Hero Lab® Online Public API

Version 1.2 – November 23, 2020

Introduction

Hero Lab Online (HLO) is the premier digital tool for character creation and management. Many gamers also make extensive use of other digital tools to further augment their experience, whether it be custom character sheet output, campaign management, and/or virtual tabletops. The goal of the Hero Lab Public API is to allow characters created within HLO to be shared with these other tools and enable a high degree of integration between them.

Building on various standards, the Public API makes it possible for client tools to directly retrieve character export data from HLO. This can be accomplished via one-time or infrequent snapshot retrieval of characters for simple purposes. Alternately, a client tool can subscribe to real-time notifications that report every change made to an array of characters for live updates.

In conjunction with this documentation, we've created two sample applications that demonstrate the basics of using the API, including real-time notifications. One sample is written in C# and the other in JavaScript. For more details on these samples, please see the appropriate appendix.

NOTE! A revision history of behavioral changes will be found at the end of this document.

Before You Dive Into the API

The Public API is fundamentally built around the HLO Export Format. All of the character data surfaced by the API is in that format, so it's imperative you familiarize yourself with that separate documentation. In fact, it's highly likely that a significant portion of your development time will be spent working with exported character data and mapping the HLO character model to the internal data model of your tool.

In light of this fact, we strongly encourage you to first determine how that process will work. To simplify the process, you can manually grab exports of various characters from within HLO, decoupling that effort entirely from integrating with the API. This approach will allow you to separately analyze the data format and develop the initial mapping to your tool.

Once you have that working, it should then be relatively straightforward to get some tokens and begin calling the API to retrieve character data. And at that point, the mapping will be in place, so you can immediately start viewing the extracted characters within your own tool.

NOTE! Documentation for the HLO Export Format is available at the URL below:

```
https://docs.herolab.online/HLO-Export-Format-Documentation.pdf
```

Conceptual Overview

The core design of the API centers on three different types of tokens. One token unlocks the API on behalf of a user. The second grants programmatic access to the API. And the third governs access to the actual data for specific characters and/or campaigns.

Connecting to the server is initiated with a ***user token***. Each user token identifies the user requesting the data from the API, along with other important details the API needs. Users must provide their user token to a tool they wish to employ, and that tool then accesses the HLO server on that user's behalf – as the user's agent.

When a client tool presents a user token to the API, it also introduces itself to the server. The API combines the two pieces of information and generates an ***access token*** that represents the combination of tool and user. This access token is then used to contact the HLO server and invoke the various endpoints that comprise the Public API.

Access to specific resources on the HLO server is achieved through ***element tokens***. These tokens behave as keys that unlock access to certain elements on the server, such as characters, cast members, and campaigns. By providing an element token to a digital tool, a user grants the tool access to that particular element. This allows users to selectively determine which elements are disclosed and even give different tokens to different tools for different purposes. For example, a player participating in multiple gaming groups might use the element token for one character in their Monday game and the element token for a separate character in their Friday game.

User Token	Provided by the HLO user for whom the tool serves as agent
Element Token	Provided by HLO users granting access to the characters/resources they control
Access Token	Acquired by the client tool to communicate with the API

NOTE! User tokens and element tokens are controlled by HLO users, and they can be revoked by the user at any time. Even if a token is entered into a tool and becomes inaccessible to the user within that tool, the user can revoke access to the token through HLO. Access tokens are acquired dynamically and used solely for communication between cllient tools and the API.

NOTE! Another way of picturing element tokens is like passwords. Each token is essentially a password that unlocks the specific element to which it applies (e.g. character, cast member, or campaign). By providing the password to another tool, that tool can then access the associated element. If a user changes their mind, they can reset the password and lock out any tool that was previously accessing the resource.

Practical Examples

The user token and element tokens are used in concert to leverage the API through a client tool. One user – often the GM – provides their user token to the tool, which the tool then presents to the HLO server, acting as the agent of the user. One or more users provide the element tokens for the characters (or campaign) that will be accessed by the tool. Once configured with the various tokens, the tool can acquire whatever information it chooses for those characters. Client tools access the API through the user token and unlock the characters via the element tokens.

A tool like a custom character sheet generator would only ever need a simple pair of user token and element token, and both tokens would usually come from the same user. In the case of a gaming group making use of HLO in conjunction with a virtual tabletop (VTT), the GM could provide their user token to the VTT, and the players could provide the element tokens for their respective characters to the VTT. The GM could additionally provide element tokens for NPCs and monsters, or the GM could simply supply a campaign token to grant access to everything within the HLO campaign. Once the tokens are in place, the VTT would have everything it needs to retrieve the data from the HLO server through the API.

Since none of these tokens expire naturally, setting up a client tool with the tokens only needs to occur once. After that, it can continue to be used indefinitely with full access to the unlocked characters through the API.

User Token

Utilizing the API starts with the ***user token***, which is required to access the services provided by the Public API. You can acquire a user token from within HLO.

A user token never expires and uniquely identifies the user that acquired it (hence the name). Consequently, we strongly encourage users to be mindful of how they handle the token and who they give it to. If your user token is compromised and used inappropriately, it will point back to you and potentially result in negative ramifications on your account. If you are concerned your user token may have been compromised, you can revoke it at any time and acquire a new one. And if you happen to misplace your user token, you can always retrieve it again through HLO.

NOTE! User tokens identify the user to the HLO server but contain no personal information and are ***not*** the same tokens used by HLO for interacting through a logged in account.

Element Tokens

Gaining access to actual content through the API requires at least one ***element token***. Each element token unlocks a specific resource within HLO, which can be a character, a cast member, or a campaign.

Element tokens behave as a key to the associated resource. Just like a house or car key, anyone with an element token can access the resource behind that token. Consequently, users should consider who they give an element token. Also like a house key, the same element token can be provided to multiple people and/or tools, granting each of them equal access through the API. There is only a single token for any HLO element, and it can be used within one or many tools.

Just like the user token, element tokens never change. Once you acquire an element token, retrieving it again will provide you with the identical token as before.

Once issued, an element token never expires. If you decide you don't want a particular element to be accessible any longer, you can revoke its token at any time. This terminates access to the element for all tools configured to use the token.

There are currently three kinds of element tokens that govern access to different resources within HLO:

Character Token	These tokens govern a specific character created outside of a campaign (also referred to as an *independent character*).
Cast Token	These tokens govern a specific cast member within a campaign. Regardless whether the cast member is on or off the stage, it can be accessed via this token.
Campaign Token	These tokens govern everything for an entire campaign. Currently, a campaign token only surfaces the PCs and any cast members that have been placed on the stage by the GM. Broader capabilities will be introduced in the future.

Accessing the API

Once you have a user token and at least one element token, you are ready to access the API. To keep things simpler at first, we recommend starting with either a cast member token or character token, as campaign tokens involve extra steps to utilize.

How It Works

Accessing the API is slightly more complicated than just using the user and element tokens. A tool must first acquire its own ***access token***, and it's through this token that the tool will communicate with the API. The tool presents the user token and identifies itself, then the API returns an access token that can be used to retrieve data from the API. The element tokens unlock the specific resources that can be retrieved.

The user token is a Java Web Token, which is a widely adopted standard (detailed fully at https://jwt.io). More specifically, the user token is a JWT ***refresh token***, and the refresh token is used to acquire the access token. The access token has a nominal lifespan, and a new access token must be acquired whenever the existing one expires. This regular process of refreshing the access token enables security checks to be easily performed at those intervals, such as detecting a revoked token.

The overall cycle operates in a very simple manner:

- A client tool is provided with a user token obtained from a user.
- The tool presents the user token to the API and acquires an access token.
- Until the access token expires, the tool includes the access token as part of subsequent API requests as proof of authentication.

- When the access token expires, the tool acquires a new access token via the user token and then resumes including the new access token as part of API requests.
- This cycle continues until the user token expires, at which point the tool must re-authenticate itself to update its user token. Currently, user tokens never expire, so there is no need to regularly authenticate.

NOTE! Using refresh and access tokens is a bit overkill at the moment. The API currently only offers read access, so there is no requirement for this approach. However, we're taking the long view and instituting the basic model from the beginning. In the future, tools will be able to make changes to HLO characters via the API, so we're laying the foundation now by instituting a suitable security model. When that capability becomes available, user tokens providing write access to content will require appropriate authentication (e.g. user login with password) and have an expiration. However, the overall refresh/access token model will remain basically the same.

Contacting the API

The HLO Public API can be reached through a collection of endpoints exposed through the domain api.herolab.online. All communication with the API requires a standard, encrypted connection, so the base URL is accessed in the form:

```
https://api.herolab.online
```

All of the API endpoints use the HTTP POST protocol, with both a JSON request and response body. To keep things simple and consistent for the diverse range of tools that may be using the API, tokens are passed in through the request body instead of via headers.

Summary of Available Endpoints

access/acquire-access-token Retrieves an access token for the provided user (refresh) token
access/verify-access-token Inspects the access token and reports whether it is valid and not expired
access/identify-game-server Retrieves the game server identifier for a particular game system
access/identify-notification-server Retrieves the host URL to use when contacting the service for a particular game system, along with the game server identifier
access/attach-game-server Establishes a relationship with the specified server, enabling subscriptions to the associated game system and abandoning any previously existing subscriptions to a different game server
access/unsubscribe-all Cancels all active subscriptions to the given game server
character/get Retrieves export data for a specific character
character/get-bulk Retrieves export data for a collection of characters
character/subscribe Registers the tool for real-time notifications covering a specific character
character/subscribe-bulk Registers the tool for real-time notifications covering a collection of characters
character/synchronize Sends the delta between the current state of a specific character on the client and its current state on the server; Allows a client tool to recover if it somehow gets out of sync with the server

character/synchronize-bulk Sends the delta between the current state of a collection of characters on the client and their current state on the server; Allows a client tool to recover if it somehow gets out of sync with the server
character/unsubscribe Stops the sending of real-time notifications for a specific character
character/unsubscribe-bulk Stops the sending of real-time notifications for a collection of characters
campaign/get-pcs Retrieves the collection of cast members that are PCs for a given campaign
campaign/get-stage Retrieves the collection of cast members that are currently on the stage for a given campaign
campaign/subscribe Registers the tool for real-time notifications encompassing cast members for the given campaign
campaign/unsubscribe Stops the sending of real-time notifications for a given campaign

API Requests

Every request submitted to the API uses an HTTP POST with a JSON body. Each request has one standard property that can optionally be specified, plus any number of additional properties that are specific to the actual request.

