REST API

Overview

Yellowfin exposes a public REST API that allows third party developers to create their own utilities, applications, and integrations with Yellowfin systems and content. This can work in tandem with existing integrations such as the JS API, SOAP API, and tight application integration, or as an entirely standalone integration tool.

The API exposes most features of several main content types such as stories, signals, discussion streams, reports (coming soon), users, and user timelines. It also has some administration capabilities to supply features such as user management, category management, import/export, system configuration, and user session management so that developers can provide their own utilities to administer and control their Yellowfin system.

Some of the possible uses of the API are to:

- integrate Yellowfin content inside a third party application, such as browsing and displaying Signals or other content, with the ability to control how it is displayed and how users interact and navigate;
- create administration utilities, such as a third-party application which can manage the configuration and administration of a Yellowfin instance without needing to login to the web interface; or,
- build a fully customized implementation of the Yellowfin application user interface, allowing developers to control how the application is displayed and interacted with, or to restrict features based on users or other specific requirements.

Key Concepts

The REST API is available under the `/api` namespace. For example, `https://yellowfin.myapp.com/api/stories`

Additionally, the suite includes RPC (Remote Procedure Call) endpoints which support workflows that are difficult to fit into the REST paradigm. These are located in the `/api/rpc` namespace.

Every API request requires an **Authorization header**. Its format is

```
YELLOWFIN ts=1600224140615 nonce=3370ddc4-37d9-41b9-9f24-ada181fdc4bf token=securityToken
```

Component	Description
`YELLOWFIN`	Custom authentication scheme. This text should match the application name, which can be set via the custom installer properties file or via Yellowfin configuration database. Note: the text must be written in uppercase, without any spaces. For example, if the App Name is 'BigFish Reporting' instead of 'YELLOWFIN', the authentication scheme should be written as 'BIGFISHREPORTING'.
`ts`	The time in milliseconds from the Unix epoch 00:00:00 UTC on 1 January 1970. This is the current time in the program which calls the API. Every programming language has a way to get the current time in this format.
`nonce`	A random UUID generated by the client.
`token`	A security token used for authenticating the user and authorizing access to the resource. Not all endpoints require this section, since some can be accessed without authorization (see documentation).

Every API request requires an **Accept header**.

- This header is used to identify the version of the API.
- Its format is specified in the API doc for each endpoint. Most resources have a JSON representation, so for example a v1 JSON resource would require `application/vnd.yellowfin.api-v1+json`
- The API is backwards compatible. Requests for a v1 resource will work even when the current API version in a Yellowfin instance is v2.

There are two security tokens which are key for consuming the API.

Token	Description
Refresh	This is an opaque security token obtained on login. Refresh Tokens do not expire and may be securely saved in the client application for obtaining access tokens.
Access	This is a JSON Web Token (JWT) which expires after 20 minutes. An access token needs to be sent in the Authorization header of nearly every API request. On expiry, the client application can use the refresh token to get a new access token.

Every API response will have one `"_links"` object. The `"_links"` object can also contain one or more link objects.

- Every link represents related resources which the user has access to.
- The client should use the link in the `"href"` attribute to access the resource rather than hard coding it in application code.
- The `"options"` array lists the HTTP methods which the user is authorised to use with the link. For example, the example above tells us that the user can read the comments list (GET) or create a new one (POST). They cannot delete all comments, which is why DELETE is not available in the `"comments"` link.


```
"_links": {
    "menu": {
        "href": "/api/menus/mobile-menu",
        "options": [
            "GET"
        ]
    },
    "self": {
        "href": "/api/stories/fcf269b0-0e14-4d15-919b-712b4143fb70",
        "options": [
            "GET"
        ]
    },
    "comments": {
        "href": "/api/stories/fcf269b0-0e14-4d15-919b-712b4143fb70/comments",
        "options": [
            "GET",
            "POST"
        ]
    },
    "share": {
        "href": "/api/stories/fcf269b0-0e14-4d15-919b-712b4143fb70/content-shares",
        "options": [
            "POST"
        ]
```


Some API responses will have an `"_embedded"` object, which can contain sub-objects that contain additional useful information related to the current resource, but which does not belong directly to that resource. These are separate resources which can have their own properties and links.

```
"_embedded": {
    "thumbnail": {
        "isPlaceholder": false,
        "_links": {
            "self": {
                "href": "/api/images/101170?thumbSize=THUMB_240_140&imageLoadType=IMAGE_THUMBNAIL",
                "options": [
                    "GET"
                ]
            }
        }
    }
}
```


top

Authentication

Most endpoints in the API require authentication. Some notable exceptions are:

- the base `/api` resource, which can be used with or without authentication to get basic api info generally used by the api-consumer application; and,
- some endpoints which supply their own inline authentication where the required credentials are passed as part of the request (see the REST API documentation for more details).

