Designing a Great API

****

Regardless of whether you are experienced in building APIs and microservices and looking for the techniques on how you can apply them with Go or you are completely new to the world of microservices, it is worth spending the time to read this chapter.

Writing an API contract feels part art, part science and, when you discuss your design with other engineers, you will most certainly agree to disagree, not to the level of tabs versus spaces, but there is certainly something personal about API contracts.

In this chapter, we will look at the two most popular options, which are RESTful and RPC. We will examine the semantics of each approach, which will equip you with the knowledge to argue your case when the inevitable discussion (read argument) occurs. Choosing between REST or RPC may be entirely down to your current environment. If you currently have services running that implement a RESTful approach, then I suggest you stick with it, likewise if you now use RPC. One thing I would suggest is that you read the entire chapter to understand the semantics, pros, and cons of each approach.

# RESTful APIs

The term **REST** was suggested by Roy Fielding in his Ph.D. dissertation in the year 2000. It stands for **Representational State Transfer** and is described as:

*"REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security and encapsulate legacy systems."*

Having an API that conforms to the REST principles is what makes it RESTful.

## URIs

One of the main components in the HTTP protocol is a URI. **URI** stands for **Uniform Resource Identifiers** and is the method by which you will access the API. You may be asking what the difference between a URI and a URL (Uniform Resource Locator) is? When I started to write this chapter, I wondered about this myself and did what any self- respecting developer would do, which is to head over to Stack Overflow. Unfortunately, my confusion only grew as there were lots of detailed answers, none of which I found particularly enlightening. Time to head over to the inner circle of hell also known as W3C standards to look up the RFC for the official answer.

In short, there is no difference, a URL is a URI that identifies a resource by its network location, and it is acceptable to interchange the terms when describing a resource entirely.

The clarification document published back in 2001

(<http://www.w3.org/TR/uri-clarification>) goes on to explain that in the early to mid-90s there was an assumption that an identifier is cast into one or two classes. An identifier might specify the location of a resource (URL) or its name (Uniform Resource Name URN) independent of location. A URI could either be a URL or a URN. Using this example, http:// would be a URL scheme and isbn: a URN scheme. However, this changed over time and the importance of the additional level of hierarchy lessened. The view changed that an individual scheme does not need to be cast into one of a discrete set of types.

The conventional approach is that http: is a URI scheme and urn: is also a URI scheme. URNs take the form urn:isbn:n-nn-nnnnnn-n, isbn: is a URN namespace identifier, not a URN scheme or a URI scheme.

Following this view, the term URL does not refer to a formal partition of URI space rather, URL is an informal concept; a URL is a type of URI that identifies a resource via its network location.

For the rest of this book, we will use the term URI and when we do we will be talking about a method to access a resource that is running on a remote server.

## URI format

RFC 3986, which was published in 2005 <https://www.ietf.org/rfc/rfc3986.txt>, defines the format that makes valid URIs:

URI = scheme "://" authority "/" path [ "?" query] ["#" fragment"] URI = [http://myserver.com/mypath?query=1#document](http://myserver.com/mypath?query=1&amp;document)

We are will use the path element in order to locate an endpoint that is running on our server. In a REST endpoint, this can contain parameters as well as a document location. The query string is equally important, as you will use this to pass parameters such as page number or ordering to control the data that is returned.

Some general rules for URI formatting:

A forward slash / is used to indicate a hierarchical relationship between resources

A trailing forward slash / should not be included in URIs Hyphens - should be used to improve readability Underscores \_ should not be used in URIs

Lowercase letters are preferred as case sensitivity is a differentiator in the path

part of a URI

The concept behind many of the rules is that a URI should be easy to read and to construct. It should also be consistent in the way that it is built so you should follow the same taxonomy for all the endpoints in your API.

## URI path design for REST services

Paths are broken into documents, collections, stores, and controllers.

### Collections

A collection is a directory of resources typically broken by parameters to access an individual document. For example:

GET /cats -> All cats in the collection GET /cats/1 -> Single document for a cat 1

When defining a collection, we should always use a plural noun such as cats or people for the collection name.

### Documents

A document is a resource pointing to a single object, similar to a row in a database. It has the ability to have child resources that may be both sub-documents or collections. For example:

GET /cats/1 -> Single document for cat 1

GET /cats/1/kittens -> All kittens belonging to cat 1 GET /cats/1/kittens/1 -> Kitten 1 for cat 1

### Controller

A controller resource is like a procedure, this is typically used when a resource cannot be mapped to standard **CRUD** (**create**, **retrieve**, **update**, and **delete**) functions.

The names for controllers appear as the last segment in a URI path with no child resources. If the controller requires parameters, these would typically be included in the query string:

POST /cats/1/feed -> Feed cat 1

POST /cats/1/feed?food=fish ->Feed cat 1 a fish

When defining a controller name we should always use a verb. A verb is a word that indicates an action or a state of being, such as feed or send.

### Store

A store is a client-managed resource repository, it allows the client to add, retrieve, and delete resources. Unlike a collection, a store will never generate a new URI it will use the one specified by the client. Take a look at the following example that would add a new cat to our store:

PUT /cats/2

This would add a new cat to the store with an ID of 2, if we had posted the new cat omitting the ID to a collection the response would need to include a reference to the newly defined document so we could later interact with it. Like controllers we should use a plural noun for store names.

### CRUD function names

When designing great REST URIs we never use a CRUD function name as part of the URI, instead we use a HTTP verb. For example:

DELETE /cats/1234

We do not include the verb in the name of the method as this is specified by the HTTP verb, the following URIs would be considered an anti-pattern:

GET /deleteCat/1234 DELETE /deleteCat/1234 POST /cats/1234/delete

When we look at HTTP verbs in the next section this will make more sense.

## HTTP verbs

The commonly used HTTP verbs are:

GET POST PUT PATCH DELETE HEAD OPTIONS

Each of these methods has a well-defined semantic within the context of our REST API and the correct implementation will help your user understand your intention.

