypt components of the URL to mask the contents of the data. This is to prevent a hacker from being able to gain information about the context of the request. E.g. if it was a request to login, and the URL looked like: "/api/login", a hacker would expect "username" and "password" as parameters somewhere in the data.

To demonstrate the masked route, this is a URL which points to the URL: **/api/CardDesignItems/5.** Secret routes must start with the secret prefix. The prefix is necessary to distinguish from the default route.

```
string prefix = CompresJSONRouteManager.WebApiSecretUrlPrefix;
string id = CompresJSONRouteManager.EncryptSecretUrlComponent("5");
string controller = CompresJSONRouteManager.EncryptSecretUrlComponent("CardDesignItems");
string url = Tools.Domain(Request.RequestContext) + "/" + prefix + "/" + controller + "/" + id;
```

```
<a href="@url">@url</a>
```

This produces the following:

/apih/VTJGc2RHVmtYMSs1U1FMYnZlWXFRNFdDQWg0SHhhaXBMZEhYKzd0VjBNYz0=/VTJGc2RHVmt YMTg2Sk5XOVFaczJmL3VtZDZ3bjBWenErdkdoWmxOM3B2WT0=

To achieve this behaviour the routes need to be setup in the global.asax. This is done so by adding the following two lines:

```
GlobalConfiguration.Configure(CompresJSONRouteManager.Register);
CompresJSONRouteManager.RegisterRoutes(RouteTable.Routes);
```

How it works

Action filters[20]

An action filter is a class which provides the ability to run code before and after an ActionResult on a MVC Controller executes. They are very powerful classes which have access to a lot of different information about the current context including route parameters and information about the connection etc.

[20] http://hackwebwith.net/asp-net-mvc-5-action-filter-types-overview/

This allows you to add logic to decide whether or not to allow a client access to that action result. This might be an action filter that requires an authorization header present, or an HTTPS connection. You can check for all these values here. They are applied to an action result via square bracket notation. The following example is a view in an MVC website which will only allow access through a secure HTTPS connection.

```
[RequireHttps]
public ActionResult SignIn()
{
    return View();
}
```

The action filters in this project are used to manipulate the incoming and outgoing data, converting normal parameters into a single processed string.

To access the context before the action result has been executed you can override the method: "onActionExecuting":

```
public override void OnActionExecuting(ActionExecutingContext filterContext)
{
```

This method is used to process the incoming data sent in a POST from the client for example. We are expecting data inside one parameter called "data". Before we can access this we need to access the HTTP body which contains all the post values:

```
Stream req = filterContext.HttpContext.Request.InputStream;
req.Seek(0, System.IO.SeekOrigin.Begin);
string httpbody = new StreamReader(req).ReadToEnd();

Dictionary<string, string> httpBodyDictionary = Converter.QueryStringToDictionary(httpbody);
```

This code retrieves the HTTP body as a string, and converts it to a useable dictionary. The data can then be accessed from the dictionary using the key: "data".

It can then process the data inside that parameter, and convert it into another dictionary containing the original values sent in the request.

```
string json = CompresJSON.DecryptAndDecompressAsNecessary(httpBodyDictionary["data"]);
var dict = new JavaScriptSerializer().Deserialize<Dictionary<string, string>>(json);
```

The value "dict" now contains the original values sent from the client.

In MVC, you usually expect the parameters to be mapped to an object automatically by the time your ActionResult code executes:

```
public JsonResult AddCustomer(Customer customer)
```

As we are not posting the parameters directly, we need to do some extra work to restore this behaviour. Without this, the user object will be empty, creating unpredictable behaviour.

From the action filter it is possible to retrieve the expected parameters from action result, along with its type.

```
var mvcActionModelParameters = filterContext.ActionDescriptor.GetParameters();
```

E.g. Using the "AddCustomer" example, this variable will now contain one key called "customer" and type: "Customer". The type is available only as a string, but you can use Reflection to create an instance of an object using the string type description.

```
var o = System.Reflection.Assembly.GetExecutingAssembly().CreateInstance(typeName);

o = Mapper.ToObject(filterContext.ActionParameters, o);

if (o != null)
{
    filterContext.ActionParameters[parameter.ParameterName] = o;
}
```

The object's properties are then populated using another method "toObject" which is able to parse values from a dictionary into the object. This object is now ready to pass into the Action Parameters dictionary which is what is passed to the action result parameters.

With all this in place, you can now use this object as you normally would. This example shows how to receive encrypted/compressed data, add the customer to the database and return the encrypted/compressed newly inserted customer.

```
[DecryptAndDecompressAsNecessary]
[EncryptAndCompressAsNecessary]
public JsonResult AddCustomer(Customer customer)
{
    new AlexDbEntities().Customers.Add(customer);
    return Json(customer);
}
```

Due to the slight differences between Web API and MVC action results, I had to create a different class for each. To use this on a Web API Action result requires using the Web API Action filters:

```
[EncryptAndCompressAsNecessaryWebApi]
[DecryptAndDecompressAsNecessaryWebApi]
```

The functionality is exactly the same, just with a slightly different code.

Custom routes

The default Web API route looks like this:

```
routeTemplate: "api/{controller}/{id}",
```

This is able to map segments of the URL into accessible values. In the case of controller and id, these help the program decide which action result, on which controller to execute.

The extra lines in global.asax are responsible for setting up the extra routes. While you can encrypt segments in a URL, you must have a hard-coded prefix. This is because the route config needs this in order to work out which route to match it to.

```
public static string WebApiSecretUrlPrefix = "apih";

config.Routes.MapHttpRoute(
    name: "SecretDefaultApi",
    routeTemplate: WebApiSecretUrlPrefix + "/{c}/{id}",
    defaults: new { id = RouteParameter.Optional },
    constraints: null,
    handler: new DecryptWebApiRouteHandler(GlobalConfiguration.Configuration)
);
```

The route looks similar to the default API route but uses a slightly different keyword as a "prefix". It also has an extra property called the handler. The handler is responsible for processing the data extracted from the URL. It does this by overriding the "SendAsync" method:

```
protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken c
{
    var routeValues = request.GetRequestContext().RouteData.Values;

    string c = CompresJSONRouteManager.DecryptSecretUrlComponent(routeValues["c"].ToString());
    routeValues["c"] = null;
    routeValues["Controller"] = c;

    if (routeValues.ContainsKey("id"))
    {
        string id = CompresJSONRouteManager.DecryptSecretUrlComponent(routeValues["id"].ToString());
        routeValues["id"] = null;
        routeValues["id"] = id;
    }

    return base.SendAsync(request, cancellationToken);
}
```

From this method it can access the route data values, apply the decryption and ensure that the "Controller" and "id" parameters are set. The route config is then able to continue routing to the appropriate action result as it normally would.

Third party dependencies

JINT[21]

JINT is a framework, accessible via the NuGet package manager which provides the ability to run JavaScript from inside the .NET environment. This tool allowed for the LZString compression algorithm to run on both the client and server in the form of JavaScript.

CrytoJS (CS library)[22]

CryptoJS is a JavaScript library with a working .NET port. It allows encryption and decryption of data using the AES algorithm, featuring a SALT to further increase the difficulty of decryption without the secret key.

Compressor & Encryptor

Structure

The "Compressor" and "Encryptor" classes, function as an interface between the business logic and compression algorithms. The purpose of the class is to provide access to the compression/encryption libraries, while complying with the separation of concerns principle.

---

[21] https://github.com/sebastienros/jint
[22] https://github.com/sytelus/CryptoJS

By enforcing a modular structure, the compression/encryption algorithms and libraries can be switched out easily, without having to change any code outside these classes.

Compression and Encryption throughout the library is mainly done via the "EncryptAndDecryptAsNeccessary", and "DecryptAndDecryptAsNecessary" methods. This function has easy access to all the complex third party algorithms via the "Compressor" and "Encryptor" interface classes:

```csharp
public static string EncryptAndCompressAsNecessary(string str)
{
    //compress

    if (CompresJSONSettings.shouldCompress)
    {
        str = Compressor.Compress(str);
    }

    //encrypt
    if (CompresJSONSettings.shouldEncrypt)
    {
        str = Encryptor.Encrypt(str);
    }


    return str;
}
```

This provides very easy to use methods from within the business logic and mask all the complex functionality of the algorithms themselves.

