Chapter 1. Authentication with Web Services via OAuth

Table of Contents

Summary

*Some REST APIs require authentication and some require us to use a more general mechanism named OAuth to avoid logins and passwords and allow three-party secure interactions. We'll describe this mechanism and how to use the `ROAuth` package in **R** to work with REST APIs requiring authentication with OAuth 1.0, We'll also illustrate how to work with OAuth 2.0 in **R**. We'll illustrate these using Dropbox and Google Storage as examples.*

Questions.

1. *get to R quicker?*

2. *get to Dropbox as the example quicker?*

3. *move the caveats and notes about how handshaking to use OAuth in **R** to end of section of the `ROAuth` package*

1. Securely Accessing Private Data with OAuth

When we use the Web we are using a client-server model where our browser or **R** (or another application) is the client and the Web server is the *provider* of the resource, e.g. an HTML document, a CGI script. In this case, the client is anonymous. The server knows the IP address of the client, but not the identity of the person making the request. For public sites and pages, this is fine as the server doesn't need to know who is requesting the page/*resource* in order to return the information. Of course, when the page or services contains data that is restricted to a specific person or group of people, the client cannot access it anonymously. Instead, the client must identify the user correctly, providing the appropriate credentials. For this, we typically use a login and password. We can specify these in an HTTP request using RCurl via the *userpwd* curl option of the form "login:password", or the *username* and *password* pair of options if we want to separate the login and password.

This simple two-way or *two-legged* authentication is quite simple and familiar. Increasingly, however, we encounter more complex and richer situations which involves three parties in the interaction. Suppose we have a user who wants to allow an application on her desktop or a Web site to access data/resources on a different Web site. The OAuth documentation hueniverse_ProtocolWorkflow describes the following nice example. Suppose we have photos on a Web site such as Flickr. We want to use an online printing service - PhotoPrint - to order physical copies of the photos. While we can download the photos from Flickr and then upload them to the printing site, PhotoPrint provides a neat feature that presents us with a list of our photos and allows us to select which ones we want to print. But how does the photo service get a list of our photos? This is private information and should require us to login to Flickr. In this case, we have three parties in the interaction. Flickr is the *provider* of the resources - the photos. We are the *user* who owns the resources (but we are not the provider). PhotoPrint is the *application* that needs access to these resources from the provider and must be given permission by us - the user and owner - to access them.

Under no circumstances do we want to give the photo service our login and password for Flickr. But we do want to authorize them to get a list of the photos, and also to be able to access the photos we select. We don't want to allow them to delete any photos, change our account in any way, etc. So we want to give them limited privileges and for a short period of time. We also want to be able to revoke these privileges or have them naturally expire. Instead of giving them our login and password, the application, user/owner and provider communicate and give the application a token, or a ticket, that grants them explicit and specific privileges. The application then uses this token in each request it makes to the provider's API so that it is allowed access the user's resources. The OAuth mechanism allows us to do all this. The OAuth guide on hueniverse.com describes OAuth in the following way: " OAuth provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.). "

OAuth is much more general, powerful and complex than the simple login-password used for client-server authentication. OAuth overcomes lots of the insecurities and limitations in basic authentication using a login and password. However, it is more complex. Not only do we have to coordinate all three parties involved to grant and gain the relevant permissions, the OAuth 1.0 mechanism requires that we sign each request to access the privileged data. This signature mechanism is quite complex. The `ROAuth` package hides these details. OAuth2 - the next generation version of the authentication mechanism - is a great deal simpler and more direct. It uses secure HTTP (HTTPS) to remove the complex signature mechanism. While OAuth2 is not a finalized specification, it is in use by significant providers such as Google, Facebook (for their Graph API), LinkedIn and will be used increasingly in the future.

Our focus on OAuth in this book is using it from within **R**. Specifically, we are looking at invoking methods offered by a provider's API so that we can access private resources belong to a user from within **R**. We use OAuth to negotiate between **R**, the user and the provider to gain an access token. We then use OAuth and this access token to invoke methods using an HTTP request. We us the `ROAuth` package, rather than RCurl directly, to make these HTTP requests in order to hide all of the extra details OAuth requires.

In this chapter, we start by looking at OAuth 1.0 and the `ROAuth` package. We'll use Dropbox's API to be able to download and upload files from and to our Dropbox account from within **R**. We illustrate OAuth 2.0 by accessing the Google Storage API. We should note that in both cases, there are packages that provide high-level **R** functions that interface to the methods of these two APIs - `rDrop` and RGoogleStorage. These functions hide much of the details of how we use OAuth in **R**. However, we illustrate some of these in this chapter. The `ROAuth` package itself hides most of the details of using OAuth (e.g. signing the contents of requests) In this chapter, we won't focus on those low-level details, but instead look at how to use OAuth in **R** via the `ROAuth` package. You can read a great deal more about the lower-level details of OAuth in the OAuth specification ???, on the Web ???, and in the book ???

1.1. The OAuth Model and R

Before we discuss the `ROAuth` package and specific **R** functions, it is important to understand how we think about the OAuth model in the context of using it from **R** to call methods in a provider's API. The general OAuth mechanism is for situations where there are three participants. There is the resource *owner/user* (i.e. us) who has to authorize an *application* to have access to the resources maintained by a *provider/server*. The user, application and provider are the three parties. When we are working within **R**, the provider is the remote Web service whose methods we want to invoke. In most cases, the owner will be the **R** user and the application will also be the combination of **R** and the owner. That an user/owner will have her own application. So while there are three parties, two of them (the user and application) are very tightly coupled. As a result, the regular three-participant OAuth approach is a lot more complicated than it need be for this situation. Indeed, a login and password would be much more convenient for us. However, since OAuth is much more flexible and also widely used in other situations, we often need to use it. Accordingly, the common approach to using OAuth in **R** is that each owner will create her own application. (We'll see how to do this for Dropbox below.) To connect all of this to OAuth's three-legged model you can think of yourself (the user) as being the resource owner, **R** as being the application and the provider, e.g. Dropbox, as being the server, or host, of the resources.

When a user creates an application, she registers it with the provider (typically via a Web page). She specifies a name and a description of the app. The application is much more than a name and description. When we register the application, the provider gives two tokens - a consumer key and secret. These uniquely identify the application.

Why should each owner have to create her own application? Instead, we might be tempted to use OAuth in the more common 3-legged manner. In the case of Dropbox, for example, we could create an **R** package (say `rDrop`) that provides an interface to the Dropbox API and which acts as a third-party separate application. The developer of the `rDrop` package would register it with Dropbox. It would have its own consumer key and secret. We could put these in the code for the package and then use them to obtain the access token by having each user grant access to their own data. This appears to make a lot more sense than having each user create her own consumer key and secret for the same **R** package. However, it is a *very* bad idea. Why? Consider the case where we do have a key and secret for the `rDrop` package. These credentials must be

in the **R** code for the package so that it could use them to perform the `handshake()`[1]. This means that other programmers could easily obtain these private key and secret and use them in their own nefarious packages. When they ask a user to grant permission for their code, they could use the key and secret from other code outside of the `rDrop` package. Users would unknowingly grant access to code that they thought was from the `rDrop`package, and the other package would then be able to do what it wanted with the access privileges, and the user would blame our package. We could use an application-specific consumer key and secret if we were running **R** code on a server and so nobody saw that code. However, once we distribute the **R** package to run on other systems not under our control, the consumer key and secret are no longer secure.