### GET

The GET method is used to retrieve a resource and should never be used to mutate an operation, such as updating a record. Typically, a body is not passed with a GET request; however, it is not an invalid HTTP request to do so.

##### Request:

GET /v1/cats HTTP/1.1

##### Response:

HTTP/1.1 200 OK

Content-Type: application/json Content-Length: xxxx

{"name": "Fat Freddie's Cat", "weight": 15}

### POST

The POST method is used to create a new resource in a collection or to execute a controller. It is typically a non-idempotent action, in that multiple posts to create an element in a collection that will create multiple elements not updated after the first call.

The POST method is always used when calling controllers as the actions of this is considered non-idempotent.

##### Request:

POST /v1/cats HTTP/1.1

Content-Type: application/json Content-Length: xxxx

{"name": "Felix", "weight": 5}

##### Response:

HTTP/1.1 201 Created

Content-Type: application/json Content-Length: 0

Location: /v1/cats/12343

### PUT

The PUT method is used to update a mutable resource and must always include the resource locator. The PUT method calls are also idempotent in that multiple requests will not mutate the resource to a different state than the first call.

##### Request:

PUT /v1/cats HTTP/1.1

Content-Type: application/json Content-Length: xxxx

{"name": "Thomas", "weight": 7 }

##### Response:

HTTP/1.1 201 Created

Content-Type: application/json Content-Length: 0

### PATCH

The PATCH verb is used to perform a partial update, for example, if we only wanted to update the name of our cat we could make a PATCH request only containing the details that we would like to change.

##### Request:

PATCH /v1/cats/12343 HTTP/1.1 Content-Type: application/json Content-Length: xxxx

{"weight": 9}

##### Response:

HTTP/1.1 204 No Body

Content-Type: application/json Content-Length: 0

In my experience PATCH updates are rarely used, the general convention is to use a PUT and to update the whole object, this not only makes the code easier to write but also makes an API which is simpler to understand.

### DELETE

The DELETE verb is used when we want to remove a resource, generally we would pass the ID of the resource as part of the path rather than in the body of the request. This way, we have a consistent method for updating, deleting, and retrieving a document.

##### Request:

DELETE /v1/cats/12343 HTTP/1.1 Content-Type: application/json Content-Length: 0

##### Response:

HTTP/1.1 204 No Body

Content-Type: application/json Content-Length: 0

### HEAD

A client would use the HEAD verb when they would like to retrieve the headers for a resource without the body. The HEAD verb is typically used in place of a GET verb when a client only wants to check if a resource exists or to read the metadata.

##### Request:

HEAD /v1/cats/12343 HTTP/1.1 Content-Type: application/json Content-Length: 0

##### Response:

HTTP/1.1 200 OK

Content-Type: application/json

Last-Modified: Wed, 25 Feb 2004 22:37:23 GMT

Content-Length: 45

### OPTIONS

The OPTIONS verb is used when a client would like to retrieve the possible interactions for a resource. Typically, the server will return an Allow header, which will include the HTTP verbs that can be used with this resource.

##### Request:

OPTIONS /v1/cats/12343 HTTP/1.1 Content-Length: 0

##### Response:

HTTP/1.1 200 OK

Content-Length: 0 Allow: GET, PUT, DELETE

## URI query design

It is perfectly acceptable to use a query string as part of an API call; however, I would refrain from using this to pass data to the service. Instead the query should be used to perform actions such as:

Paging Filtering Sorting

If we need to make a call to a controller, we discussed earlier that we should use a POST request as this is most likely a non-idempotent request. To pass data to the service, we should include the data inside of the body. However, we could use a query string to filter the action of the controller:

POST /sendStatusUpdateEmail?$group=admin

{

""message": "": "All services are now operational\nPlease accept our apologies for any inconvenience caused.\n

The Kitten API team""

}

In the preceding example, we would send a status update email with the message included in the body of the request, because we are using the group filter passed in the query string we could restrict the action of this controller to only send to the admin group.

If we had added the message to the query string and not passed a message body, then we would potentially be causing two problems for ourselves. The first is that the max length for a URI is 2083 characters. The second is that generally a POST request would always include a request body. Whilst this is not required by the HTTP specification, it would be expected behavior by the majority of your users.

## Response codes

When writing a great API, we should use HTTP status codes to indicate to the client the success or failure of the request. In this chapter, we will not be taking a comprehensive look at all the status codes available; there are many resources on the Internet that have this information. We will provide some sources for further reading, what we will do is look at the status codes that you as a software engineer will want your microservice to return.

Currently, it is a generally held consensus that this is good practice as it allows the client to immediately determine the status of a request without having to dive into the request body to gain further insight. In the instance of a failure and APIs that always return a 200 OK response to the user with a message body containing further information is not good practice as it requires the client to have to inspect the body to determine outcome. It also means that the message body will contain additional information other than the object that it should represent. Consider the following bad practice:

Bad request body:

POST /kittens RESPONSE HTTP 200 OK

{

""status":": 401,

""statusMessage": "": "Bad Request""

}

Successful request:

POST /kittens

RESPONSE HTTP 201 CREATED

{

""status":": 201, ""statusMessage": "": "Created",", ""kitten":": {

""id": "": "1234334dffdf23",",

""name": "": "Fat Freddy'sFreddy's Cat""

}

}

Imagine if you were writing a client for the preceding request, you need to add logic to your application to check the status node in the response before you could read and process the returned kitten.

Now consider something even worse: And even worse failure:

POST /kittens RESPONSE HTTP 200 OK

{

""status":": 400,

""statusMessage": "": "Bad Request""

}

And even worse success:

POST /kittens RESPONSE HTTP 200 OK

{

""id": "": "123434jhjh3433",",

""name": "": "Fat Freddy'sFreddy's Cat""

}

If your API author had done something like the preceding example, you need to check to see if the response that has been returned is an error or the kitten that you were expecting. The number of WTFs per minute you would utter whilst coding a client for this API would not endear you to its author. These might seem like extreme examples, but there are instances like this out in the wild, at some point in my career I'm fairly sure I have been guilty of such a crime, but then I had not read this book.

