OAuth 2.0 for Client-side Web Applications

This document explains how to implement OAuth 2.0 authorization to access Google APIs from a JavaScript web application. OAuth 2.0 allows users to share specific data with an application while keeping their usernames, passwords, and other information private. For example, an application can use OAuth 2.0 to obtain permission from users to store files in their Google Drives.

This OAuth 2.0 flow is called the *implicit grant flow*. It is designed for applications that access APIs only while the user is present at the application. These applications are not able to store confidential information.

In this flow, your app opens a Google URL that uses query parameters to identify your app and the type of API access that the app requires. You can open the URL in the current browser window or a popup. The user can authenticate with Google and grant the requested permissions. Google then redirects the user back to your app. The redirect includes an access token, which your app verifies and then uses to make API requests.

Google APIs Client Library and Google Identity Services

If you use Google APIs client library for JavaScript (/api-client-library/javascript) to make authorized calls to Google, you should use Google Identity Services (/identity/oauth2/web/guides/overview) JavaScript library to handle the OAuth 2.0 flow. Please see Google identity Services' token model (/identity/oauth2/web/guides/use-token-model), which is based upon the OAuth 2.0 *implicit grant* flow.

Note: Given the security implications of getting the implementation correct, we strongly encourage you to use OAuth 2.0 libraries such as Google identity Services' token model (/identity/oauth2/web/guides/use-token-model) when interacting with Google's OAuth 2.0 endpoints. It is a best practice to use well-debugged code provided by others, and it will help you protect yourself and your users.

Rest of this page details how to interact with Google's OAuth 2.0 endpoints directly **without** using any OAuth 2.0 library.

To enable an API for your project:

1. Open the API Library (https://console.developers.google.com/apis/library) in the Google API Console.
2. If prompted, select a project, or create a new one.
3. The API Library lists all available APIs, grouped by product family and popularity. If the API you want to enable isn't visible in the list, use search to find it, or click **View All** in the product family it belongs to.
4. Select the API you want to enable, then click the **Enable** button.
5. If prompted, enable billing.
6. If prompted, read and accept the API's Terms of Service.

Create authorization credentials

Any application that uses OAuth 2.0 to access Google APIs must have authorization credentials that identify the application to Google's OAuth 2.0 server. The following steps explain how to create credentials for your project. Your applications can then use the credentials to access APIs that you have enabled for that project.

1. Go to the Credentials page (https://console.developers.google.com/apis/credentials).
2. Click **Create credentials > OAuth client ID**.
3. Select the **Web application** application type.
4. Complete the form. Applications that use JavaScript to make authorized Google API requests must specify authorized **JavaScript origins**. The origins identify the domains from which your application can send requests to the OAuth 2.0 server. These origins must adhere to Google's validation rules (#origin-validation).

Identify access scopes

Scopes enable your application to only request access to the resources that it needs while also enabling users to control the amount of access that they grant to your application. Thus, there may be an inverse relationship between the number of scopes requested and the likelihood of

If your public application uses scopes that permit access to certain user data, it must complete a verification process. If you see **unverified app** on the screen when testing your application, you must submit a verification request to remove it. Find out more about unverified apps (https://support.google.com/cloud/answer/7454865) and get answers to frequently asked questions about app verification (https://support.google.com/cloud/answer/9110914) in the Help Center.

Obtaining OAuth 2.0 access tokens

The following steps show how your application interacts with Google's OAuth 2.0 server to obtain a user's consent to perform an API request on the user's behalf. Your application must have that consent before it can execute a Google API request that requires user authorization.

Step 1: Redirect to Google's OAuth 2.0 server

To request permission to access a user's data, redirect the user to Google's OAuth 2.0 server.

OAuth 2.0 Endpoints (#oauth-2.0-endpoints)

Generate a URL to request access from Google's OAuth 2.0 endpoint at `https://accounts.google.com/o/oauth2/v2/auth`. This endpoint is accessible over HTTPS; plain HTTP connections are refused.