Login

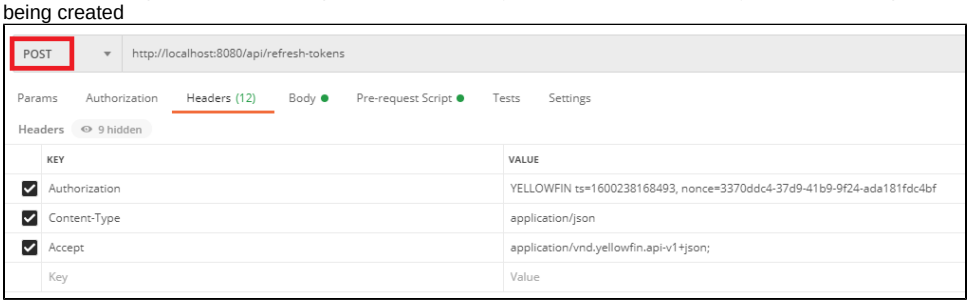
The REST API uses refresh tokens and access tokens to authenticate and authorize access to resources. These tokens can be generated in a few different ways.

Standard Login

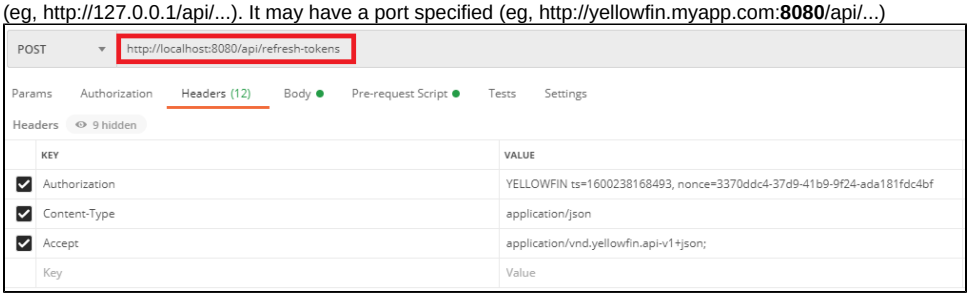
The standard login flow passes user credentials to the server to authenticate and generate a refresh token, and as a convenience also an access token. The refresh token can then be used to create access tokens, which can be used to access other resources.

Rather than a session, a refresh token is used to identify a user. A consumer must create a refresh token and obtain an access token before they can use other REST endpoints. Creating a refresh token can be thought of as a login process.

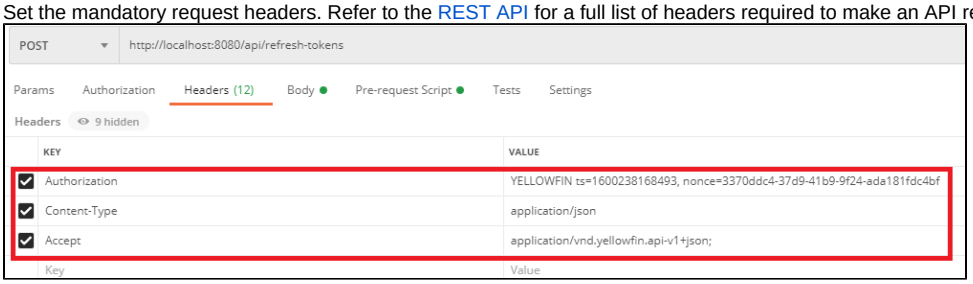
1. Use the HTTP operation POST. Requests that create any kind of resource will always use a POST operation. In this case, a refresh token is being created



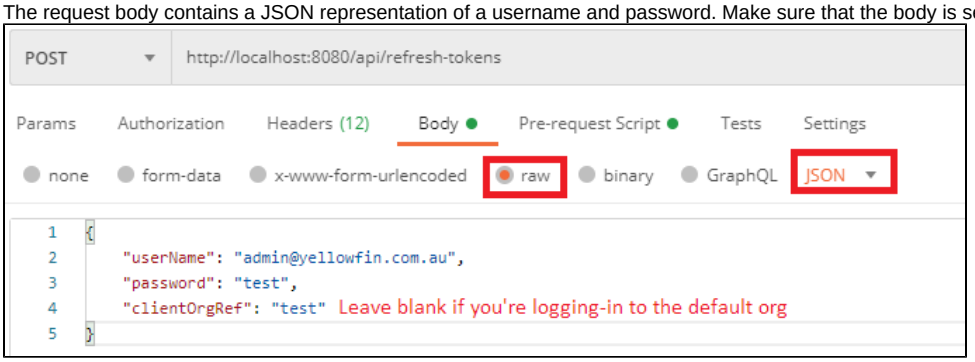
2. Enter the URL of the refresh token endpoint. A valid URL will always have either a name (eg, http://yellowfin.myapp.com/api/...) or an IP address (eg, http://127.0.0.1/api/...). It may have a port specified (eg, http://yellowfin.myapp.com:**8080**/api/...)



