



External Interface API

Version 1.5

Revision Table

Revision	Date	TSW Version	Notes
1.5	29 Sept 2026	TSW6 Build 493	First version of document

Overview

The External Interface API allows external systems to interface in to Train Sim World 6 while the title is running. Using a JSON over TCP style interface, external systems can retrieve data in real time from the simulation as well as make certain changes into it such as adjusting lever positions.

As at 1.5 of the API, the primary focus is about interfacing with the running Simugraph simulation and much of the central design goals are about supporting external hardware implementations of train cab environments. There are some additional supporting functions such as controlling the Weather as well.

The mechanic for access is sending an HTTP style request over TCP with name/value pair parameters. The response will take the form of a JSON block. Access to the API needs to be enabled via a command line parameter and there is a private key that must be shared between the game and the external applications to enable a basic form of security.

Please note that this document is technical by nature and will not serve as a tutorial for learning TCP, HTTP, JSON or other concepts and additional reading may be needed as required.

Contents

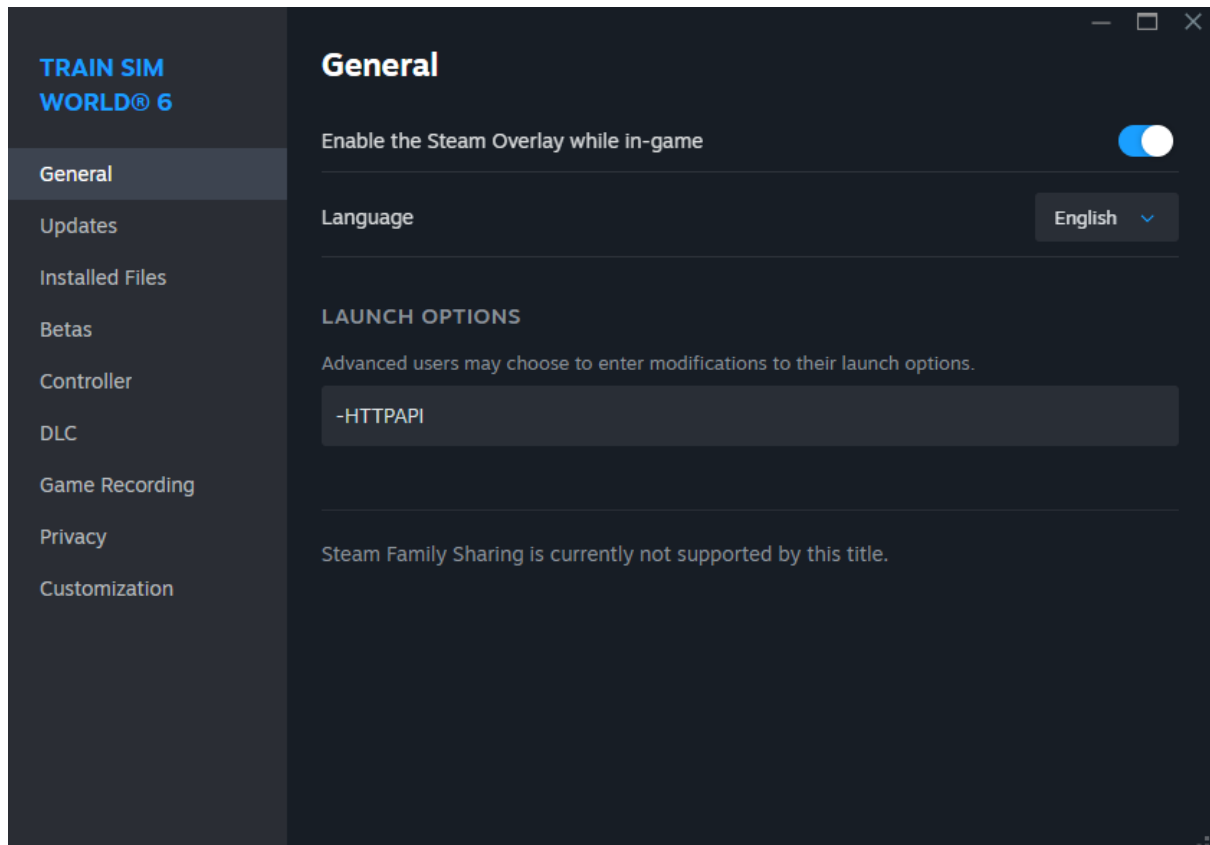
Revision Table.....	1
Overview	1
Connection and Security	3
Release Builds	4
Development Builds	4
Connection	6
Opening Network Access to Other Computers.....	6
Rate Limits & Throttling	7
curl Example.....	8
Postman Example	9
Top Level Commands - /info.....	11
Top Level Commands - /list.....	12
Reading Data - /get.....	15
Example: Retrieving the Throttle Lever Setting	16
Writing Data	17
Subscriptions	20
Making Subscriptions	20
Reading Subscriptions	21
Removing a Subscription.....	22
Weather	24
Time of Day.....	26
Virtual Rail Driver	27
Driver Aid	28
PlayerInfo.....	28
Track Data.....	29
Other Data	30
Other Information	31
Player Train	31
All Player Vehicle Locations.....	32

Connection and Security

To enable the External Interface API, the title must be launched with the command line parameter “-HTTPAPI”. This can be done within Steam as follows.

Find the title in your Steam Library, right click on it and select Properties.

From the dialog that opens, select the General tab (which should be the default) and observe the “LAUNCH OPTIONS” section. Add the parameter there, as shown:



With this in place, load the game up to the main menu once and then exit. This will ensure that the communication API key is generated.

In a “Release” build of the game vs a “Development” build of the game, the location of the generated key file will be different.

Most people accessing this API will be using a Release build. If Dovetail Games have specifically advised you are on a Development build then the alternative location will apply for all profile and save data including the Comm API Key.

Release Builds

To find the generated key, go to the following location:

Documents\My Games\TrainSimWorld6\Saved\Config

Here you will find a file CommAPIKey.txt

Development Builds

To find the generated key, locate the install folder for the game and navigate to a Saved\Config folder. For example, if your Train Sim World 6 installation in Steam is located here:

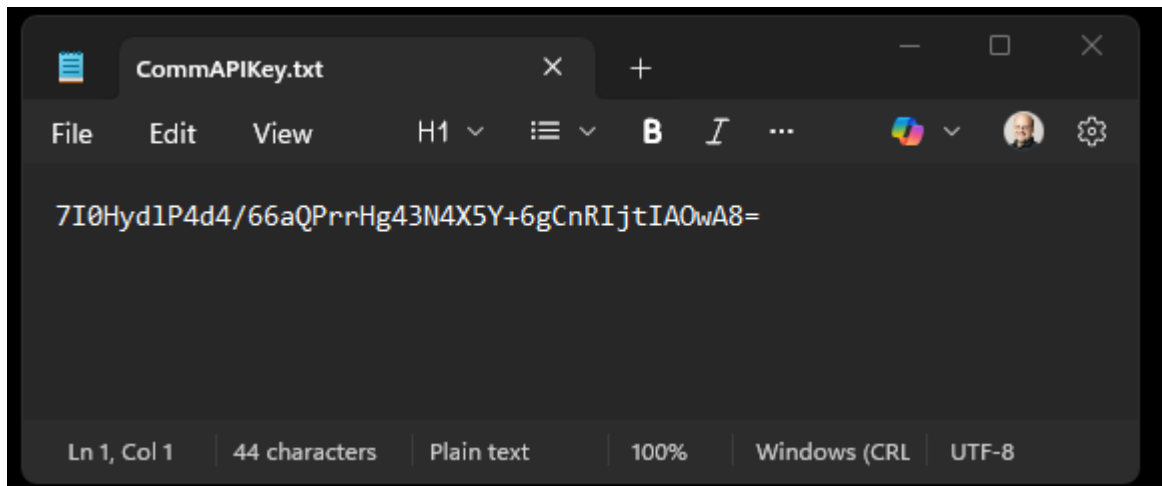
D:\SteamLibrary\steamapps\common\TrainSimWorld6

Then you should navigate from the above into this folder:

WindowsNoEditor\TS2Prototype\Saved\Config

Here you will find the file CommAPIKey.txt

Open the file in notepad and it will contain a string. This string is the key that must be sent as part of all requests to the API or they will be rejected.



The key includes all standard ASCII characters in the file but does not include any CR/LF characters at the end.

In the above case, the key is “7I0Hyd1P4d4/66aQPrrHg43N4X5Y+6gCnRIjtIAOwA8=”

Note: The Key can change at any time, and if the CommAPIKey.txt file is deleted a new (different) key will be created. Every time the file is created on any system, it's generated randomly. In any usage of the External Interface API this must be taken into account.

It is recommended that the external software seek this file out and dynamically read it as part of its initialisation, so that anyone using the software doesn't need to worry about these specifics or about updating it should it change.