Standard Request Properties

callerId	(int; optional) Client-tool determined value that is parroted back in the response (detailed separately)

With the lone exception of the initial request to acquire the access token, every other request requires that the access token be included.

Additional Standard Request Properties

accessToken	(string) Access token acquired from the user token

API Responses

If an invalid endpoint is invoked or clearly bogus inputs are provided to an endpoint, the response will typically be a suitable HTTP error (e.g. 404). However, as long as the minimally necessary inputs are submitted to a valid endpoint, the response will be an OK status (200), and the response body will contain details of what occurred. If the inputs were invalid in some way, or if anything went wrong with the request, the response will indicate the nature of the problem. All endpoints return a JSON response body with a few standard properties that explain what transpired.

Standard Response Properties

callerId	(int) Whatever value was included in the request is parroted back in the response
result	(int) Result code identifying what happened with the request; See the master list of all result codes in an appendix for details of what each result code means
severity	(int) Severity level of the result, which will be one of a few set values; These are detailed in the same appendix with the result codes
error	(string; optional) If an error occurred, this will be a plain text explanation of what went wrong that will hopefully be helpful in correcting the problem; Omitted if the request was successful

Example

Below is an example response from a failed request:

```
{
  "result": 7110,
  "severity": 150
  “error”: “Invalid element token received”
}
```

Acquiring an Access Token

There is a single endpoint for acquiring the access token that is subsequently used in all other requests made to the API. This endpoint takes a user token and the tool name as input and provides an access token in response.

```
https://api.herolab.online/v1/access/acquire-access-token
```

The tool name is important, as it identifies the tool to the API. If there are multiple tools accessing the API from the same IP address, each tool can be distinguished and treated separately. In addition, as the API is expanded, the tool identity will become central to various future operations.

NOTE! The tool name should be chosen with some care. It should be something that is unique to the tool, so that another tool won’t inadvertently use the same value. In addition, the name should be stripped of any details that will change over time, such as a version number. Lastly, the tool name may be publicly displayed within HLO, so the name should distinguish the tool to HLO users.

Request Properties

refreshToken	(string) User token for which to retrieve the access token
toolName	(string) Name that uniquely identifies the tool to the API (50 characters maximum); Must remain constant for the life of the tool (e.g. does not include version numbers or other details that will change)
lifespan	(int; optional) Specifies a brief lifespan for the issued access token (in seconds); Values longer than a short period are rejected, making this only useful for testing purposes; Zero yields the default lifespan (as does omitting the property)

Response Properties

accessToken	(string) Access token issued for use with other API endpoints

Example

The request body should look something like the following:

```
{
  "refreshToken": "<user token goes here>",
  “toolName”: “some tool”,
  “lifespan”: 0
}
```

The response will look similar to this:

```
{
  "accessToken": "<access token appears here>",
  "callerId": 0,
  "result": 0,
  "severity": 1
}
```

Retrieving Element Data

You've now got your access token. It's time to do something with an element token and retrieve a character through the API. If you haven't done so already, please take the time to familiarize yourself with the HLO Export Format, since the character data you retrieve will be provided in that format.

Overview

Retrieving data is performed on one or more specific characters at a time, which can be either independent characters or cast members from a campaign. The initial retrieval must always be the full character, and that can be quite a lot of data. After the initial retrieval, a client tool should thereafter request only the changes that have occurred since the last time data was retrieved. These two methods are referred to as ***full retrieval*** and ***differential retrieval***, respectively.

Each character has a version. Every time a change is made to a character, its version is incremented. Whenever retrieval occurs, the current version of the character is included within the export data, so the caller knows what version it possesses. On the next retrieval, that version can be specified as the baseline in the request, and the API will return only the changes that have actually occurred since that version (if any). The net result is a much more efficient dynamic between the tool and the API.

When retrieving multiple characters, separate results are returned for each character. It's quite possible that some characters will have been changed while others have not, and every character will have its own version, so each character is clearly distinct in the response.

Frequent Polling is Actively Discouraged

An important detail to keep in mind is that retrieving the data for many characters can quickly put a heavy load on the server. Used appropriately, differential retrieval can help to mitigate that load, but it certainly doesn't eliminate it. Consequently, tools that frequently poll the API are actively discouraged by the API's design.

Whenever a request is processed, the number of characters and nature of the data actually retrieved (full or differential) determine how soon the caller is allowed to make its next request. This wait time is communicated back within the response so that the caller knows and can schedule its next request accordingly. If the caller does not wait at least the indicated interval before trying again, its subsequent request will fail. Additionally, the wait time will restart anew, which will further delay getting the next updates reported to the tool.

Retrieval of character data sporadically to sync up a client tool should work smoothly. Infrequent polling will work well for certain types of tools that don't require "live" updates. And polling will typically be the way most tools first begin their integration with the API. However, any tool that wants to stay updated on a steady basis will likely find that polling provides updates too slowly for a responsive user experience. For these kinds of tools, the API offers an alternative solution – the ability to subscribe to real-time notifications that report every change as it occurs to each character. For more details, please see the chapter on Real-Time Notification.

Differential Retrieval

When differential retrieval is performed, it's entirely possible that nothing has actually changed within a character. For example, the player is engaged in a roleplaying scene or has stepped away from the game for a time. This situation is reported in the response.

It's also possible that nothing *meaningful* has changed within a character since the last update. A classic example is when a player toggles an ability on and then off, yielding no net difference. In this case, the version has changed, but nothing else has, so the lack of any net change is reported within the differential export.

If sufficient time has elapsed between requests, or a player is rapidly making many changes to the character, yet another scenario can arise. More changes may accrue than are retained in the character's change history on the server, meaning that no differential export can be provided. When this occurs, the API simply returns a full export, and the baseline version indicated in the response is set to zero to indicate this event (i.e. it will not equal the baseline version submitted in the request).

Retrieving Cast Members via Campaign Tokens

Campaign tokens grant access to the various elements governed by the campaign, but a client tool needs to identify what those elements actually are. At the moment, there are two endpoints for that purpose, and more are planned for the future. One endpoint retrieves all the PCs for the campaign, while the other retrieves the list of cast members that are currently on the stage within HLO.

Once the list of cast members is known, their actual contents can be obtained via the character retrieval endpoints. This is achieved by specifying both the campaign token and cast member id. The campaign token grants access to the campaign, and the cast member itself identifies the specific character to be retrieved.

NOTE! Working with campaigns is really best handled through subscription notifications, since the caller is immediately informed whenever cast members are added to and removed from the stage.

Requesting a Single Character

Retrieving the export data for a single character utilizes the "get" endpoint. The request body specifies the character token or cast member token to retrieve, along with the ever-present access token. Optionally, the request can specify an individual actor to retrieve and/or the baseline version to be used for a differential export.

```
https://api.herolab.online/v1/character/get
```

Request Properties

accessToken	(string) Access token acquired from the user token
elementToken	(string) Either a character token or a cast member token to retrieve the export for
actor	(string; optional) Identifies the specific actor for which the export should be retrieved; If omitted or null, the export retrieves information for all actors within the character portfolio
baseline	(int; optional) Baseline version that a differential export should report changes since; If omitted or zero, the full export is retrieved for the character

Response Properties

wait	(int) Minimum interval that the caller must wait until making another retrieval request to the API (in milliseconds)
status	(int) Indicates the status of the requested character and the nature of the retrieved export data (if any); See the appendix for the values that can be returned
export	(string; optional) Export data for the character (or null/omitted if none is returned)

Example

An actual request body that retrieves a differential export relative to version 42 for the primary actor might look similar to the sample below:

```
{
  "accessToken": "<access token goes here>",
  “elementToken”: “$sDCoHBm~@Sa#”
..”actor”: “actor.1”,
  “baseline”: 42
}
```

The corresponding response will look similar to this:

```
{
  "wait": 1500,
  “status”: 1,
  “export”: “<export data goes here>”,
  "result": 0,
  "severity": 1
}
```

Requesting Multiple Characters in Bulk

Retrieving the data for multiple characters at a time is significantly more efficient and flexible. However, it also introduces a few extra wrinkles compared to retrieving a single character. The “get-bulk” endpoint is used.

```
https://api.herolab.online/v1/character/get-bulk
```

The most important difference is that bulk retrieval is the only way to retrieve characters using campaign tokens, and you will have to first obtain the cast members of the campaign for which you want to retrieve character data. Please see the next two sections for ways of accomplishing that.

The other big difference is that bulk retrieval submits a list of characters to retrieve, including optional actor and baseline version constraints. Correspondingly, the response returns a list of the characters with their details. The actual structure of each requested character is quite similar to retrieving a single character, with one additional property, and the request includes a list of these objects. The structure of each returned character is also similar to the response when retrieving a single character, with the response containing a list of these objects.