One might consider putting the consumer key and secret in compiled code. Again, this is a problem if you give the source code for the package to others to install as an **R** package. If you just provide binaries, people can still use a program such as `strings` to extract those values. Furthermore, if the **R** code in the package can query the key and secret from the native code and then use them within **R** code, then an **R** user can step through **R** code and discover this private data directly. To avoid this, the compiled code would have to do all the signing of the HTTP requests directly itself. There are libraries to help with this (liboauth and an **R** package that uses it), but this is undesirable for several reasons.

1.2. Creating/Registering an Application with the Provider

Before we can access a provider's API in **R**, we need to register an application with the provider. This will give us the consumer key and secret that uniquely identifies the application to the provider. How we register the application differs slightly for each provider, but the basic steps are the same. We login to the provider's Web page using our account for that site, find the relevant Web page for registering developer applications and fill in a form. Assuming there are no errors, the provider will show you the consumer key and secret pair for the new application. You must store these and ensure that they are private, i.e. nobody else can read them.

In order to follow along with our example, you can register an application for Dropbox. We visit the page https://www.dropbox.com/developers/apps and click on the button in the middle of the page entitled "Create an App". Note that this page also lists the existing applications we have registered and we can find their information, including the key and secret, should we have lost them. We can also delete them and so revoke access for those applications. We specify a name for the application. Almost any name is fine. You might use the login name for the Dropbox account prefixed by the letter "R". We'll use "RJaneDoe" for this chapter. You also provide a brief description. It is important to change the default "Access level" option to "Full Dropbox". This is how we control the scope of the privileges. We click on the "Create" button and the Dropbox site shows us a new page with the key and secret for our new application. We will need to record those. It is important to keep these private.

2. The `ROAuth` package

Rather than describing the OAuth mechanism and its low-level details, we'll start by focusing on how we use it in **R** via the `ROAuth` package. After we see it in practice, we'll give a high-level overview of two aspects of how OAuth works (negotiating the access token and signing the HTTP requests) in the hope that this demystifies the steps and helps when things go awry.

[1]There is no point in locating them at a URL as they are still accessible to others. Similarly, keeping them in the package but not in the **R** code still makes them accessible. Basically, anytime we have an open source package, it cannot restrict access to the keys. Instead, we must have each user provide the key and secret and that is what we are doing when they register their own application and get their own pair of key and secret.

2.1. The Basic Workflow in R for OAuth 1.0

There are three pieces to the computational model for **R** that you need to understand to make use of *ROAuth*. The first is that we use the *oauth*() function to create an **R** object that contains the application's consumer key and secret, and information about how to communicate with the provider. The key and secret are for our "application", and are associated with our account on the provider, e.g. Dropbox. The second step is to use the consumer key and secret to get an access token for making actual HTTP requests to the provider's API. The *handshake*() function does the negotiating, involving the user to grant the permissions via a Web page on the provider's site. The final piece is that each request needs to use the credentials object to sign the content of the request. This will happen automatically by using *OAuthRequest*() function for each call to a method in the provider's API.

The basic sequence of operations in pseudo-code are

```
cred = oauth(key, secret, requestURL, authURL, accessURL)
cred = handshake(cred)
OAuthRequest(cred, methodURL,
             listOfArgs,
             method = "GET",
             curl_option, curl_option,
             curl = curlHandle)
```

We first combine the application information for the OAuth provider. Then we negotiate between the application, provider and user to get an access token. Then we use this access token to invoke methods in the provider's API.

2.1.1. Creating the Application *OAuthCredentials* Object

The *oauth*() function assembles the necessary details in order to be able to obtain an access token from the provider. The *oauth*() function typically requires five arguments. The first two are the application's consumer key and secret that were created by the provider when we registered our application. The remaining three arguments are URLs that tell the *oauth*() function how to negotiate for an access token. These are the request URL, authorize URL and access URL. We'll see later that OAuth uses these in sequence to obtain the final access token that has been verified by the user, i.e. owner of the resources we are going to access. For Dropbox, these URLs are listed and explained on the API reference page. Each OAuth provider will provide documentation the tells us its URLs for these tasks.

We can now call the *oauth*() function as

```
cred = oauth(consumerKey = cKey, consumerSecret = cSecret,
             requestURL = "https://api.dropbox.com/1/oauth/request_token",
             authURL = "https://www.dropbox.com/1/oauth/authorize"
             accessURL = "https://api.dropbox.com/1/oauth/access_token/",
             post = FALSE)
```

We assume here that the consumer key and secret are stored in the variables **cKey** and **cSecret**. This is a good practice as they are secret and should never be visible in code. (See below for ideas about how to store them.)

oauth() just creates an **R** object of class *OAuthCredentials*. It does not communicate with the provider - Dropbox - at this point.

Move or remove

The negotiations to get the token and secret necessary for making the actual requests on behalf of the owner involve several steps. They also involve additional security that involves signing the contents of the request in such a way that the receiver can verify that they came from the correct source and were not intercepted and changed, or being re-submitted (e.g. transferring money for a second time). This signature process also involves several steps and some less familiar technologies, e.g. digital signatures. The `ROAuth` package hides these complexities from you, and allows you to focus on invoking the Web service methods.

2.1.2. The Handshake to Get the User Permission and Access Token

Now that we have specified all the details, we can start the negotiation for authorization and the relevant tokens. For Dropbox, we do this by calling the handshake method:

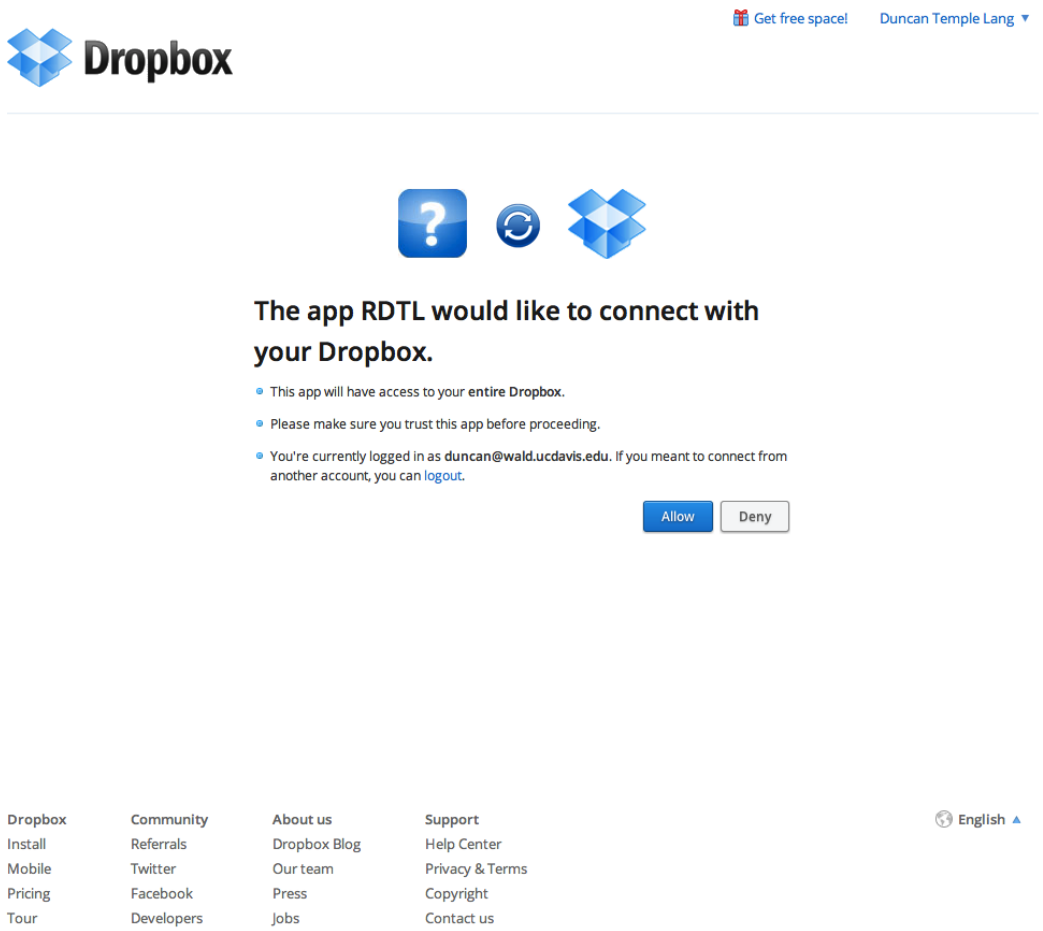
```
cred = handshake(cred, post = FALSE,
                 verify = "hit return in R when you authorize access"))
```

(We can also use the form

```
cred = cred$handshake(post = FALSE, verify = "...")
```

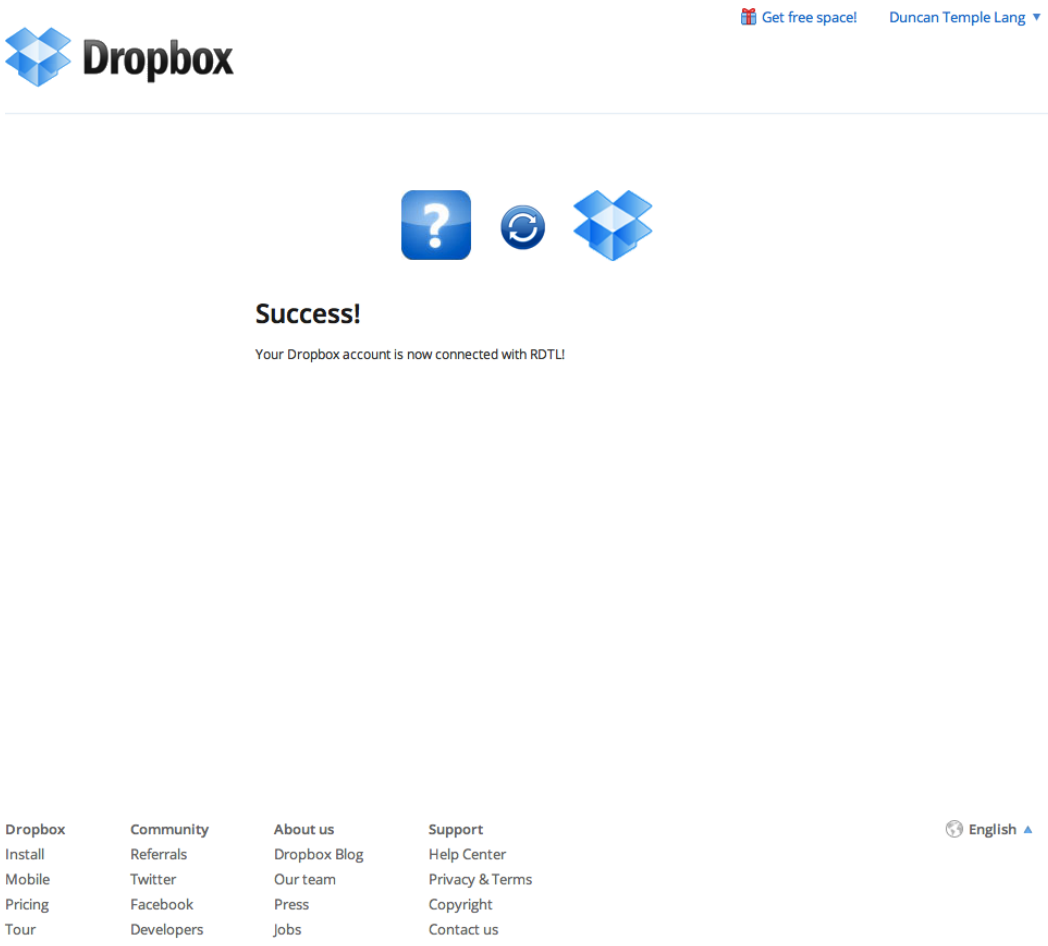
) After we call this function, there is a slight delay and then we see the message that asks us to authorize the access using our message in the *`verify`* argument. Then, our Web browser comes to the foreground and we are displayed a page asking us to grant permission to the application we registered with the provider, Dropbox. If we are not already logged into our account on Dropbox in our Web browser, the browser will direct us to the login page and after entering our login and password, the actual page with which we grant permission to our application will appear. It looks something like

Figure 1.1.



We then grant permission by clicking the "Allow" button and we see the next page

Figure 1.2.



For Dropbox, we return to the **R** prompt and hit enter so that the handshake operation can proceed.

We should note that most OAuth providers will issue a new token after you grant permissions to the application. This will be shown on a Web page when you grant access. You then copy that from the browser back into **R** as the *oauth*() function needs this token before it can proceed to the next stage to get the final access token. Dropbox is unusual in that it doesn't issue an new token. This is why we use the *verify* parameter to specify a message to prompt the user to hit the return key after they granted permission to the application. Generally, *oauth*() will prompt the user with something like

```
To enable the connection, please direct your web browser to:
https://www.dropbox.com/1/oauth/authorize?oauth_token=xxxxxxxxxx
When complete, record the PIN given to you and provide it here,
or hit enter:
```

This tells the user which URL to visit in case she has to visit this manually and also to copy the new token back to **R**.

It is important to note that the `handshake()` function returns the *updated* `OAuthCredentials` object with the additional key and secret that allows us to make requests. It is essential that you assign the object to an **R** variable to be able to use it in future calls. The `OAuthCredentials` object you created with the call to `oauth()` does not have these key and secret, just your own consumer key and secret. So if you do not assign the updated version, it will be as if you never went through the handshake process.

The `post` argument in the call to `handshake()` controls how we perform the handshaking HTTP requests, either using GET or POST operations. POST is the default approach as many OAuth servers use POST and this is the recommended approach. We had to override it in the case of Dropbox.