In the “Example” chapters, it will be shown how to use the Comm API Key as part of requests.

Note: There is never any reason to share your Comm Key with anyone. Doing so will potentially permit them access to connect to your game, and if any vulnerabilities were ever identified within TSW could enable worse things to happen. If anyone asks you for the Comm Key, you should not share it under any circumstances. If you think it might have gotten out at all, delete the file so that a new one is auto generated and restart the game.

Connection

Connections to the API are made via a TCP socket connection to port 31270.

By default, only 127.0.0.1:31270 is opened for connection, meaning the socket cannot be connected to from another computer on the network.

Note that only standard TCP is supported, not HTTPS or any other form of Secure Encrypted communication, so be sure to always use “http” and not “https” in any request URLs.

Requests will include HTTP headers, a HTTP method such as GET/POST/PATCH etc and any name/value pair parameters. The response will then come back as a JSON block.

Opening Network Access to Other Computers

By default, only a socket is opened on the 127.0.0.1 “local” network interface to enhance security as this inherently prevents any external connections from being possible.

For some use cases however it may be required or advantageous to have the external software running on a different PC to the PC that the game is running on. To make this work, the socket must be configured to open on all network interfaces.

To achieve this, go to the home folder for the game data, this will be in the same place as the DTGCommKey.txt was located.

Where the DTGCommKey.txt was in Saved\Config, you should now go one further level deep to Saved\Config\WindowsNoEditor.

Inside here you will find engine.ini which is one of the internal configuration files for the game. It's a text file openable in your normal favourite text editor.

Within here, you need to add a new section to the INI file:

```
Paths=../../../../TS2Prototype/Plugins/DLC/TTC_MK1/Content
Paths=../../../../TS2Prototype/Plugins/DLC/EuropeanContentPack
Paths=../../../../TS2Prototype/Plugins/DLC/NL_Content_Pack/Con

[Audio]
UnfocusedVolumeMultiplier=1.000000

[HTTPServer.Listeners]
DefaultBindAddress=0.0.0.0

[WindowsApplication.Accessibility]
StickyKeysHotkey=True
ToggleKeysHotkey=True
FilterKeysHotkey=True
```

Specifically note the new highlighted “HTTPServer.Listeners” section. Setting the “bind” address to 0.0.0.0 instructs the game to bind to *all* network interfaces.

Save the file, restart the game and you should now be able to connect to it from other computers.

It is important that you consider the security implications of an open socket however the API isn't capable of much so risk is minimal. In a professional environment we would recommend use of Firewalls to ensure that only expected devices are able to access the machine on this port.

Rate Limits & Throttling

Note that the title does not impose any throttling, but every request will have *some* impact on the general running of the title. How much this is will depend on the hardware being used. While a general use of the API will have no perceptible impact, it's certainly possible to have a noticeable FPS impact on the title if massive numbers of requests are spammed. There is no reason to spam the API at these kinds of rates however. Part of the development process for any external software should include evaluation of request rates vs any impact to the experience, and adjusting the implementation or setting minimum hardware specifications accordingly.

curl Example

curl is not required to use the API but is a handy command line tool that can allow for some simple aspects to be tested quickly outside of writing code.

curl is a suggested tool for testing purposes, however if you run into any technical difficulties installing or using it you will need to refer to curl support systems for advice.

curl for Windows can be obtained from the following URL:

<https://curl.se/windows/>

Implementations are available for a wide range of other operating systems.

On Windows, the ZIP file can be downloaded, extracted and then the “curl” executable is in the “bin” folder. It does not get installed like other Windows applications and is strictly for command line use, without any windows GUI.

Simple usage example

curl

```
-H "DTGCommKey: 7I0Hyd1P4d4/66aQPrrHg43N4X5Y+6gCnRIjtIA0wA8="
"http://localhost:31270/get/CurrentDrivableActor.Function.HUD_GetSpeed"
```

(note: command line should be on a single line, it's broken out here for ease of reading in the document).

Response when the key is incorrect:

```
C:\Windows\System32>curl -H "DTGCommKey: 7I0Hyd1P4d4/66aQPrrHg43N4X5Y+6gCnRIjtIA0wA8=" "http://localhost:31270/get/CurrentDrivableActor.Function.HUD_GetSpeed"
{"errorCode": "dtg.comm.InvalidKey", "errorMessage": "API Key for request doesn't match CommAPIKey.txt in the game config directory."}
C:\Windows\System32>
```

Response when the key is correct:

```
C:\Windows\System32>curl -H "DTGCommKey: SLP+e2eg59UAxks+E7VYHFuaUMFkKI0Hq5oxPkksYEY=" "http://localhost:31270/get/CurrentDrivableActor.Function.HUD_GetSpeed"
{"Result": "Success", "Values": {"Speed (ms)": 4.5457782745361328}}
C:\Windows\System32>
```

Note that the return value to this request is in meters per second, the above result is roughly 10mph which is what the locomotive was doing at the time.

Postman Example

Postman is not required to use the API but is a handy visual tool with full GUI for trying out various requests easily and understanding responses before writing code.

Postman is a suggested tool for testing purposes, however if you run into any technical difficulties installing or using it you will need to refer to the Postman support systems for advice.

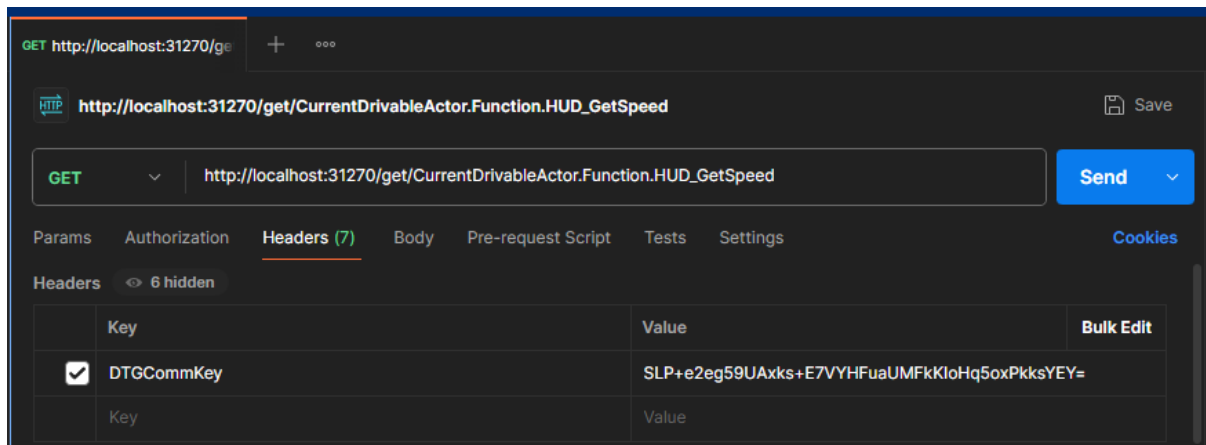
Postman can be downloaded for free from the following URL:

<https://www.postman.com/downloads/>

Once downloaded, run the installer as with any other Windows application.

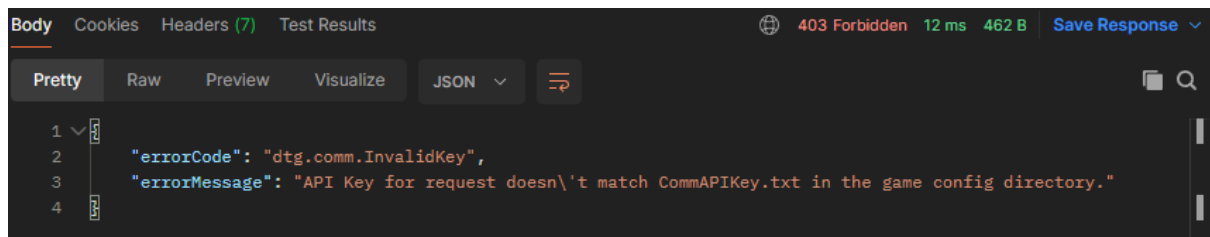
Note that Postman has some premium options and a server hosted environment for saving queries and other advanced features. While there is some benefit to using those they are certainly not essential and the tool can be used perfectly fine in its most basic form without them using the “Lightweight API Client”.

Setup a request similar to the following:



Note that on the “Headers” tab, the DTGCommKey parameter must be set up. If you need any parameters for requests, those will go in the “Params” tab.