Individual Character Properties in Request

elementToken	(string) A character token, cast member token, or campaign token for which to retrieve export data
castId	(string; optional) Unique id of a cast member for which to retrieve export data; Only applicable when a campaign token is specified as the element token and must otherwise be null or the empty string; Id must have been obtained via a separate API endpoint that retrieves cast ids for a campaign token
actor	(string; optional) Identifies the specific actor for which the export should be retrieved; If omitted or null, the export retrieves information for all actors within the character portfolio
baseline	(int; optional) Baseline version that a differential export should report changes since; If omitted or zero, the full export is retrieved for the character

Request Properties

accessToken	(string) Access token acquired from the user token
characters	(array) Array of one or more objects with the property structure defined above

Individual Character Properties in Response

elementToken	(string) Identical token passed in for the corresponding requested character
castId	(string; optional) Identical token passed in by the request
status	(int) Indicates the status of the requested character and the nature of the reported export data; See the appendix for the values that can be returned
export	(string; optional) Export data for the character (or null/omitted if none is returned)

Response Properties

wait	(int) Minimum interval that the caller must wait until making another retrieval request to the API (in milliseconds)
characters	(array) Array of one or more objects with the property structure defined above

Example

An actual request body that retrieves a single character might look similar to the one below:

```
{
  "accessToken": "<access token goes here>",
  “characters”: [
    {
      “elementToken”: “$sDCoHBm~@Sa#”
      “castId”: null,
    ..”actor”: “actor.1”,
      “baseline”: 42
    }
  ]
}
```

The corresponding response might look similar to this:

```
{
  "wait": 1500,
  “characters”: [
    {
      “elementToken”: “$sDCoHBm~@Sa#”
      “castId”: null,
      “status”: 1,
      “export”: “<export data goes here>”
    }
  ],
  "result": 0,
  "severity": 1
}
```

Retrieving Cast Members on the Stage

Obtaining the cast members that currently reside on a campaign’s stage is accomplished via the “get-stage” endpoint.

```
https://api.herolab.online/v1/campaign/get-stage
```

The contents of the stage change infrequently and sporadically. Consequently, knowing when to retrieve the list of cast members is typically best left to the user to determine instead of any type of polling. When a GM starts a session and adds the PCs to the stage is a perfect time. So is when the GM enacts a script that puts a bunch of monsters and NPCs onto the stage for a new scene.

NOTE! Campaigns work exceptionally well when subscription notifications are employed, since the caller is immediately informed whenever cast members are added to and removed from the stage. You are strongly encouraged to leverage this alternate approach.

When retrieving the cast members on the stage, the request needs only the campaign token. The response returns the ids of the cast members currently on the stage, along with the wait interval.

Request Properties

accessToken	(string) Access token acquired from the user token
campaignToken	(string) Campaign token for which to retrieve the cast members currently on the stage

Response Properties

wait	(int) Minimum interval that the caller must wait until making another retrieval request to the API (in milliseconds)
castList	(array) Array of one or more strings that identify the ids of the cast members on the stage

Example

An actual request body that retrieves the cast members on the stage might look similar to the one below:

```
{
  "accessToken": "<access token goes here>",
  “campaignToken”: “$CDYw0vq~@Sa#”
}
```

The corresponding response might look similar to this:

```
{
  "wait": 1500,
  “castList”: [
    “$DDygM9w~@Sa#”
  ],
  "result": 0,
  "severity": 1
}
```

Retrieving Cast Members for PCs

A frequently fundamental need for client tools will be acquiring all of the PCs for the game session. When an HLO campaign is used, this is achieved via the “get-pcs” endpoint.

```
https://api.herolab.online/v1/campaign/get-pcs
```

Once all of the PCs are acquired through this call, they can be accessed when they are both on the stage and off the stage. This makes it possible to retrieve the characters even when the GM has not started a session that would normally make them available via the stage.

NOTE! All other cast members are currently limited to access only while they are on the stage. The only way to gain access while off-stage is to obtain an explicit cast member token for each cast member wanted in this manner.

Similar to the preceding endpoint, the request needs only the campaign token to obtain the PCs, and the response returns the ids of the corresponding cast members, plus the wait interval.

Request Properties

accessToken	(string) Access token acquired from the user token
campaignToken	(string) Campaign token for which to retrieve the PC cast members

Response Properties

wait	(int) Minimum interval that the caller must wait until making another retrieval request to the API (in milliseconds)

castList	(array) Array of one or more strings that identify the ids of the PC cast members

Example

An actual request body that retrieves the cast members for a campaign's PCs might look similar to the one below:

```
{
  "accessToken": "<access token goes here>",
  “campaignToken”: “$CDYw0vq~@Sa#”
}
```

The corresponding response might look similar to this:

```
{
  "wait": 1500,
  “castList”: [
    “$DDygM9w~@Sa#”
  ],
  "result": 0,
  "severity": 1
}
```

Real-Time Notification

Notifications in real-time are the optimal way of integrating with HLO's API. The client tool tells the API what elements it wants to monitor, then sits back and waits. Whenever a user takes an action or something occurs within the HLO server that the client may care about, the API sends a notification with the details. There's no polling and no wasted communications. It all happens on a just-in-time basis.

Basic Workflow

The preceding example is a bit simplified, but accurate. The basic model centers on the client tool setting up a real-time notification channel with the HLO server using SignalR. Once that's in place, the client attaches to the appropriate game server and then subscribes to notifications pertaining to the explicit set of element tokens it cares about. When operations are performed upon the resources governed by those tokens, a notification is sent to the tool with the details. The tool then determines whatever actions are appropriate based on the data received.

When a character first appears to a subscribed client, the API will bring the client up to date. Thereafter, all changes made to the character are reported as they occur, allowing the client to incorporate them. If a client gets out of sync for some reason, it can re-synchronize itself very easily.

When a campaign is subscribed, additional notifications enter the mix. Any time new characters are added to the stage, the client is informed. The same applies whenever existing characters are removed from the stage. The net effect is that the subscribed tool is kept in perfect sync with whatever actions are occurring in HLO.

SignalR

The technology underlying the API's real-time notification is SignalR, which is an open-source library developed by Microsoft that is specifically targeted for real-time web-based notifications. It's built on WebSockets, handles connection management automatically, and gracefully degrades to less-desirable transport methods, as needed. SignalR also provides JavaScript and .Net client implementations, so the vast majority of tools will be able to integrate with it quickly using the provided packages. We've created sample clients in both C# and JavaScript to demonstrate this capability (see the appendix).

Using SignalR entails setting up a connection with the HLO server. Once established, SignalR allows the API to send messages to the client tool whenever it chooses. The connection is automatically maintained. It also fully supports authentication using the same access token acquired to make requests of the API.

The SignalR connection is used exclusively for sending notifications to the client tool. All requests submitted to the API are performed via the assorted REST endpoints via HTTP POST.

Full documentation on SignalR will be found at the URL below. HLO is built upon ASP.NET Core 2.2, so make sure all client behaviors are compatible with that version.

```
https://docs.microsoft.com/en-us/aspnet/core/signalr?view=aspnetcore-2.2
```

Before Connecting to SignalR

Before you can connect to SignalR, you first need to know what URL to utilize. HLO utilizes multiple servers, and different game systems are accessed through different servers. Normally, this is all completely transparent to users. It's also completely transparent to all of the REST endpoints of the API. Unfortunately, it's not transparent when connecting to SignalR.

We must be able to change and adapt how we manage the servers behind the scenes, but we also need to ensure that we never break tools that integrate with the API. We've therefore created an endpoint whose sole purpose is to identify the URL to be used to connect to SignalR for a particular game system.

```
https://api.herolab.online/v1/access/identify-notification-server
```

This endpoint simply needs to know one of the element tokens that the client tool will be subscribing to. Based on this initial information, the appropriate server can be identified and reported back to the client.

Request Properties

accessToken	(string) Access token acquired from the user token
elementToken	(string) Any element token that will be subscribed to for notifications

Response Properties

host	(string) Host domain that must be used to connect to SignalR for the given element token
gameServer	(string) Identifier associated with the game server for the element token, which can then be attached to and subscribed

Example

A sample request body retrieving the host to use for connecting to SignalR will look like the following:

```
{
  "accessToken": "<access token goes here>",
  “elementToken”: “$CDYw0vq~@Sa#”
}
```

The corresponding response might look similar to this:

```
{
  "wait": 1500,
  “host”: “sf-api.herolab.online”,
  “gameServer”: “@Sa#”,
  "result": 0,
  "severity": 1
}
```

NOTE! The “gameServer” property can also be used in conjunction with the various endpoints for monitoring server status, which are detailed in an appendix.

Connecting to SignalR

Once the host is identified, connecting to SignalR is simply a matter of synthesizing the appropriate URL. The SignalR endpoint is named “api-signalr”, and SSL communications are required, so the constructed URL might look something like the one below:

```
https://sf-api.herolab.online/api-signalr
```

The above URL would then be passed in when configuring the connection to SignalR.

The actual process of connecting to SignalR requires using one of the client packages that Microsoft provides for that express purpose. Please refer to the appropriate documentation for using the client package. The accompanying sample client applications can also be used as a guide.

NOTE! After the SignalR connection is established, everything ought to “just work”. In case there are issues, the API provides a few endpoints over the SignalR connection that can be used to incrementally verify all aspects of the connection. These are detailed in an appendix.

Reconnection Protocol, Part 1

A variety of common situations will result in the SignalR connection being dropped, and the SignalR mechanism makes it easy to reconnect. However, there is one situation where reconnecting won’t be possible and where SignalR does a poor job handling it. This occurs whenever the access token expires or otherwise becomes invalid.