Compressor

The compressor uses JINT to execute code from the LZString. LZString was originally chosen as it have a C# implementation as well as the JavaScript one but unfortunately didn't work as expected.

The compressor contains two easily to use functions:

```csharp
public static string Compress(string str)
```

```csharp
public static string Decompress(string str)
```

The Compressor bridges the gap between the business logic and library code and accesses the relevant library. The compress function returns a base64 encoded string, and decompress expects a base64 encoded input string:

```
if (CompresJSONSettings.compressionMethod == CompressionMethod.LZ77)
{
    var data = Convert.FromBase64String(str);
    var decompressedData = LZ77.Decompress(data);
    return Converter.BytesToString(decompressedData);
}

else if (CompresJSONSettings.compressionMethod == CompressionMethod.GZip)
{
    var data = Convert.FromBase64String(str);
    var decompressedData = GZip.Decompress(data);
    return Converter.BytesToString(decompressedData);
}
```

As the solution in its current stage requires LZString in the backend, it therefore requires access to the JavaScript via JINT.

```
else if (CompresJSONSettings.compressionMethod == CompressionMethod.LZString)
{
    Dictionary<string, object> args = new Dictionary<string, object>() {
        { "x" , str }
    };
    return JavaScriptAnalyzer.runJavaScriptFunctionWithArgs("Decompress", args).ToString();
}
```

It is done so via the class "JavaScriptAnalyzer" which functions as the interface between the JavaScript and C#. The JavaScript analyser executes the JavaScript and returns the string value to the Compressor.

The backend is setup so that the compression method can be easily switched via an Enum in the "CompresJSONSettings" class:

```
public static CompressionMethod compressionMethod = CompressionMethod.LZString;
```


Encryptor

The Encryptor's structure is setup very similarly to the Compressor, providing a bridge between the libraries and application code.

The Encryptor has two methods, providing easy access to the third party functionality (CryptoJS):

```csharp
public static string Encrypt(string str)
{
    var p = new CryptoJS();
    var encrypted = p.OpenSSLEncrypt(str, CompresJSONSettings.EncryptionKey);
    return Converter.Base64Encode(encrypted);
}

public static string Decrypt(string str)
{
    var p = new CryptoJS();
    var decoded = Converter.Base64Decode(str);
    return p.OpenSSLDecrypt(decoded, CompresJSONSettings.EncryptionKey);
}
```

These methods encrypt the input string using the globally defined encryption key (and SALT inside the library) and return it as a base64 encoded string. The decryption expects a base64 encoded string as the input.

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[23]. 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:

```
import ABToolKit
```

Usage

Making a request

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

```
CompresJsonRequest.create("http://alex.bechmann.co.uk/compresjson/api/Customers/1", parameters: nil, method: .GET)
```

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

You can add code in the callback by adding a handler (e.g. onDownloadSuccess):

```
CompresJsonRequest.create("http://alex.bechmann.co.uk/compresjson/api/Customers/5", parameters: nil, method: .GET)

.onDownloadSuccess { (json, request) -> () in

    println(json)
}
```

This will convert the processed JSON from the response:

```
{"data":"U2FsdGVkX1/LkZ+SBQ04LhRH8AhUTJbDIdT70xFOTDH08RZBffI7cE21fGqxAEhAfqW4B1ZAmhgZgSrlV5sn
tacZqwsVjkdAlced2Eb66p5UgMhas11MvFccCYU5E1oYh4Wex9kAY6lGW6z0vOArj0FOpOgDmKs8ewqGZSY7qxn0+TqzB
YYTT0LgaUSGHKc4MTkGr2FGNQ2RZdtW9IPeL5Doq2mVaz896rRLiqsTsFWTf2dL7fk8gP/KBm5CdcWnFpZORzyVZOLwFO
ccoZMJE9T47NnzSwECENqWGnZW4eoyu5fCz991tAQdglvLxaDQjBtQcunN4qqDgJersudr9MC69/TbTm+PsW7tt4tB/ri
WkdUaglAZ43VbTICwBYYdXHBB0d/kO+55uacNgzKzRBPKzJY5TgLDTzRrmD7WTNMK30k25ip9gv1CZVarbK6tQidKQR8F
ciuZxBXNSsUAgF542qANNFKJQnneG9Qmj2AeLDrubXAbUR/FCHjf5y7nW0a4Ymz7TcpOvEu6svhAeDET1etm6HnTEofns
Nl1/VV22NO4kUHzf710fvBAq61pRKKL5Fz04DM2LVPNaiZ58fJN+ix7c6jogLk6h3AznzyLSxmp2oVlNmJhCThPNLsWOw
JJZEJDiAxuszSgzgoRjGbd1fjW880pZDdwU9/rNBWt562sdcq2uqNdFlR2JlpqX81un2qBI8dLBWavrEFhvWe4BsL5ddm
8Y8PknD0Z0wvciPZeT1cvDP9WlnhdQVTvJmOGJTNFyxXJj88dGByEOfEAlFGfljy4WFDKABaKK+fTu/mTulf6eIs41W/n
PvgjouTdAAyM2b1jAala+eXN5lSaUINKHNsHqbpDi7nvIP4So2qxeAv/ohEADpQ+RAUos1Lm0FrwSr35BwR90chADNbPP
dlfLHLvpFmpOuIymu4="}
```

Into normal JSON and print it to the console:

```
{
    "Address2" : null,
    "CountryCode" : null,
    "CustomerName" : "Sebastian",
    "Address3" : null,
    "Password" : null,
    "CustomerID" : 5,
    "CurrencyCode" : "GBP",
    "Address4" : null,
    "Orders" : [

    ]
```

All Output ⌄   🗑 | ☐☐
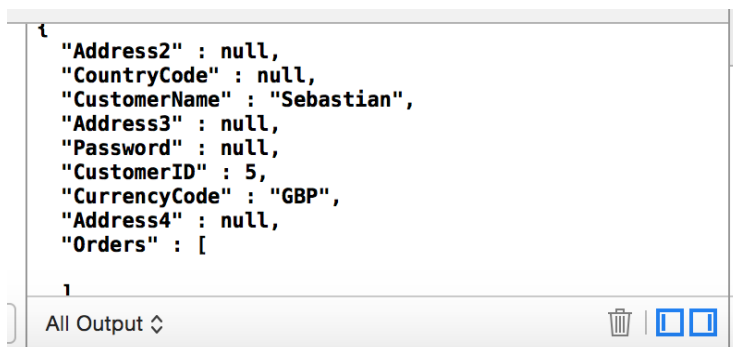
The values could then be unpacked using SwiftyJSON syntax:

```
let customer = Customer()
customer.CustomerID = json["CustomerID"].intValue
customer.CustomerName = json["CustomerName"].stringValue
```

Multiple handlers

You can add "onDownloadSuccess", "onDownloadFailed", and "onDownloadFinished" handlers in

any order.

```
CardDesignItem.webApiGetMultipleObjects(CardDesignItem.self, completion: { (objects) -> () in

    self.items = objects

})?.onDownloadFinished({ () -> () in

    refreshControl?.endRefreshing()
    self.tableView.reloadData()

}).onDownloadFailure({ (error, alert) -> () in

    alert.show()
})
```

- **"onDownloadSuccess"** gives you access to the JSON response.
- **"onDownloadFinished"** is useful for cleaning up the UI (e.g. hiding loader animations) regardless of whether the request failed or not.
- **"onDownloadFailed"** provides you with a pre-setup UIAlert containing the error message and a retry button. You can edit the message manually via the alert object, or use this code space to generate your own error handling.