As with all of the OAuth calls we are making, we are using an HTTP request and we can customize this with Curl options. You can specify these in any OAuth call (`handshake()` or `OAuthRequest()`) as individual arguments or as a list of options via the `.opts` parameter. We can also pass a Curl handle to use for the request via the `curl` parameter.

2.1.3. Invoking Methods with the OAuth access token - `OAuthRequest()`

Once we have completed the handshake to acquire the authentication tokens, we can use it repeatedly to call privileged methods in the provider's API. We do this by calling the `OAuthRequest()` function. The first argument is the `OAuthCredential` object (with the newly updated access key and secret). The second is the URL for the request. Next, we pass a list or character vector of the arguments for the call. We can also specify the type or `method` of the HTTP request, i.e. GET, POST, PUT, DELETE, etc. The default is GET. For example, the Dropbox API has a method for getting meta-information about the user's account. (See https://api.dropbox.com/developers/reference/api.) This method takes no inputs and is a regular "GET" request. So we can invoke it with

```
val = OAuthRequest(cred, "https://api.dropbox.com/1/account/info")
```

The result is a JSON string and we can convert this to an **R** object with

```
fromJSON(val)
```

Again, we can specify Curl options or a Curl handle to control the HTTP request. One can pass options to control how the HTTP request is performed via the **...** parameter of `OAuthRequest()`. For example, we can specify the user-agent for the request and also instruct the request to follow redirects with

```
val = OAuthRequest(cred, "https://api.dropbox.com/1/account/info",
                   httpheader = c('User-agent' = 'rDrop'),
                   followlocation = TRUE, verbose = TRUE)
```

These arguments are passed on to `oauthGET()`, `oauthPOST()` and similarly named functions, selected based on the value of the `method` argument.

In most cases we'll pass actual arguments to parameterize the method. For example, if we want to create a new folder in Dropbox, we have to specify the root folder (either "dropbox" or "sandbox") and the path or name of the new folder. This is a POST request rather than a GET operation. So we can invoke this as

```
val = OAuthRequest(cred,
                   "https://api.dropbox.com/1/fileops/create_folder",
                   c(root = "dropbox", path = "NewFolder"),
                   method = "POST",
                   httpheader = c('User-agent' = 'rDrop'),
                   followlocation = TRUE, verbose = TRUE)
```

Again we specify Curl options, but the interesting argument to our function is the collection of individual arguments passed to the REST method. These are root and path. Here they are passed as a vector, but this can also be a list.

We've mentioned that OAuth 1.0 uses a regular HTTP request, but involves a somewhat complex process of signing the requests and information to the header in the HTTP request. The `OAuthRequest`() function doesn't require input from us to deal with this and uses the information in the `OAuthCredentials` object to take care of all of the details on our behalf. For the curious, we'll discuss the details later in this chapter.

There are some sites that do not require the user-authorization step in the OAuth negotiations. They use OAuth so that they can reliably identify the application making the requests, or simply when there is no specific user but general access. They may want to do this for auditing purposes or to provide different services for different applications, e.g. provide faster response for premium applications. In these cases, we can simply omit the authentication URL when creating the `OAuthCredentials` object, or directly in the call to `handshake`(). For example, the site http://term.ie/oauth/example provides a test site for OAuth1. It does not require user authentication as there is no user. It uses the literal strings "key" and "secret" as the consumer key and secret for all applications. We can obtain our access token using the call

```
cred = oauth("key", "secret",
             requestURL = "http://term.ie/oauth/example/request_token.php",
             accessURL = "http://term.ie/oauth/example/access_token.php")
```

We simply omit the `authURL` in the call. Then we can call `handshake`() and this performs the two-step request to gain the access token.

2.2. Using an Access Token Across R Sessions

¿¿¿¿¿
[Done] I moved so that it is beside Keeping the Consumer Key and Secret Private. This, that and Extending the OAuthCredentials Class and An Alternative Syntax for Invoking OAuth Requests could go in their own section as separate from the mechanics of OAuth 1.0 in ***R****. Done*
????

At the end of a successful call to `handshake`(), the `cred` object will have the information to make authorized requests to access the protected resources. It would be nice to only have to do this handshake and grant permissions for a particular application a single time. We can achieve this by saving the `cred` variable in **R**'s usual way, i.e. via the `save`() function. Then we can load it into other **R** sessions and the tokens will still be valid (unless the user has revoked the permissions on the server). Of course, one can also call the `handshake`() function again and repeat the negotiations and authorization. Remember that the contents of this object are secret and so it is important to be careful that nobody else can read the saved file. If somebody does access it, they can call any of the provider's API methods.

2.3. Keeping the Consumer Key and Secret Private

We have to provide our application's key and secret in the call to `oauth`(). It is important that these remain private and are not shared with other people. As a result, they should not appear in code that can be read by another person. There are several approaches to help with this. One is to set these values as an **R** option, typically in our .Rprofile file that is read when **R** starts. (It is important to ensure that this file is not readable by anybody else on your computer and also that you don't display your options in an **R** session.) Our code can refer to those options and then the values will never be seen in the code. For example, we can call `oauth`() with

```
oauth(getOption("DropboxKey"), getOption("DropboxSecret"),
        requestURL, accessURL, authURL)
```

Similarly, we could also assign the values to variables and then refer to those in our call *oauth*() as we did above

2.4. Extending the *OAuthCredentials* Class

It is convenient to combine and store the consumer key and secret along with the URLs needed for the OAuth negotiations to gain an access token. By combining them in a *OAuthCredentials* object, we can easily pass all of these values to *handshake*(). Of course, it is just as easy to pass them to *handshake*() individually, e.g.

```
cred = handshake(c(consumerKey, consumerSecret),
                 "https://api.dropbox.com/1/oauth/request_token",
                 "https://www.dropbox.com/1/oauth/authorize",
                 "https://api.dropbox.com/1/oauth/access_token/")
```

and this also works. We can also use the resulting *OAuthCredentials* object `cred` in another call to *handshake*() to obtain a new access token.

One of the benefits of using the *OAuthCredentials* object in the first place is that we can define simple sub-classes of it for different services. For example, we might define one for Dropbox and another for Mendeley using

```
setClass("DropboxCredentials", contains = "OAuthCredentials")
setClass("MendeleyCredentials", contains = "OAuthCredentials")
```

The purpose of these classes is that we can create more specific *OAuthCredential* objects and pass them to **R** functions that are wrappers for the Web service methods for that particular service. These functions need only check that the credentials are of the appropriate class rather than checking the URL string for the request. For example, we might implement downloading a file from Dropbox as

```
dropbox_get =
function(cred, filename, binary = NA, ..., curl = getCurlHandle())
{
   if(!is(cred, "DropboxCredentials"))
     stop("invalid credentials")

   OAuthRequest(cred, "https://api-content.dropbox.com/1/files/",
                c(root = "dropbox",  path = filename), binary = binary,
                ..., curl = curl)
}
```

Using a sub-class is good practice as it catches mistakes of using credentials for one service with another and getting quite confused. The way we have written this wrapper function above (*dropbox_get*()) also illustrates several related good practices. The function takes the credential object and then explicit arguments for the parameters expected by the REST request. Importantly, it also takes a Curl handle via the *curl* parameter and any number of other arguments (via the **...** parameter). It uses these in each HTTP request within the function. This allows the caller to control the HTTP request and also reuse an existing Curl handle.