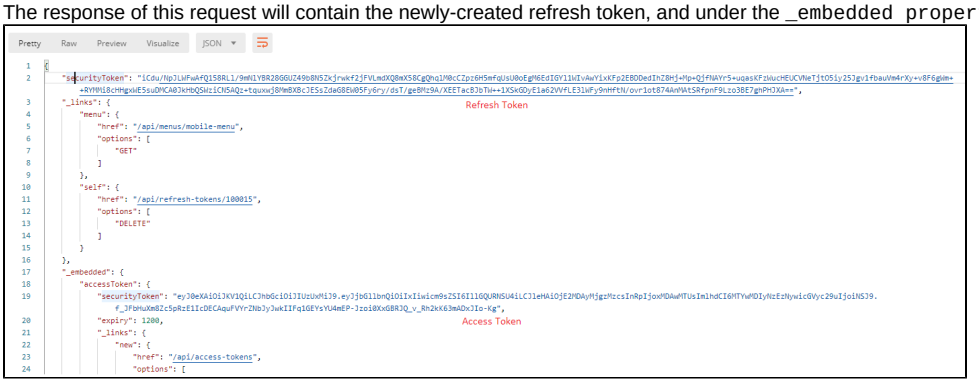
3. Set the mandatory request headers. Refer to the REST API for a full list of headers required to make an API request



4. The request body contains a JSON representation of a username and password. Make sure that the body is sent as raw JSON



The response of this request will contain the newly-created refresh token, and under the `_embedded property`, an access token.



The client application should securely store these tokens. It should also store the `"self"` link as it will be needed for logging out.

Single Sign-On

The API provides single sign-on (SSO) functionality within the REST API itself. This allows a REST user to logon another REST user, generating a refresh token for that user, which can be forwarded on to that user (or more likely another application) to allow use of the API. This endpoint supports inline authentication for the REST user who is making the SSO request, and supports simple authentication (see the Troubleshooting note below about SSO errors with noPassword SSO).

Onboarding

This is essentially another form of SSO that can be done from the web application. Administrators can generate onboarding tokens for specific users to be passed to an external application that can then pass that token into their authentication flow rather than requiring the user's credentials to generate a refresh token. If a token is passed with a refresh token POST request (and it's not a normal REST SSO request from above), then this token is verified against the token created in the web application to logon the user.

Web SSO

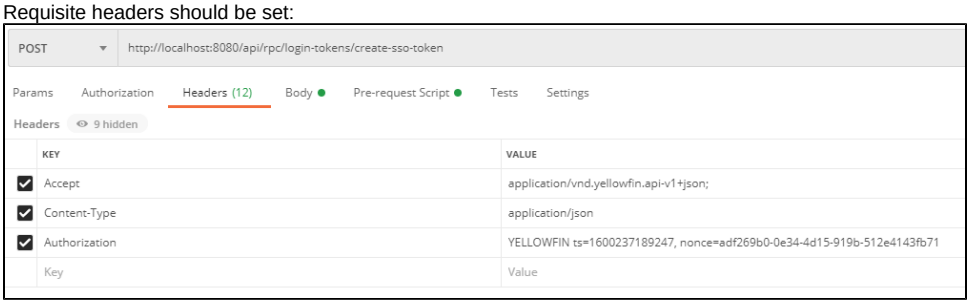
Both the web application and the JavaScript API use a different token system to handle single sign-on, which means that you cannot use these tokens interchangeably. Thankfully, the REST API provides several ways to generate the Web SSO tokens required (called login tokens in the REST API). There are three main ways to generate one of these tokens.

Generation method	Description
Generate a token for any other user while logged in	Passing a user's credentials in the request to /login-tokens will authenticate that user and create a Web SSO token for that user. This can be used by an admin account to logon other non-admin users.
Generate a token for the current logged-in REST user	This uses the same flow as above, but if no secondary user credentials are passed, then a token is generated for the current user instead. This is useful for integrations to redirect the current user to JS API or web content.