What the author in their best intention has done is try to take the HTTP status codes too literally. W3C RFC2616 states that the HTTP status code relates to the attempt to understand and satisfy the request

(<https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1.1>); however, this is a little ambiguous when you look at some of the individual status codes. Modern consensus is that it is OK to use HTTP status codes to indicate the processing state of an API request not just the server's ability to process the request. Consider how we could make these requests better by implementing this approach.

A good example of a failure:

POST /kittens

RESPONSE HTTP 400 BAD REQUEST

{

""errorMessage": "": "Name should be between 1 and 256 characters in length and only contain [A-Z] - ['-.]"'-.]"

}

A good example of a success:

POST /kittens

RESPONSE HTTP 201 CREATED

{

""id": "": "123hjhjh2322",",

""name": "": "Fat Freddy'sFreddy's cat""

}

This is far more semantic; the user only ever needs to read the response in the instance of a failure if they require further information. In addition to this we can provide a standard error object that is used across all the endpoints of our API, which provides further but non- required information to determine why a request failed. We will look at error objects in a little while, but for now let's look at HTTP status codes more in depth.

### 2xx Success

2xx status codes indicate that the clients request has been successfully received and understood.

#### 200 OK

This is a generic response code indicating that the request has succeeded. The response accompanying this code is generally:

GET: An, an entity corresponding to the requested resource

HEAD: The, the header fields corresponding to the requested resource without the message body

POST: An, an entity describing or containing the result of the action

#### 201 Created

The created response is sent when a request succeeds and the result is that a new entity has been created. Along with the response it is common that the API will return a Location header with the location of the newly created entity:

201 Created

Location: https://api.kittens.com/v1/kittens/123dfdf111

It is optional to return an object body with this response type.

#### 204 No Content

This status informs the client that the request has been successfully processed; however, there will be no message body with the response. For example, if the user makes a DELETE request to the collection then the response may return a 204 status.

### 3xx Redirection

The 3xx indicate class of status codes indicates that the client must take additional action to complete the request. Many of these status codes are used by CDNs and other content redirection techniques, however, code 304 can exceptionally useful when designing our APIs to provide semantic feedback to the client.

#### 301 Moved Permanently

This tells the client that the resource they have requested has been permanently moved to a different location. Whilst this is traditionally used to redirect a page or resource from a web server it can also be useful to us when we are building our APIs. In the instance that we rename a collection we could use a 301 redirect to send the client to the correct location.

This however should be used as an exception rather than the norm. Some clients do not implicitly follow 301 redirect and implementing this capability adds additional complexity for your consumers.

#### 304 Not Modiﬁed

This response is generally used by a CDN or caching server and is set to indicate that the response has not been modified since the last call to the API. This is designed to save bandwidth and the request will not return a body, but will return a Content-Location and Expires header.

### 4xx Client Error

In the instance of an error caused by a client, not the server, the server will return a 4xx response and will always return an entity that gives further details on the error.

#### 400 Bad Request

This response indicates that the request could not be understood by the client due to a malformed request or due to a failure of domain validation (missing data, or an operation that would cause invalid state).

#### 401 Unauthorized

This indicates that the request requires user authentication and will include a WWW- Authenticate header containing a challenge applicable to the requested resource. If the user has included the required credentials in the WWW-Authenticate header, then the response should include an error object that may contain relevant diagnostic information.

#### 403 Forbidden

The server has understood the request, but is refusing to fulfill it. This could be due to incorrect access level to a resource not that the user is not authenticated.

If the server does not wish to make the fact that a request is not able to access a resource due to access level public, then it is permissible to return a 404 Not found status instead of this response.

#### 404 Not Found

This response indicates that the server has not found anything matching the requested URI. No indication is given of whether the condition is temporary or permanent.

It is permissible for the client to make multiple requests to this endpoint as the state may not be permanent.

#### 405 Method Not Allowed

The method specified in the request is not allowed for the resource indicated by the URI. This may be when the client attempts to mutate a collection by sending a POST, PUT, or PATCH to a collection that only serves retrieval of documents.

#### 408 Request Timeout

The client did not produce a request within the time that the server is prepared to wait. The client may repeat the request without modification at a later time.

### 5xx Server Error

Response status codes within the 500 range indicate that something has gone "Bang", the server knows this and is sorry for the situation.

The RFC advises that an error entity should be returned in the response explaining whether this is permanent or temporary and containing an explanation of the error. When we look at our chapter on security we will look at the recommendation about not giving too much information away in error messages as this state may have been engineered by a user in the attempt to compromise your system and by returning things such as a stack trace or other internal information with a 5xx error can actually help to compromise your system. With this in mind it is currently common that a 500 error will just return something very generic.

#### 500 Internal Server Error

A generic error message indicating that something did not go quite as planned.

#### 503 Service Unavailable

The server is currently unavailable due to temporary overloading or maintenance. There is a rather useful pattern that you can implement to avoid cascading failure in the instance of a malfunction in which the microservice will monitor its internal state and in the case of failure or overloading will refuse to accept the request and immediately signal this to the client. We will look at this pattern more in chapter xx; however, this instance is probably where you will be wanting to return a 503 status code. This could also be used as part of your health checks.

## HTTP headers

Request headers are a really important part of the HTTP request and response process and implementing a standard approach helps your users to transition from one API to another. In this sub section, we will not cover all the possible headers that you can use in your API, but we will look at the most common headers for full information on the HTTP protocol please take a look at RFC 7231 <https://tools.ietf.org/html/rfc7231>. This document contains a comprehensive overview of the current standard.

### Standard request headers

Request headers provide additional information for the request and the response of your API. Think of them like metadata for the operation. They can be used to augment other data for the response that does not belong in the body itself such as the content encoding. They can also be utilized by the client to provide information that can help the server process the response. Where possible we should always use the standard headers as this gives consistency to your user and provides them with a common standard across multiple endpoints from many different vendors.