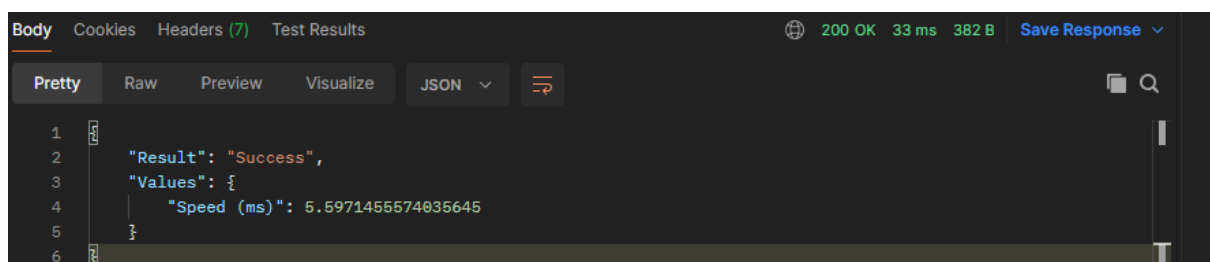
If the request is made with an invalid key, this is the kind of response you will see a 403 Forbidden was returned along with the following JSON body:



The screenshot shows a REST client interface with the 'Body' tab selected. The status bar at the top indicates a '403 Forbidden' response with a time of '12 ms' and a size of '462 B'. The response body is displayed in 'Pretty' format as a JSON object with two fields: 'errorCode' and 'errorMessage'.

```
1 {
2   "errorCode": "dtg.comm.InvalidKey",
3   "errorMessage": "API Key for request doesn't match CommAPIKey.txt in the game config directory."
4 }
```

If the request is made with a valid key, this is the kind of response you will see a 200 OK response, with the usable data in the JSON body:

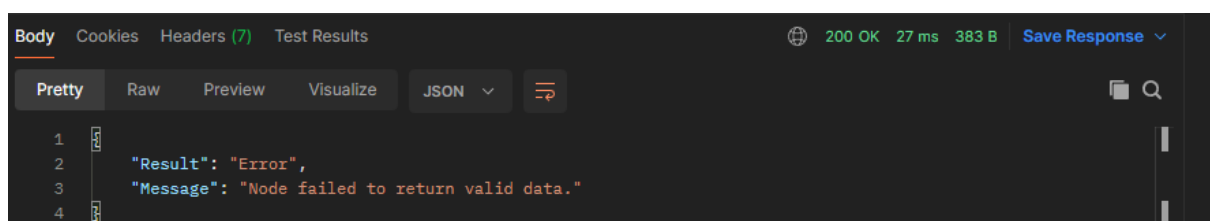


The screenshot shows a REST client interface with the 'Body' tab selected. The status bar at the top indicates a '200 OK' response with a time of '33 ms' and a size of '382 B'. The response body is displayed in 'Pretty' format as a JSON object with 'Result' and 'Values' fields.

```
1 {
2   "Result": "Success",
3   "Values": {
4     "Speed (ms)": 5.5971455574035645
5   }
6 }
```

Once again, notice that the result is in meters per second.

If you make an invalid request for data such as malforming the request path, you may get a response like the following:



The screenshot shows a REST client interface with the 'Body' tab selected. The status bar at the top indicates a '200 OK' response with a time of '27 ms' and a size of '383 B'. The response body is displayed in 'Pretty' format as a JSON object with 'Result' and 'Message' fields.

```
1 {
2   "Result": "Error",
3   "Message": "Node failed to return valid data."
4 }
```

The majority of examples used in this document were created using Postman.

Top Level Commands - /info

The External Interface API is designed to be self-documenting because of the changeable nature of environments and trains within the title. It's suggested that developers make use of tools like Postman to discover how the API works in the cases they are exploring for specifics, this document will not cover all the API calls possible for all the trains in the title.

Results from queries issued by the game supersede any written documentation and should be checked first.

At the highest level, the /info API call will return all the top level commands that the system supports.

Method	Path	Description
GET	/info	Get information about available commands
GET	/list	List all valid paths for commands
GET	/get	Get a value for a node's End Point
PATCH	/set	Set a value on certain writeable endpoints
GET	/subscription	Read a subscription set. (see later documentation on how Subscriptions work)
POST	/subscription	Create a subscription
DELETE	/subscription	Unsubscribe from a particular path.
GET	/listsubscriptions	List all active subscriptions. Doesn't retrieve values.
POST	/execdev	Run a console command, this does not work in Release builds, only in Development builds.

Top Level Commands - /list

The /list path will return all valid paths that commands can be used against. The resulting data set can be quite large. For example, while running an ES44C4 on Cajon Pass, the result was around 415 kilobytes of JSON.

This is an important way to navigate what is available.

The list here is where you would discover “CurrentDrivableActor” as shown in the previous examples:

```
14959 "NodePath": "Root/CurrentDrivableActor",
14960 "NodeName": "CurrentDrivableActor",
14961 "Nodes": [
```

Beneath that, you will see every node that can be interacted with, for example:

```
"NodePath": "Root/CurrentDrivableActor/Coupler_F",
"NodeName": "Coupler_F"
```

Would be an example of the node needed to interact with the front coupler, usually used to release or lock the knuckle.

The result set is a tree and is not fully expanded in a top level /list request. For example, you can see that within “CurrentDrivableActor” there is a “Simulation” node that is flagged as collapsed:

```
"NodePath": "Root/CurrentDrivableActor/Simulation",
"NodeName": "Simulation",
"CollapsedChildren": 173
```

Here, we can request this node specifically by issuing
/list/CurrentDrivableActor/Simulation :

```
1  ✓ 1
2    "Result": "Success",
3    "NodePath": "CurrentDrivableActor/Simulation",
4    "NodeName": "Simulation",
5    "Nodes": [
6      {
7        "Name": "BrakeRigging1"
8      },
9      {
10       "Name": "BrakeRigging2"
11     },
12     {
13       "Name": "CoolingSystem"
14     },
15     {
16       "Name": "Bogie1"
17     },
18     {
19       "Name": "Axle11"
20     },
21   ],
22 }
```

This then provides the details of nodes inside Simugraph and other simulation elements that can be interacted with.

Once again, you can then add this node to the request to drill down further and further.
E.g. /list/CurrentDrivableActor/Simulation/Axle11/Wheel :

```
1  ✓ 1
2    "Result": "Success",
3    "NodePath": "CurrentDrivableActor/Simulation/Axle11/Wheel",
4    "NodeName": "Wheel",
5    "Nodes": [],
6    "Endpoints": [
7      {
8        "Name": "PinType",
9        "Writable": false
10     },
11     {
12       "Name": "NetTorque_NM",
13       "Writable": false
14     },
15     {
16       "Name": "LocalTorque_NM",
17       "Writable": false
18     },
19     {
20       "Name": "NetTractiveEffort",
21       "Writable": false
22     },
23   ],
24 }
```

While “Nodes” form the *path* to the request, “Endpoints” are where the data itself is stored.

This means you could not issue, for example:

```
/list/CurrentDrivableActor/Simulation/Axle11/Wheel/NetTorque_NM
```

Even though it is in the list, it's an Endpoint, not a node. Indeed the Nodes array is empty, there are no further child nodes here, just data on the Endpoints.

To read that data, the /get command is used.

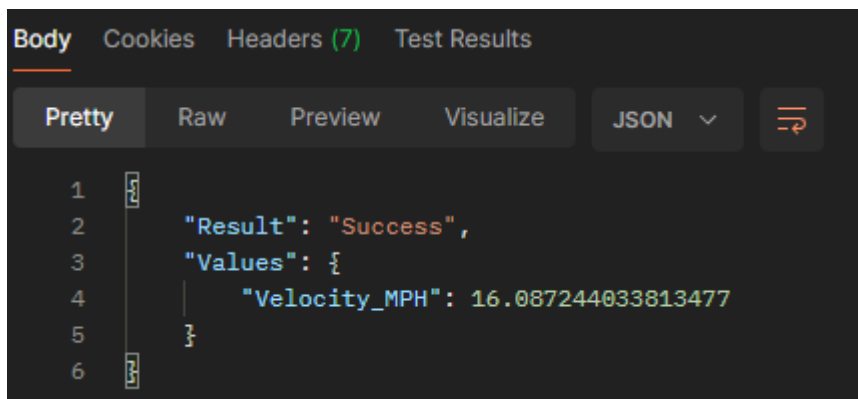
Reading Data - /get

Once you have located the information you want to get access to you'll need to use the `/get` command to retrieve it.

Extending from the previous example, we can find there is an Endpoint underneath Wheel called `Velocity_MPH`. To retrieve its value we issue the following command:

`/get/CurrentDrivableActor/Simulation/Axle11/Wheel.Velocity_MPH`

Note that for an End Point, the period (".") symbol is used.



The observant will notice that this is an entirely different way of retrieving the speed of the vehicle compared to the examples previously which returned in meters per second and this because it's *not* returning the speed of the vehicle.

This command is returning the speed effected by the rotation speed of the specific axle chosen. If this axle were to be locked up or spinning, it would *not* report the actual speed of the vehicle.

As you delve further into the simulation it is crucial to be fully aware of what a value might actually be telling you rather than what it can be easily mistaken for.

All EndPoints can have their values retrieved in this way.

Example: Retrieving the Throttle Lever Setting

Using `/list/CurrentDrivableActor` we can see there is a node “Throttle(Lever)”.