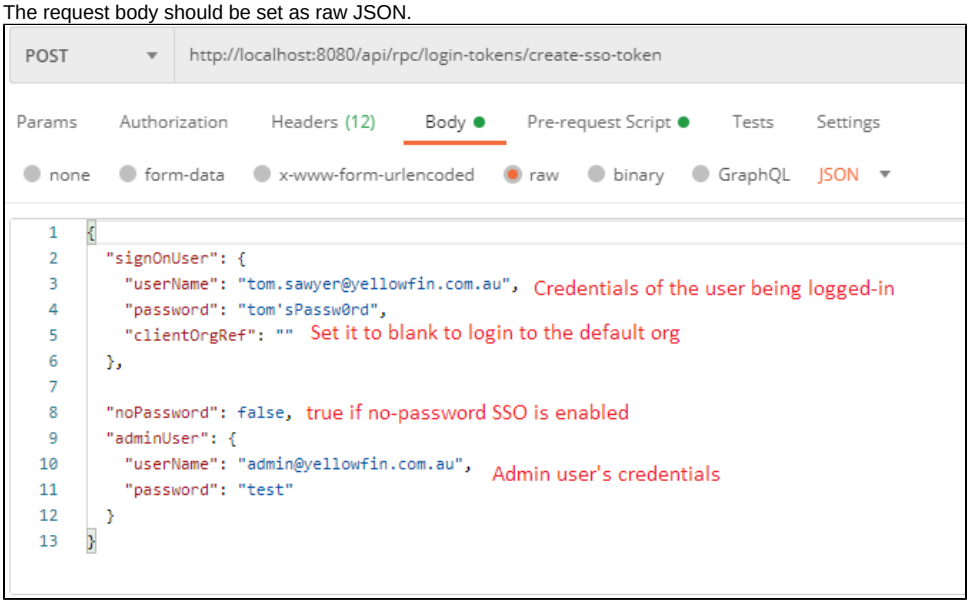
Generate a token for any other user while not logged in	Using the /rpc/login-tokens endpoint allows the use of inline authentication to pass both an admin user's credentials as well as the credentials of the user to be logged in.

A popular use-case for the API is Web SSO. A couple of API endpoints are available for generating a login token. The generated token can be used to login to Yellowfin's browser interface. The simplest way to do this is to use the RPC endpoint POST /login-tokens/create-sso-token.

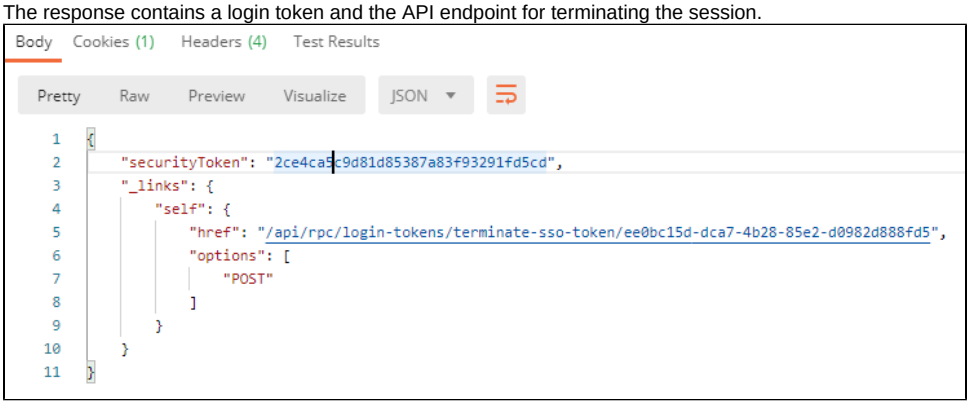
- Use the POST method and set the URL to /login-tokens/create-sso-token
- Requisite headers should be set:



- The request body should be set as raw JSON.



- The response contains a login token and the API endpoint for terminating the session.



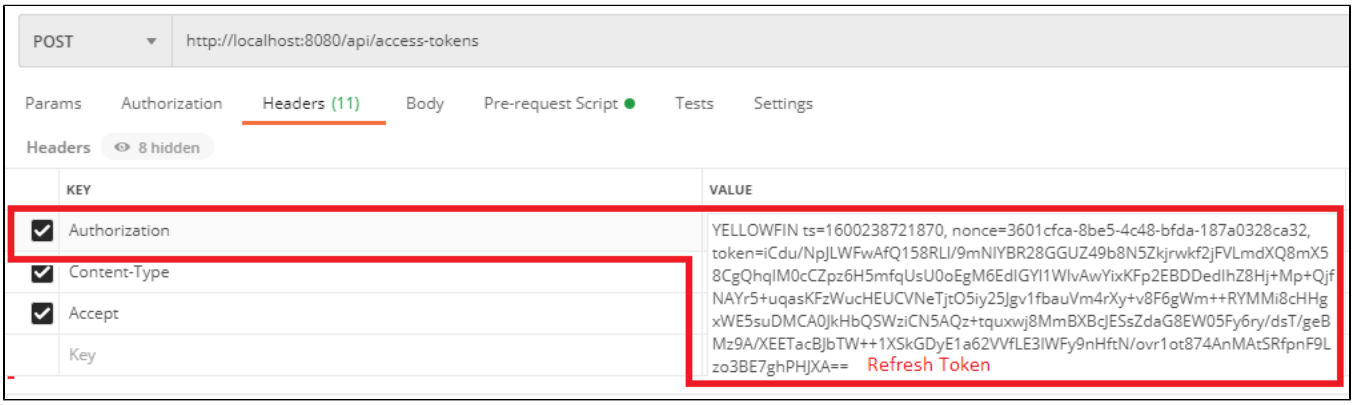
- The token may be used for logging into the Yellowfin Web UI or the JavaScript API. See Redirecting to Yellowfin with the Login Token.