The way this scenario plays out is that SignalR simply drops the connection. It provides no feedback whatsoever regarding why the connection was dropped – it simply terminates it. Any attempt to reconnect will also fail without any explanation. Consequently, the following protocol should be followed whenever a SignalR connection is dropped and an attempt is made to reconnect.

- The first step should always be to verify the token via the special endpoint provided for this purpose (see below).
- If the access token ***is not*** valid, acquire a new access token.
- If the access token ***is*** valid, the failure is due to some other reason (e.g. internet connection issues).
- After assessing the situation and potentially acquiring a new access token, attempt to reconnect with any new token.

Assessing whether the access token is valid can be achieved via the "verify-access-token” endpoint.

```
https://api.herolab.online/v1/access/verify-access-token
```

Request Properties

accessToken	(string) Access token acquired from the user token

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below simply provides the access token for verification.

```
{
  "accessToken": "<access token goes here>"
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Reconnection Protocol, Part 2

The previous section covered the logic necessary to reestablish the SignalR connection itself, but there's more to the problem than just that. During the lapse in connection, it's quite possible that the client tool will miss some number of notifications. More dramatically, when the connection drops, it is also possible that all subscriptions established by the client tool will be completely lost. For example, when the HLO server recycles, everything is reset, and no knowledge of the previous subscriptions will exist on the server.

What's a client tool to do about all this? Fortunately, the solution is pretty simple. Whenever a client tool reconnects to the server, it should simply re-attach and re-subscribe to everything. The client knows what element tokens it was subscribed to, and it knows the current state it possesses for each character. Armed with that knowledge, the client simply attaches to the same game server and subscribes to all the same tokens, specifying the versions it possesses. If a campaign subscription is employed, the client is informed of the current version of all characters and can synchronize itself as needed to acquire any missed notifications.

To accommodate the above solution, the API has no difficulty with a client attaching to a game server to which it is already attached, nor with subscribing to a token that is already subscribed. In fact, if the client subscribes with a baseline version that is not up-to-date, the server will implicitly treat it as a synchronize request to bring the client current.

So there are three new bullets to add to the sequence outlined in the previous section, which should be performed once the connection is reestablished:

- Re-attach to the game server for the previously held subscriptions.
- Re-subscribe to all previously held subscriptions, providing the current version possessed for each character.
- When a campaign subscription is employed, after receiving the notification about which characters are on the stage, synchronize any characters for which the client has become out-of-date.

Testing SignalR Connection Handling

During client tool development, the ability to efficiently test and debug the SignalR connection logic will have a meaningful impact on the process. Access tokens don't expire for an hour or two, so a developer would normally need to setup a fair amount of instrumentation to properly test the connection. However, if an access token could be acquired with a very short lifespan (e.g. a few seconds or minutes), most or all of that instrumentation would be unnecessary.

This is exactly why the "lifespan" property was included as part of the "acquire-access-token" endpoint – to make the connection testing significantly easier. The "lifespan" property allows a very short expiration time to be specified, which is then embedded into the issued access token. When testing out the handling of access token expiration and the loss of a connection, this property can be leveraged to trigger expiration after only a brief delay.

Subscribing and Controlling Notifications

There are a variety of endpoints whose purpose is to tell the API what information to monitor and what notifications should be sent. This chapter outlines each of those endpoints.

NOTE! The following restrictions apply to all client subscriptions:

- Subscriptions of any form are not allowed until a client tool has established a SignalR connection through which to receive notifications.
- Only a single game system can be subscribed to at any time. An attempt to subscribe to a token from a second game system will be rejected.
- Before subscriptions to element tokens can be established, the client must attach to the appropriate game server.

Attach to the Game Server

As the first operation that a client tool will perform after the SignalR connection is in place, the “attach-game-server” endpoint designates the game system server to which it will be submitting further element token subscriptions. Once a client has attached itself to a game server, subscriptions to element tokens of the corresponding game system become enabled.

```
https://api.herolab.online/v1/access/attach-game-server
```

If the same game server is re-attached while subscriptions already exist to that game server, no changes occur. However, if a new game server is attached while subscriptions exist to a different game server, all of those subscriptions are immediately cancelled.

The only property that is needed for this endpoint is the desired game server, and the identifier for that is returned as part of the “identify-notification-server” response.

Request Properties

accessToken	(string) Access token acquired from the user token
gameServer	(string) Identifier for the game system server to be subscribed

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below attaches the indicated game server.

```
{
  "accessToken": "<access token goes here>",
  “gameServer”: “@Sa#”
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Subscribing to a Single Character

In the same manner as with the “get” and “get-bulk” endpoints, there are two endpoints for subscribing to notifications for character and cast member tokens. The “subscribe” endpoint subscribes to a single element.

```
https://api.herolab.online/v1/character/subscribe
```

NOTE! Subscriptions to characters might ***not*** immediately send a notification with details about the character, even if the client tool is out-of-date. Notifications are only sent when a character is actually loaded into HLO for use. Consequently, if the character is not loaded, nothing will be sent. The moment that the user triggers the load to occur, the subscriber will then be notified and brought up to date.

When re-subscribing after a connection is temporarily lost, it will frequently be the case that the client tool already has a recent version of the character. It's possible the character is out-of-date, but the tool only needs changes since the version it possesses. Consequently, the client should specify whatever version it knows about for the character.

Request Properties

accessToken	(string) Access token acquired from the user token
elementToken	(string) Either a character token or a cast member token to be subscribed
baseline	(int; optional) Baseline version that an initial synchronization should report changes since; If omitted or zero, the client is assumed to possess nothing for the character

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below establishes a subscription where the client possesses version 42 of the character.

```
{
  "accessToken": "<access token goes here>",
  “elementToken”: “$sDCoHBm~@Sa#”,
  “baseline”: 42
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Subscribing to Multiple Characters

When the client tool needs to subscribe to more than one character and/or cast member, it's much more efficient to simply submit them all together via the “subscribe-bulk” endpoint.

```
https://api.herolab.online/v1/character/subscribe-bulk
```

NOTE! Unlike the “get-bulk” endpoint, this one is limited to character tokens and cast member tokens. There is a separate endpoint for subscribing to a campaign.

Just like the “subscribe” endpoint above, only two properties are necessary for each character being subscribed: the element token and baseline version.

Individual Character Properties in Request

elementToken	(string) Either a character token or cast member token to be subscribed
baseline	(int; optional) Baseline version that an initial synchronization should report changes since; If omitted or zero, the client is assumed to possess nothing for the character

Request Properties

accessToken	(string) Access token acquired from the user token
characters	(array) Array of one or more objects with the property structure defined above

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

An actual request body that subscribes to a single character might look similar to the one below:

```
{
  "accessToken": "<access token goes here>",
  “characters”: [
    {
      “elementToken”: “$sDCoHBm~@Sa#”,
      “baseline”: 42
    }
  ]
}
```

The corresponding response might look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Subscribing to a Campaign

Only a single campaign can be subscribed at one time, and it’s accomplished via the separate “subscribe” endpoint for campaigns.

```
https://api.herolab.online/v1/campaign/subscribe
```

Subscribing to a campaign entails simply identifying the campaign token. Everything else occurs in conjunction with the campaign.

Immediately upon subscribing, a notification is sent that identifies all of the cast members that are currently on the stage, including their basic details. This gives the client placeholder information and the version so that the client can determine whether it already has the necessary data for each character. If not, then the client can use the “synchronize” endpoint to retrieve whatever information it is missing. Thereafter, all changes to the stage and the characters on it are reported via notifications.

Request Properties

accessToken	(string) Access token acquired from the user token
campaignToken	(string) Campaign token to be subscribed

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below establishes a subscription to a campaign.

```
{
  "accessToken": "<access token goes here>",
  “campaignToken”: “$CDYw0vq~@Sa#”
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Synchronizing a Single Character

There may be times when a client tool gets out-of-sync with the HLO server for a particular character. If this occurs, the "synchronize" endpoint should enable the client tool to get back into sync.

```
https://api.herolab.online/v1/character/synchronize
```

Synchronizing is similar to the "get" endpoint that retrieves all changes to a character relative to a baseline version. By specifying the version known by the client, only the changes are reported to get the client back up-to-date. The one major difference from the "get" endpoint is that the results are delivered via a notification instead of being returned directly.

Request Properties

accessToken	(string) Access token acquired from the user token
elementToken	(string) A character token, cast member token, or campaign token for which to synchronize the export data
castId	(string; optional) Unique id of a cast member for which to synchronize the export data; Only applicable when a campaign token is specified as the element token and must otherwise be null or the empty string; Id must have been obtained via a separate notification that discloses cast ids for a campaign token
baseline	(int; optional) Baseline version that the synchronization should report changes since; If omitted or zero, the full export is retrieved for the character

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below requests synchronization of the cast member for a campaign relative to baseline version 42.

```
{
  "accessToken": "<access token goes here>",
  "elementToken": "$CDYw0vq~@Sa#",
..”castId”: “mWB5zq4f”,
  “baseline”: 42
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Synchronizing Multiple Characters

If a client tool must synchronize multiple characters at the same time, the "synchronize-bulk" endpoint can be used.

```
https://api.herolab.online/v1/character/synchronize-bulk
```

Just like the "synchronize" endpoint above, three properties are provided for each character being synchronized: the element token, optional cast id, and baseline version. The results are delivered via a subsequent notification instead of being returned directly.

Individual Character Properties in Request

elementToken	(string) A character token, cast member token, or campaign token for which to synchronize the export data

castId	(string; optional) Unique id of a cast member for which to synchronize the export data; Only applicable when a campaign token is specified as the element token and must otherwise be null or the empty string; Id must have been obtained via a separate notification that discloses cast ids for a campaign token
baseline	(int; optional) Baseline version that the synchronization should report changes since; If omitted or zero, the full export is retrieved for the character

Request Properties

accessToken	(string) Access token acquired from the user token
characters	(array) Array of one or more objects with the property structure defined above

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below requests synchronization of the cast member for a campaign relative to baseline version 42.