The Google authorization server supports the following query string parameters for web server applications:

Parameters	
`client_id`	**Required** The client ID for your application. You can find this value in the API Console Credentials page (https://console.developers.google.com/apis/credentials).
`redirect_uri`	**Required**

Parameters	
	Note that the `http` or `https` scheme, case, and trailing slash ('`/`') must all match.
`response_type`	**Required** JavaScript applications need to set the parameter's value to `token`. This value instructs the Google Authorization Server to return the access token as a ***name=value*** pair in the fragment identifier of the URI (`#`) to which the user is redirected after completing the authorization process.
`scope`	**Required** A space-delimited list of scopes that identify the resources that your application could access on the user's behalf. These values inform the consent screen that Google displays to the user. Scopes enable your application to only request access to the resources that it needs while also enabling users to control the amount of access that they grant to your application. Thus, there is an inverse relationship between the number of scopes requested and the likelihood of obtaining user consent. We recommend that your application request access to authorization scopes in context whenever possible. By requesting access to user data in context, via incremental authorization (#incrementalAuth), you help users to more easily understand why your application needs the access it is requesting.
`state`	**Recommended** Specifies any string value that your application uses to maintain state between your authorization request and the authorization server's response. The server returns the exact value that you send as a `name=value` pair in the URL fragment identifier (`#`) of the `redirect_uri` after the user consents to or denies your application's access request. You can use this parameter for several purposes, such as directing the user to the correct resource in your application, sending nonces, and mitigating cross-site request forgery. Since your `redirect_uri` can be guessed, using a `state` value can increase your assurance that an incoming connection is the result of an authentication request. If you generate a random string or encode the hash of a cookie or another value that captures the client's state, you can validate the response to additionally ensure that

Parameters	
	Enables applications to use incremental authorization to request access to additional scopes in context. If you set this parameter's value to `true` and the authorization request is granted, then the new access token will also cover any scopes to which the user previously granted the application access. See the incremental authorization (#incrementalAuth) section for examples.
`enable_granular_consent`	**Optional** Defaults to `true`. If set to `false`, more granular Google Account permissions (https://developers.googleblog.com/2018/10/more-granular-google-account.html) will be disabled for OAuth client IDs created before 2019. No effect for newer OAuth client IDs, since more granular permissions is always enabled for them.
`login_hint`	**Optional** If your application knows which user is trying to authenticate, it can use this parameter to provide a hint to the Google Authentication Server. The server uses the hint to simplify the login flow either by prefilling the email field in the sign-in form or by selecting the appropriate multi-login session. Set the parameter value to an email address or `sub` identifier, which is equivalent to the user's Google ID.
`prompt`	**Optional** A space-delimited, case-sensitive list of prompts to present the user. If you don't specify this parameter, the user will be prompted only the first time your project requests access. See Prompting re-consent (/identity/protocols/oauth2/openid-connect#re-consent) for more information. Possible values are: `none` — Do not display any authentication or consent screens. Must not be specified with other values. `consent` — Prompt the user for consent.

```
https://accounts.google.com/o/oauth2/v2/auth?
 scope=https%3A//www.googleapis.com/auth/drive.metadata.readonly&
 include_granted_scopes=true&
 response_type=token&
 state=state_parameter_passthrough_value&
 redirect_uri=https%3A//oauth2.example.com/code&
 client_id=client_id
```

After you create the request URL, redirect the user to it.

JavaScript sample code

The following JavaScript snippet shows how to initiate the authorization flow in JavaScript without using the Google APIs Client Library for JavaScript. Since this OAuth 2.0 endpoint does not support Cross-Origin Resource Sharing (CORS), the snippet creates a form that opens the request to that endpoint.

```
/*
 * Create form to request access token from Google's OAuth 2.0 server.
 */
function oauthSignIn() {
  // Google's OAuth 2.0 endpoint for requesting an access token
  var oauth2Endpoint = 'https://accounts.google.com/o/oauth2/v2/auth';

  // Create <form> element to submit parameters to OAuth 2.0 endpoint.
  var form = document.createElement('form');
  form.setAttribute('method', 'GET'); // Send as a GET request.
  form.setAttribute('action', oauth2Endpoint);

  // Parameters to pass to OAuth 2.0 endpoint.
  var params = {'client_id': 'YOUR_CLIENT_ID',
                'redirect_uri': 'YOUR_REDIRECT_URI',
                'response_type': 'token',
                'scope': 'https://www.googleapis.com/auth/drive.metadata.reac
                'include_granted_scopes': 'true',
                'state': 'pass-through value'};

  // Add form parameters as hidden input values
```

```
    document.body.appendChild(form);
    form.submit();
  }
```

Step 2: Google prompts user for consent

In this step, the user decides whether to grant your application the requested access. At this stage, Google displays a consent window that shows the name of your application and the Google API services that it is requesting permission to access with the user's authorization credentials and a summary of the scopes of access to be granted. The user can then consent to grant access to one or more scopes requested by your application or refuse the request.