Access Tokens

Creating an access token is a very similar process to creating a refresh token. To create one:

- use the POST operation
- use the URL of the access token endpoint

- use the same headers as the refresh token request
    - the Authorization header must specify a refresh token, with a property named token



The refresh token response provides an access token to make it easier to start consuming the API after login.

Logout

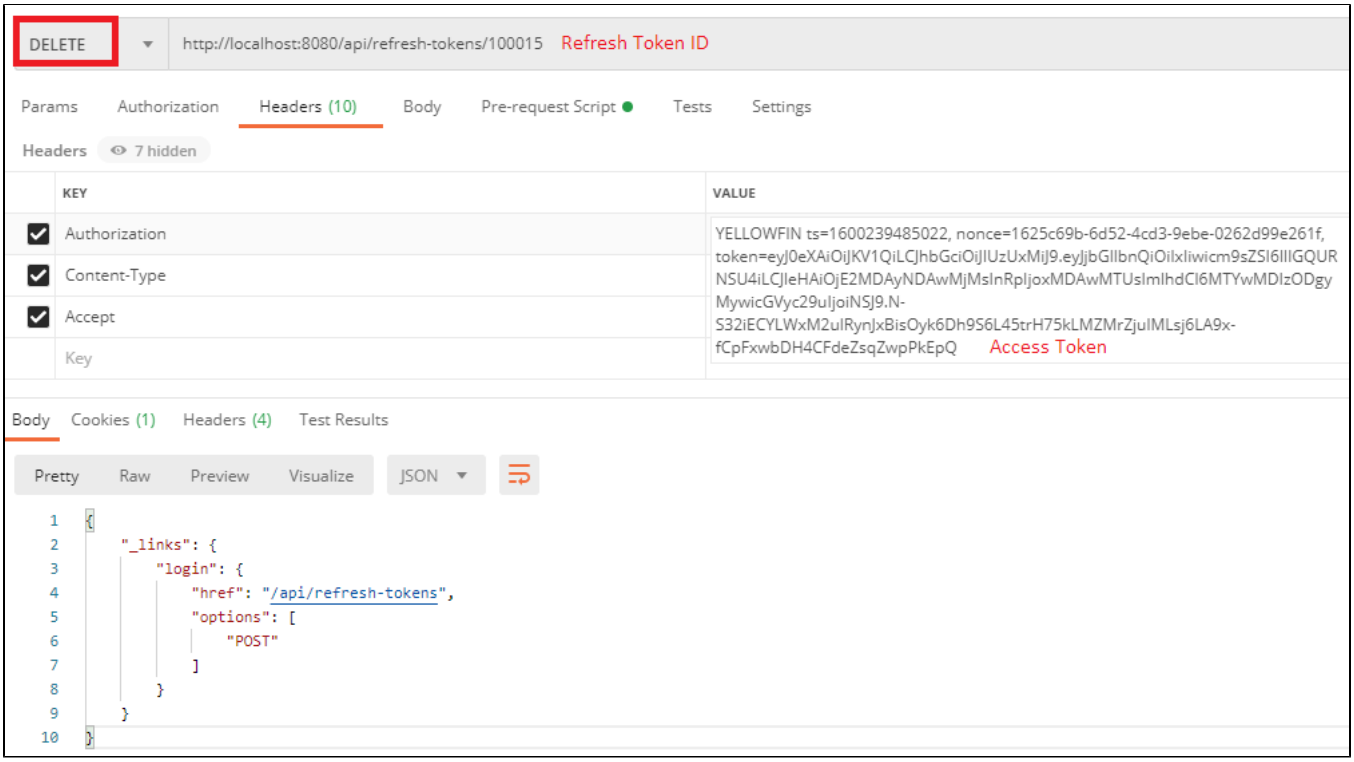
Any logon flow that results in a refresh token being generated can be logged out in the same way. Single log-off (SLO) can be achieved by passing another user's token ID to be deleted by the /refresh-tokens endpoint as long as the currently logged-in user has rights to do so.

It is also possible to SLO a Web SSO session by passing the token ID (returned when the login token was created) to the /login-tokens DELETE endpoint. The same access restrictions apply.

The response of the POST/refresh-tokens request will contain the information required to effectively "log out" of the REST API — a call to delete that refresh token. The response of the POST/refresh-tokens request contains a `_links` property.

The options array in the `"self"` link lists which operations can be performed on the new refresh token. There should only be one — `"DELETE"`. Calling DELETE /refresh-tokens will effectively log the user out of the REST API.