#### Authorization - string

Authorization is one of the most commonly used request headers, even if you have a public read only API I advise you to ask the user to authorize their requests. By requesting that the user authorizes a request, you have the capability to perform operations such as user level logging and rate limiting. Quite often you may see authorization conducted with a custom request header such as "X-API-Authorization". I would recommend you do not use this approach as the standard Authorization header as specified by the W3C RFC 2616 (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>) has all the capability we need. Many companies such as Twitter and PayPal use this header to authenticate requests let's. Let's look at a simple example from Twitter's developer documentation to see how this can be implemented:

Authorization:

OAuth oauth\_consumer\_key="="xvz1evFS4wEEPTGEFPHBog",", oauth\_nonce="="kYjzVBB8Y0ZFabxSWbWovY3uYSQ2pTgmZeNu2VS4cg",", oauth\_signature="="tnnArxj06cWHq44gCs1OSKk%2FjLY%3D",", oauth\_signature\_method="="HMAC-SHA1",", oauth\_timestamp="="1318622958",",

oauth\_token="="370773112- GmHxMAgYyLbNEtIKZeRNFsMKPR9EyMZeS9weJAEb",",

oauth\_version="="1.0""

The header is in the form of [Authorization method] [Comma separated URL encoded values]. This clearly informs the server that the authorization type is OAuth and the various components of this authorization follow this in a comma delaminated format. By following this standard approach you can enable your consumers to use a third- party library that implements this standard and thus save them the work of having to build a bespoke implementation.

#### Date

Timestamp of the request in RFC 3339 format.

#### Accept - content type

The requested content type for the response, such as:

application/xml text/xml application/json

text/javascript (for JSONP)

#### Accept-Encoding - gzip, deﬂate

REST endpoints should always support gzip and deflate encoding, when applicable.

Implementing gzip support in Go is relatively straightforward; we showed how it is possible to implement middleware into your microservices in Chapter 1, *Introduction to Microservices*. In the following example, we will use this technique to create a gzip response writer.

The core of writing a response in a gzipped format is the compress/gzip package, which is part of the standard library. It allows you to create a Writer interface that implements ioWriteCloser wrapping an existing io.Writer, which writes to the given writer using the gzip compression:

func NewWriter(w io.Writer) \*Writer

To create our handler we are going to write the NewGzipHandler function, this returns a new http.Handler that will wrap our standard output handler.

The first thing we need to do is create our own ResponseWriter that embeds

http.ResponseWriter.

Example 2.1 chapter2/gzip/gzip\_deflate.go:

1. type GzipResponseWriter struct {
2. gw \*gzip.Writer
3. http.ResponseWriter 71}

The core method for this is the implementation of the Write method:

1. func (w GzipResponseWriter) Write(b []byte) (int, error) {
2. if \_, ok := w.Header()["()["Content-Type"];"]; !ok {
3. // If content type is not set, infer it from the uncompressed body.

|  |  |  |
| --- | --- | --- |
| 76 |  | w.Header().Set("("Content-Type",", http.DetectContentType(b)) |
| 77 |  | } |
| 78 |  | return w.gw.Write(b) |
| 79 | } |  |

If you look at the implementation for Write in the standard http.Response struct there is a whole load of stuff going on in there that we neither want to lose or re-implement because the gzip.Writer object is created with a writer when we call Write on it, it then in turn calls write on http.Response and we lose none of the complexity.

Internally in our NewGzipHandler our handler checks to see if the client has sent the Accept-Encoding header and if so we will write the response using the GzipResponseWriter method if the client has requested uncompressed content then we only call ServeHttp with the standard ResponseWriter:

1. type GZipHandler struct {
2. next http.Handler 42 }

43

1. func (h \*GZipHandler) ServeHTTP(w http.ResponseWriter, r \*http.Request)

{

1. encodings := r.Header.Get("("Accept-Encoding")") 46
2. if strings.Contains(encodings, ""gzip")") {
3. h.serveGzipped(w, r)
4. } else if strings.Contains(encodings, ""deflate")") {
5. panic("("Deflate not implemented")")
6. } else {
7. h.servePlain(w, r) 53 }

54 }

55

1. func (h \*GZipHandler) serveGzipped(w http.ResponseWriter, r

\*http.Request) {

1. gzw := gzip.NewWriter(w)
2. defer gzw.Close() 59
3. w.Header().Set("("Content-Encoding", "", "gzip")")
4. h.next.ServeHTTP(GzipResponseWriter{gzw, w}, r) 62 }

63 func (h \*GZipHandler) servePlain(w http.ResponseWriter, r \*http.Request) 64 {

65 h.next.ServeHTTP(w, r) 66 }

This is by no means a comprehensive example and there are many open source packages like the one from the team at the NY Times (<https://github.com/NYTimes/gziphandler>), which manages this for you.

As a little programming test, why not try and modify this example to implement DEFLATE.

### Standard response headers

All services should return the following headers.

Date: The date that the request was processed in RFC 3339 format.

Content-Type: The content type of the response.

Content-Encoding: gzip or deflate.

X-Request-ID/X-Correlation-ID: Whilst you may not directly request your clients to implement this header it may be something that you add to requests when you call downstream services. When you are trying to debug a service that is running in production it can be incredibly useful to be able to group all the requests by a single transaction ID. A common practice that we will see when we look at logging and monitoring is to store all logs in a common database such as Elastic Search. By setting the standard way of working when building many connected microservices that they pass the correlation ID with each downstream call you will be able to query your logs in Kibana or another log query tool and group them into a single transaction:

X-Request-ID: f058ebd6-02f7-4d3f-942e-904344e8cde

### Returning errors

In the instance of failure, users of your API should be able to write one piece of code that handles errors across different endpoints. A standard error entity will help your consumers by enabling them to write DRY code whenever an error caused by client or server occurs.