2.5. An Alternative Syntax for Invoking OAuth Requests

It can be convenient to think of the request as being a part of the *OAuthCredentials* object. We can call the *OAuthRequest*() function as

```
cred$OAuthRequest(url, params, methods, ...)
```

This helps to avoid the situation where we forget to pass the **cred** object as the first argument, but instead just focus on the parameters for the specific method we are calling.

The *OAuthRequest*() function looks at the *method* and determines which helper function to invoke, e.g. *oauthGET*() or *oauthDELETE*(). We can by-pass this by using the invocation form

```
cred$get(url, params, ...)
```

where we substitute the "get" with any of post, put, delete, head to specify the method.

Note that this form of invocation - `cred$OAuthRequest()` - is also compatible with the original reference-class based *OAuth* class in the package. That particular implementation can cause problems when the credential objects are serialized and reused after the *ROAuth* package has been changed and improved. Instead of using the new package's methods, the older object uses the older methods from the earlier version of the package. That approach is still available in the package but, for various reasons, we encourage people to use the S4 class *OAuthCredentials* and the *handshake*() function for OAuth 1.0.

2.6. Low-level Details of OAuth 1.0

In this section, we'll give a brief description of two important aspects of OAuth. The first is how we use the application's consumer key and secret to get an access token that we can use to make actual HTTP requests to access the protected data. The second is how we digitally sign the HTTP requests to ensure their integrity and validity.

2.6.1. The OAuth 1.0 Handshake

We use the *handshake*() function to allow the application to obtain the important access token to access the user's resources on the provider. It can then use this to call methods in the provider's API to access the user's resources. We don't need to understand how the *handshake*() function gets the access token to use it. However, we explain the details next for those who are interested or need to know.

Generally in a call to *oauth*(), we have to specify the request URL, the authentication URL and also the access URL for OAuth. However, we only seem to visit one page in the Web browser. It is reasonable to ask what is the purpose of these URLs? The client uses the request, authorization and access URLs in three sequential steps. In each of these, the purpose is to get a token which it can use in the next step. At the end, the application has a token it can then use to access the resources owned by the user and made available via the provider. The user is us, say JaneDoe. The application is RJaneDoe that we registered with the provider, Dropbox. When we registered the application, we were given a key and secret for the application and this is the key-secret pair we pass to *oauth*(). For accessing Dropbox, we would create the *OAuthCredentials* with a call such as

```
cred =
  oauth(consumerKey = cKey, consumerSecret = cSecret,
        requestURL =   "https://api.dropbox.com/1/oauth/request_token",
```

```
        authURL =  "https://www.dropbox.com/1/oauth/authorize",
        accessURL = "https://api.dropbox.com/1/oauth/access_token/")
```

If it helps, we can think of the *OAuthCredentials* object as the application in the OAuth workflow. We, the **R** user, are the user in the OAuth workflow. Of course, Dropbox is the provider, and the files in our Dropbox account are the resources.

Step 1 - application asks provider for an unverified request token. When we call the *handshake*() function with the value of the `cred` variable, the application (our `cred` object) contacts the provider (Dropbox) using the request URL to ask the provider for a *request token*. This is a token which the application can use to ask the user for permission to access the resources on the provider.

The application asks the provider for the request token using its consumer key and secret. The provider checks these are valid application identifiers and issues the token. If the key or secret is not correct, the provider doesn't issue a token and application won't be able to move to the next step.

Step 2 - user grants authorization to the application. The application takes the unverified request token and asks the user to authorize it via the provider's Web site, specifically the authorization URL (given by the *authURL* argument in *oauth*()). This requires the user to be logged into the provider's Web site and so this might first bring the user to the provider's login page and then to the actual authorization URL. When the user grants the permissions requested by the application, the provider issues a verified request token. This is the string that the user copies from the browser back to the **R** session where the *oauth*() function is waiting before it can proceed to the third and final step.

Dropbox, in fact, does not issue a new token for the verified request token. In this case, we can skip the step of copying the new verified request token to **R**, since there is no new token.

Step 3 - application exchanges the verified request token for an access token. The final step in the handshake involves the application exchanging the user-verified request token for an access token that it can use to actually access the user's resources on the provider. It sends the user-verified request token (along with the application's own key and secret) to the provider via the access URL (given by the *accessURL* parameter). The provider returns a new access token. The application then stores that. The application can then use this access token in any later calls to methods in the provider's API in order to access the user's resources.

As we mentioned, the entire handshake process focuses on tokens and there are three tokens - the (unverified) request token, the verified request token and the access token. The user is involved in the second step; only the application and provider are involved in the negotiations for the first and third step.

2.6.2. The OAuth 1.0 Digital Signature Details

The *ROAuth* package also digitally signs each HTTP request performed via *OAuthRequest*() and its helper functions (*oauthGET*(), *oauthPOST*(), etc.). This digital signature involves combining the URL of the method request, all of the parameters in the request, the access token, the consumer key, a time stamp to identify to the server when the request was constructed (and avoid people replaying them at a later date). These are combined into a single string and then the "signature" or hash-based authentication code (HMAC) (or other signing methods) for that string is computed using the consumer secret. This string is then attached to the request in the HTTP header via an Authorization field so that the server can verify the contents of the request and ensure that they haven't been modified by anybody else. *Show an actual request and the headers that it contains*

<duncan>If you want to do this OK with me, but I don't think it's essential</duncan>

The server verifies the signature by looking up its own copy of the secrets associated with the consumer key and the access token. It then has all of the information that the signer had and so can reproduce the signature. If they do not match, the server rejects the request. The signatures may not match because of an error in the client's signing code (or indeed the server too). However, when the code is correct, any mismatch indicates that somebody changed the actual request, e.g. one or more of the parameters. Others cannot create an appropriate new signature because they do not have the consumer and access secret. The server also checks the time stamp and will reject a request if it is from too long ago. It has to allow for some network latency delay in the request, but it will reject a request if it exceeds some threshold. The signature also uses a "nonce", a random number whose sole purpose is to be used just once. This nonce is contained in the signature and in the Authorization field. The server examines the nonce and looks at its records. If the nonce was used before, the request is rejected as being invalid. This prohibits a malicious man-in-the-middle from intercepting the request and replaying it another time, e.g. transferring money from an account.

There are many more details in the signing process which we have not discussed here, such as (lexicographically) ordering the parameters in the entire request, escaping certain characters by converting them to base-64, etc.

3. OAuth 2.0

Up to now, we have described OAuth 1.0. As we mentioned, it is powerful but the details are complex, specifically the digital signing of each request. Fortunately, `ROAuth` package hides these. However, OAuth2 is a good deal simpler as it uses secure HTTP (HTTPS) and avoids the need for a complicated signature process. Because it is quite simple, we can manually and explicitly implement the necessary OAuth2 steps to both obtain a token and to use the token to make a request. We'll show how to do this using Google Storage as a specific example. You can read the code to get the idea. However, to run the code you will need to register an application with Google Storage API. Before that, you will need to enable the Google Storage service for your Google login. See for instructions on how to do this.