Note: Every rail vehicle could have its own unique naming standard, while there is *generally* consistency it’s not guaranteed so should be checked in any use case.

You can pull the details of this as follows:

`/list/CurrentDrivableActor/Throttle(Lever)`

This will return all the endpoints.

Within Throttle(Lever) there are a wide range of End Points including `Function.GetCurrentNotchIndex`, accessed as follows:

`/get/CurrentDrivableActor/Throttle(Lever).Function.GetCurrentNotchIndex`

Which might return something like this, where my ES44C4 is indeed in Notch 4:

```
1  [
2    "Result": "Success",
3    "Values": {
4      "ReturnValue": 4
5    }
6  ]
```


Writing Data

Some EndPoints support their data being set externally, usually ones mapped to user interactive controls. Others, such as valve pressures etc, wouldn't make sense to be able to set their values.

When retrieving the list of End Points from a node, each one will tell you if it supports being written to or not.

```
{
  "Name": "ObjectClass",
  "Writable": false
},
{
  "Name": "InputValue",
  "Writable": true
},
}
```

To write to an End Point, you would add a parameter “Value” to the request and put the new value in there.

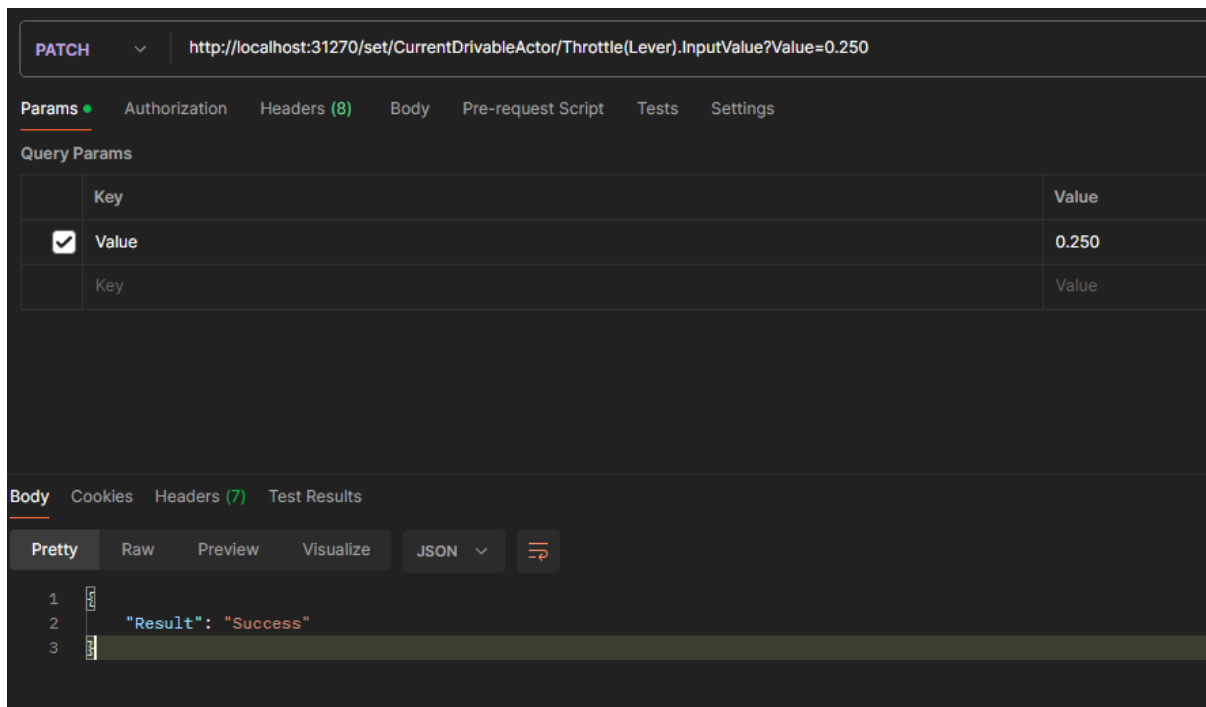
Thus, if reading the “InputValue” End Point looks like this:

The screenshot shows a REST client interface with a GET request to `http://localhost:31270/get/CurrentDriveableActor/Throttle(Lever).InputValue`. The Headers tab is active, showing a table with one header: `DTGCommKey` with value `SLP+e2eg59UAxks+E7VYHFuaUMFkKloHq5oxPkksYEY=`. The Body tab is also active, showing a JSON response in Pretty format:

```
1 {
2   "Result": "Success",
3   "Values": {
4     "InputValue": 0
5   }
6 }
```

A small tooltip labeled "speedomet" is visible on the right side of the response body.

Setting it would look like this:



Note the method has been changed the “PATCH” and we’re using /set now as the top level command. Under “Params”, a new Key named “Value” has been added and we’ve set it to 0.250. When you run this, the Throttle lever in the cab will immediately change to reflect the new state.

The initial observation one might have from this is why 0.250 when the throttle on an ES44C4 goes from notch 0 to 8.

We can go back to the API to get this answer. Remember that the API needs to be usable to control trains with a wider variety of types of control systems. One train might have 8 notches, another might have a smooth 0..100% range, another might just have three notches.

```
/get/CurrentDrivableActor/Throttle(Lever).Function.GetMinimumInputValue
```

This will return the minimum value the controller can have.

```
/get/CurrentDrivableActor/Throttle(Lever).Function.GetMaximumInputValue
```

This will return the maximum value the controller can have.

```
/get/CurrentDrivableActor/Throttle(Lever).Function.GetNotchCount
```

This will return the number of notches the controller has.

Taking these functions and running them on the ES44C4 over the API, we find that:

GetMinimumInputValue	0
GetMaximumInputValue	1
GetNotchCount	9

Bearing in mind that 0 is one of the notches, if you divide 1 by 8 you get a notch size of 0.125.

If you issue the command with a value of 0.125 you will then see the train correctly set itself to Notch 1 on the throttle. Set it to 0.750 and you will get Notch 6.

It is therefore crucial to understand the range, any notches etc when setting a value by querying the meta data first.

Note that if you send a value outside of the notches, it will slip to a nearby notch. So for example if you set it to 0.2 it will move to Notch 2 as it “falls” to the 0.250 notch position. In this way you can directly map exact lever positions from hardware levers and let the title figure out the resulting notch position, or you can send specific notch values in by calculating the appropriate values to send. Whichever is more convenient for your application.

Internally within Train Sim World, controllers have both an “Input” and an “Output” position. The “Input” position is always what is used to influence the control from the outside whether that’s from the API, keyboard or joystick controls.

In the case of the ES44C4 Throttle(Lever) End Point, the Output value refers to the real-world value so if you set the Input Value to 0.125 then the Output Value will return 1. This could be useful in representing various lever positions back within external systems in readouts for example.

Lastly, it’s important to note that all API calls reflect or impact the current status of the train. So if your application sets the throttle to 0.250 (Notch 2) and then someone uses the keyboard on the game PC to move the throttle, when you read the current output value or current notch position back it will correctly reflect the position the game has for it. You may need to factor this in with periodic re-captures of control positions back to your application or at least account for overriding them and it not being an unhandled condition.

Subscriptions

With many external applications there are many use cases where the application will be seeking updates on numerous values regularly not just one or two. For example, regular updates on RPM, Speedometer, Brake pressure gauges and so forth.

Rather than making lots of repeated identical requests, the subscription system allows the external application to subscribe to various end points and then get a bulk update on all of them whenever needed, in a single request.

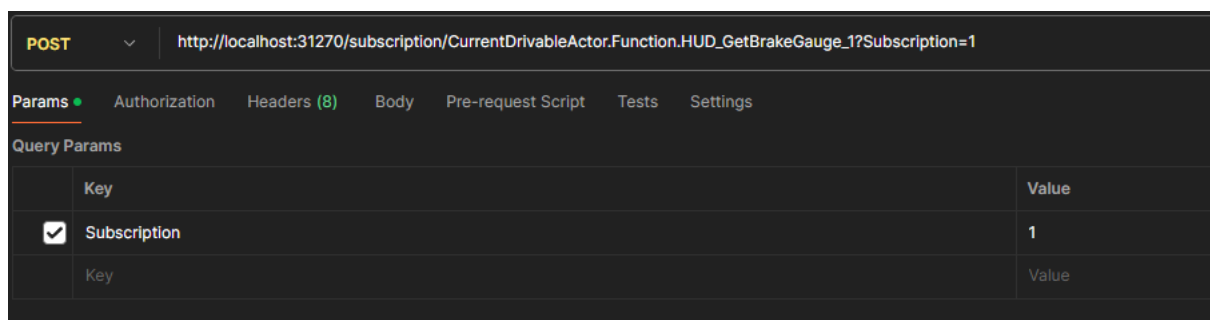
Making Subscriptions

To subscribe to a value you need to send a POST request to `/subscription` with the detail of the End Point you want to subscribe to, and a Subscription ID. You can choose your own subscription ID's, and then use those later for other operations.