Activating "alert.show()" will bring up an alert box like this:
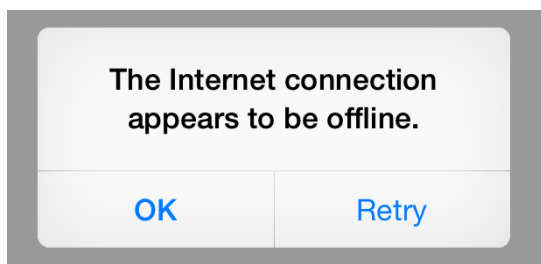
JSON Mapping

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:

```swift
class User: CompresJSONObject {

    var id = 0
    var Username = ""
    var Email = ""
    var Password = ""
```

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):

```swift
let customer = Customer.createObjectFromJson(json)
```

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:

```swift
var fontID = 0
```

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

```swift
self.registerKey("fontID", jsonKey: "FontID")
```

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

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. "Address.self" and "Product.self" refer to user defined swift classes. (Including JSON key parameter if necessary). When it maps to the class, it will use the defined mappings setup from inside each of those classes.

```swift
self.registerClass(Product.self, forKey: "Products")
self.registerClass(Address.self, propertyKey: "address", jsonKey: "Address")
```

You might also have a date, formatted as a string. You can choose to use the global default for the date format, or define it per key. To use the global default you would have to set first, this should be done in the AppDelegate.

```
JSONMappingDefaults.sharedInstance().dateFormat = DateFormat.DateTimeWithSeconds.rawValue
```

It is important to make sure that this format is accurate because otherwise the date formatter will through an exception, and the app will crash. Once this is set, dates can be registered to property keys:

```
self.registerDate("DateOfBirth")
```

If you need to specify the format you can do so. You can also use the DateFormat enum helper, or write it yourself.

```
self.registerDate("DateOfBirth", format: DateTime.DateTimeWithSeconds.rawValue)
```

```
self.registerDate("DateOfBirth", format: "dd/MM/yyyy HH:mm:ss")
```

This example shows how these methods can be combined to map a JSON response to a class[24].

If your model looked like this:	And the JSON you were expecting looked like this:
| ```swift
class User: JSONObject {

    var id = 0
    var Username = ""
    var Email = ""
    var Password = ""

    var DateOfBirth :NSDate = NSDate()
    var Products = Array<Product>()
    var address = Address()
``` | ```json
{
    "UserID":5,
    "Username":"Alex",
    "DateOfBirth":"20/09/1991",
    "Products":[
        {
            "ProductID":1509,
            "ProductName":"Pencil"
        },
        {
            "ProductID":1352,
            "ProductName":"Pen"
        }
    ],
    "Address":{
        "Ad1":"32 Haughgate Close"
    }
}
``` |

You would put the following into the "registerClassesForJsonMapping" method:

```swift
override func registerClassesForJsonMapping() {

    self.registerKey("id", jsonKey: "UserID")
    self.registerDate("DateOfBirth", format: DateFormat.Date)
    self.registerClass(Product.self, forKey: "Products")
    self.registerClass(Address.self, propertyKey: "address", jsonKey: "Address")
}
```

The properties which match by default, do not need to be specified here. In this case "Username", "Email" and "Password" would be mapped automatically.

Automating the REST URL's

The REST URL's are managed within the "WebApiManager" class. A set of RESTful routing URLs can be setup by overriding the "webApiUrls()" function:

---

[24] [show entire class in appendix]

```
override class func webApiUrls() -> WebApiManager {

    return WebApiManager().setupUrlsForREST("Users")
}
```

The Web Api Manager takes the domain from the global defaults which is setup inside the AppDelegate, unless overridden.

```
WebApiDefaults.sharedInstance().domain = "http://topik.ustwo.com"
```

The "webApiRFestObjectID" function is fired whenever the Web Api manager needs the object's unique identifier. It should be setup to return whichever value should identify the object. This example returns the property "id" from the User class:

```
override func webApiRestObjectID() -> Int? {

    return id
}
```

The function "setupUrlsForREST" with the argument "Users" will generate these URL's when called from the following swift methods:

Swift method	Action	Method	URL
User.webApiUrls().getMultipleUrl()	Get multiple	GET	http://topik.ustwo.com/Users
User.webApiUrls().getUrl(5)	Get	GET	http://topik.ustwo.com/Users/5
User.webApiUrls().insertUrl()	Insert	POST	http://topik.ustwo.com/Users
User.webApiUrls().updateUrl(9)	Update	PUT	http://topik.ustwo.com/Users/9
User.webApiUrls().deleteUrl(122)	Delete	DELETE	http://topik.ustwo.com/Users/122

These URL's can be accessed to send your own requests, or you can use the in-built CRUD methods on the CompresJSONObject class. You can change the values via the swift objects as you would normally, and then all the "compresJSONWebApiUpdate" method to send the data to the Web API. This object returns the CompresJSONRequest so you can chain the code handlers in the same way as a normal CompresJsonRequest.

```
item.ItemText = textFields[0].text
item.fontID = textFields[1].text.toInt()!
item.CardDesignID = textFields[2].text.toInt()!

var request: CompresJsonRequest?

if item.CardDesignItemID == 0 {

    request = item.webApiInsert()
}
else {

    request = item.webApiUpdate()
}

request?.onDownloadSuccess({ (json, request) -> () in

    self.item = CardDesignItem.createObjectFromJson(json)
    self.navigationController?.popViewControllerAnimated(true)

}).onDownloadFailure({ (error, alert) -> () in

    alert.show()
})
```

This example shows how to add some logic and save the request to an instance variable and executing a save or update request depending on the item's ID.

How it works

JSONObject

The JSONObject class is derived from "NSObject" which is the base for all classes in swift. It contains a number of methods to convert to and from JSON and dictionary objects, aswell as handling the Web API CRUD methods.

Converting from JSON

SwiftyJSON[25] (Dependency)

SwiftyJSON is a library which is used to handle extraction of data from a JSON response. Any JSON string can be converted into a "SwiftyJSON" object using the class "JSON". It contains the

---

[25] (SwiftyJSON, n.d.)

information inside an instance of this class and has some advantages over using a normal String/AnyObject dictionary.

It tackles the issues defined in the problem statement with mapping JSON data to a dictionaries, and reading its values. SwiftyJSON allows you to safely read the values without throwing a crash if something is missing. This behaviour is optional so you can decide if you would like a crash or not if keys are accessed without values present in the JSON string.

You can bypass serialization code shown in the example (See Redundant code in common tasks) with:

```
let json = JSON(data: dataFromNetworking)
if let userName = json[0]["user"]["name"].string{
  //Now you got your value
}
```

If you use .string to access the value, the program will crash if it does not find a value there. In the example above that problem is tackled by wrapping it around an "if let[26]" statement. You can replace this with .stringValue to return an empty string in the same scenario, also preventing a crash:

```
let name = json["name"].stringValue
```

Mapping

SwiftyJSON is used at this stage, only to convert a JSON string (or NSData) into a useable String/AnyObject dictionary.

```
public class func createObjectFromJson< T : JSONObject >(json:JSON) -> T {
```

This method has a parameter "T" for the type, which is part of a generic function[27]. The parameter is parsed indirectly, by getting the type from whichever class it was fired from.

E.g. If you invoke it from the class "User", T will be of type "User". This allows you to gain information about the class (if fired from a derived class) from the base class. This extra information

---

[26] (Apple, n.d.) Optional Chaining
[27] (Apple, n.d.) Generics

includes access to the properties which are used in the object mapping. This means the same code can set the values of whatever properties are included in the derived class.