```
{
  "accessToken": "<access token goes here>",
  “characters”: [
    {
      “elementToken”: “$CDYw0vq~@Sa#”,
    ..”castId”: “mWB5zq4f”,
      “baseline”: 42
    }
  ]
}
```

The corresponding response might look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Unsubscribing a Single Character

As the counterpart of subscribing, the “unsubscribe” endpoint removes a subscription to a single character or cast member token.

```
https://api.herolab.online/v1/character/unsubscribe
```

Request Properties

accessToken	(string) Access token acquired from the user token
elementToken	(string) Either a character token or a cast member token to be unsubscribed

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below abandons a subscription of a character.

```
{
  "accessToken": "<access token goes here>",
  “elementToken”: “$sDCoHBm~@Sa#”
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Unsubscribing Multiple Characters

When multiple character and/or cast member tokens need to be unsubscribed, submit them all together via the "unsubscribe-bulk" endpoint.

```
https://api.herolab.online/v1/character/unsubscribe-bulk
```

Request Properties

accessToken	(string) Access token acquired from the user token
characters	(array) Array of one or more strings that identify the ids of the character and/or cast member tokens to unsubscribe

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below unsubscribes a character.

```
{
  "accessToken": "<access token goes here>",
  "characters": [
    "$sDCoHBm~@Sa#"
  ]
}
```

The corresponding response might look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Unsubscribing a Campaign

If there's a need to unsubscribe a campaign, there's an "unsubscribe" endpoint available.

```
https://api.herolab.online/v1/campaign/unsubscribe
```

Request Properties

accessToken	(string) Access token acquired from the user token
campaignToken	(string) Campaign token to be subscribed

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below unsubscribes a campaign.

```
{
  "accessToken": "<access token goes here>",
  “campaignToken”: “$CDYw0vq~@Sa#”
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Unsubscribing Everything

When a client tool no longer needs to subscribe to anything, such as before it exits, it should cancel all of its subscriptions. This is accomplished via the “unsubscribe-all” endpoint, which unsubscribes character tokens, cast member tokens, and campaign tokens, alike.

```
https://api.herolab.online/v1/access/unsubscribe-all
```

Request Properties

accessToken	(string) Access token acquired from the user token
gameServer	(string) Identifier for the game system server to be fully unsubscribed

All subscription- and notification-related requests report everything through notifications. As such, there are no response properties other than the standard ones for every request.

Example

The request body below fully unsubscribes all subscriptions possessed by a client.

```
{
  "accessToken": "<access token goes here>",
  “gameServer”: “@Sa#”
}
```

The corresponding response will look similar to this:

```
{
  "result": 0,
  "severity": 1
}
```

Notifications Received

Once a client tool has setup a subscription to an element token, various notifications will start being sent. This chapter details each of the notifications that might be received by a client tool.

General Structure

All notifications are unified under a single umbrella within the API. A single method within the client tool is invoked for all the different notifications. This method receives two parameters. The first parameter identifies the specific notification received. The second parameter is the payload, which varies and depends entirely on the notification.

The type of notification (first parameter) is sent as an integer value. This integer will be one of the values defined within the Api_Notify enumerated set, for which the possible values will be found in an appendix.

The payload is always sent as a serialized JSON object. The actual object depends entirely on the notification type, and every notification has its own object structure. The object can be deserialized and accessed once its type is known.

Consequently, the client tool behavior should be to first inspect the notification type (first parameter). Based on that value, determine the proper object type for the payload, and finally deserialize the payload into the actual object.

Character Loaded

Whenever a monitored character is loaded into HLO through the actions of a user, the client tool is sent this notification. If a character is already loaded when the client subscribes to it, this notification is also sent. The net behavior is that any time a character is "new" to the client, it will receive this notification.

The notification identifies the character and its current version, allowing them to be inspected before consuming the export data. It also provides the full export data for the character.

Notification Properties

charId	(string) Unique id of the character; This is the same value within the export data
current	(int) Current version of the character; This is the same value within the export data
export	(string) Full export data for the character

Character Update

Any time a characters is edited within HLO, the client tool is sent this notification. Every time the version changes, a separate notification is sent with the effects of that one change.

If a client tool gets out-of-sync with the server and invokes the "synchronize" or "synchronize-bulk" endpoint, a character update notification is sent with the necessary details to bring the client tool up-to-date. The "status" property of the response is only really applicable in this scenario, since it will always be a "delta" in the standard case.

Notification Properties

charId	(string) Unique id of the character; This is the same value within the export data
current	(int) Current version of the character; This is the same value within the export data
status	(int) Nature of the character, which impacts the export data reported; See the appendix for details (Change_Status)
export	(string; optional) Export data for the character, the nature of which is given by the status

Characters Appear on Stage

Whenever a campaign subscription is in place, the state of the stage is reported. When new characters appear on the stage, this notification is sent.

Once characters are added to the stage, they begin being loaded for use by the server. When each character is finally loaded, the client tool will receive a "load" notification with the full details.

However, if the number of characters is large and/or the server is under load, the process of loading all the characters may take some time to complete. While it's waiting, the client tool may want to provide some initial feedback to the user to accurately reflect the new characters being added within HLO. That's where this notification comes in.

Basic information for each of the characters added to the stage is reported. This allows the client tool to create placeholders that identify the new characters to users. Since some characters will comprise more than one actor (e.g. animal companions or drones), all of the actors for each character are included, as well. So the overall structure is a list of characters, with a list of actors provided for each character.

Actor Properties

actor	(string) Unique id of the actor; This is the same value that will appear within the export data when the character load notification arrives
name	(string) Display name of the actor, allowing for use in a placeholder

Character Properties

charId	(string) Unique id of the character; This is the same value that will appear within the export data when the character load notification arrives
current	(int) Current version of the character; This is the same value within the export data
jersey	(int; optional) Jersey number assigned to the character, which is used to distinguish between multiples of the same character (e.g. bandit #1, bandit #2, etc.)
actors	(array) Array of one or more objects with the actor property structure defined above

Notification Properties

characters	(array) Array of one or more objects with the character property structure defined above

Characters Disappear from Stage

When a campaign subscription is in place and characters disappear from the stage, this notification is sent. All it consists of is a list of the characters that have been removed.

Notification Properties

characters	(array) Array of one or more strings containing the id of each character that disappeared

Stage Contention

Certain operations on campaigns relating to the stage may take a non-trivial amount of time to complete. During these operations, all of the characters involved are temporarily locked to ensure the integrity of data is maintained. These intervals are referred to as ***stage contention***.

Periods of stage contention may be important to some clients. Contention does not preclude subscription-oriented requests from being submitted. However, contention may defer their completion until after the lock is released. To keep the client tool apprised, this notification is sent whenever the contention state is entered or exited for a subscribed campaign.

Notification Properties

campaignToken	(string) Campaign token for which the contention state has transitioned
isContention	(bool) Whether the campaign is now in or out of contention

Shutdown Initiated

Through the normal course of events, the HLO server will temporarily go offline. Deployments of new features necessitate this, and the server also performs a “recycle” operation once every night that clears its state for a fresh start.

Users of HLO are warned of these events through the application. Client tools are afforded a similar heads-up when server shutdown has begun through this notification.

Notification Properties

nature	(int) Nature of the shutdown; See the appendix for details (Shutdown_Nature)
downtime	(int) Anticipated period for which the server will be down (in minutes); This value is an estimate and is typically a cautious one; When the server recycles, it immediately restarts, so the downtime will be as close to zero as possible (typically about a minute)

Retrieval Error

Despite everyone's best efforts, errors can occur when retrieving the data requested by a client tool (directly or indirectly). Retry logic is used internally to avoid errors like this, but sometimes an error will take too long to recover. If this occurs, the retrieval error notification is sent, which simply indicates the character for which the data could not be acquired. The client should simply make its original request again (e.g. synchronize).

Notification Properties

charId	(string) Unique id of the character that encountered the error

Element Token Revoked

It's possible that a user will revoke an element token to which a client tool has subscribed. When this occurs, the client tool is sent this notification, after which no further notifications will be sent relating to the element token.

Notification Properties

elementToken	(string) Element token that the user has revoked

User Token Revoked

It's similarly possible that a user will revoke the user token through which a client tool has connected to the HLO server. When this occurs, the client tool is sent this notification, after which the access token will become invalid and any further attempts to reconnect using the token will fail.

Due to its nature, there are no properties associated with this notification.

Notification Properties

<none>	n/a

Miscellaneous Considerations

This chapter is simply a collection of various topics that are not central to the API but are still important concerns to many tool developers integrating with the API.

CORS Support

The API fully supports Cross-Origin Resource Sharing (CORS). This is a browser security mechanism that limits the ability of a web application to access different sites without explicit authorization from those sites to do so. All API endpoints, including all SignalR endpoints, enable their use within any web application talking to any collection of servers. CORS is a

browser-enforced set of restrictions, and the API simply authorizes its use in whatever manner a client tool chooses. Consequently, there is nothing that a web-based client tool needs to do. It should all "just work".

Element Tokens

There are some caveats to keep in mind when working with element tokens. They should not arise often, but they could lead to some head-scratching, so they are outlined here.

- Character and campaign tokens are only accessible to the user that owns that resource. However, cast member tokens can be acquired by both the player controlling the character ***and*** the GM. The token is identical if both retrieve it, as it represents the identical resource within HLO. The wrinkle with this is that the cast token can be revoked by ***either*** the player or GM, which then invalidates the token for ***both***, since it's the same token.
- When an element's ownership or control is transferred to another user (e.g. a campaign PC), any token issued for that element remains in full force and effect. The new owner must explicitly revoke the token if that is desired. The assumption is that most situations where ownership and/or control are transferred will benefit from the token being preserved. The notable exception to this behavior is when an independent character (outside of a campaign) is introduced into a campaign without making a copy. In this scenario, the original character becomes something completely different within HLO, so any character token is automatically revoked and must be issued anew for the cast member.
- When an element's status is changed (e.g. retired or trashed), any associated token is automatically revoked. If the element is restored, a new token must be acquired.

Obtaining Tokens

Both user tokens and element tokens are obtained exclusively through the HLO interface. Log in to HLO normally, and you'll find each element token in a suitable location for its type (e.g. character, cast member, campaign). Wherever you can acquire an element token, the ability to acquire a user token will appear alongside it.

The various element tokens can be found in the following locations:

Character Token	First load the character into HLO so that you can edit it. Now open the Application menu in the upper right (the gear icon) and select the "Export/Integrate" option.
Cast Token	Locate the cast member within the campaign and open the preview of that cast member (simply clicking on the cast member should open the preview). In the upper right, click on the "Integrate" option.
Campaign Token	Open the campaign and go to the "Settings" tab on the left. At the bottom of the central panel, click the "Integrate" button in the "Other" block.

Providing a Caller Id

The client tool using the API may operate in a purely asynchronous manner and want to correlate responses it receives with the original requests that were submitted. To facilitate this technique, the API allows the caller to specify a "caller id" within every request. This id is limited to an integer, but it can be any value the client chooses. When a response is sent back, the response will then contain the identical caller id, making it easy for the caller to make the association.

For the client submitting the request, the caller id is included as a simple property in the request body, looking something like below:

```
{
  …,
  "callerId": 123,
  …
}
```

When submitted, the caller id is simply parroted back within the response, looking basically the same as the request.

Supplement Information in Export Data

Characters on the stage within HLO track a variety of supplemental details that are unique to being on the stage. Some of this additional information will be included in the export data sent by the API.

At the moment, there is only one piece of information added, which is the jersey number assigned to the character. It is only included for characters that actually possess a jersey number in HLO, which means characters that are non-unique and can appear on the stage more than once (e.g. “bandit #1”, “bandit #2”, etc.). When applicable, the jersey number will appear within the “portfolio” section of the export data, as the same value is assigned equally to any minions of the main character.

Synchronizing Jersey Number to External Tool

As described in the previous section, HLO includes the jersey number assigned to each character within the export data. HLO automatically assigns a jersey number to each non-unique character as it is added to the stage (i.e. PCs and unique NPCs are exempt and have no jersey number). By default, characters are assigned a progressively increasing jersey in the order the character is added to the stage. The net result is that a scene with 3 goblins and 3 orcs might number them as “goblin #1”, “goblin #2”, “goblin #3”, “orc #4”, “orc #5”, and “orc #6”.

Some tools, most notably VTTs, have pre-assigned numbering using a different scheme. For example, in the scene above, the goblins might be numbered 1-3, and the orcs might also be numbered 1-3, with each grouping having its own sequence. Correlating the HLO jersey numbers to the numbering within the tool can get a little complicated when this occurs, and it can also get a little confusing for users switching between the two products.

HLO makes it possible for users to bridge this difference by synchronizing the jersey numbers issued by HLO to match those used by the external tool. The “base jersey” number can be assigned to non-unique NPCs within scene scripts and when putting ad hoc NPCs on the stage. For example, in the scenario above, the scene script can be edited to specify a base jersey number of 1 for the orcs and goblins alike. When the scene is enacted onto the stage within HLO, the three orcs will be issued jersey numbers starting at 1 (i.e. #1-#3), and the same will be done for the goblins. The net result is characters on the stage that can be readily correlated to their counterpart within the external tool.

Save-Point Restoration

When a GM restores a save-point for a campaign, quite a bit of complexity is involved on the HLO server. When a client tool has a subscription to that campaign, the API distills all that complexity into a collection of notifications that should be straightforward to digest.

- A notification is sent that all characters on the stage have disappeared.
- A notification is sent with the list of restored characters that have now appeared on the stage.
- As each of the restored characters is reloaded by the server, a notification is sent with all the export data of the character.

Caveats When Receiving Character Notifications

There are a few special situations that can arise in conjunction with character notifications, as outlined in the following topics.

Missed Notifications

Due to the vagaries of internet communications, it is possible (albeit unlikely) that a client will miss a notification containing an update for a character. For example, a temporary internet connection failure could cause this to occur. If the client detects that the baseline version reported for a change is more recent than the current version possessed, the client should immediately synchronize the character by retrieving the differential between the version possessed and the latest version reported.

Differential Export Omits Certain Properties

The current assumption of the API is that a client tool subscribing to notifications maintains its own representation of each character. As changes are reported, they are incorporated into the local representation of the tool. Since the tool has its own representation, it means the tool already has its own descriptions for all the game elements portrayed within the export data for a character. And that means that the tool has no need for those properties, which make up a significant portion of the data transmitted within each character change notification.

Given the above logic, the “description” and “summary” properties are omitted from all different export data reported to the client. If your tool needs this data for some reason, please let us know, and we can investigate making the inclusion of those properties optional for a client tool.

Extraneous Notifications

When a client tool unsubscribes from monitoring a character, previous requests may already be in the pipeline. Consequently, it is possible that previously queued notifications may continue to be delivered for a brief period. If this occurs, the client should simply ignore any notifications that it no longer cares about.

Another situation where this will occur is when a cast member the client already knows about is added to the stage. Whenever any character appears on the stage, a client tool monitoring the campaign will be sent full export data for that character. If the client tool already has everything for the character from while it was off-stage, it will receive all the data again, which can simply be ignored.

Older Baseline Version

In highly unusual cases, the differential export included in an update notification may specify an older baseline version that the client actually possesses (or requested). If this occurs, the client should be able to safely apply the differential export without incident. All the properties will reflect their latest values, so the only thing that might occur is some fields will specify their existing values.

The only incorrect results that can arise will happen when an item is moved. The item will always reside in its correct location in the export data, but the old location it was moved from may be reported incorrectly. The client should be able to readily detect and handle this case. This situation can only occur when an item is moved multiple times, such as in the following scenario:

- At version 12, item “Child” resides beneath item “Parent1”.
- At version 13, item “Child” is moved beneath item “Parent2”.
- At version 14, item “Child” is moved beneath item “Parent3”.

In in the above scenario, if the client is synced to version 13 and instead receives a differential export relative to version 12, the “Child” item will be correctly reported as residing beneath “Parent3”. However, the “Child” item will also be reported as having been moved out from beneath item “Parent1”, which is where it lived at version 12.

Redundant Data in a Notification

Also in highly unusual cases, the API may report redundant information. For example, a differential export may be received with a baseline version of 16 and a current version of 18. This may be followed by a differential export with a baseline of 17 and a current version of 18. When this occurs the second block of data holds redundant state that the client already possesses, and the client can simply ignore the unnecessary data.

Multiple Instances of the Same Tool and Polling

The expectation is that client tools will only utilize polling in a limited manner. This has two important implications. First of all, if the same tool is used with the ***same*** tokens from different IP addresses, the API can detect it and will deny the requests. Secondly, if the same tool is used with the ***same*** user token by multiple computers with the same IP address, the wait times imposed will compound each other. In both of these instances, the client tool is repeatedly polling for the same data from multiple computers.

The proper solution for a tool that intends to be used in this manner is to subscribe to the appropriate tokens and receive notifications. Sending the same change notifications to multiple targets when the data is acquired is highly efficient, and that's exactly the usage model that we're striving for with integrating client tools through the API.

Connection Limits

As it is brand new, we don't have any clear sense of how extensively or widely the Public API will be adopted. We have therefore adopted nominal limits on the number of concurrent SignalR connections allocated to client tools, with the expectation that those limits will likely need to be revised in coordination with tool developers.

When a limit is reached, an error implicating the connection limit is reported by the "verify-access-token" endpoint to explain connection attempts are failing. If users of your tool encounter this error, please contact us so we can work out how best to address it. Ideally, if you are anticipating a wide release of integration with your tool, please inform us in advance so we can implement a suitable plan to fully support the launch.

Appendix 1: Severity and Result Codes

This section details the various result codes (severity and specific error) returned by the API.

Severity

Responses can indicate a range of severity ratings from the following list:

1 – Success Request completed successfully.
50 – Info Request completed, but something also occurred that is worthy of flagging.
100 – Warning Request completed, but something occurred that was probably not expected.
150 – Error Request failed in some way.
200 – Failure Request failed in a way that has cascading implications and could impact subsequent requests.

Result Codes

The exact outcome of each request will report one of the following result codes:

7000 – Api_Unchanged Request yielded no actual change or effect.
7011 – Api_Usage_Error Caller provided inputs that were invalid in some way, such as possessing out-of-range values.
7022 – Api_Precluded Request cannot be performed due to some pre-requisite or situational state that precludes it.
7100 – Api_Bad_Api_Token The provided token is invalid in some way, typically having expired or been revoked. NOTE! If a refresh token is invalid, a new access token must be acquired through the product. NOTE! If an access token is invalid, a fresh access token must be acquired via the endpoint provided for this purpose. Access tokens have a nominal lifespan and will expire regularly.
7101 – Api_Bad_Tool_Name The provided tool name is invalid.
7102 – Api_Bad_Game_Server The provided game server identifier is invalid.
7110 – Api_Bad_Element_Token Specified element token is either invalid, revoked, or inaccessible to the caller.
7111 – Api_Bad_Campaign Specified campaign token is either invalid or does not identify an accessible campaign.
7113 – Api_Bad_Cast_Member Specified cast member token is either invalid or does not identify an accessible cast member.
7113 – Api_Bad_Character Specified independent character token is either invalid or does not identify an accessible character.
7200 – Api_Status_Inaccessible Requested resource is inaccessible due to its status, such as a campaign or cast member being retired or trashed.
7201 – Api_Not_Subscribed No subscription exists for the requested token.
7210 – Api_Demo_Account Requested capability is not available to a demo or expired account.
7250 – Api_Campaign_Contention Request could not be completed due to a transient resource contention on the campaign, the stage, or characters on the stage.
7800 – Api_Caller_Throttled The caller is hitting the API endpoints faster than allowed and has been denied access. NOTE! When this occurs, the previous wait time is restarted anew, so slow down sending requests. NOTE! If a caller persistently requires throttling on a protracted basis, additional lockout periods will be applied that become progressively longer.
7801 – Api_Caller_Throttled_Overlap Same as Api_Caller_Throttled, except the caller made simultaneous/overlapping requests
7805 – Api_Caller_Lockout_Tier Persistent need for throttling has resulted in a prolonged lockout at the next tier. NOTE! Each tier imposes a progressively longer lockout period. Wait for the lockout to expire before utilizing the API further.

7810 – Api_Overlapping_Client_Lockout Multiple callers are polling the same token(s) concurrently, which is not allowed. NOTE! This caller has been locked out for a little while and must wait until the lockout elapses.
7890 – Api_Connection_Limit_Exceeded Connection from client tool was denied due to reaching the maximum limit on connections. NOTE! Contact support for assistance in resolving the problem.
7900 – Api_Not_Enabled All operations of the public API are currently unavailable. Try again later.
7977 – Api_Internal_Error An internal error occurred. If this error persists, please open a support ticket for assistance.
7999 – Api_Unspecified_Error An error occurred that does not have a suitable API translation defined for it.

Appendix 2: Enum Types

The API uses a small number of enumerated types in responses and notifications. This section details each of these types.

API Notification Type (Api_Notify)

Every notification sent to a subscribed client identifies its nature and includes a payload specific to that nature. The notification type will be one of the values below:

-999 – User_Token_Revoked The user token was revoked and all associated subscriptions have been cancelled.
-86 – Element_Token_Revoked An element token was revoked for the element and the associated subscription has been cancelled.
-1 – Retrieval Error Something went wrong while retrieving the character, so new export data is not available. Attempt to retrieve the character anew by synchronizing it.
1 – Shutdown Initiated Shutdown of the server has begun, and all subscriptions will be need to be re-established once the server comes back up.
10 – Character_Load The character has been loaded for use, and the full export is reported.
11 – Character_Update The character has been modified by its controller, and the new delta is reported. NOTE! This can also be triggered in response to a subscription or an explicit synchronize request from the client.
101 – Stage_Appear (campaigns subscriptions only) One or more characters have just been added to the stage.
102 – Stage_Disappear (campaigns subscriptions only) One or more characters have just been removed from the stage.
120 – Stage_Contention (campaigns subscriptions only) The campaign stage has either entered or exited a state of contention. During contention, most operations upon the stage are temporarily suspended while the current operation is completed.

Character Change Status (Change_Status)

The status of a requested character will always be one of the following values:

-1 – Missing Character is not valid or is inaccessible, which can occur for a variety of reasons; No export data is returned
0 – Unchanged No changes have occurred with the character since the given baseline version; No export data is returned
1 – Delta The character has likely changed relative to the baseline version, returning a differential export (which may itself be empty)
2 – Complete Either a baseline version of zero was request or the character has changed relative to the baseline version, with a full export being returned

Imminent Shutdown Behavior (Shutdown_Nature)

When the server informs a subscribed client it has initiated shutdown, the nature of the shutdown will be one of the following values:

0 – Unknown The server is going down for an unspecified reason or length of time
1 – Maintenance Shutdown is for maintenance and has the accompanying expected downtime
2 – Recycle Shutdown is due to a regular server recycle, and the server should be up quickly (typically about a minute)
3 – Unplanned An unplanned shutdown was triggered that is unlikely to start back up until someone takes a look and resolves an unexpected problem on the server

Appendix 3: Troubleshooting SignalR Connection Issues

Typically, once the SignalR connection is established, all communications over that connection should “just work”. It’s folly to simply make that assumption, so the API provides a few special endpoints over the SignalR connection that are designed to facilitate diagnosis of problems by incrementally adding complexity and isolating where the problem lies. Each of the sections in this chapter details one of these endpoints, and practical examples of their use can be found in both of the sample client applications that accompany this documentation.

Ping the Server

The first step is verifying that a primitive endpoint on the SignalR connection can be invoked. The “ping” endpoint takes no parameters and returns no value. If anything goes wrong with contacting the server over the SignalR connection, an error is triggered that ought to shed some light on the nature of the problem.

Parrot a Value

The second step is invoking a SignalR endpoint that does something trivial and returns a value. The “parrot” endpoint takes a single integer parameter and returns that same value. As above, if anything goes wrong with the communication in either direction, an error is triggered that should help uncover what’s wrong.

Parrot a Value with Authentication

The third step is to introduce authentication into the mix. The “authParrot” endpoint is identical to the “parrot” endpoint above, with one difference. This endpoint requires that the caller provide a valid access token within the appropriate header of the request. Problems at this juncture almost always point to an issue with the token itself or the way it’s being provided to the server.

Send a Notification Message

The fourth and final step is to trigger the server to send a notification message back to the client. The “inform” endpoint takes two parameters. The first is the name of the endpoint on the client side that should be called to receive the notification. The second is a message string that is delivered to the client via that notification endpoint. The client-side endpoint should be defined as accepting a list/array of strings, and the message will be sent as the first and only value in that list.

Once this step is operating correctly, everything is in place and ought to work for receiving notifications from the HLO server.

Appendix 4: Sample C# Client Application

To help get you jump-started and provide a concrete example of how to utilize the API, a sample client application has been written in C# for Microsoft’s .Net platform.

Basic Details

A download containing all the necessary files will be found at the URL below:

```
https://docs.herolab.online/hlo-api-csharp-client-sample.zip
```

The following assumptions and requirements apply to this sample:

- The sample code includes a solution and project that require Visual Studio 2017 (or newer) to load. The Community Edition is free.
- The sample targets the .Net framework 4.7.2, but older versions can likely be utilized.
- Various nuget packages are relied upon by the sample, which are all freely available. The most important of these is the SignalR client package, which provides all the connection logic for communicating with the API. The only package not from Microsoft is the NewtonSoft JSON package, which is used to serialize and deserialize the JSON objects transmitted to and from the server.

The following behaviors are core to the way the sample operates:

- All output from the sample is routed to the console window for simplicity.
- Validation checks rely simply on the use of Debug.Assert to verify that values are as expected. If an assert fails, then something has gone wrong at the point where it occurs. If everything is working, the no asserts should fail. A proper application will need to perform suitable handling of any errors that may arise.

There are two source files comprising the sample, with the following purposes:

Program.cs	Main program code that contacts the server and orchestrates everything
Api_Types.cs	Assorted class definitions representing the various object payloads that are submitted to and returned from the API and that appear within the sample

How It Works

If everything is working correctly, the sample should start up and go through the progression of setting everything up with the HLO server. Along the way, it will write a few messages to the console window as various steps are completed. Once the SignalR connection is in place, an element token is subscribed and the sample simply listens for notifications pertaining to that element, writing them out to the console window as they are received.

The overall behavior of the sample should be readily discernible from the comments found within the source files. A high-level summary can be distilled as the following:

- Contact the API to acquire an access token that will be used in all subsequent calls of the API.
- Retrieve the proper URL to be used when contacting SignalR by providing the element token to be monitored.
- Setup the SignalR connection and then open that connection.
- Invoke a few SignalR requests as a progressive proof-of-life that the connection is fully working as intended.
- Subscribe to the specified element token.
- Listen for notifications pertaining to that token and emit them to the console window.
- At this point, start making changes to the character for which the element token was provided and see the corresponding notifications appear in the console window.

Running the Sample

Before the sample application can be run, it must first be configured with a few pieces of information that you will need to provide. These variables will be found near the top of the “program.cs” file.

s_tool_name	Enter a suitable name for your client tool
s_user_token	Paste in a user token acquired from within HLO; See the Miscellaneous Considerations chapter for where to acquire the user token
s_element_token	Paste in an element token acquired from within HLO; See the Miscellaneous Considerations chapter for where to acquire the element token

Once you’ve got everything ready, build the application and run it. The console window will display the progress made. When the element subscription is in place, start making changes to the character to which the element token pertains and watch them appear in the console window.

It’s now time to start doing something useful with all those notifications in your own tool!

Appendix 5: Sample JavaScript Client Application

We’ve provided a second concrete example of utilizing the API through a sample client application. This one has been written in Javascript so it can be run in a web browser. Similar code with only relatively small changes should also work in a Node.js environment (e.g. on a server running Node.js).

Basic Details

A download containing all the necessary files will be found at the URL below:

```
https://docs.herolab.online/hlo-api-js-client-sample.zip
```

The following assumptions and requirements apply to this sample:

- The sample code requires Node.js to be installed in order to build and run it (any version after about 10.15.3 should work). Node.js is free to install from https://nodejs.org/ (the latest LTS version is recommended), and automatically includes the Node Package Manager (npm).
- The sample targets the latest version of modern browsers such as Chrome, Firefox, Edge, and Safari, though older versions and other browsers can likely be utilized.
- Various npm packages are relied upon by the sample, which are all freely available and browsable at https://www.npmjs.com/. The most important of these is the ASP.NET Core SignalR client package @aspnet/signalr which is browsable at https://www.npmjs.com/package/@aspnet/signalr. This package provides all the connection logic for communicating with the API. The sample uses version 1.0.3 which is known to be compatible with the Hero Lab Online server. Versions up through the latest version 1.1.4 appear like they may also work based on limited testing but we haven't tested this extensively. The other packages are development dependencies used to build and run the sample.

The following behaviors are core to the way the sample operates:

- All output from the sample is routed to the console of the browser's F12 development tools for simplicity.
- Validation checks rely simply on the use of a simple assert function defined in the sample to verify that values are as expected. If an assert fails, then something has gone wrong at the point where it occurs, and a console error will be printed. If everything is working, then no asserts should fail. A proper application will need to perform suitable handling of any errors that may arise.

There are several files comprising the sample, with the following purposes:

src/main.js	Main program code that contacts the server and orchestrates everything
dist/bundle.js	Compiled version of main.js that includes all dependencies and is loaded by the browser
index.html	HTML file loaded by the browser, which includes a <script> tag to load the bundle.js file
package.json	Package definition file for npm, listing all dependencies and development dependencies
package-lock.json	Package lock file generated by npm to record exactly which packages to install
webpack.config.js	Configuration file for webpack, which is the tool used to build the bundle.js file

How It Works

If everything is working correctly, the sample should start up and go through the progression of setting everything up with the HLO server. Along the way, it will write a few messages to the console window as various steps are completed. Once the SignalR connection is in place, an element token is subscribed and the sample simply listens for notifications pertaining to that element, writing them out to the console window as they are received.

The overall behavior of the sample should be readily discernible from the comments found within the source files. A high-level summary can be distilled as the following:

- Contact the API to acquire an access token that will be used in all subsequent calls of the API.
- Retrieve the proper URL to be used when contacting SignalR by providing the element token to be monitored.
- Setup the SignalR connection and then open that connection.
- Invoke a few SignalR requests as a progressive proof-of-life that the connection is fully working as intended.
- Subscribe to the specified element token.
- Listen for notifications pertaining to that token and emit them to the console window.

- At this point, start making changes to the character for which the element token was provided and see the corresponding notifications appear in the console window.

Running the Sample

Before the sample application can be run, it must first be configured with a few pieces of information that you will need to provide. These constants will be found near the top of the “main.js” file.

TOOL_NAME	Enter a suitable name for your client tool
USER_TOKEN	Paste in a user token acquired from within HLO; See the Miscellaneous Considerations chapter for where to acquire the user token
ELEMENT_TOKEN	Paste in an element token acquired from within HLO; See the Miscellaneous Considerations chapter for where to acquire the element token

Next, you will need to install Node.js on your system by downloading it from https://nodejs.org/. We recommend using the latest LTS version, and installing using the default settings.

Next, you will need to open a command prompt to the folder where the sample was unzipped to (the folder where the package.json file is), and execute the following commands one at a time:

```
npm install
npm run build
npm run serve
```

The “npm install” command will first install all of the dependencies of the project into a node_modules subfolder of the project. This step may take a few minutes. The “npm run build” command will build the bundle.js file. The “npm run serve” command will start up a local HTTP server via the npm “http-server” package.

At this point, you should see output similar to the following:

```
Staring up http-server, serving ./
Available on:
  http://192.168.1.4:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

You can then navigate to any of the listed URLs in your browser and open up the browser console in the F12 developer tools. The console window will display the progress made. When the element subscription is in place, start making changes to the character to which the element token pertains and watch them appear in the console window.

Note: You may see a warning in the console (e.g. in Chrome) like “DevTools failed to load SourceMap”. It is safe to ignore this warning for the purposes of the sample application. For applications you develop, you can enable or disable source maps in whatever way you prefer to avoid this warning.

It’s now time to start doing something useful with all those notifications in your own tool!

Appendix 6: Hero Lab Online Server Status

The HLO server offers a few additional endpoints that a client tool may find useful. All but one of these endpoints provide state information about the running server. If the server is offline, the final endpoint can be used to assess the nature and duration of the outage.

Custom Request Header

In order to utilize these endpoints, the client must identify the correct game server being interacted with for appropriate routing. This is accomplished by including a custom header within the submitted HTML request. The various game

servers can be in different states at any given time (e.g. deploying an update to one while the others are live), so each must be properly addressed. For most requests, the correct server can be implicitly inferred, but not for status inquiries that don't contain the necessary context for accurate routing.

If subscription notifications are employed, the client already has the game server value it needs to provide. This is the "gameServer" property returned by the "identify-notification-server" endpoint. If ad hoc retrieval is utilized, the client must retrieve this value using the "identify-game-server" endpoint detailed below.

The custom request header is "X-Game-Server", and it must be included with each of the status requests. An example of the request header is as follows:

```
X-Game-Server: @Sa#
```

Identifying the Game Server

Client tools that retrieve data as needed (i.e. not via subscription notification) will need to specify the correct game server within the custom request header. This value is obtained via the "identify-game-server" endpoint.

```
https://api.herolab.online/v1/access/identify-game-server
```

Request Properties

accessToken	(string) Access token acquired from the user token
elementToken	(string) Any element token for the game system that is currently being monitored

Response Properties

gameServer	(string) Identifier associated with the game server for the element token, which can then be used within the custom header

Example

A sample request body retrieving the game server will look like the following:

```
{
  "accessToken": "<access token goes here>",
  "elementToken": "$CDYw0vq~@Sa#"
}
```

The corresponding response might look similar to this:

```
{
  "wait": 1500,
  "gameServer": "@Sa#",
  "result": 0,
  "severity": 1
}
```

Proof-of-Life

If the server can be pinged, then the next level of assessment is a simple proof-of-life check. The "hello" endpoint is invoked as a GET request and will respond with a fixed, non-empty string if the server is operating at the most basic level or higher.

```
https://api.herolab.online/state/hello
```

NOTE! Remember to specify the "X-Game-Server" header with the appropriate value.

Basic Operational Status

The “status” endpoint is also a GET request and will report whether the server is in a maintenance state or generally available. If in maintenance mode, the server will return a Service Unavailable (503) status code. Otherwise, it will return an OK (200) status.

```
https://api.herolab.online/state/status
```

NOTE! Remember to specify the “X-Game-Server” header with the appropriate value.

Login Acceptance Status

Depending on the server’s state, it may or may not be accepting logins from new users. The “is-live” endpoint is invoked as a GET request and returns a simple JSON response body.

```
https://api.herolab.online/state/is-live
```

Response Properties

live	(bool) Whether the server is currently accepting new logins from users

NOTE! Remember to specify the “X-Game-Server” header with the appropriate value.

Next Planned Outage

We strive to announce maintenance outages, such as deployment of new features, at least a day in advance. When a planned outage is announced, it can be learned by a client tool via the “next-outage” endpoint. This endpoint is invoked as a GET request and returns simple JSON response body with the details.

```
https://api.herolab.online/state/next-outage
```

Response Properties

nextOutage	(int) The scheduled time of the next outage, as a standard 32-bit time value (i.e. number of seconds since 00:00 hours, Jan 1, 1970 UTC)
nextDowntime	(int) Estimated length of the maintenance period (in minutes)

NOTE! This only identifies the next planned outage for maintenance purposes. Occasionally, a hotfix is required for which the outage announcement is only made a short time in advance. And the nightly recycle of the servers is not announced via the mechanism. It simply occurs on schedule, with the server coming back up immediately.

NOTE! Remember to specify the “X-Game-Server” header with the appropriate value.

Outage Status While Offline

When the HLO server goes offline, such as during a new feature deployment, it will become temporarily unreachable. In this situation, a client tool can go to an alternate source to determine the outage status of the server and assess the nature of the outage.

```
https://status.herolab.online
```

The above URL provides a human-readable statement regarding the current status of the HLO server. To retrieve a status value for programmatic consumption, use the endpoint below, which will return either “normal” or “maintenance”.

```
https://status.herolab.online/status
```

NOTE! The status server is only intended for use when something appears to be wrong with the HLO server. The status server will never undergo maintenance concurrently with the HLO server, but it will undergo maintenance itself, which means it will occasionally be offline for that purpose. So the proper protocol should be to contact the HLO server normally. If the HLO server is unreachable, check the status server as a fallback.

NOTE! In an extreme circumstance, the status server could be unreachable when the HLO server is also offline. An example might be our hosting provider having a complete network failure that causes all of our servers to become unreachable at the same time.

NOTE! A client tool can know in advance when the next scheduled maintenance will occur, as outlined in the preceding section. A client tool can also be told exactly when the HLO server is going offline, whether for maintenance or some other purpose. See the separate chapter on Real-Time Notification for further details.

NOTE! Remember to specify the “X-Game-Server” header with the appropriate value.

Appendix 7: Revision History

This appendix identifies the various behavioral changes that have been introduced into the Public API and when those revisions occurred. Only behavioral changes are listed here. Documentation changes that don’t involve the functionality of the API (e.g. typos, formatting, clarifications) are not included here.

V1.1 2020/10/29	• The “gameServer” property is now returned by the “identify-notification-server” endpoint • The “attach-game-server” endpoint is introduced as a required part of the connection and subscription workflow • The “unsubscribe-all” endpoint is available for use • HLO supports the “base jersey” value to synchronize the jersey number of characters put onto the stage with the “token value” used by various VTTs
V1.2 2020/11/23	• All server status endpoints now require a custom header in the request. • The “identify-game-server” endpoint has been added in conjunction with the custom header for status endpoints.