Your application doesn't need to do anything at this stage as it waits for the response from Google's OAuth 2.0 server indicating whether any access was granted. That response is explained in the following step.

Errors

Requests to Google's OAuth 2.0 authorization endpoint may display user-facing error messages instead of the expected authentication and authorization flows. Common error codes and suggested resolutions are listed below.

`admin_policy_enforced`

The Google Account is unable to authorize one or more scopes requested due to the policies of their Google Workspace administrator. See the Google Workspace Admin help article Control which third-party & internal apps access Google Workspace data (https://support.google.com/a/answer/7281227) for more information about how an administrator may restrict access to all scopes or sensitive and restricted scopes until access is explicitly granted to your OAuth client ID.

`disallowed_useragent`

The authorization endpoint is displayed inside an embedded user-agent disallowed by Google's OAuth 2.0 Policies (/identity/protocols/oauth2/policies#browsers) use Android libraries such as Google Sign-In for Android (/identity/sign-in/android) or OpenID Foundation's AppAuth for Android (https://openid.github.io/AppAuth-Android/).

Web developers may encounter this error when an Android app opens a general web link in an embedded user-agent and a user navigates to Google's OAuth 2.0 authorization endpoint from your site. Developers should allow general links to open in the default link handler of the operating system, which includes both Android App Links (https://developer.android.com/training/app-links) handlers or the default browser app. The Android Custom Tabs (https://developer.chrome.com/docs/android/custom-tabs/overview/) library is also a supported option.

`org_internal`

The OAuth client ID in the request is part of a project limiting access to Google Accounts in a specific Google Cloud Organization (https://cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy#organizations). For more information about this configuration option see the User type (https://support.google.com/cloud/answer/10311615#user-type) section in the Setting up your OAuth consent screen help article.

`invalid_client`

The origin from which the request was made is not authorized for this client. See **`origin_mismatch`** (#authorization-errors-origin-mismatch).

`invalid_grant`

When using incremental authorization (#incrementalAuth), the token may have expired or has been invalidated. Authenticate the user again and ask for user consent to obtain new tokens. If you are continuing to see this error, ensure that your application has been configured correctly and that you are using the correct tokens and parameters in your request. Otherwise, the user account may have been deleted or disabled.

`origin_mismatch`

The scheme, domain, and/or port of the JavaScript originating the authorization request may not

The `redirect_uri` passed in the authorization request does not match an authorized redirect URI for the OAuth client ID. Review authorized redirect URIs in the Google API Console Credentials page (https://console.developers.google.com/apis/credentials).

The scheme, domain, and/or port of the JavaScript originating the authorization request may not match an authorized JavaScript origin URI registered for the OAuth client ID. Review authorized JavaScript origins in the Google API Console Credentials page (https://console.developers.google.com/apis/credentials).

The `redirect_uri` parameter may refer to the OAuth out-of-band (OOB) flow that has been deprecated and is no longer supported. Refer to the migration guide (/identity/protocols/oauth2/resources/oob-migration) to update your integration.

`invalid_request`

There was something wrong with the request you made. This could be due to a number of reasons:

- The request was not properly formatted
- The request was missing required parameters
- The request uses an authorization method that Google doesn't support. Verify your OAuth integration uses a recommended integration method

Step 3: Handle the OAuth 2.0 server response

OAuth 2.0 Endpoints (#oauth-2.0-endpoints)

The OAuth 2.0 server sends a response to the `redirect_uri` specified in your access token request.

If the user approves the request, then the response contains an access token. If the user does not approve the request, the response contains an error message. The access token or error message is returned on the hash fragment of the redirect URI, as shown below:

- An access token response:

parameter was specified in the access token request, its value is also included in the response.

- An error response:

```
https://oauth2.example.com/callback#error=access_denied
```

Note: Your application should ignore any additional, unrecognized fields included in the query string.