Adding an End Point is done one at a time.

`/subscription/CurrentDrivableActor.Function.HUD_GetBrakeGauge_1`

Add a parameter to the request named "Subscription" and set its value to, e.g. 1.



The screenshot shows a REST client interface with a POST request to `http://localhost:31270/subscription/CurrentDrivableActor.Function.HUD_GetBrakeGauge_1?Subscription=1`. The 'Params' tab is selected, showing a table of query parameters.

Key	Value
<input checked="" type="checkbox"/> Subscription	1
Key	Value

We can repeat the same command with different end points, let's add three more:

`/subscription/CurrentDrivableActor.Function.HUD_GetBrakeGauge_2`

`/subscription/CurrentDrivableActor.Function.HUD_GetAmmeter`

`/subscription/CurrentDrivableActor.Function.HUD_GetIsSlipping`

With each of these, ensure to pass the *same* value for the Subscription parameter so that it all goes in to the same subscription.

Remember that on game-startup there are no subscriptions (they do not get persisted between sessions) so your application will have to ensure that it's clearing and resetting subscriptions appropriately.

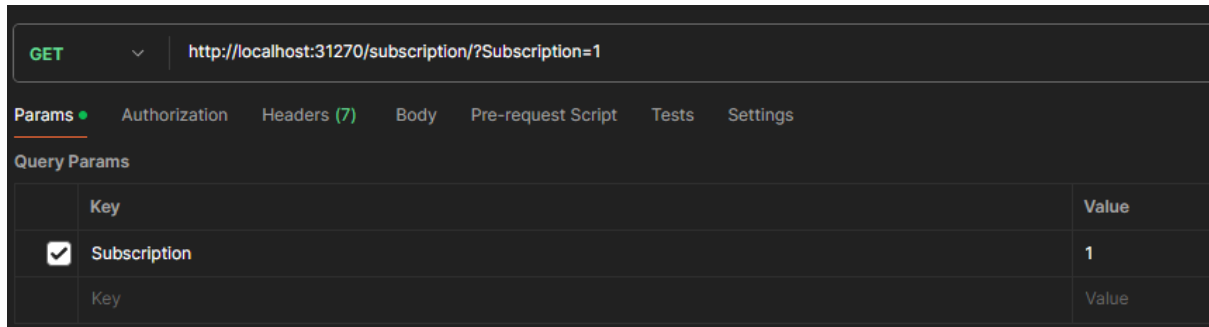
Reading Subscriptions

To read the values from a subscription you simply request it via a GET method:

/subscription

Put a parameter named Subscription and set it to 1 to match the subscription we created and it will then return everything in it.

For this request:



The screenshot shows a REST client interface with a GET request to `http://localhost:31270/subscription/?Subscription=1`. The 'Params' tab is active, showing a single query parameter: `Subscription=1`.

Key	Value
Subscription	1

The following is returned:

```
1  {
2    "RequestedSubscriptionID": 1,
3    "Entries": [
4      {
5        "Path": "CurrentDrivableActor.Function.HUD_GetBrakeGauge_1",
6        "NodeValid": true,
7        "Values": {
8          "WhiteNeedle (Pa)": 720444.4375,
9          "RedNeedle (Pa)": 0
10       }
11     },
12     {
13       "Path": "CurrentDrivableActor.Function.HUD_GetBrakeGauge_2",
14       "NodeValid": true,
15       "Values": {
16         "WhiteNeedle (Pa)": 721133.75,
17         "RedNeedle (Pa)": 0
18       }
19     },
20     {
21       "Path": "CurrentDrivableActor.Function.HUD_GetAmmeter",
22       "NodeValid": true,
23       "Values": {
24         "Amps": 0
25       }
26     },
27     {
28       "Path": "CurrentDrivableActor.Function.HUD_GetIsSlipping",
29       "NodeValid": true,
30       "Values": {
31         "IsSlipping": false
32       }
33     }
34   ]
35 }
```

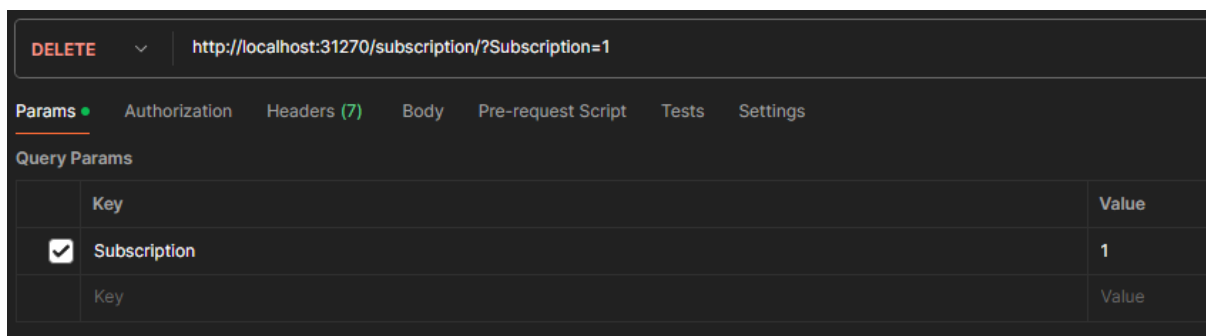
As you can see, what would have been four separate requests each iteration previously, is now one. You can have multiple subscriptions set up to cater for different polling periods, e.g. perhaps there is data you want every second and other data you want 60 times a second, separating them out into different subscriptions can limit needless extra load from both the title and your application.

Removing a Subscription

Issue a request with the method as “DELETE” to:

/subscription

With the parameter “Subscription” and the value set accordingly, to remove the subscription.



The screenshot shows a REST client interface with the following details:

- Method: DELETE
- URL: http://localhost:31270/subscription/?Subscription=1
- Params: Authorization, Headers (7), Body, Pre-request Script, Tests, Settings
- Query Params table:

	Key	Value
<input checked="" type="checkbox"/>	Subscription	1
	Key	Value

Result on success:

```
1 {
2   "SubscriptionID": 1,
3   "Action": "SubscriptionSetDeleted"
4 }
```

Having removed the subscription, if you now try and request data from it you will get an error.

```
1 ✓ {
2   "errorCode": "dtg.comm.NoSuchSubscription",
3   "errorMessage": "Could not find requested subscription ID"
4 }
```

If you try and remove a subscription that does not exist, you will also get an error:

```
1 {
2   "errorCode": "dtg.comm.NoSuchSubscription",
3   "errorMessage": "Could not find requested subscription ID"
4 }
```

As a general guide, the boot of an application should start by issuing removal requests for all its standard subscriptions to ensure they're going (e.g. in case the app is restarted without the game being restarted). Errors in this case are expected for a fresh start of the game. After issuing those removal requests whether they were present or not before you can be confident they are now gone. The application can now register new requests using those ID's without worrying about if any data is getting doubled up.

Weather

The Weather Manager for the game can be accessed via the External Interface API.

As seen under the top level “/list” results, “WeatherManager” is an available node. If we query that with /list/WeatherManager we can see a list of End Points.

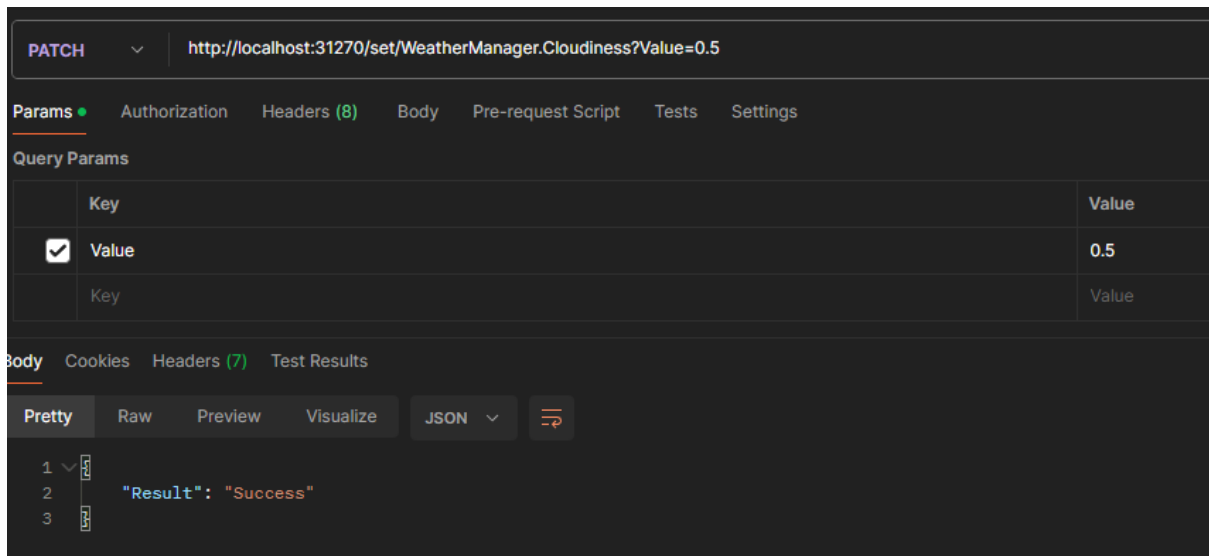
```
[{"Result": "Success",
  "NodePath": "WeatherManager",
  "NodeName": "WeatherManager",
  "Nodes": [],
  "Endpoints": [
    {
      "Name": "Reset",
      "Writable": true
    },
    {
      "Name": "Temperature",
      "Writable": true
    },
    {
      "Name": "Cloudiness",
      "Writable": true
    },
    {
      "Name": "Precipitation",
      "Writable": true
    }
  ]
}]
```

You can retrieve values as normal, with /get/WeatherManager.Cloudiness:

```
[{"Result": "Success",
  "Values": {
    "Cloudiness": 0.24387374520301819,
    "CloudinessOverridden": 1
  }
}]
```

You can even set them, as they are marked Writable = true.