The JSONObject class method "createObjectFromJson" expects a JSON object as the parameter, converts it to a dictionary, and fires the "createObjectFromDict" method.

```
public class func createObjectFromDict< T : JSONObject >(dict: Dictionary<String, AnyObject?>) -> T {

    var obj = T()
    obj.setPropertiesFromDictionary(dict)
    return obj
}
```

createObjectFromDict creates and returns an instance of the class after firing "setPropertiesFromDictionary", which handles the mapping to the object.

```
func setPropertiesFromDictionary(dict: Dictionary<String, AnyObject?>){
```

The first thing this method does is fire the delegate method: "registerClassesForJsonMapping". This fires the method defined in the Swift class (if overridden), to get all the information needed to start mapping.

```
jsonMappingDelegate?.registerClassesForJsonMapping?()
```

As the method is aware of the class type, it can access the class properties. It then loops through "keysWithTypes", a method which returns an array of objects containing the each property as a string value, as well as its type description.

```
for k in keysWithTypes() {
```

With this information it's able to use the jsonKey to access the dictionary it received from the JSON, and retrieve the correct value. If the jsonKey is not specifically set, it is simply the same as the propertyKey.

```
if let dictionaryValue: AnyObject? = dict[jsonKey]{
```

The function "setValue", a built in NSObject method, can then be fired using the propertyKey. It allows you set values to a property via a string representation of the property.

```
self.setValue(dictionaryValue, forKey: propertyKey)
```

It also does some processing on the value if necessary. For example, if you specified in your class that it expects a date for that jsonKey, it will convert the string into an NSDate object using your defined date format, before firing "setValue".

```
var date = NSDate.dateFromString(dict[propertyKey]! as! String, format: JSONMappingDefaults.sharedInstance().dateFormat)
self.setValue(date, forKey: propertyKey)
```

Converting to JSON

It can also reverse this process. This is needed when you want to convert a Swift object into key/value pairs to send as parameters to a Web API for inserting or updating information.

```
for key in self.keys() {
```

It loops through all the properties names via the "keys" method and extracts the value from the object using "valueForKey" which is a built-in NSObject method. It sets the value to the return value "dict" using the defined jsonKey. This means that the parameter used will be the same for posting data as it is for reading it.

```
dict[jsonKey] = self.valueForKey(key)
```

JsonRequest

JsonRequest is a class I made to function as an interface for an HTTP request. It allows you to post and retrieve data easily to and from a server and automatically map the response into usable JSON objects. The request itself is "modular" allowing the actual HTTP request code to be replaced if necessary with no changes required to the business logic within any apps. This was originally created using NSURLRequest (a built in request object) but replaced with Alamofire, a third party framework providing more functionality. JsonRequest is designed to utilize the functionality of third party libraries while keeping the markup consistent.

Alamofire[28]

At the moment, Alamofire handles the actual HTTP request. It is a very powerful library which can handle HTTP requests as well as file upload/downloads etc. It makes it easy to send a request, specifying the HTTPMethod via an enum, and handle the response data and error messages.

```
Alamofire.request(.GET, "http://httpbin.org/get")
        .response { (request, response, data, error) in
                println(request)
                println(response)
                println(error)
            }
```

Method chaining

When you make a request, you usually need to add code handlers for a successful request, a failed request and for when it finished. You would often add a response parameter to the function, in which you pass a "closure". A closure is a piece of code which you pass in, and can execute from inside that function.

The closure would provide you with a space to handle response data and error objects, requiring you to manually check if the error was nil, before continuing.

```
if let err = response.error {
    println("error: \(err.localizedDescription)")
    return //also notify app of failure as needed
}
if let res = response.responseObject {
    println("response: \(res)")
}
```

You could get around this by having multiple closures for success and for error, and finished. This starts to make the code look untidy, as you don't always need to access all three closures. I tackled this by implementing "method chaining". This allows you to subscribe to as many events as possible by chaining response handlers:

---

[28] (Alamofire, n.d.)

To implement this, the methods "onDownloadSuccess", "onDownloadFinished" and "onDownloadFailed" methods all return "Self" (the JsonRequest instance), and save the closures code in an array, inside the class:

```
public func onDownloadSuccess(success: (json: JSON, request: JsonRequest) -> ()) -> Self {
    self.succeedDownloadClosures.append(success)
    return self
}
```

When the request returns successfully, it loops through the array of closures, and executes them.

```
func succeedDownload(json: JSON) {

    for closure in self.succeedDownloadClosures {

        closure(json: json, request: self)
    }
}
```

The following example demonstrates how to parse parameters into the request, specifying the HTTP method and adding two response handlers:

```
var params = [
    "CustomerID" : 5,
    "Address1" : "10 High Street",
    "EMail" : "sebra54@gmail.com",
    "Discount" : 100,
    "CurrencyCode" : "GBP",
    "CustomerName" : "Sebastian",
]

JsonRequest.create("http://mysite.com/sample/api/Customers/5", parameters: params, method: .PUT)

.onDownloadSuccess { (json, request) -> () in

    println("request successful")

}.onDownloadFinished { () -> () in

    println("request finished")
}
```

CompresJsonRequest

This class inherits from "JsonRequest", but has some of the methods overridden in order to extend the functionality. When the HTTP request is executed in a "CompresJsonRequest", it overrides the

passed parameters, serializes them into a JSON string and processes that string with the necessary encryption & compression.

Inheriting from the base class complies with the DRY principle by preventing the need to re-write redundant code with the use of a base class. It is however also complying with DAMP as the overridden properties on the CompresJsonRequest class do contain some very similar code to the base class. The base class function is not trying to cater for both scenarios.

It then sends just one parameter "data" in the actual request, and embeds the processed JSON string inside here. This is now in a format that the backend will understand.

```
if let params = self.parameters {

    var err: NSError?
    var json: String = NSJSONSerialization.dataWithJSONObject(params, options: nil, error: &err)!.toString()

    json = CompresJSON.encryptAndCompressAsNecessary(json)

    self.parameters = Dictionary<String, AnyObject>()
    self.parameters!["data"] = json
}
```

It also adds some logic to the request before it fires the callbacks, to decrypt/decompress the data:

```
let json = JSON(data: data! as! NSData)

let encryptedJson = json["data"].stringValue
let unencryptedJson = CompresJSON.decryptAndDecompressAsNecessary(encryptedJson)

if let dataFromString = unencryptedJson.dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: false){

    let unpackedJson = JSON(data: dataFromString)

    self.succeedDownload(unpackedJson)
}
```

It reads from the "data" property of the response, processes the data inside that string, and converts that string value into a JSON object before firing the callback.

You can also access the underlying Alamofire request via the property: "alamofireRequest" on a JsonRequest, and continue the method chain:

```
}).onDownloadFailure { (error, alert) -> () in

    alert.show()
    println(error)

}.alamofireRequest?.responseJSON(options: .allZeros, completionHandler: { (request, response, js

    println("Request URL: \(request.URL!)")
    println("StatusCode: \(response!.statusCode)")
    println(json)
})
```

By accessing the Alamofire request object directly, you are able to inspect the actual values and URL's being used in the HTTP request, including status code, headers, parameters, data size etc. This allows you to utilize the functionality of the third party library.

E.g. the above code outputs the following to the console:

```
Request URL: http://alex.bechmann.co.uk/compresjson/apih/
VTJGc2RHVmtYMS9vQ2ZMcTR5S01rWXNBN2FJVGFOTTVua0FxMUVLMFBDST0=
StatusCode: 200
Optional({
    data = "U2FsdGVkX1/HnEwCjfT/
4I4FJkhSiD3n0MJ6LmsUaQnGDcN9qJhTy8KBiTVo5ap4KeLdD1yS7r2/
wwC8gtm2ETkO4MAlzdq7QpPFAYcwRfkgWa661OQFrD3D36KbaeHIbDnvYkxCz7S
y282gj21FblZnAgGPIsoLgJN+MaJPSLvb0/yfaEMO7awrA/pSsVcf3nCML
+02CgNwwppQo/
0F8cLi03zVWoFghFhVBn3VYkD9C37rBWJ4+jsDXgo7oWCNnhDQe7aJz4wYcnx5+
i4dL+eTxAn0zjj8nmB7POoD
```

All Output ◇

This shows you that it is accessing the Web API using an encrypted route and the raw data string from the response.

CompresJSONObject

This is derived from JSONObject and its purpose is to extend the functionality in order to communicate with the Web API while utilizing the compression and encryption methods from the CompresJsonRequest class. It overrides all the "factory" methods and Web API CRUD methods. It does this to ensure that those methods fire a "CompresJsonRequest", instead of a "JsonRequest". By using a CompresJsonRequest, everything stays the same except that it transports the data in the compressed/encrypted state.