Sample OAuth 2.0 server response

You can test this flow by clicking on the following sample URL, which requests read-only access to view metadata for files in your Google Drive:

```
https://accounts.google.com/o/oauth2/v2/auth?
 scope=https%3A//www.googleapis.com/auth/drive.metadata.readonly&
 include_granted_scopes=true&
 response_type=token&
 state=state_parameter_passthrough_value&
 redirect_uri=https%3A//oauth2.example.com/code&
 client_id=client_id
```
(https://accounts.google.com/o/oauth2/v2/auth?scope=https%3A//www.gc

After completing the OAuth 2.0 flow, you will be redirected to `http://localhost/oauth2callback`. That URL will yield a `404 NOT FOUND` error unless your local machine happens to serve a file at that address. The next step provides more detail about the information returned in the URI when the user is redirected back to your application.

Calling Google APIs

OAuth 2.0 Endpoints
(#oauth-2.0-endpoints)

You can try out all the Google APIs and view their scopes at the OAuth 2.0 Playground (https://developers.google.com/oauthplayground/).

HTTP GET examples

A call to the `drive.files` (/drive/v2/reference/files/list) endpoint (the Drive Files API) using the `Authorization: Bearer` HTTP header might look like the following. Note that you need to specify your own access token:

```
GET /drive/v2/files HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer access_token
```

Here is a call to the same API for the authenticated user using the `access_token` query string parameter:

```
GET https://www.googleapis.com/drive/v2/files?access_token=access_token
```

`curl` examples

You can test these commands with the `curl` command-line application. Here's an example that uses the HTTP header option (preferred):

```
$ curl -H "Authorization: Bearer access_token" https://www.googleapis.com/
```

Or, alternatively, the query string parameter option:

```
$ curl https://www.googleapis.com/drive/v2/files?access_token=access_token
```

JavaScript sample code

In this code snippet, the `access_token` variable represents the token you have obtained to make API requests on the authorized user's behalf. The complete example (#example) demonstrates how to store that token in the browser's local storage and retrieve it when making an API request.

```
var xhr = new XMLHttpRequest();
xhr.open('GET',
    'https://www.googleapis.com/drive/v3/about?fields=user&' +
    'access_token=' + params['access_token']);
xhr.onreadystatechange = function (e) {
  console.log(xhr.response);
};
xhr.send(null);
```

Complete example

OAuth 2.0 Endpoints (#oauth-2.0-endpoints)

This code sample demonstrates how to complete the OAuth 2.0 flow in JavaScript without using the Google APIs Client Library for JavaScript. The code is for an HTML page that displays a button to try an API request. If you click the button, the code checks to see whether the page has stored an API access token in your browser's local storage. If so, it executes the API request. Otherwise, it initiates the OAuth 2.0 flow.

For the OAuth 2.0 flow, the page follows these steps:

1. It directs the user to Google's OAuth 2.0 server, which requests access to the `https://www.googleapis.com/auth/drive.metadata.readonly` scope.
2. After granting (or denying) access to one or more requested scopes, the user is redirected to the original page, which parses the access token from the fragment identifier string.
3. The page uses the access token to make the sample API request.

listed as **OAuth 2.0 Demo for Google API Docs**.

