

# TS2 Technical Manual

## Table of Contents

1. Introduction .....	2
2. Architecture .....	2
2.1. The simulation server .....	2
2.2. Clients .....	3
2.3. The Editor .....	3
3. Simulation model .....	4
3.1. Options .....	4
3.2. Track Items .....	6
3.3. Routes .....	13
3.4. Train Types .....	14
3.5. Services .....	15
3.6. Trains .....	17
3.7. Signal Library .....	19
3.8. Message Logger .....	29
4. Writing a simulation .....	29
4.1. Load the Signal Library .....	30
4.2. Setup the layout .....	30
4.3. Define Routes .....	33
4.4. Define Train Types .....	34
4.5. Define Services .....	34
4.6. Define Trains .....	34
5. Websocket API .....	35
5.1. URI .....	35
5.2. Initializing a websocket connection .....	35
5.3. Requesting data from the server .....	36
5.4. Server Event Notifications .....	40
6. Developing a Client .....	42
6.1. Getting the simulation dump .....	42
6.2. Register listeners .....	42
6.3. Get renotified to sync the UI .....	43
6.4. Drawing trains and activated routes .....	43
6.5. Interact with the simulation .....	43

# 1. Introduction

This document describes the internals of TS2. It is aimed both at simulation writers and developers who want to interact with the simulation through its API.

## 2. Architecture

TS2 is composed of 3 main components:

- The simulation server
- Clients
- The editor

### 2.1. The simulation server

This is the central piece of the software. It is shipped with the TS2 release.

It can be run:

- Locally for a standalone game
- Remotely on another PC or even on a cloud server for multi-player gaming

The TS2 sim server tasks are

- To centralize the game data, that is:
  - Scenery layout (The track items, and how they are connected to each other, their states, etc.)
  - Possible and set routes
  - Services and Trains
- To enforce interlocking on :
  - Route setting/unsetting
  - Change of points position
  - Update of signals aspects
- To compute train behaviour, speed and position
- Make the clock tick

The TS2 sim server communicates with clients through websocket.

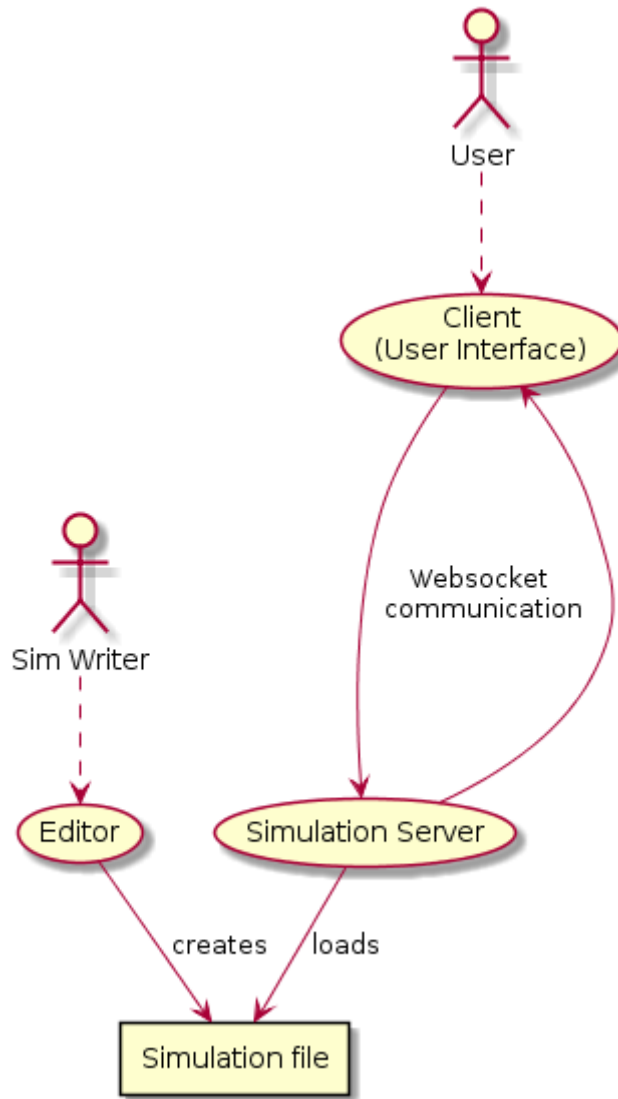


Figure 1. TS2 overall architecture

## 2.2. Clients

Clients are software that interact with the simulation server, typically to provide a user interface.

Many clients can be connected at the same time to the same simulation server.