Due to the fact that a CompresJSONRequest returns the unpacked JSON, you can easily switch between using a JSONObject and a compresJSONObject simply by changing which class your object inherits from. This is possible because they both take in the same parameters, and return the same values.

Web API Manager

Any object which inherits from JSONObject also has a static Web API Manager available, as long as it implements the required delegate methods. It is static property because the URL's should be accessible without an instance of the class e.g. for GET requests. (Described in usage). All the URL's conform to the REST Web API standards.

Using the object class, you can create a REST URL, parsing the ID if necessary:

```
Customer.webApiUrls().updateUrl(5)
```

The delegate method "webApiUrls()" returns whatever you setup inside your class definition…

```
override class func webApiUrls() -> WebApiManager {

    return WebApiManager().setupUrlsForREST("Customers")
}
```

… and creates the appropriate URL:

```
return validRestUrlSet() ? "\(getDomain())/api/\(restKey!)/\(id)" : nil
```

e.g. http://alex.bechmann.co.uk/compresjson/api/Customers/5

At the moment, "api" or "apih" in the URL is hard coded, assuming that this library is communicating with a .NET project with the CompresJSON C# library installed. This behaviour is temporary and will be replaced with more customization options for other API's.

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.

```
public static string EncryptionKey = "7e4bac048ef766e83f0ec8c079e1f90c2eb690a9";

public static bool ShouldCompress = true;
public static bool ShouldEncrypt = true;
```

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.

```
let settings = CompresJSON.sharedInstance().settings

settings.encryptionKey = "7e4bac048ef766e83f0ec8c079e1f90c2eb690a9"
settings.shouldCompress = true
settings.shouldEncrypt = true
settings.encryptUrlComponents = true
```

As the frontend isn't stateless like a web 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 can only be set to true if the routes have been setup in the "global.asax" file.

Analysis

The library is fully functional, although not completely optimized yet. It allows for the transportation of secure and compressed data between a client and server. To fully evaluate the library, some testing has been done.

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. The data is extracted using Entity Framework and accessible from the internet via "scaffolded" Web API 2 controllers. The only modifications to the controllers is to force a JSON response. This is necessary as the library does not support other data formats yet. As little as possible changes were made to anything else in the project, to help demonstrate its applicability to standard MVC applications.

Speed and data size performance

For the final testing, I ran the speed tests available at: http://alex.bechmann.co.uk/compresjson.

It runs a series of requests 20 times, recording the average response time as well as the data size. The requests progressively larger, and the purpose is to find out whether or not there is a benefit from the compression and encryption processing.

It compares the processed response with a normal JSON response. It also tests compression only, encryption only, and both together to determine where if at all, it bottlenecks. The test was done three times, for the different implemented compression algorithms GZip, Deflate and LZString.

As only GZip and LZString were successful in decompressing on the client, I am going to focus on the results for those in particular.

GZip (Working in browser)

The following graph shows the number of objects against the data size of a response, using different combinations of encryption and compression:

Response data size (kb)

Number of objects in response	1	2	5	10	20	40	80	160	240	480
Normal	0.17	0.33	0.79	1.59	3.33	6.64	13.4	26.49	39.54	80.29
Compressed (GZip)	0.16	0.19	0.23	0.31	0.55	0.79	1.55	2.67	3.57	7.14
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49	8.91	17.91	35.37	52.76	107.1
GZip + AES	0.24	0.28	0.32	0.45	0.76	1.07	2.1	3.6	4.78	9.55
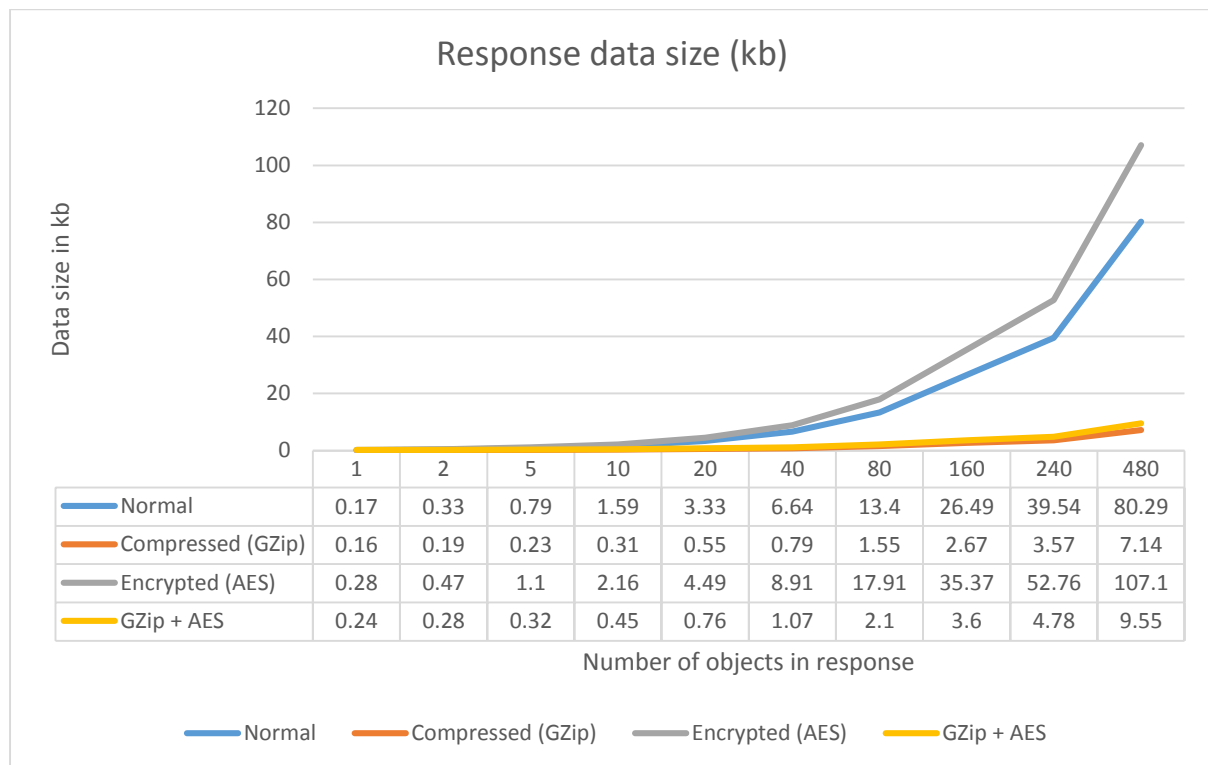
This library, with GZip compression was the more effective of the two working implementations. The data was size was successfully reduced after a request of only 1 object (compression only) and two obects (both), returned as part of a processed JSON object array. The benefits become much more apparent, with a larger response.

This allows you to get substantially more information from the same volume of data as shown from the graph. The encryption was also very effective, adding very little overhead to the raw compression.

A normal request for 80 objects, which might be used to populate a TableView, used 13.4kb. Using the CompresJSON processing, you could download 480 objects using only 12.7kb (at no cost in speed). This was smaller, and also benefitted from the extra security measures of the AES encryption, which are very promising results!

The following graph shows the number of objects against the response time, using different combinations of encryption and compression:

	1	2	5	10	20	40	80	160	240	480
Normal	0.24	0.25	0.21	0.18	0.21	0.28	0.32	0.66	0.62	0.99
Compressed (GZip)	0.14	0.16	0.14	0.15	0.16	0.16	0.15	0.17	0.17	0.29
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26	0.35	0.46	0.59	0.64	1.43
GZip + AES	0.14	0.15	0.2	0.14	0.13	0.14	0.18	0.25	0.21	0.28

The response time graph shows very similar correlations. The graph shows less of a smooth curve, because measuring response time can be influenced by a number of hard-to-measure things like network traffic and server compiling etc.