/set/WeatherManager.Cloudiness with a Value from 0 to 1. Note that the method in this case is PATCH as we're using /set.



In-game weather will instantly adjust, and in the case of the above command you will see the cloud level change along with any other impacts of that cloudiness change such as lighting etc.

Currently, the Weather Manager allows the following parameters to be set:

Temperature	Current temperature, this will affect whether “precipitation” is rain or snow for example.
Cloudiness	How cloudy the sky is.
Precipitation	How much rain or snow is falling.
Wetness	How wet is the ground.
GroundSnow	How much snow is settled on the ground.
PiledSnow	How much piled snow is there (this is only infrequently used on routes themselves however)
FogDensity	How much fog there is. Note some of the best results are between 0.0 and 0.05, subtlety is often best for Fog.

Time of Day

Using `/get/TimeOfDay.Data` you can retrieve a range of information about the current time of day in the in-game world, in a variety of formats.

```
"Result": "Success",
"Values": {
  "LocalTime": 234526449958791,
  "LocalTimeISO8601": "0001-09-29T10:37:24.995-08:00",
  "GMTOffset": -8,
  "WorldTime": 234814449958791,
  "WorldTimeISO8601": "0001-09-29T18:37:24.995Z",
  "DayPercentage": 0.41179186105728149,
  "SunriseTime": "+05:46:38.086",
  "SolarNoonTime": "+11:39:42.222",
  "SunsetTime": "+17:32:46.358",
  "SunPositionAzimuth": 244.53294372558594,
  "SunPositionAltitude": 50.019737243652344,
  "MoonPositionAzimuth": 332.44680786132812,
  "MoonPositionAltitude": 52.152915954589844,
  "OriginLatitude": 34.498329162597656,
  "OriginLongitude": -117.55481719970703,
  "SystemTime": 6.3894754840129997e+17,
  "SystemTimeISO8601": "2025-09-29T15:00:40.130Z"
}
```

None of these parameters support being written to.

Virtual Rail Driver

For some applications, the mechanism used by the Rail Driver hardware from PI Engineering is actually sufficient for a simple external piece of hardware or software, so rather than having to dive into all the specifics of various trains and cope with that variation there is an abstraction layer which presents the same access that the Rail Driver uses.

To use the Virtual Rail Driver interface you will need to enable it via the API as doing so will disable the “real” Rail Driver interface.

Using a PATCH command, use this to enable it:

```
/set/VirtualRailDriver.Enabled
```

Set the “Value” parameter to “true”.

Once it’s enabled you can then use other End Points on the VirtualRailDriver, for example:

```
PATCH /set/VirtualRailDriver.Throttle?Value=0.5
```

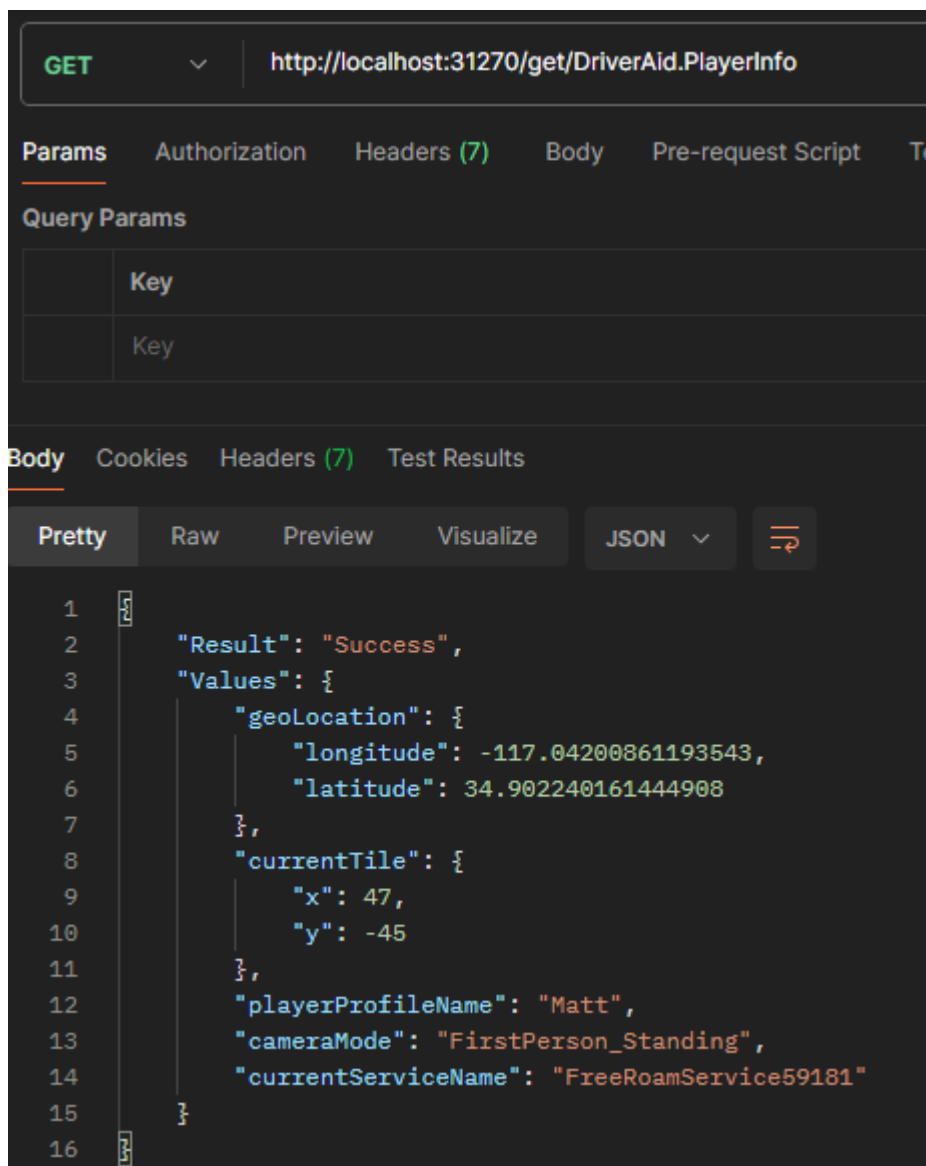
Driver Aid

The driver aid gives access to information about the line up ahead of the player train. This includes gradient information and whether there are tunnels.

Driver Aid offers three end points: Data, PlayerInfo and TrackData.

PlayerInfo

Player Info returns key information about the player, including the current Lat and Long in the world. Information about “Tile” coordinates are internal systems used within the game.



Track Data

TrackData contains upcoming track information.

The screenshot shows a REST client interface with a GET request to `http://localhost:31270/get/DriverAid.TrackData`. The response is a JSON object with the following structure:

```
1 {
2   "Result": "Success",
3   "Values": {
4     "lastPlayerPosition": {
5       "height": 65043.984375,
6       "distanceToHeight": 266908.65625,
7       "bTunnelFound": false
8     },
9     "trackHeights": [
10      {
11        "height": 64280,
12        "distanceToHeight": 0,
13        "bTunnelFound": false
14      },
15      {
16        "height": 64280,
17        "distanceToHeight": 4924.28125,
18        "bTunnelFound": false
19      },
20      {
21        "height": 64312.4453125,
22        "distanceToHeight": 60129.4375,
23        "bTunnelFound": false
24      }
25    ]
26  }
27 }
```

Other Data

The “Data” End Point contains other information about entities and data on the track ahead of the player, this includes where the next signal is, where the next speed limit is, what the various aspects and limits and also then some of the following signal and speed limits.