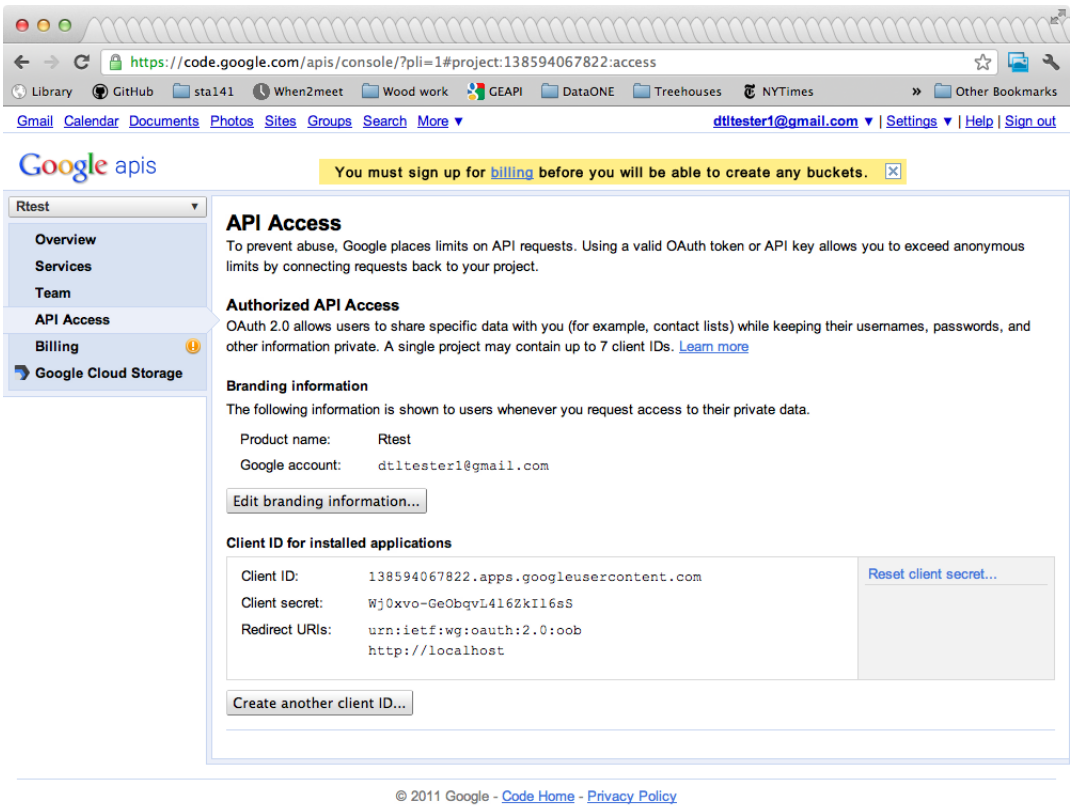
3.1. Google Storage

Google Storage is a RESTful Web service that allows people to store, access and share data in the cloud in a secure manner, similar in spirit to Amazon's Simple Storage Service (S3) [???]. We can upload, download and copy files in buckets (corresponding to folders) and query, set and change permissions (access control lists) using HTTPS requests. Authentication is done using OAuth 2.0. See https://developers.google.com/storage/ for more details about setting up an account, using the API, etc. The RGoogleStorage package provides an **R** interface to this API and hides much of the OAuth 2.0 details. However, in this section, we'll look at how the package performs the authentication to illustrate how we can work with OAuth 2.0 in **R** for other providers.

The workflow for using OAuth 2.0 has a basic similarity to that of OAuth 1.0. We still have the application, user/owner and provider. In this case, Google Storage API is the provider. We, again, have to register an application with the provider. As with OAuth 1.0, this results in a private key and secret so must be done for each user wanting to access Google Storage from within **R**. We create the application via Google's API console[???https://code.google.com/apis/console#access]. The steps involve creating a new project, selecting the API or service of interest, specifying a product name, setting the "Application type" option to "Installed application" (rather than a Web application or Service account) and generating an OAuth 2.0

client id. We won't describe specifically how to do all of this as there is up-to-date documentation on the API Web page. The successful creation of an application rewill be a Web page that looks like the page shown in Figure 1.3 (page 15).

Figure 1.3. Creating an application for accessing Google Storage



Having created the application to access Google Storage, we need the values of the Client ID and Client secret in **R** in order to get authorization and access tokens.

The important information we will need are the Client ID and the Client secret. We assign these to an **R** variable or set them as `options()` in **R** so we can refer to them without showing them in the code.

3.1.1. Getting the User's Permission and the Authorization Token

We now have our application set up. Next, we need to have the user (also us) authorize access to our Google Storage account and its buckets and files. As with OAuth 1.0, this is a negotiation between all three parties - the user, the application and the provider. However, it is a little simpler than with OAuth 1.0. We'll leverage the Web browser to have the user visit a Web page to grant our application the authorization it needs.

To do this, we construct the URL to visit in **R** and tell the user's Web browser to view that page via the `browseURL()` function in **R**. We cannot do this with an RCurl request as we need the Web browser so that the user can login to their Google account and to interact with the page to grant authorization and obtain the authorization token.

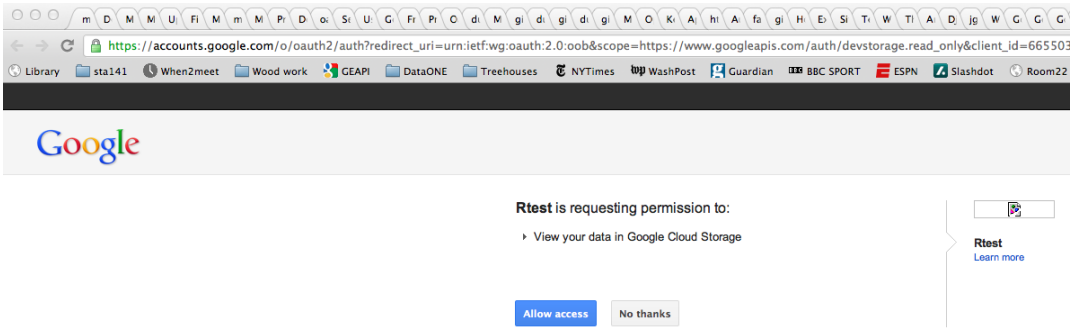
To create the URL, we have to combine our application identifier (Client ID from above) and also information indicating the scope, or set of permissions, we want access for, e.g. read, read and write, or just write. We send the request to https://accounts.google.com/o/oauth2/auth. Note that we are using HTTPS in all of our OAuth 2.0 requests. We add on the different parameters as part of a GET request, i.e. as name=value pairs, separated from each other via '&' and separated from the URL by '?'. So the URL might look something like

```
https://accounts.google.com/o/oauth2/auth?
  redirect_uri=urn:ietf:wg:oauth:2.0:oob&
  scope=https://www.googleapis.com/auth/devstorage.read_only&
  client_id=xxxxxxx.apps.googleusercontent.com&
  response_type=code
```

We of course replace the "xxxxxxx" in the client_id parameter value with the actual identifier for our application. We have also set the redirect_uri parameter to the reserved string 'urn:ietf:wg:oauth:2.0:oob'; this URI indicates that there is no Web page to callback to and that the application is a stand-alone/desktop application. The "oob" in this URI stands for "out of band". This means the token will be returned to us directly in the Web browser rather than the browser being redirected to a Web page specifically for our application.