The Microsoft API guidelines recommend the following format for these entities:

{

""error":": {

""code": "": "BadArgument",",

""message": "": "Previous passwords may not be reused",", ""target": "": "password",",

""innererror":": a {

""code": "": "PasswordError",", ""innererror":": {

""code": "": "PasswordDoesNotMeetPolicy",", ""minLength": "": "6",",

""maxLength": "": "64",", ""characterTypes": ["":

["lowerCase","","upperCase","","number","","symbol"],"], ""minDistinctCharacterTypes": "": "2",", ""innererror":": {

""code": "": "PasswordReuseNotAllowed""

}

}

}

}

}

##### ErrorResponse: Object

The ErrorResponse is the top level object which will be returned by our response and contains the following fields:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Type** | **Required** | **Description** |
| error | Error |  | The error object. |

##### Error: Object

The Error object is the detail for our error response; it provides full detail for the reason that the error occurred:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Type** | **Required** | **Description** |
| Code | String  (enumerated) |  | One of a server-defined set of error codes. |
| message | String |  | A human-readable representation of the error. |
| Target | String | - | The target of the error. |
| Details | Error[] | - | An array of details about specific errors that led to this reported error. |
| innererror | InnerError | - | An object containing more specific information than the current object about the error. |

##### InnerError: Object

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Type** | **Required** | **Description** |
| Code | String | - | A more specific error code than was provided by the containing error. |
| innererror | InnerError | - | An object containing more specific information than the current object about the error. |

Microsoft has provided an excellent API guidelines resource, you can read more about returning errors by looking at the following link:

<https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#51-errors>

## Accessing APIs from JavaScript

Web browsers implement a sandbox mechanism that restricts resources in one domain from accessing resources in another. For example, you may have an API that allows the modification and retrieval of user data and a website that provides an interface for this API. If the browser did not implement the "same-origin policy" and assuming the user did not log out of their session then it would be possible for a malicious page to send a request to the API and modify it without you knowing.

To get around this, there are two methods that can be implemented by your microservice to allow this access, **JSONP** which stands for (**JSON with Padding**) and **CORS** (**Cross-Origin** **Resource Sharing**).

### JSONP

JSONP is pretty much a hack, and it is implemented by most browsers that do not implement the later CORS standard. It is restricted to GET requests only and works by getting round the issue that while XMLHTTPRequest is blocked from making requests to third-party servers, there are no restrictions on HTML script elements.

A JSONP request inserts a <script src="..."> element into the browsers DOM with the API's URI as the src target. This component returns a function call with the JSON data as a parameter, and when this loads, the function executes passing the data to the callback.

JavaScript callback is defined in the code:

function success(data) { alert(data.message);

}

This is the response from the API call:

success({"({"message":"":"Hello World"})"})

To denote a request for data to be returned as JSONP, generally the

callback=functionName parameter is added to the URI, in our example this would be

/helloworld?callback=success. Implementing this is particularly straightforward let's take a look at our simple Go helloworld example and see how we can modify this to implement JSONP.

One thing to note is the Content-Type header that we are returning. We are no longer returning application/json as we are not returning JSON we are actually returning JavaScript so we must set the Content-Type header accordingly:

Content-Type: application/javascript

Example chapter2/jsonp/jsonp.go:

Let's take a quick look at an example of how we can send JSONP with Go, our response object is going to be exactly the same as the ones in Chapter 1, *Introduction to Microservices*:

1. type helloWorldResponse struct {
2. Message string `json:":"message"`"` 20 }

The difference is all in the handler, if we look at line **30** we are checking to see if there is a callback parameter in the query string. This would be provided by the client and indicates the function they expect to be called when the response is returned:

1. func helloWorldHandler(w http.ResponseWriter, r \*http.Request) {
2. response := helloWorldResponse{Message: ""HelloWorld"}"}
3. data, err := json.Marshal(response)
4. if err != nil {
5. panic("("Ooops")") 28 }

29

1. callback := r.URL.Query().Get("("callback")")
2. if callback != """" {
3. r.Headers().Add("("Content-Type", "", "application/javascript")")
4. fmt.Fprintf(w, "%"%s(%s)",)", callback, string(data))
5. } else {
6. fmt.Fprint(w, string(data)) 36 }

37 }

To return our response in JSONP format all we need to do is wrap the standard response to a JavaScript function call. In line **33**, we are taking the callback function name that was passed by the client and encapsulating the response we would normally send. The resultant output would look something like this:

##### Request:

GET /helloworld?callback=hello

##### Response:

hello({"message":"Hello World"})

### CORS

Assuming your users are using a desktop browser that has been released in the last five years, or a mobile browser such as iOS 9 or Android 4.2+, then implementing CORS will be more than enough. [http://caniuse.com/#feat=cors](http://caniuse.com/#feat%3Dcors) says that it is over 92% of all Internet users. I was looking forward to bashing IE for the lack of full adoption; however, since this has been supported since IE8 I will have to complain about mobile users.

CORS is a W3C proposal to standardize cross-origin requests from the browser. It works by the browsers built in HTTP client making an OPTIONS request to a URI before the real request.

If the server at the other end returns a header that contains the origin of the domain from which the script is being loaded, then the browser will trust the server and will allow a cross-site request to be made:

Access-Control-Allow-Origin: origin.com

Implementing this in Go is quite straightforward and we could create a middleware to globally manage this for us. For simplicity, in our example we have hard coded this into the handler:

Example 2.2 chapter2/cors/cors.go

1. if r.Method == ""OPTIONS"" {
2. w.Header().Add("("Access-Control-Allow-Origin", "\*")", "\*")
3. w.Header().Add("("Access-Control-Allow-Methods", "", "GET")")
4. w.WriteHeader(http.StatusNoContent)
5. return

30 }

In line **25**, we detect if the request method is OPTIONS and instead of returning the response we return the Access-Control-Allow-Origin header that the client is expecting. In our example, we are simply returning \\*, which means all domains are allowed to interact with this API. This is not the safest implementation and quite often you will request your API users to register the domains that will be interacting with the API and restrict the Allow- Origin to only include those domains. In addition to the Allow-Origin header we are also returning the following:

Access-Control-Allow-Methods: GET

This tells the browser that it can only make GET requests to this URI and that it is forbidden to make POST, PUT, and so on. This is an optional header, but it can be used to enhance your user's security when interacting with the API. One thing to note is that we are not sending back a 200 OK response we are using 204 No Content since it is invalid to return a body with an OPTIONS request.