To run this code locally, you need to set values for the `YOUR_CLIENT_ID` and `YOUR_REDIRECT_URI` variables that correspond to your authorization credentials (#creatingcred). The `YOUR_REDIRECT_URI` variable should be set to the same URL where the page is being served. The value must exactly match one of the authorized redirect URIs for the OAuth 2.0 client, which you configured in the API Console Credentials page. If this value doesn't match an authorized URI, you will get a `redirect_uri_mismatch` error. Your project must also have enabled the appropriate API (#enable-apis) for this request.

```
<html><head></head><body>
<script>
  var YOUR_CLIENT_ID = 'REPLACE_THIS_VALUE';
  var YOUR_REDIRECT_URI = 'REPLACE_THIS_VALUE';
  var fragmentString = location.hash.substring(1);

  // Parse query string to see if page request is coming from OAuth 2.0 serve
  var params = {};
  var regex = /([^&=]+)=([^&]*)/g, m;
  while (m = regex.exec(fragmentString)) {
    params[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
  }
  if (Object.keys(params).length > 0) {
    localStorage.setItem('oauth2-test-params', JSON.stringify(params) );
    if (params['state'] && params['state'] == 'try_sample_request') {
      trySampleRequest();
    }
  }

  // If there's an access token, try an API request.
  // Otherwise, start OAuth 2.0 flow.
  function trySampleRequest() {
    var params = JSON.parse(localStorage.getItem('oauth2-test-params'));
    if (params && params['access_token']) {
      var xhr = new XMLHttpRequest();
      xhr.open('GET',
          'https://www.googleapis.com/drive/v3/about?fields=user&' +
          'access_token=' + params['access_token']);
      xhr.onreadystatechange = function (e) {
        if (xhr.readyState === 4 && xhr.status === 200) {
    }
  }

  /*
   * Create form to request access token from Google's OAuth 2.0 server.
   */
  function oauth2SignIn() {
    // Google's OAuth 2.0 endpoint for requesting an access token
    var oauth2Endpoint = 'https://accounts.google.com/o/oauth2/v2/auth';

    // Create element to open OAuth 2.0 endpoint in new window.
    var form = document.createElement('form');
    form.setAttribute('method', 'GET'); // Send as a GET request.
    form.setAttribute('action', oauth2Endpoint);

    // Parameters to pass to OAuth 2.0 endpoint.
    var params = {'client_id': YOUR_CLIENT_ID,
                  'redirect_uri': YOUR_REDIRECT_URI,
                  'scope': 'https://www.googleapis.com/auth/drive.metadata.re
                  'state': 'try_sample_request',
                  'include_granted_scopes': 'true',
                  'response_type': 'token'};

    // Add form parameters as hidden input values.
    for (var p in params) {
      var input = document.createElement('input');
      input.setAttribute('type', 'hidden');
      input.setAttribute('name', p);
      input.setAttribute('value', params[p]);
      form.appendChild(input);
    }

    // Add form to page and submit it to open the OAuth 2.0 endpoint.
    document.body.appendChild(form);
    form.submit();
  }
</script>

<button onclick="trySampleRequest();">Try sample request</button>
</body></html>
```

Validation rules	
Scheme (https://tools.ietf.org/html/rfc3986#section-3.1)	JavaScript origins must use the HTTPS scheme, not plain HTTP. Localhost URIs (including localhost IP address URIs) are exempt from this rule.
Host (https://tools.ietf.org/html/rfc3986#section-3.2.2)	Hosts cannot be raw IP addresses. Localhost IP addresses are exempted from this rule.
Domain (https://tools.ietf.org/html/rfc1034)	• Host TLDs (Top Level Domains (https://tools.ietf.org/id/draft-liman-tld-names-00.html)) must belong to the public suffix list (https://publicsuffix.org/list/). • Host domains cannot be "`googleusercontent.com`". • JavaScript origins cannot contain URL shortener domains (e.g. `goo.gl`) unless the app owns the domain.
Userinfo (https://tools.ietf.org/html/rfc3986#section-3.2.1)	JavaScript origins cannot contain the userinfo subcomponent.
Path (https://tools.ietf.org/html/rfc3986#section-3.3)	JavaScript origins cannot contain the path component.
Query (https://tools.ietf.org/html/rfc3986#section-3.4)	JavaScript origins cannot contain the query component.
Fragment (https://tools.ietf.org/html/rfc3986#section-3.5)	JavaScript origins cannot contain the fragment component.
Characters	JavaScript origins cannot contain certain characters including: • Wildcard characters (`'*'`) • Non-printable ASCII characters • Invalid percent encodings (any percent encoding that does not follow URL-encoding form of a percent sign followed by two hexadecimal digits) • Null characters (an encoded NULL character, e.g., `%00`,

resources at the time you need them. To enable that practice, Google's authorization server supports incremental authorization. This feature lets you request scopes as they are needed and, if the user grants permission for the new scope, returns an authorization code that may be exchanged for a token containing all scopes the user has granted the project.

For example, an app that lets people sample music tracks and create mixes might need very few resources at sign-in time, perhaps nothing more than the name of the person signing in. However, saving a completed mix would require access to their Google Drive. Most people would find it natural if they only were asked for access to their Google Drive at the time the app actually needed it.