Note that a valid access token is required to perform this operation. It must be included in the `token` property of the Authorization header.



top

Accessing Resources

Please consult the API doc for the headers that need to be specified for each endpoint, along with mandatory and optional parameters. Please consult the API doc for the headers that need to be specified for each endpoint, along with mandatory and optional parameters.

Base API Resource

This endpoint describes the current state of the API itself, represented as a standard REST resource. It is available to both logged-in and non-logged-in users, so it can be accessed with or without an access token. It returns API information such as the current API version, server information such as the current application version (see note below). The endpoint also returns all the top-level resources that are available to the current user (if no user is logged in, it returns the top-level endpoints which can be accessed by a user who is not logged in).

> Sometimes available endpoints can differ between application versions for the same REST version, so it is important to pay attention to this value as well as the API version.

Standard Resources

Almost all standard resources require access token authentication, and are mostly represented as JSON (see the REST API documentation for endpoint specifics). These generally share a standard structure with object properties as well as `_links`, and `_embedded` properties for navigation and additional useful information.

Most list endpoints implement a filtering system that uses JSON-serialized objects to pass filtering information to the endpoint. These endpoints are also generally accompanied by a direct child /metadata resource which describes the filter values available to the current user. See the REST API documentation for each endpoint for more information.

There are several endpoints that provide a fields parameter, which can control which model fields are returned in the API response. These are sent as a serialized JSON list of property names that should be included. This feature is not uniformly implemented across all endpoints, and not all fields are able to be excluded on all endpoints. See the REST API documentation to find the details for each endpoint.

RPC Resources

RPC (remote procedure call) endpoints are similar to standard resource endpoints in that they follow the same rules for authentication (some RPC endpoints have inline authentication; see the REST API documentation) and response structures, but which do not follow the REST paradigm. Instead of representing a resource directly, these represent a procedure to be run on the server. All RPC endpoints use the POST operation.

The reason these exist is that they implement features that are very difficult to fit into the REST paradigm, such as endpoints which require stateful interactions, or complex multi-request call structures. These generally are features that have been migrated from existing SOAP services or the web application interface and are very difficult to refactor cleanly. Some of these features may be converted to a fully RESTful implementation in future.

top

Request Body Parameters

Most endpoints that have a POST operation available require passing parameters in the request body, unless they are simple enough that they do not require any additional parameters. There are two main situations which can cause the required format of the request body to differ:

- Requests that require application/form-data are generally requests that require file upload of some sort. This means that the body should be passed as form-data and that it should use the correct encodings. Any sub-objects that do not refer to file upload parameters should be encoded as raw JSON within the form-data.
- All other requests which don't have file upload parameters will use raw JSON in the request body.

Please see the REST API documentation for the exact requirements of each endpoint.

top

Login Flow Example

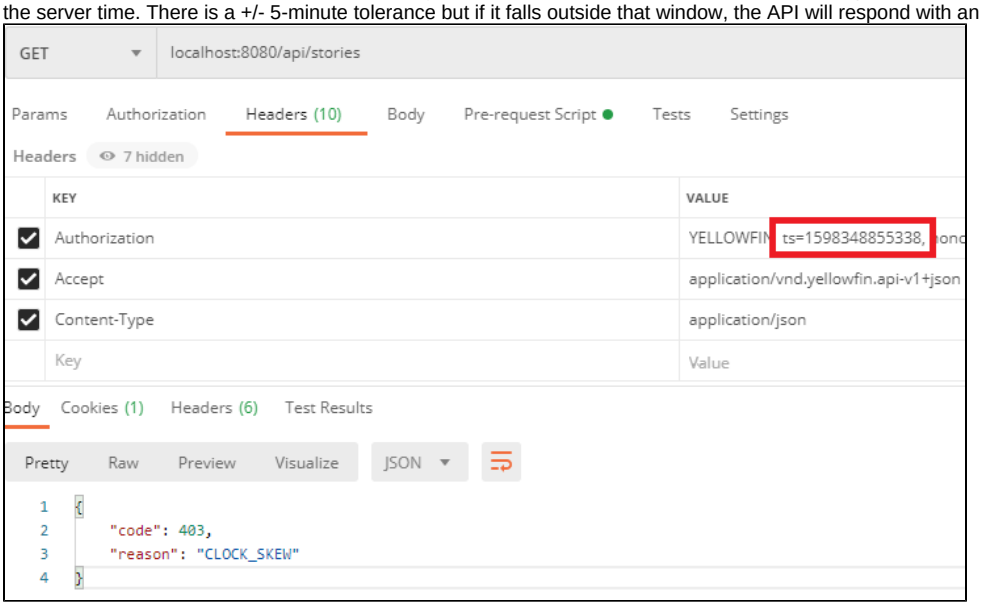
In general, most use cases of the API are going to need to follow a similar approach to logging in to the REST API. This approach constitutes the initial connection and user login to the REST API. Please note that subsequent connections will not need to generate another refresh token unless the token has been invalidated by a user- or administrator-forced logout. In the case of applications that support features that may be ahead of some servers, it's a good idea to check that the version of the server has not changed between sessions.