# RPC APIs

RPC stands for remote procedure call; it is a method of executing a function or method on a remote machine. RPC has been around since the dawn of time and there are many different types of RPC technology some of which relies on there being an interface definition (SOAP, Thrift Protocol Buffers). This interface definition can make it easier to generate client and server stubs for different technology stacks. Generally, the interface is defined using a **DSL** (**domain specific language**) and a generator program will use this to create application clients and servers.

Where REST needs to use HTTP as a transport layer, RPC is not bound by this constraint, and while it is possible to send RPC calls over HTTP, you can use the lightness of TCP or even UDP sockets if you choose to.

RPC has seen a resurgence in use lately with many large-scale systems built by the likes of Uber, Google, Netflix, and so on are using RPC. Due to the speed and performance that you can get from the lower latency from not using HTTP and the smaller message size attained by implementing a binary message format rather than JSON or XML.

The detractors of RPC mention the tight coupling that can occur between the client and the server in that if you update the contract on the server then all the clients need to be updated too. With many modern RPC implementations this is less of a problem and in fact is no less a problem than you can have with RESTful APIs. Whilst old technology such as JMI was tightly bound, requiring the client and the server to share the same interface, modern implementations such as Protocol Buffers marshal the object sensibly and will not throw an error should there be minor differences. Thus by following the standard guidelines in the *Versioning APIs* section you have no less a problem than if you were implementing a RESTful API.

One of the benefits of RPC is that you can quickly generate a client for your users, this allows an abstraction from both the transport and the message type and allows them to depend upon an interface. As the creator you can change the underlying implementation of your application such as a move from Thrift to Proto buffers, without requiring the client to do anything other than use the latest version of your provided client. Versioning also allows you to retain the same backward compatibility that you can achieve with REST.

## RPC API design

Some of the principles we have just discussed for creating a good RESTful API can also apply to RPC. However; one of the main differences is that you may not be using HTTP as your transport; therefore you are not always going to be able to use HTTP status codes as an indicator of success or failure. **RPC** stands for **Remote Procedure Call** and dates way back before the Internet. It was originally conceived as a way to execute a procedure that could be running in a separate application on the same machine or even potentially on the network. While we take this for granted now, back in the 90s this was cutting edge.

Unfortunately, frameworks such as CORBA and Java RMI gave RPC a bad name and even now if you speak to an opponent of RPC they will most likely bring these two frameworks up. The benefits, however, were performance, using binary serialization is incredibly efficient on the network and we no longer have the tight coupling that RMI and CORBA enforced. We are also not trying to do anything too clever; we are no longer attempting to share an object across two processes we are taking a more functional approach, that is, methods that return immutable objects. This gives us the best of both worlds; the simplicity of interoperation and the speed and small payload of binary messages.

### RPC message frameworks

These days we are no longer coupled to having the same interface implementation on both the client and the server, this would not adhere to our mantra of independently versionable and deployable. Thankfully frameworks are more flexible we can take the same approach as we do with REST, it is OK to add, however, removing elements or changing the signatures of a method must trigger a version update.

#### Gob

We have already looked at gob in the previous chapter but as a quick re-cap, the gob format was specifically designed to facilitate Go to Go-based communication and was structured around the idea of something easier to use and possibly more efficient than the likes of protocol buffers, this comes at a cost of cross-language communication.

##### gob object definition:

type HelloWorldRequest struct { Name string

}

More information about gob can be found in the Go documentation at [https://golang.org](https://golang.org/pkg/encoding/gob/)

[/pkg/encoding/gob/](https://golang.org/pkg/encoding/gob/)

#### Thrift

The Thrift framework was created by Facebook and was open sourced in 2007. It is currently maintained by the Apache Software Foundation. The main aims of Thrift are:

**Simplicity**: Thrift code is straightforward and approachable, free of unnecessary dependencies

**Transparency**: Thrift conforms to the most common idioms in all languages **Consistency**: Niche, language-specific features belong in extensions, not in the core library

**Performance**: Strive for performance first, elegance second

This is a thrift service definition:

struct User {

1: string name,

2: i32 id,

3: string email

}

struct Error { 1: i32 code,

2: string detail

}

service Users {

Error createUser(1: User user)

}

Find more information on Apache Thrift at [https://thrift.apache.org](https://thrift.apache.org/).

#### Protocol Buﬀers

Protocol Buffers are a Google product, and they have just entered their third revision. Protocol Buffers take the approach of providing a DSL that the generator (written in C) reads and can generate client and server stubs for over ten languages, the primary ten are maintained by Google and encompass: Go, Java, C, JavaScript for NodeJS.

Protocol Buffers is a pluggable architecture, so it is possible to write your own plugins to generate all kinds of endpoints not just RPC; however, RPC is the main use case as they are coupled to the gRPC framework.

gRPC was designed by Google to be a fast and language agnostic RPC framework, which originated from an internal project where latency and speed were of the utmost importance in Google's architecture. By default, gRPC uses protocol buffers as the method for serializing and de-serializing structured data. An example of this DSL is shown in the following example.

Protocol buffer service definition:

service Users {

rpc CreateUser (User) returns (Error) {}

}

message User {

required string name = 1; required int32 id = 2; optional string email = 3;

}

message Error {

optional code int32 = 1 optional detail string = 2

}

Find more information on Protocol Buffers at

<https://developers.google.com/protocol-buffers/>.

#### JSON-RPC

JSON-RPC is an attempt at a standard way of representing objects for RPC using JSON. This removes the need to decode any proprietary binary protocol at the expense of transfer speed. There is no requirement for any particular client or server to serve this data format, TCP sockets, and the ability to write strings that pretty much most all programming languages can manage are all you require.

Unlike Thrift and Protocol Buffers, JSON-RPC sets the standard for the message serialization.

JSON-RPC implements some nice features that allow the batching of requests; every request contains an id parameter, which is established by the client. When the server responds it will return the same identifier allowing the client to understand to which request a response relates.