In this case, at sign-in time the app might request the `openid` and `profile` scopes to perform basic sign-in, and then later request the `https://www.googleapis.com/auth/drive.file` scope at the time of the first request to save a mix.

The following rules apply to an access token obtained from an incremental authorization:

- The token can be used to access resources corresponding to any of the scopes rolled into the new, combined authorization.
- When you use the refresh token for the combined authorization to obtain an access token, the access token represents the combined authorization and can be used for any of the `scope` values included in the response.
- The combined authorization includes all scopes that the user granted to the API project even if the grants were requested from different clients. For example, if a user granted access to one scope using an application's desktop client and then granted another scope to the same application via a mobile client, the combined authorization would include both scopes.
- If you revoke a token that represents a combined authorization, access to all of that authorization's scopes on behalf of the associated user are revoked simultaneously.

Caution: choosing to include granted scopes will automatically add scopes previously granted by the user to your authorization request. A warning or error page may be displayed if your app is not currently approved to request all scopes that may be returned in the response. See Unverified apps (https://support.google.com/cloud/answer/7454865) for more information.

The following code snippet demonstrates how to do that. The snippet assumes that you have stored the scopes for which your access token is valid in the browser's local storage. (The complete example (#example) code stores a list of scopes for which the access token is valid by setting the **`oauth2-test-params.scope`** property in the browser's local storage.)

The snippet compares the scopes for which the access token is valid to the scope you want to use for a particular query. If the access token does not cover that scope, the OAuth 2.0 flow starts. Here, the **`oauth2SignIn`** function is the same as the one that was provided in step 2 (#redirecting) (and that is provided later in the complete example (#example)).

```
var SCOPE = 'https://www.googleapis.com/auth/drive.metadata.readonly';
var params = JSON.parse(localStorage.getItem('oauth2-test-params'));

var current_scope_granted = false;
if (params.hasOwnProperty('scope')) {
  var scopes = params['scope'].split(' ');
  for (var s = 0; s < scopes.length; s++) {
    if (SCOPE == scopes[s]) {
      current_scope_granted = true;
    }
  }
}

if (!current_scope_granted) {
  oauth2SignIn(); // This function is defined elsewhere in this document.
} else {
  // Since you already have access, you can proceed with the API request.
}
```

Revoking a token

In some cases a user may wish to revoke access given to an application. A user can revoke access by visiting Account Settings (https://myaccount.google.com/permissions). See the Remove site or app access section of the Third-party sites & apps with access to your account (https://support.google.com/accounts/answer/3466521#remove-access) support document for more

OAuth 2.0 Endpoints
(#oauth-2.0-endpoints)

To programmatically revoke a token, your application makes a request to `https://oauth2.googleapis.com/revoke` and includes the token as a parameter:

```
$ curl -d -X -POST --header "Content-type:application/x-www-form-urlencoded"
        https://oauth2.googleapis.com/revoke?token={token}
```

The token can be an access token or a refresh token. If the token is an access token and it has a corresponding refresh token, the refresh token will also be revoked.

Note: Google's OAuth 2.0 endpoint for revoking tokens supports JSONP and form submissions. It does not support Cross-origin Resource Sharing (CORS).

If the revocation is successfully processed, then the HTTP status code of the response is `200`. For error conditions, an HTTP status code `400` is returned along with an error code.

The following JavaScript snippet shows how to revoke a token in JavaScript without using the Google APIs Client Library for JavaScript. Since the Google's OAuth 2.0 endpoint for revoking tokens does not support Cross-origin Resource Sharing (CORS), the code creates a form and submits the form to the endpoint rather than using the `XMLHttpRequest()` method to post the request.

```
function revokeAccess(accessToken) {
  // Google's OAuth 2.0 endpoint for revoking access tokens.
  var revokeTokenEndpoint = 'https://oauth2.googleapis.com/revoke';

  // Create <form> element to use to POST data to the OAuth 2.0 endpoint.
  var form = document.createElement('form');
  form.setAttribute('method', 'post');
  form.setAttribute('action', revokeTokenEndpoint);

  // Add access token to the form so it is set as value of 'token' parameter
  // This corresponds to the sample curl request, where the URL is:
  form.submit();
}
```

Note: Following a successful revocation response, it might take some time before the revocation has full effect.

Last updated 2023-11-07 UTC.