Once we have created the URL string in **R**, we pass it to `browseURL()` to display this in our Web browser. This will show us the page to grant permissions, bringing us first to the login page, if necessary, of our Google account and grant the permissions. The page to grant permission will look like the screenshot displayed in Figure 1.4 (page 16).

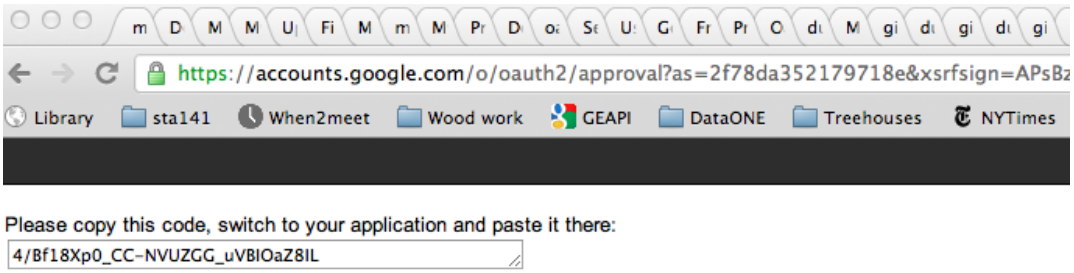
Figure 1.4. An Application Asking a User for Access



This is the Web page the user is shown when an application asks for access to the user's files. The user clicks on one of the buttons to permit or deny access.

We click on the "Allow access" button and then we are shown a page with a single text field containing the all important access token. The page will be similar to that shown in Figure 1.5 (page 17).

Figure 1.5. Getting the Authorization Token



This is the next page the user sees in the Web browser when granting access to an application. The user copies the authorization token in the text field back to **R** so that the application can exchange it for an access token to be able to make requests to the API.

We then cut-and-paste this string into **R** and assign it to a variable, say

```
token = "4/Bf18Xp0_CC-NVUZGG_uVBIOaZ8IL"
```

Since this is just a string and it doesn't tell us anything about its purpose or content, we like to define a class for representing the string and indicating its purpose. When we forget where we got the value, we can check the class. To do this, we could use the following code:

```
setClass('OAuth2PermissionToken', contains = 'character')
token = new('OAuth2PermissionToken', token)
```

Better yet, we can make this a Google token or even a Google Storage token with

```
setClass('GoogleOAuth2Token', contains = 'OAuth2PermissionToken')
setClass('GoogleStorageToken', contains = 'GoogleOAuth2Token')
token = new('GoogleStorageToken', token)
```

We of course only need to define the class once and create different instances at any time.

3.1.2. Exchanging the Authorization Token for an Access Token

We now need to exchange the authorization token for an access token. The user is no longer involved in this negotiation which is simply between our application and Google. Of course, we are doing this in **R**, so we are involved but as the application. For this step, we need to use our client identifier and also its secret. We send an HTTPS POST request to https://accounts.google.com/o/oauth2/token, and include as arguments:

1. the authorization token we just received from the user,

2. the client id and client secret identifying our application,

3. the grant_type as 'authorization_code', and

4. the redirect_uri as before ('urn:ietf:wg:oauth:2.0:oob')

We don't need or want the Web browser at this point as the user is not involved. Instead, we send the request directly from **R** to Google. We can use *postForm*() to do this with the code

```
args = c(client_id = client_id,
         client_secret = client_secret,
         grant_type = 'authorization_code',
         code = as(token, "character"),
         redirect_uri = 'urn:ietf:wg:oauth:2.0:oob')
txt = postForm('https://accounts.google.com/o/oauth2/token',
                .params = args, style = "POST")
```

The result contains the actual access token. Google returns the information in JSON format, such as

```
{
  "access_token" :
     "ya29.AHES6ZR9ZrB4kqY2W9iGm2fm5QNtCqaI7FJ7VTPAK3qDRQ",
  "token_type" : "Bearer",
  "expires_in" : 3600,
  "refresh_token" : "1/bVto2xxzbv_Pz8kHluGR5idfTy8qdiSFBvwIVsUEvYM"
}
```

Other providers may use **XML** or another format.

The JSON content gives us the access token and also its type (Bearer) and when it expires (in seconds). Some providers will just return these and the token will be valid forever. Most providers, however, will have the token expire after a given time, say 60 minutes as here. Rather than have to have the user grant permissions again after the access token expires, OAuth 2.0 allows the provider to return a refresh token along with the access token. The application can use this to renew the access token when it expires, without the user being involved. We'll discuss precisely how to do this later.

To get the access token from the JSON content, we can use the *fromJSON*() function and then save the information as an S4 class (see ??? on how to use this function). We define a class *OAuth2AuthorizationToken* that holds the access token, the refresh token and also the expiration time. We calculate this time by adding the value of the expires_in field to the current time `Sys.time()`.

3.1.3. Using the Access Token in an API Request

We are now going to use the access token to access data in Google Storage. First, let's list the contents of an existing bucket named "proj1". To do this, we send an HTTPS request to https://commondatastorage.googleapis.com/proj1 We put the access token and our client identifier into the header of the HTTP request. This looks something like

```
                                                Authorization
"OAuth ya29.AHES6ZR9ZrB4kqY2W9iGm2fm5QNtCqaI7FJ7VTPAK3qDRQ"
                                             x-goog-project-id
```

```
                                    "xxxxxxxxxxxx"
                                             Date
                        "March, 29 Mar 2012 13:11:16 PDT"
                                    x-goog-api-version
                                                 "2"
```

The Authorization field contains the access token in the *OAuth2AuthorizationToken* object that we obtained above. We prefix this access token with the string "OAuth ". The project ID is just the numbers identifying our application. We add the current date and time so that Google can verify that the request is not being "replayed" at some later time. Finally, we specify the version of the Google API to which we are sending the request. Our request could be made manually as

```
hdr = c(Authorization =
          "OAuth ya29.AHES6ZR9ZrB4kqY2W9iGm2fm5QNtCqaI7FJ7VTPAK3qDRQ"
        'x-goog-project-id' = clientID,
        Date = format(Sys.time(), "%B, %d %b %Y %H:%M:%S %Z"),
        'x-goog-api-version' =  "2")
url = "https://commondatastorage.googleapis.com/proj1"
txt = getURLContent(url, httpheader = hdr,
                    useragent = "RGoogleStorage")
```

The result is an **XML** document that contains an element for each "file" in the bucket (corresponding to a folder). We can then process this and convert it to a data frame in **R** listing the name of the document, when it was last modified, its size in bytes, the owner's name and unique identifier.