This is a JSON-RPC serialized request:

{

"jsonrpc": "2.0",

"method": "": "Users.v1.CreateUser", "params": {

"name": "Nic Jackson", "id": 12335432434

},

"id": 1

}

This is a JSON-RPC serialized response:

{

"jsonrpc": "2.0",

"result": {...}, "id":": 1

}

Find more information on JSON-RPC 2.0 at <http://www.jsonrpc.org/specification>.

### Filtering

When we looked at RESTful APIs we discussed the concept of using the query string to perform filtering actions such as:

Paging Filtering Sorting

Obviously, if we are writing an RPC API, we do not have the luxury of a query string; however, implementing these concepts is incredibly useful. As long as we are consistent there is no reason at all that we cannot define a parameter on our request object for the filter condition:

{

"jsonrpc": "2.0",

"method": "": "Users.v1.GetUserLog", "params": {

"name": "Nic Jackson", "id": 12335432434,

"filter": {

"page\_start":": 1, //optional "page\_size"" : 10, //optional "sort": "name DESC" //optional

},

"id": 1

}

This is just an example and you will probably choose to implement something specific to your own needs, however, the key is consistency. If we use this same object for every method, we can be reasonably sure that our users will be cool with this.

# Versioning APIs

API versioning is something you should think about from the very beginning and avoid as long as you can. In general, you will need to make changes to your API, however, having to maintain *n* different versions can be a royal pain in the backside, so doing the upfront design thinking at the beginning can save you a whole load of trouble.

Before we look at how you can version your API, which is quite straightforward let's look at when you should version.

You would increment your API version number when you introduce a breaking change. Breaking changes include:

Removing or renaming APIs or API parameters

Changing the type of an API parameter, for example, from integer to string Changes to response codes, error codes, or fault contracts

Changes to the behavior of an existing API

Things that do not involve a breaking change include:

Adding parameters to a returned entity Adding additional endpoints or functionality

Bug fixes or other maintenance that does not include items in the breaking changes list

## Semantic versioning

Microservices should implement the Major versioning scheme. Quite often, designers will elect to only implement a Major version number and imply .0 for the minor version as according to the semantic versioning principles [http://semver.org](http://semver.org/) a Minor version would generally indicate the addition of functionality that has been implemented in a backwards compatible way. This could be adding additional endpoints to your API. It can be argued that since this would not affect the client's ability to interact with your API you should not worry about Minor versions and only concentrate on major as the client will not need to request a particular version without these additions in order to function.

When versioning APIs I think it is cleaner to drop the minor version and only concentrate on major version. We would take this approach for two reasons:

The URI becomes more readable, and dots are only used as network location separators. When using an RPC API dots are only used to separate API.VERSION.METHOD and make everything easier to read.

We should be inferring through our API versioning that change is a big thing and has an impact on the function of the client. Internally we can still use Major.Minor; however, this does not need to be something to the client as they will not have the capability to elect to use minor versions of the API.

## Versioning formats for REST APIs

To allow the client to request a particular API version, there are three common ways you can do this.

It can be done as part of the URI:

https://myserver.com/v1/helloworld

It can also be done as a query string parameter:

https://myserver.com/helloworld?api-version=1

Finally, It can be done by using a custom HTTP header:

GET https://myserver.com/helloworld api-version: 2

Whichever way you implement versioning is up to you and your team, but it should play a big part in your upfront design thinking. Once you have decided on an option stick, to it as providing a consistent and great experience for your consumers should be one of your primary goals.

## Versioning formats for RPC APIs

Versioning RPC can be a little more difficult as most likely you are not using HTTP as your transport. However, this is still possible. The best way to deal with this is the namespace of your handlers.

In the go base packages, you have the capability to give your handler a name,

Greet.v1.HelloWorld.

### Naming for RPC

With RPC you do not have the luxury of using HTTP verbs to confer the intent of the API, for example, you have the collection users. With an HTTP API you would be able to split up the various actions using GET, POST, DELETE, and so on. This is not possible with an RPC API and you need to think in the same way as if you were writing methods inside your Go code, so for example:

GET /v1/users

The preceding code might be written as an RPC method as follows:

Users.v1.Users

GET /v1/users/123434

Alternatively, it might be written as an RPC method as follows:

Users.v1.User

Sub collections become a little less semantic, whereas in a RESTful API you would be able to do the following:

GET /v1/users/12343/permissions/1232

You cannot do this with an RPC API and you must explicitly specify the method as a separate entity:

Permissions.v1.Permission

The method name also needs to infer the action that the API is going to perform; you cannot rely on the use of HTTP verbs, so in the instance that you have a method that can delete a user you would have to add the delete verb into the method call, for example:

DELETE /v1/users/123123

The preceding code would become:

Users.v1.DeleteUser

# Object type standardization

Whether you are using custom binary serialization, JSON, or JSON-RPC you need to think about how your user is going to handle the object at the other side of the transaction. Many of the serialization packages Protocol Buffers such as protocol buffers and Thrift that use stubs to generate client code will happily deal with serialization of simple types such as Dates into native types that enable your consumer to easily use and manipulate these objects. However, if you are using JSON or JSON-RPC there is no concept of a Date as a native type therefore it can be useful to fall back to ISO standards which the user of the client can easily deserialize. The Microsoft API design guidelines provide some good advice on how to handle Dates and Durations.

## Dates

When returning a date, you should always use the DateLiteral format and preferably the Iso8601Literal. If you do need to send back a date in a format other than Iso8601Literal, then you can use a StructuredDateLiteral format, which allows you to specify the kind as part of the returned entity.