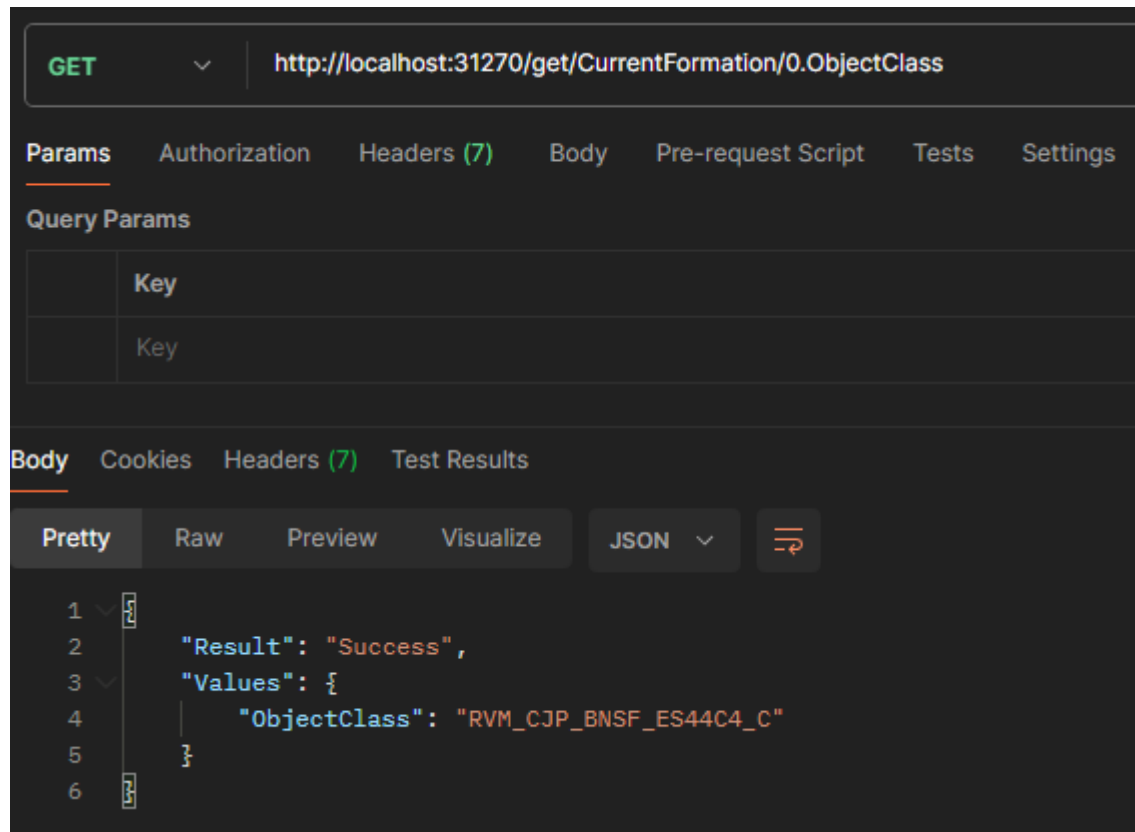
The screenshot shows a REST client interface with a GET request to `http://localhost:31270/get/DriverAid.Data`. The response is displayed in the 'Body' tab, formatted as JSON. The JSON object contains a 'Result' field set to 'Success' and a 'Values' field which is a complex object representing track data.

```
1 {
2   "Result": "Success",
3   "Values": {
4     "signalSeen": true,
5     "speedLimitSeen": false,
6     "distanceToSignal": 209465.21875,
7     "distanceToNextSpeedLimit": 81.448699951171875,
8     "gradient": 0.10000000149011612,
9     "signalAspectClass": "Stop",
10    "signalPropertyGuid": "8FB0F57F41469AC86FA528931EE03040",
11    "bSignalIsPermissive": false,
12    "speedLimit": {
13      "value": 9.9999995620235262e+24
14    },
15    "nextSpeedLimit": {
16      "value": 13.411199569702148
17    },
18    "currentSpeedLimitSource": "TrackSpeedLimit",
19    "trackMaxSpeed": {
20      "value": 13.411199569702148
21    },
22    "serviceMaxSpeed": {
23      "value": 3.4028234663852886e+38
24    },
25    "formationMaxSpeed": {
26      "value": 3.4028234663852886e+38
27    },
28  }
29 }
```

Other Information

Player Train

To find out the Rail Vehicle Class of the train being driven by the player, issue the following request:



In this case, the player is driving the ES44C4 from the Cajon Pass route.

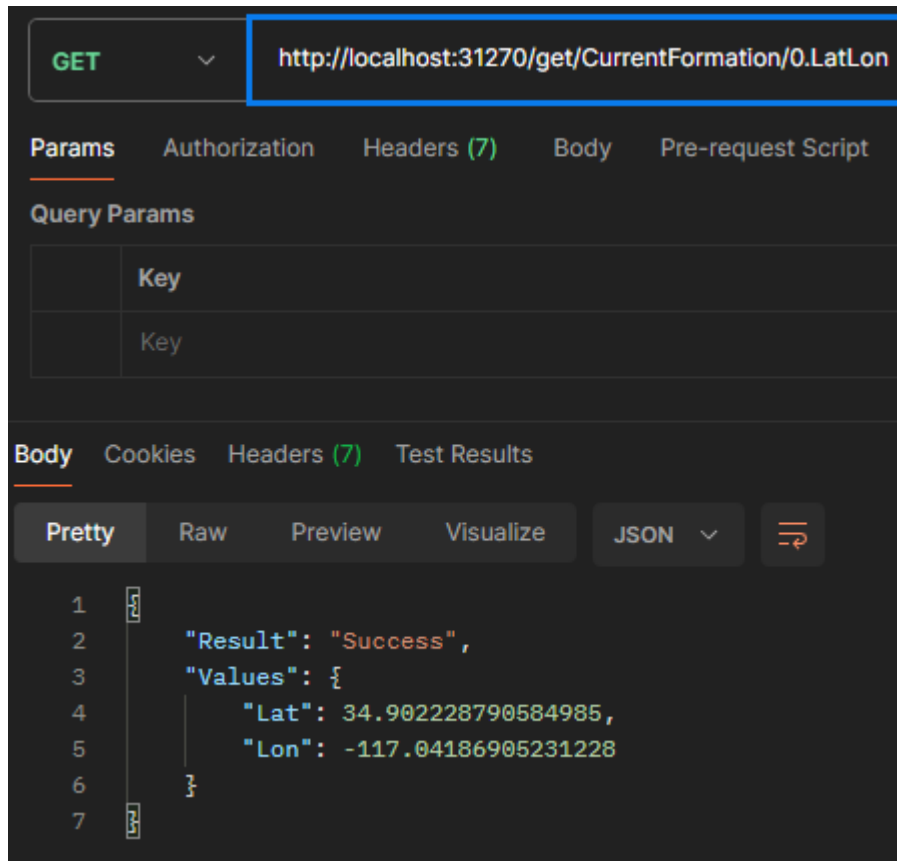
Note that CurrentFormation, being used in this way, will allow API access to any of the vehicles on the player formation.

Note: It is not always the case that the driven vehicle is formation index 0. For example, in the case of a multiple unit that drives into a terminus station and then leaves the same way back out again it is technically now reversing within the game logic. If it were a 3 car unit with indices 0, 1 and 2 – then the player would be driving index 0 on the way in and index 2 on the way out.