The data shows that requests for up to 20 or so items are very similar, even when the data size is shown to be less. I suggest that this is due to the extra processing required on the server to the data. Similarly to the data size graph, the benefits become stronger with the larger requests. Measurable benefits seem to kick in at a between 10 and 20 objects.

LZString (Working on iOS device)

The following graph shows the number of objects against the data size of a response, using different combinations of encryption and compression:

Response data size (kb)

	1	2	5	10	20	40	80	160	240	480
Normal	0.17	0.33	0.79	1.59	3.33	6.64	13.4	26.49	39.54	80.29
Compressed (LZString)	0.25	0.36	0.63	1.07	2	3.25	5.6	9.01	12	21.32
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49	8.91	17.91	35.37	52.76	107.1
LZString + AES	0.37	0.51	0.87	1.45	2.7	4.37	7.49	12.05	16.03	28.45

Number of objects in response
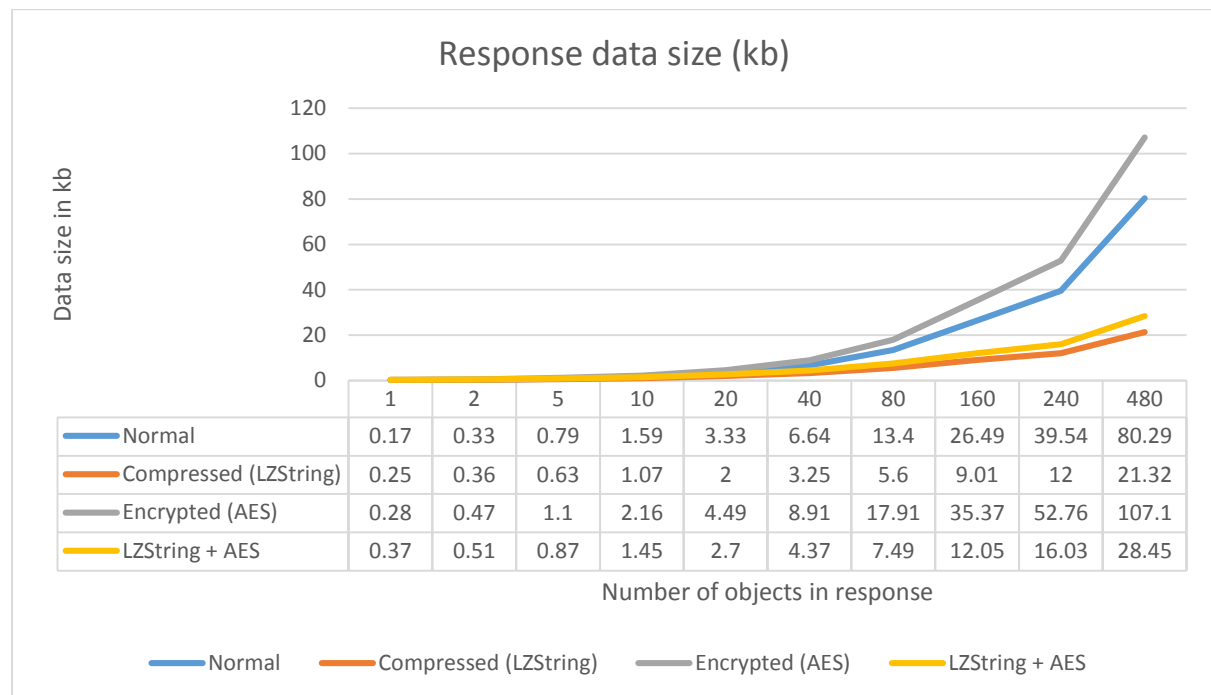
The compression from LZString is not as effective as GZip or Deflate. The compressed data is between two and three times larger than the other algorithms. The benefits, while using LZString are noticeable at 10 objects, compared to 2 with GZip.

The following graph shows the number of objects against the response time, using different combinations of encryption and compression:
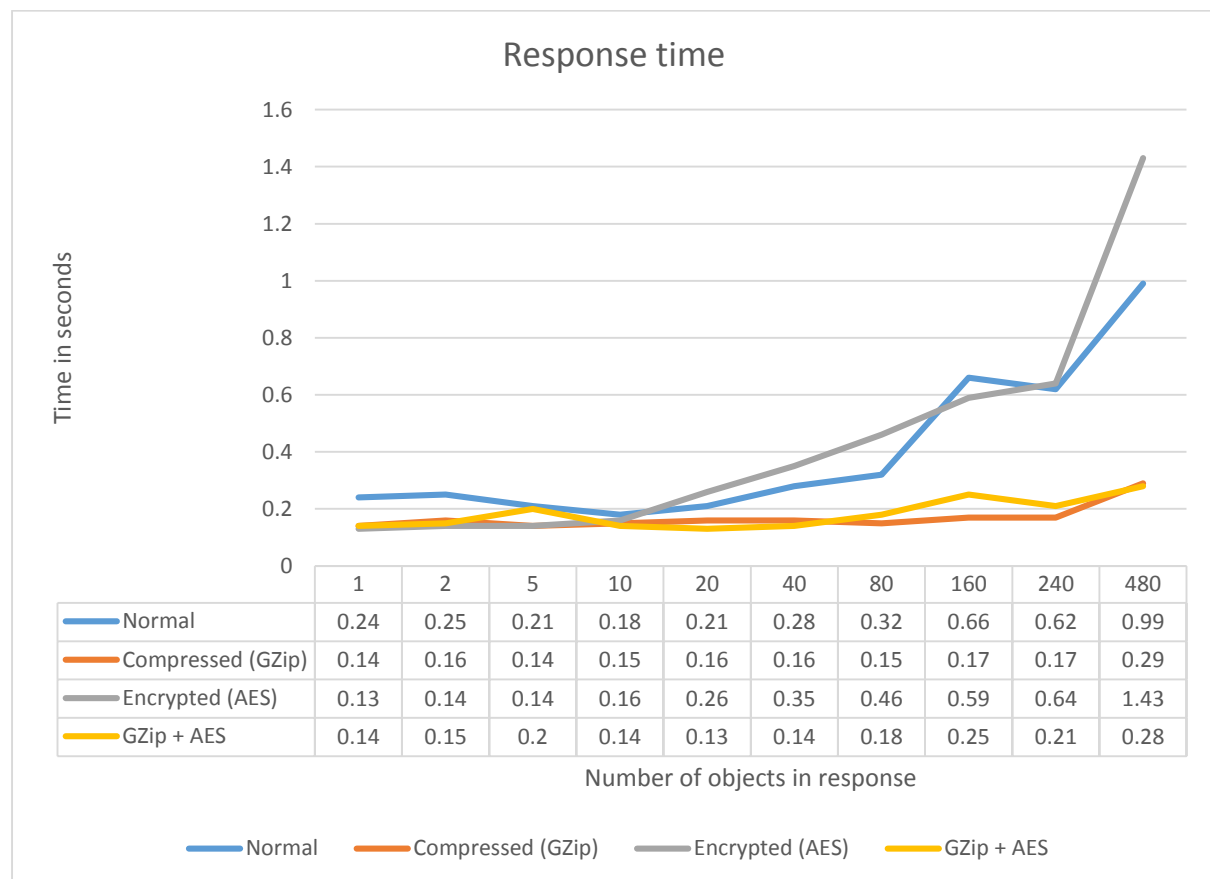
Response time

Time in seconds

	1	2	5	10	20	40	80	160	240	480
Normal	0.24	0.25	0.21	0.18	0.21	0.28	0.32	0.66	0.62	0.99
Compressed (LZString)	0.27	0.35	0.42	0.52	0.79	1.2	1.94	3.27	4.52	8.3
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26	0.35	0.46	0.59	0.64	1.43
LZString + AES	0.38	0.39	0.42	0.53	0.83	1.28	1.98	3.64	4.92	10.27