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://doc.s3.amazonaws.com/2006-03-01">
  <Name>proj1</Name>
  <Prefix/>
  <Marker/>
  <IsTruncated>false</IsTruncated>
  <Contents>
    <Key>bar</Key>
    <LastModified>2011-05-18T13:05:11.187Z</LastModified>
    <ETag>"5578833a0c6cb26394a1414140718cab"</ETag>
    <Size>12</Size>
    <StorageClass>STANDARD</StorageClass>
    <Owner>
      <ID>00b4903a97f8e9...0edc6aaee</ID>
      <DisplayName>Duncan Temple Lang</DisplayName>
    </Owner>
  </Contents>
  <Contents>
    <Key>myPlot</Key>
    <LastModified>2011-05-18T12:59:26.190Z</LastModified>
    <ETag>"0e49b507686b4ad978ef53832c11c157"</ETag>
    <Size>14184</Size>
    <StorageClass>STANDARD</StorageClass>
    <Owner>
      <ID>00b4903a97f8e9...edc6aaee</ID>
```

```
      <DisplayName>Duncan Temple Lang</DisplayName>
    </Owner>
  </Contents>
</ListBucketResult>
```

We have manually created the content for the *httpheader* setting in several places above. We should do this in a function since the format is the same in each of these places. The `makeHeader()` function in the RGoogleStorage package can be used for this task.

Let's look at an API method where we can create a document within a bucket. We specify the path of the file we want to create (or over-write) and send the request to https://commondatastorage.googleapis.com/path/to/file. Here we are sending content from **R** to Google Storage. We do this via a PUT operation. We get the content of the document either from a file or an **R** object in memory and then upload it using the Curl options *readfunction* and *infilesize*. Again we create our own value for *httpheader* including the Authorization token and the client secret.

In this case, we are writing to the storage facility. For this we need write privileges to the user's Google Storage account. However, we obtained our token for reading only. This means we need to create a new permission and access token to be able to perform this operation. To do this, we repeat the steps in Section 3.1.1 (page 15). The only thing we need to change is the scope, which we set to "https://www.googleapis.com/auth/devstorage.read_write".

We haven't covered how to send arguments in a request other than upload the contents of a file. It turns out that in the Google Storage API, this is how we pass arguments other than identifying the bucket or document in the URL of the request. For instance, to download the contents of a document in a bucket, we use the full URL to that document, e.g. https://commondatastorage.googleapis.com/proj1/abc. To get the access control list (ACL), or in other words, the permissions for a bucket, we use the "acl" parameter as part of a GET request, e.g.
<duncan>I am not sure if the parameters in the request should have some sort of markup or not. I just surrounded this one in quotes. Let me know, if you would like to mark it up and I can revisit.</duncan>

```
getForm("https://commondatastorage.googleapis.com/phase1/foo",
           acl = "", .opts = list(httpheader = oauth2Header))
```

where `oauth2Header` is our header including the token, client secret and date as above. ***FIXME*** When setting an ACL on a bucket, we actually upload an **XML** document that describes the permissions. When copying a document, we specify the target via the URL and the original document by specifying its path in the x-goog-copy-source field in the HTTP header.

3.1.4. Refreshing an OAuth2 Access Token

As we saw when we got the access token, we also received the number of seconds until the token expired and a refresh token. We put the expiration time in the `OAuth2AuthorizationToken` object. When we use this object in one of our functions, we should check to see if this access token has expired (by comparing the expiration time to the current time). If it has, we can use the `refresh_token` to obtain a new access token. This is very similar to the step we used to exchange the original user-granted token for our access token. We POST a request to https://accounts.google.com/o/oauth2/token with the client id and secret. Instead of the `code` and `redirect_uri` arguments, we specify arguments named "refresh_token" and "grant_type". For the former, we provide the value of the refresh token we were given; for the latter, we specify the string 'refresh_token'. So a request might look like

```
args = c(client_id = getOption("Google.storage.ID"),
```

```
         client_secret = getOption("Google.storage.Secret"),
         grant_type = 'refresh_token',
         refresh_token = token@refresh_token)

postForm('https://accounts.google.com/o/oauth2/token',
          .params = args, style = "POST")
```

This again returns a new access token and the expiration duration in JSON form and we can again turn this into an *`OAuth2AuthorizationToken`*.

We can encapsulate getting and refreshing a token into a function so that the details are hidden from the caller. The *`getAuth`*() function in the RGoogleStorage package does this and the function *`getPermission`*() takes care of making the initial request and guiding the caller to the Web browser to grant the permissions.

We should write our functions for API methods so that they check the expiration and update the *`OAuth2AuthorizationToken`* as necessary. Unfortunately, if any of these functions update the token, then that function may find it difficult to return the token as part of the regular value returned from the API method called. For example, suppose we have a function *`download`*() to retrieve a document from Google Storage. This function should return the contents of the document, but if it refreshes the token, that new access token will be discarded when the function returns. This is fine as the other functions can also refresh an expired token when they are called, but there may be a lot of unnecessary refresh requests. Instead, we could issue a warning and ask the user to explicitly refresh the token before the next function call. However, an alternative is to use a mutable object to represent the token and refresh token, i.e. the *`OAuth2AuthorizationToken`*. We would pass this mutable object to our functions which would refresh them if necessary. Since they update that mutable object, the new token and expiration time are in the original object passed to the function and there is no need for the function to return this additional information. This is an example where using a reference class is appropriate simply because we cannot easily return the updated token as well as the actual result of a function.

4. Further Reading

<duncan>Did you want to write a short paragraph as to why you are pointing the reader to these various sources and which parts are particularly worth paying attention to?</duncan>

There are several good books and online documents that describe OAuth 1.0 and OAuth 2.0.

[1] Eran Hammer. 2011. *The OAuth 1.0 Guide*. http://hueniverse.com/oauth/guide

This guide is a terrific explanation of OAuth 1.0 written in a manner that explains it to users of OAuth in very clear terms. It is written by the primary author of the OAuth 1.0 specification. The guide explains all the details of OAuth 1.0, from high-level concepts to details about how to sign each request. There is even an interactive document to illustrate all the steps of the signing process for those who need to understand this.

[2] Johnathan LeBlanc. 2011. *Programming Social Applications*. Building Viral Experiences with OpenSocial, OAuth, OpenID, and Distributed Web Frameworks. O'Reilly Media / Yahoo Press. Sebastopol CA. 978-1449394912.

Chapter 9 of this book provides a very readable description of OAuth 1.0 and also OAuth 2.0. If the discussion at hueniverse.com is not entirely clear, this book should help to clarify those difficult concepts from a different perspective. The book is also interesting for other topics it covers.

[4] . *Using OAuth 2.0 to Access Google APIs*. https://developers.google.com/accounts/docs/OAuth2

Google has a good overview of OAuth 2.0 and how to use it. This is quite clear and comprehensive.

[5] Eran Hammer. 2010. *OAuth 1.0 Specification*. Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc58492070-1721.

This is the official specification for OAuth 1.0. As such it is a little more pedantic than explanatory. It is a good reference when writing the code to implement OAuth 1.0.