The informal Iso8601Literal format is the simplest method to use and should be understandable by almost any client consuming your API:

{"date": "2016-07-14T16:00Z"}

The more formal StucturedDateLiteral does not return a string, but an entity that contains two properties, kind and value:

{"date": {"kind": "U", "value": 1471186826}}

The permissible kinds are:

C: **CLR**; number of milliseconds since midnight January 1 00

E: **ECMAScript**; number of milliseconds since midnight, January 1, 1970

I: **ISO 8601**; a string limited to the ECMAScript subset

O: **OLE Date**; integral part is the number of days since midnight, December 31, 1899, and fractional part is the time within the day (0.5 = midday)

T: **Ticks**; number of ticks (100-nanosecond intervals) since midnight January 1, 1601

U: **UNIX**; number of seconds since midnight, January 1, 1970

W: **Windows**; number of milliseconds since midnight January 1, 1601

X: **Excel**; as for O, but the year 1900 is incorrectly treated as a leap year, and day 0 is "January 0 (zero)"

### Durations

Durations are serialized to conform with ISO 8601 and are represented by the following format:

P[n]Y[n]M[n]DT[n]H[n]M[n]S

P: This is the duration designator (historically called "period") placed at the start of the duration representation

Y: This is the year designator that follows the value for the number of years

M: This is the month designator that follows the value for the number of months

W: This is the week designator that follows the value for the number of weeks

D: This is the day designator that follows the value for the number of days T: This is the time designator that precedes the time components of the representation

H: This is the hour designator that follows the value for the number of hours

M: This is the minute designator that follows the value for the number of minutes

S: This is the second designator that follows the value for the number of seconds

For example, P3Y6M4DT12H30M5S represents a duration of "three years, six months, four days, twelve hours, thirty minutes, and five seconds".

### Intervals

Again part of the ISO 8601 specification is if you need to receive or send an interval you can use the following format:

Start and end, such as 2007-03-01T13:00:00Z/2008-05-11T15:30:00Z

Start and duration, such as 2007-03-01T13:00:00Z/P1Y2M10DT2H30M Duration and end, such as P1Y2M10DT2H30M/2008-05-11T15:30:00Z Duration only, such as P1Y2M10DT2H30M, with additional context information

Find more information on JSON serialization of dates and times at [https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#113-json-](https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#113-json-serialization-of-dates-and-times) [serialization-of-dates-and-times](https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#113-json-serialization-of-dates-and-times).

# Documenting APIs

Documenting APIs is incredibly useful whether you intend the API to be consumed internally by other teams in your company, external users, or even only yourself. You will thank yourself for spending the time to document the operations of the API and keep this up to date. Keeping documentation up to date should not be an arduous task. There are many applications that can generate documentation automatically from your source code, so all you need to do is run this application as part of your build workflow.

## REST based-based APIs

Currently three primary standards are fighting it out to become the queen of REST API documentation:

Swagger

API Blueprint RAML

### Swagger

Swagger was designed by SmartBear and has been chosen to be part of the Open API Initiative; this potentially gives it the greatest chance of adoption as a standard for documenting RESTful APIs. The Open API Initiative ([https://openapis.org](https://openapis.org/)) however is an industry body and whether it gains the recognition that the W3C has around web standards is probably dependent on more big names joining.

Documentation is written in YAML, and various code generation tools can both write Swagger documentation from source code as well as being able to generate client SDKs. The standard is comprehensive in its feature list and is also relatively simple to write as well as being well understood by the developer community.

The code example of Swagger is shown as follows:

/pets:

get:

description: Returns all pets from the system that the user has access

to

produces:

- application/json responses:

'200''200':

description: A list of pets. schema:

type: array items:

$ref: ''#/definitions/Pet'Pet'

definitions:

Pet:

type: object properties:

name: type: string

description: name of the pet

Find more information on Swagger at [http://swagger.io](http://swagger.io/).

### API Blueprint

API Blueprint is an open standard designed by Apiary and released under the MIT license. It ties closely into Apiary's products. However, it can be used on its own, and there are a variety of open source tools that read and write the format.

Documentation is written in Markdown, which can make authoring the documentation feel a little more natural rather than dealing with nested layers of objects.

The code example for API Blueprint is shown as follows:

FORMAT: 1A

# Data Structures ## Pet (object)

+ name: Jason (string) - Name of the pet. # Pets [/pets]

Returns all pets from the system that the user has access to'to'

## Retrieve all pets [GET]

+ Response 200 (application/json)

+ Attributes (array[Pet])

Find more information on API Blueprint at [https://apiblueprint.org](https://apiblueprint.org/).

### RAML

**RAML** stands for **RESTful API Modelling Language** and is written in YAML format. It aims to allow the definition of a human-readable format that describes resources, methods, parameters, responses, media types, and other HTTP constructs that form the basis of your API.

The code example for RAML is shown as follows:

#%RAML 1.0

title: Pets API

mediaType: [ application/json] types:

Pet:

type: object properties:

name:

type: string

description: name of the pet

/pets:

description: Returns all pets from the system that the user has access to get:

responses:

200:

body: Pet[]

Find more information on RAML at [http://raml.org](http://raml.org/).

## RPC based-based APIs

With RPC APIs there is an argument that your contract is your documentation, in the following example we define the interface using the protocol buffers DSL and would make any necessary comments to assist the consumer as required. The predominant theory to follow is one of self-documenting code that your methods and parameter names should infer intent and enough description to negate the use of comments.

Protocol buffer example:

// The greeting service definition. service Users {

// Create user creates a user in the system with the given User details,

// it returns an Error message which will be nil on a successful operation

rpc CreateUser (User) returns (Error) {}

}

// Person describes a user entity message User {

// name is a required field and represents the name of required string name = 1;

// id is the unique identifier for the user in the sytem required int32 id = 2;

// email is the users email address and is an optional field optional string email = 3;

}

message Error {

optional code int32 = 1 optional detail string = 2

}

Which standard you choose is entirely dependent on you, your workflow, your team standards, and your users. It will vary from case to case, however, once you choose an approach in the same way as you do with naming conventions you should stick to a consistent style across all of your APIs.

# Summary

In this chapter, we did not spend much time looking at code; however, we have looked at some essential concepts around writing a great API, which is as important as being able to write the code.

The bulk of this chapter has been concerned with RESTful APIs as unlike RPC we need to be a little more descriptive in their use. We also have the capability to leverage the principles of HATEOAS, which we do not have when using RPC.

In the next chapter, we will start to look at some of the fantastic frameworks that exist in the Go community, so we can start applying these principles and furthering our advancement to microservice mastery.