Number of objects in response

Normal   Compressed (LZString)   Encrypted (AES)   LZString + AES
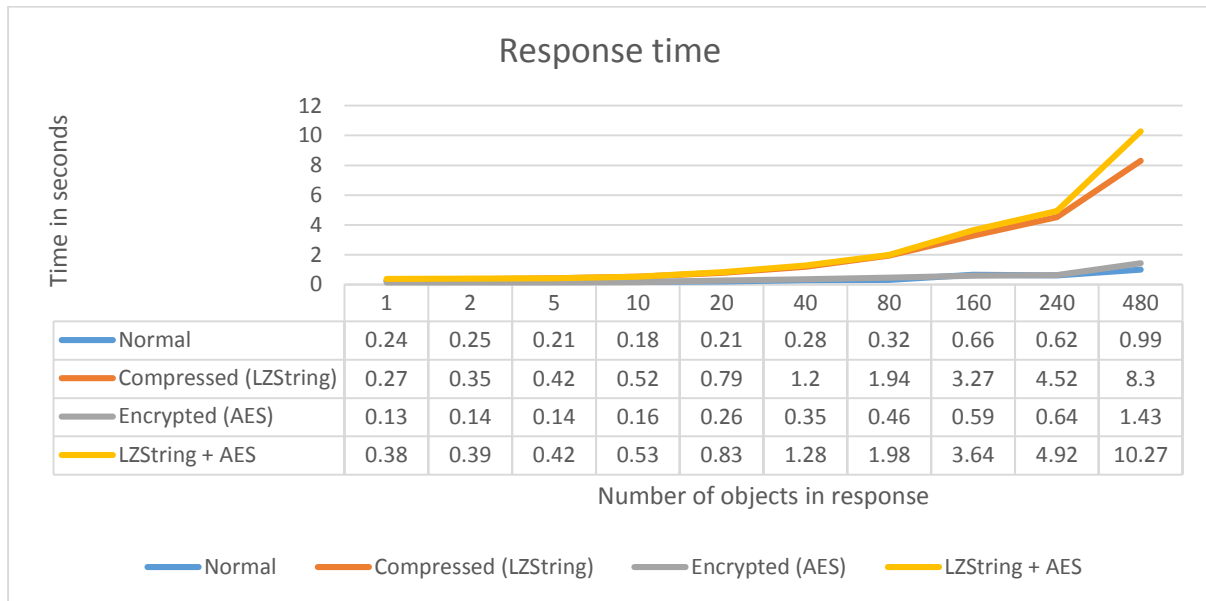
A big performance hit is also taken on the server with LZString. I think this is due slow performance of JINT (Running JavaScript on backend) dependency. The processing time required to compress the data is more than the speed benefit achieved from the reduced data size. While the time is still usable on the smaller requests, the larger ones start to feel very sluggish.

Discussion

Library installation

While drag and drop is a very temporary library installation technique, it is valid and is used by many people. Some people even prefer it as they know the code won't get updated, requiring them to maintain their own code. Although not implemented, it was planned to package it into a class library and could be imported by adding a reference to the .dll from another project.

The CocoaPod setup was very successful, and it was successfully integrated into a test project with all its dependencies and compiled with no problems at all.

Use of coding principles

I am satisfied that both libraries are written in accordance with the coding principles. They provide functionality with as little as possible code for a developer, in accordance with DRY. Method names are descriptive, following naming conventions for both C# and Swift. They can be applied to existing projects with minimal changes required.

There is also a clear separation of concerns, as both libraries are very modular, allowing new dependencies such as compression and encryption algorithms to be changed easily, and do not interfere with the separation of concerns within a MVC or Swift application either with all processing logic hidden away inside action filters.

I am satisfied that they are intuitive and that a developer will be able to adopt it without too much hassle because of the accordance to accepted coding principles and established library installation methods.

Encryption

The way the encryption is applied is effective in some areas, but does have its limitations. The HTTP body is encrypted, but the headers are left open. This could cause confusion if a developer doesn't realise this, because HTTP headers are often used for authentication tokens which could be problematic if they are intercepted. Also while the URL is partly masked, it doesn't mask the domain of the server which adds some context to a request for a potential hacker.

The AES encryption key is also hard coded into the code on both the server, and in the iOS device. As it uses a symmetric key, just hacking one device, or server increases the possibility to decrypt all past

data, had it been sniffed and recorded for example. Ultimately the security is lessened unless the encryption is rotated as often as possible.

REST

While the API follows REST conventions, there are some aspects which are not accounted for. The URL generation and use of HTTP verbs follows the convention for all CRUD functionality except PATCH, which is not quite available as part of the JSON Object and Web API Manager, but it can be be done manually via a JsonRequest or CompresJsonRequest if the necessary actions are in place on the backend.

To be truly RESTful it should allow transfer of many types of multimedia, including files, images and XML data etc. At the moment, the library is limited to modifying JSON data. It also isn't completely crash safe on the backend, meaning PARSE errors will crash the request, and you won't receive the correct status code in the response.

As authentication handling is not handled by the library, the burden falls on a developer to implement this themselves. It is possible to add extra action filters and shouldn't affect any of the functionality.

Compromises

I had numerous problems finding libraries for both compression and encryption which would produce the same results on both the server (C#) and the client (Swift/Objective-C). It was not feasible to create my own implementation of the algorithms due to a lack of time. This was an issue because I was not able to gain enough control of the libraries I found, to match their settings and they were producing different results, making communication impossible.

From my research, I found that the most effective compression was using DEFLATE. However I was not able to find a library for both server and client side which produced the same results. So I had to use a compromise, in the form of a GZIP implementation and a library called LZString. LZString is pure JavaScript and is able to run on both the server and client using JavaScript.

LZString is quite a slow implementation due to the dependency on JINT (to run JavaScript in .NET). I was able to get GZip partially implemented in the browser only and was able to decompress the data from the server. GZip is very slightly larger than Deflate but by an extremely small margin (roughly 2 bytes).

Data Format

JSON was chosen as the format for sending data. It was used to send the original data, as well as sending the processed data from server to front end. As the data in its processed state, is not human readable, it doesn't make sense to use a human-readable format like JSON. Sending the data as a string in this way also meant relying on Base64 encoding (adding to the data size). The raw transportation of the processed data could have been done in another way, e.g. An array of bytes for example.

CompresJSON vs HTTPS

CompresJSON is designed to be an alternative to HTTPS, preferring ease of installation vs optimal security. It offers quite good security, effective compression and restores much of the behaviour you would expect from non-encrypted data transfer.

However, it restores server side behaviour by manually attempting to emulate it from inside the action filters, e.g. mapping form data to usable objects. This behaviour is not as effective as the built in MVC object mapping which you would expect from vanilla MVC and will not be improved with future updates.

Also even though the mark-up is minimal, HTTPS requires no extra code and is much less likely to affect any of your backend API logic at all, in any situation.
HTTPS can be combined with compressing data using Content-Encoding, which is a more effective, and widely accepted standard. This makes it much easier to decode the data on the frontend, as a swift NSURLConnection has built in functionality to decompress the data.

As the HTTPS wraps around the request data, it also complies with the correct ordering of compression and encryption for maximum efficiency, tested earlier in this project.

While HTTPS has its drawbacks, in some cases it might be worth the investment for any medium sized app. However I think that this library still has its uses for smaller apps where heavy security is not a necessity. Due to the way the library is built, it doesn't require a backend with this library installed. The current compression and encryption is just a layer on top of a functioning library meaning all of the other benefits including object mapping and response handling can still be used with a future HTTPS backend. This means you could use the CompresJSON format in the beginning of a project and upgrade to a secure connection at a later stage, with few code changes.

Conclusion

From working on this project I have devised a set of good practice rules which I recommend should be followed in any project, regardless of whether or not this library is used.

- Don't send unnecessary fields in your data response
- Use techniques reduce what data is transported e.g Removing redundant key data
- Cache static data
- Utilize deflate compression
- Use some form of encryption
- De-serialize JSON data into local object representations

The problem statement was:

Are there viable optimizations to be made on the transportation of JSON data in terms of reducing the data size, and can it be combined with adding security to provide an easy, low cost way to improve the way data is sent between a web server and client?