1. Probe the server for the API version.
    - This can be done by making a GET request to the base /api endpoint (see the Base API Resource section on this page). The result returned here will determine the API version and application version of the server, which allows the API consumer to interact with whichever supported versions they are using that fall within the server's supported version ranges.
    - Note that the base /api endpoint did not exist in versions 1.0 and 1.1. Versions 1.0 and 1.1 are essentially identical and should be almost 100% compatible (please see the documentation for each version for differences), so receiving a 404 response from this endpoint should indicate that the server is running one of these versions, and falling back to v1 will work in all cases except for the very minor changes from v1.1.
2. Generate a refresh token via any of the available methods (see the Login section on this page).
3. Use the automatically pre-generated access token from the refresh token response, or create your own if required (pre-generated token may be expired if no activity has happened for a while).
4. Now you can use the access token to access all of the API resources which require this kind of token.

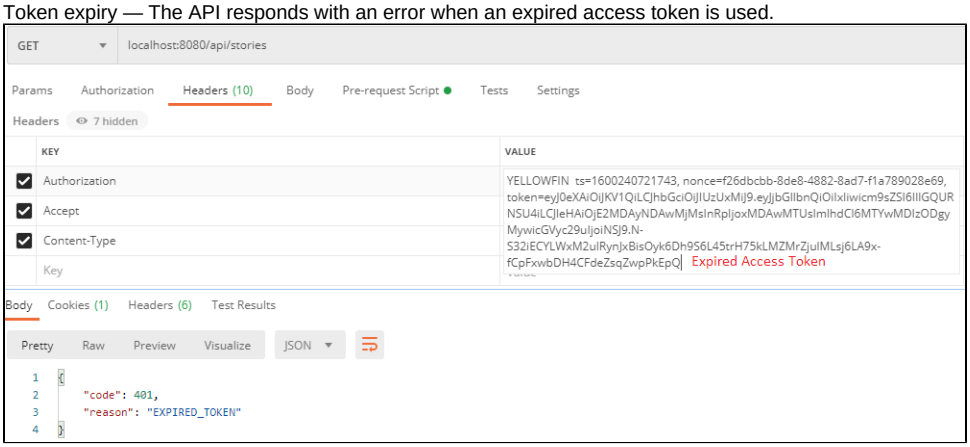
top

Troubleshooting

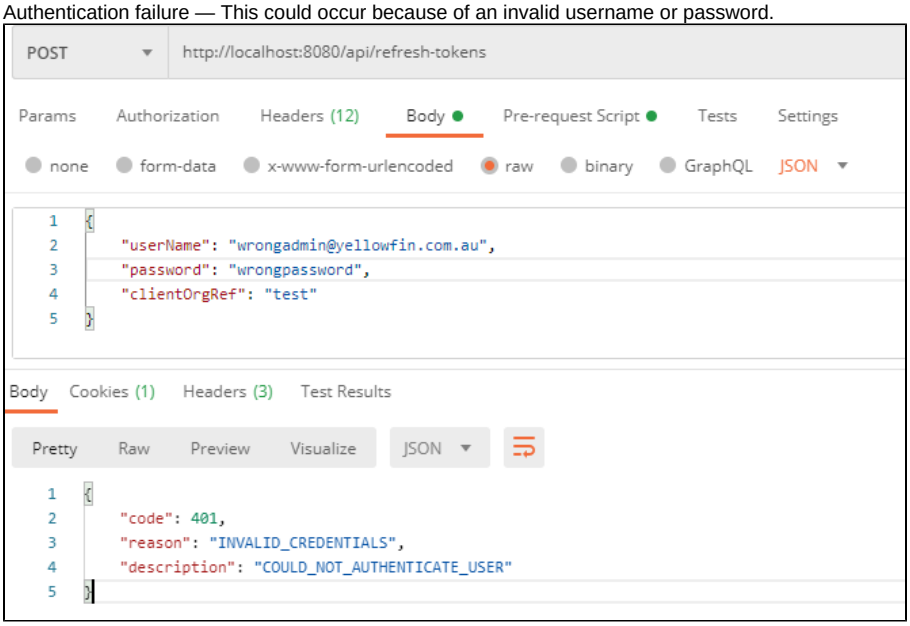
- Clock Skew — This is one of the most commonly-encountered errors. It is because the timestamp in the Authorization header is not in sync with the server time. There is a +/- 5-minute tolerance but if it falls outside that window, the API will respond with an error.