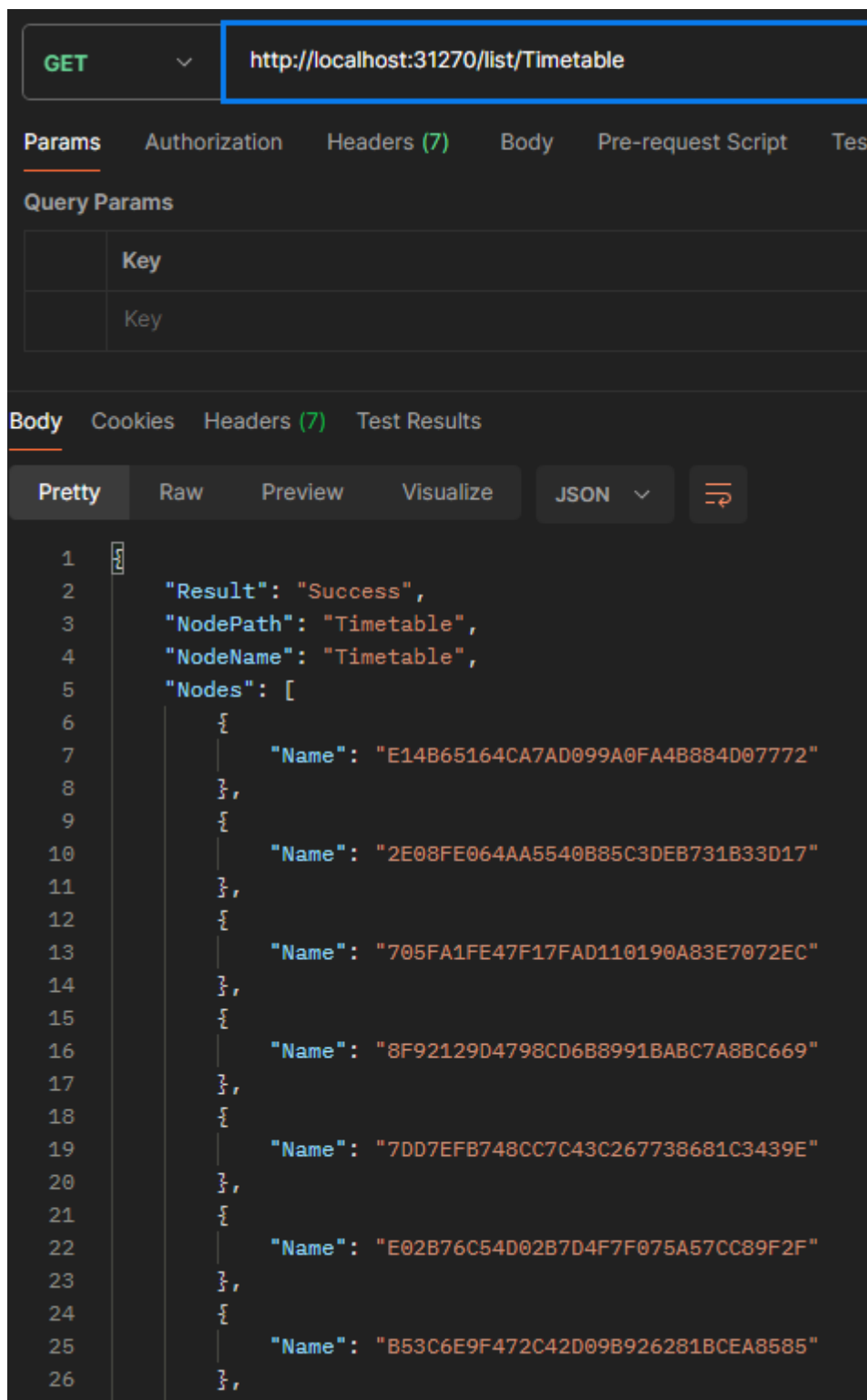
All Player Vehicle Locations

Every vehicle contains an end point “LatLon”.

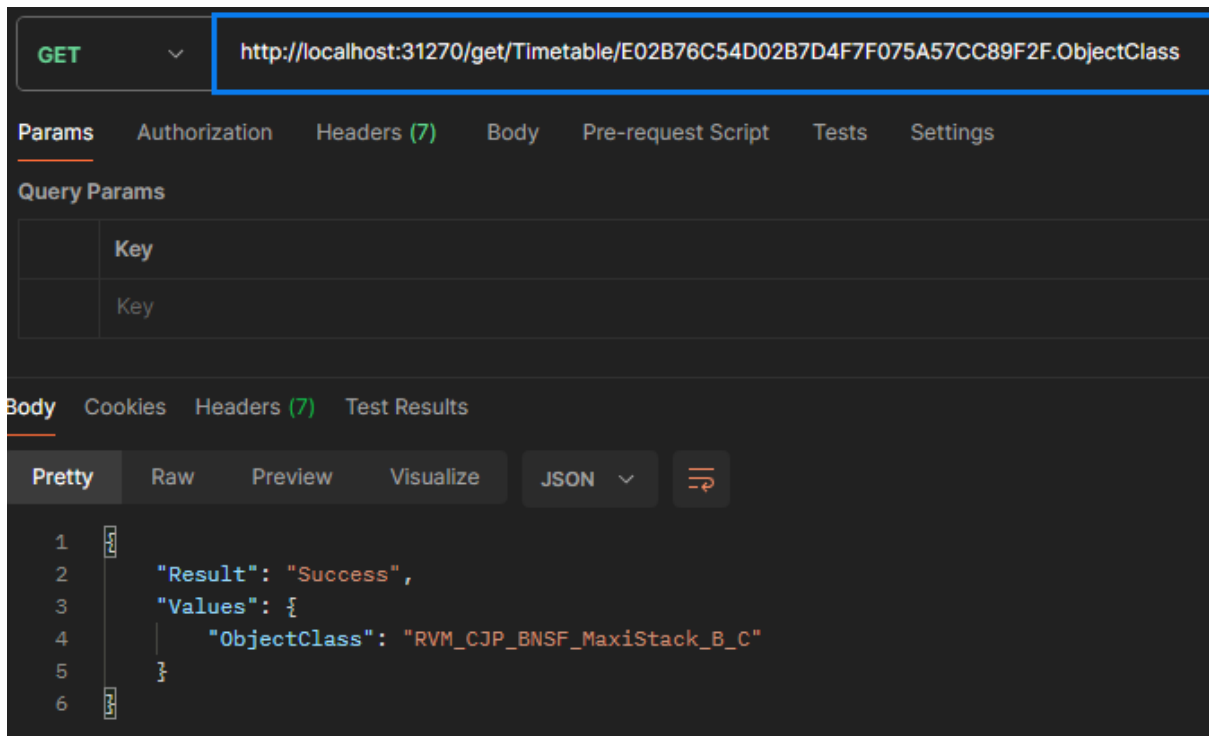
You could for example iterate through all of the vehicles in CurrentFormation and retrieve the Lat Lon using a query such as:



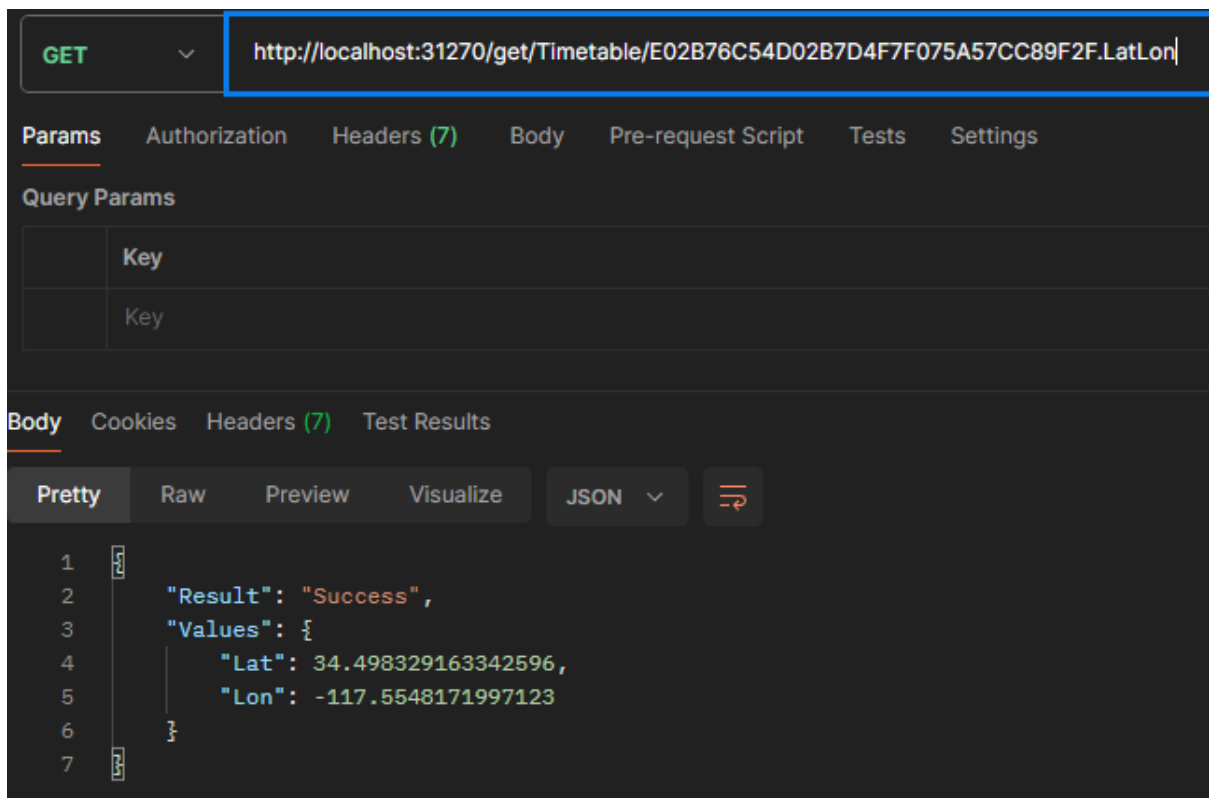
Every vehicle in the active timetable can be iterated through, though it is not easy to find specific vehicles of interest.



In this example, a large chunk of data is returned with the unique ID's of every vehicle in the timetable. Each one is a Rail Vehicle, so you could query each one for Lat Lon and vehicle class for example.



The vehicle queried is part of a Maxistack container set.



And this is where the vehicle is in the world.