In response to the problem statement, although it is difficult to say which methods are the best, there are definitely optimizations to be made in the transportation of data, and wouldn't necessarily only involve JSON. Reducing the data size through compression is be very effective, and can definitely be combined with security.

As with anything, low cost solutions are generally of a lesser quality, but as far as that is an acceptable compromise this library can address these issues.

Perspective

Future plans for this include:

- **Furthering the research**: Time constraints resulted in some compromises having to be made, and some research not completed. If I was to continue this project I would invest some more time on some of the research into JSON optimizations and encryption algorithms as well as other data formats to see if there were some more appropriate methods available.
- **Supporting other languages**: Ports could be made for other back and front-end languages to allow this to be used in other places such as Android, windows phone or other applications.
- **Unit tests**: Unit tests should be in place to allow any changes to the algorithms or business logic to be stress tested quickly and effectively.

References

(GitHub), p., n.d. *LZString.* [Online]
Available at: http://pieroxy.net/blog/pages/lz-string/index.html
[Accessed May 2015].

Alamofire, n.d. *Alamofire.* [Online]
Available at: https://github.com/Alamofire/Alamofire
[Accessed May 2015].

Anon., n.d. *Blogspot.* [Online]
Available at: http://web-resource-optimization.blogspot.dk/2011/06/json-compression-algorithms.html
[Accessed 19 May 2015].

Gilbertson, S., 2011. *arstechnica.* [Online]
Available at: http://arstechnica.com/business/2011/03/https-is-more-secure-so-why-isnt-the-web-using-it/
[Accessed 15 May 2015].

Microsoft, 2015. *Introduction to REST and .net Web API.* [Online]
Available at: http://blogs.msdn.com/b/martinkearn/archive/2015/01/05/introduction-to-rest-and-net-web-api.aspx
[Accessed 15 May 2015].

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].

Taylor, S., n.d. *Three Principles of Coding Clarity.* [Online]
Available at: http://archive.vector.org.uk/art10009750
[Accessed 12 May 2015].

Triana, M., 2013. *Using HTTP Verbs correctly in your REST Web API.* [Online]
Available at: http://micheltriana.com/2013/09/30/http-verbs-in-a-rest-web-api/
[Accessed May 2015].

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

Wikipedia, n.d. *AES.* [Online]

Available at: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[Accessed May 2015].

Wikipedia, n.d. *GZip.* [Online]

Available at: http://en.wikipedia.org/wiki/Gzip

[Accessed 04 May 2015].

Wikipedia, n.d. *LZ77 and LZ78.* [Online]

Available at: http://en.wikipedia.org/wiki/LZ77_and_LZ78

[Accessed May 2015].

Youse, M., n.d. *AES.* [Online]

Available at: http://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard

[Accessed May 2015].

Appendix

User class

Inherits from JSONObject

```swift
import UIKit

class User: JSONObject {

    var id = 0
    var Username = ""
    var Email = ""
    var Password = ""

    var DateOfBirth: NSDate = NSDate()
    var Products = Array<Product>()
    var address = Address()


    override func registerClassesForJsonMapping() {

        self.registerKey("id", jsonKey: "UserID")
        self.registerDate("DateOfBirth", format: DateFormat.Date)
        self.registerClass(Product.self, forKey: "Products")
        self.registerClass(Address.self, propertyKey: "address", jsonKey: "Address")
    }


    // MARK: - Web api methods

    override class func webApiUrls() -> WebApiManager {

        return WebApiManager().setupUrlsForREST("Users")
    }

    override func webApiRestObjectID() -> Int? {

        return id
    }
}
```

Deflate (LZ77)

Response time

	1	2	5	10	20	40	80	160	240	480
Normal	0.24	0.25	0.21	0.18	0.21	0.28	0.32	0.66	0.62	0.99
Compressed (Deflate)	0.26	0.2	0.21	0.18	0.19	0.2	0.19	0.22	0.27	0.33
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26	0.35	0.46	0.59	0.64	1.43
Deflate + AES	0.13	0.14	0.14	0.22	0.14	0.15	0.16	0.18	0.22	0.3

Number of objects in response

Response times

	1	2	5	10	20
Normal	0.24	0.25	0.21	0.18	0.21
Compressed (Deflate)	0.26	0.2	0.21	0.18	0.19
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26
Deflate + AES	0.13	0.14	0.14	0.22	0.14

Number of objects in response

Response data size

	1	2	5	10	20	40	80	160	240	480
Normal	0.17	0.33	0.79	1.59	3.33	6.64	13.4	26.49	39.54	80.29
Compressed (Deflate)	0.14	0.17	0.2	0.29	0.52	0.76	1.53	2.65	3.54	7.12
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49	8.91	17.91	35.37	52.76	107.1
Deflate + AES	0.22	0.26	0.3	0.41	0.72	1.05	2.07	3.57	4.76	9.51

Response data size (kb)

GZip

Response time

	1	2	5	10	20	40	80	160	240	480
Normal	0.24	0.25	0.21	0.18	0.21	0.28	0.32	0.66	0.62	0.99
Compressed (GZip)	0.14	0.16	0.14	0.15	0.16	0.16	0.15	0.17	0.17	0.29
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26	0.35	0.46	0.59	0.64	1.43
GZip + AES	0.14	0.15	0.2	0.14	0.13	0.14	0.18	0.25	0.21	0.28

Number of objects in response

Response time

	1	2	5	10	20
Normal	0.24	0.25	0.21	0.18	0.21
Compressed (GZip)	0.14	0.16	0.14	0.15	0.16
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26
GZip + AES	0.14	0.15	0.2	0.14	0.13

Number of objects in response

75

Response data size (kb)

	1	2	5	10	20	40	80	160	240	480
Normal	0.17	0.33	0.79	1.59	3.33	6.64	13.4	26.49	39.54	80.29
Compressed (GZip)	0.16	0.19	0.23	0.31	0.55	0.79	1.55	2.67	3.57	7.14
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49	8.91	17.91	35.37	52.76	107.1
GZip + AES	0.24	0.28	0.32	0.45	0.76	1.07	2.1	3.6	4.78	9.55

Number of objects in response

Response data size (kb)

	1	2	5	10	20
Normal	0.17	0.33	0.79	1.59	3.33
Compressed (GZip)	0.16	0.19	0.23	0.31	0.55
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49
GZip + AES	0.24	0.28	0.32	0.45	0.76

Number of objects in response

76

LZString

Response time

	1	2	5	10	20	40	80	160	240	480
Normal	0.24	0.25	0.21	0.18	0.21	0.28	0.32	0.66	0.62	0.99
Compressed (LZString)	0.27	0.35	0.42	0.52	0.79	1.2	1.94	3.27	4.52	8.3
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26	0.35	0.46	0.59	0.64	1.43
LZString + AES	0.38	0.39	0.42	0.53	0.83	1.28	1.98	3.64	4.92	10.27

Number of objects in response

Response time

	1	2	5	10	20
Normal	0.24	0.25	0.21	0.18	0.21
Compressed (LZString)	0.27	0.35	0.42	0.52	0.79
Encrypted (AES)	0.13	0.14	0.14	0.16	0.26
LZString + AES	0.38	0.39	0.42	0.53	0.83

Number of objects in response

77

Response data size (kb)

	1	2	5	10	20	40	80	160	240	480
Normal	0.17	0.33	0.79	1.59	3.33	6.64	13.4	26.49	39.54	80.29
Compressed (LZString)	0.25	0.36	0.63	1.07	2	3.25	5.6	9.01	12	21.32
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49	8.91	17.91	35.37	52.76	107.1
LZString + AES	0.37	0.51	0.87	1.45	2.7	4.37	7.49	12.05	16.03	28.45

Number of objects in response

Response data size (kb)

	1	2	5	10	20
Normal	0.17	0.33	0.79	1.59	3.33
Compressed (LZString)	0.25	0.36	0.63	1.07	2
Encrypted (AES)	0.28	0.47	1.1	2.16	4.49
LZString + AES	0.37	0.51	0.87	1.45	2.7

Number of objects in response

78