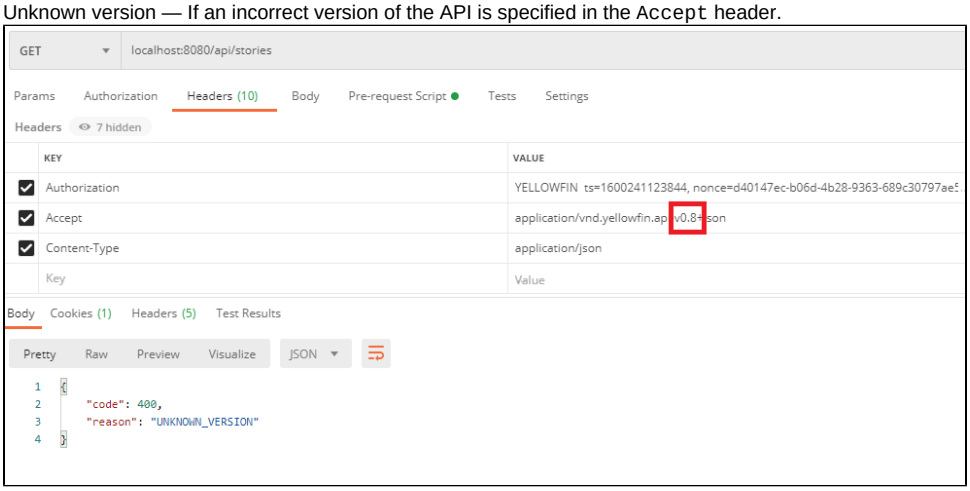
- Token expiry — The API responds with an error when an expired access token is used.



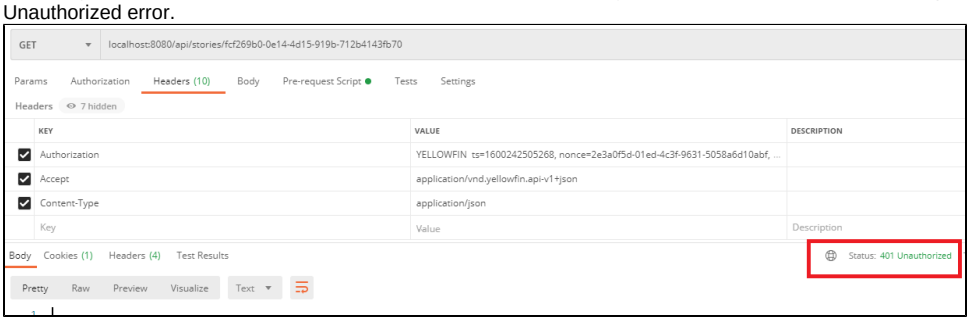
- Authentication failure — This could occur because of an invalid username or password.



- Unknown version — If an incorrect version of the API is specified in the `Accept` header.



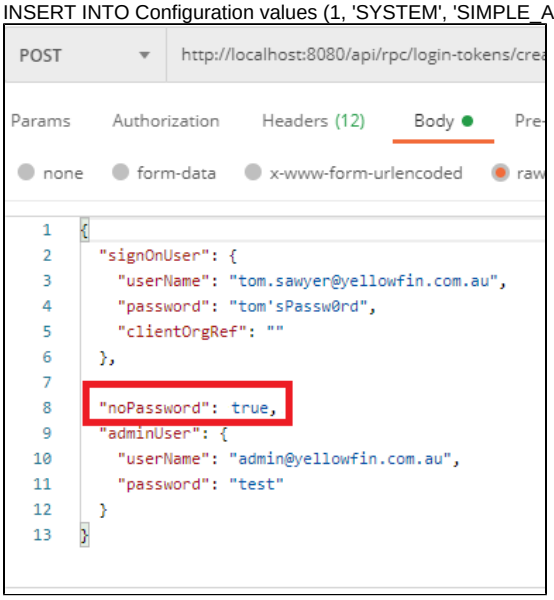
- Licensing error — Content services such as GET/stories/uuid, are only available when a server licence is present. If not, the API will return a 401 Unauthorized error.



- CORS — This is generally not a problem for the REST API because CORS applies only to browsers. A web browser is not a recommended REST client as it is not easy to securely store tokens.

- SSO Errors — Ensure that credentials and org reference are correct. If `noPassword` authentication is being used, ensure that it has been enabled on the server. This is done by inserting a record into the Configuration table **and restarting Yellowfin**.

  INSERT INTO Configuration values (1, 'SYSTEM', 'SIMPLE_AUTHENTICATION', 'TRUE');



- Error 500 Internal Server Error — This is a generic error message which indicates that something went wrong on the server. Contact support with the error trace in the server logs for more information.

The full documentation of the current REST services is available in our external developer site. Click here to access it.

top