The TS2 release ships two clients:

- The standard Python client, which is the main "all-in-one" client that is also able to spawn simulation servers and give access to the editor.
- A minimalist web client which is included with the simulation server and mainly used for API testing. This client can be accessed at <http://localhost:22222> when the sim server is running.

## 2.3. The Editor

The editor is part of the standard Python client and is used to create or modify simulations with a graphical user interface.

## 3. Simulation model

A simulation in TS2 is modelled by the following objects:

- Options
- Track items
- Routes
- Train Types
- Services
- Trains
- Signal Library
- Message Logger

### 3.1. Options

This is a list of options for the simulation. They must be set at simulation writing time, but can be modified during the simulation.

The default values in the table below are set by the editor. The simulation server itself has no default and expect all options to be set.

Key	Default Value	Description
<code>title</code>		Title of the simulation
<code>description</code>		Detailed description of the simulation targeted at the user.
<code>clientToken</code>	client-secret	Client token to connect to the simulation. Unless you want to run a public instance of TS2 you can leave it to the default value.
<code>version</code>	0.7	Defines the version of the file format. Do not change this value.
<code>timeFactor</code>	5	The number of seconds elapsed in the simulation for each real seconds. This value can be set between 1 and 10.
<code>currentTime</code>	06:00:00	Current time inside the simulation. When writing a simulation this will be the time when the simulation starts. During the simulation run, this value is updated every 500ms.
<code>warningSpeed</code>	8.33	Speed (in metres per second) a train driver will observe when given a "Proceed with caution" manual order from the dispatcher. Default value is 30 km/h.
<code>currentScore</code>	0	This value is the current penalty score of the simulation. Obviously, it should be set to 0 when writing a simulation.

Key	Default Value	Description
defaultMaxSpeed	44.44	This speed (in metres per second) will be used by the simulation whenever a track item has a maximum speed of 0. Default value is 160 km/h.
defaultMinimumStopTime	[(45, 75, 70), (75, 90, 30)]	The time in seconds a train will normally stop at a station. It can be a single value in seconds, or a <a href="#">delay generator</a> .
defaultDelayAtEntry	[(-60, 0, 50), (0, 60, 50)]	The delay in seconds a train will have by default when entering the area. It can be a single value in seconds, or a <a href="#">delay generator</a> . If the value is negative, the train will be early.  This value can be overridden train by train.
trackCircuitBased	false	This value defines the way the trains will be represented on the layout. If it is true, each track item will be considered as a track circuit and will be either marked free or occupied. If it is false, the occupied area will show the real position of the train.  This option should be set to true if you care about realism.
defaultSignalVisibility	100	Distance in metres at which a driver can see a signal and will start taking it into account.
wrongPlatformPenalty	5	Penalty points that will be added to the score each time a train stops at a wrong platform.
wrongDestinationPenalty	100	Penalty points that will be added to the score each time a train is not routed out of the area at the correct exit point.
latePenalty	1	Penalty points that will be added to the score per minute lost in the area. Delay at entry is subtracted from the actual delay to define it.

Delay generators are expressions that will yield a random value according to a specified distribution.

They are composed of a list of triplets such as:

```
[(45, 75, 70), (75, 90, 30)]
```

For each triplet, the values are in order:

- Minimum value
- Maximum value
- Percentage of occurrence

In the example above, the expression means:

- 70% of the time the value will be between 45 and 75
- 30% of the time the value will be between 75 and 90

Inside each triplet, the value is yielded with a uniform distribution.

## 3.2. Track Items

The layout of the tracks in the area is defined by 8 track item types:

- Line
- Signal
- Points
- Platform
- Place
- End
- InvisibleLink
- Text

Each type has "definition attributes" which can be set with the editor and "technical attributes" which are returned by the simulator through the API.

### 3.2.1. Common Attributes

All items share the following attributes.

#### Definition Attributes

Technical Name	Attribute Name in Editor	Description
id	ID	Unique ID of the item. The editor sets it automatically and it cannot be modified by the user.
__type__	Type	Type of the item. The type of an item cannot be changed.
name	Name (or Text)	Name of the item as known in the real world (e.g. signal number).
x	Position (or Point1)	Position of the item on the x-axis.
y	Position (or Point1)	Position of the item on the y-axis.  <b>WARNING</b> y-axis increases from top to bottom.
maxSpeed	Maximum speed (m/s)	Maximum speed allowed on this item in metres per second.
realLength	Real Length (m)	Length of this item in real life (in metres).
conflictTiId	Conflict item ID	Set to the ID of another item to prevent route setting on both items at the same time. This feature is typically used to interlock track crossovers without points.

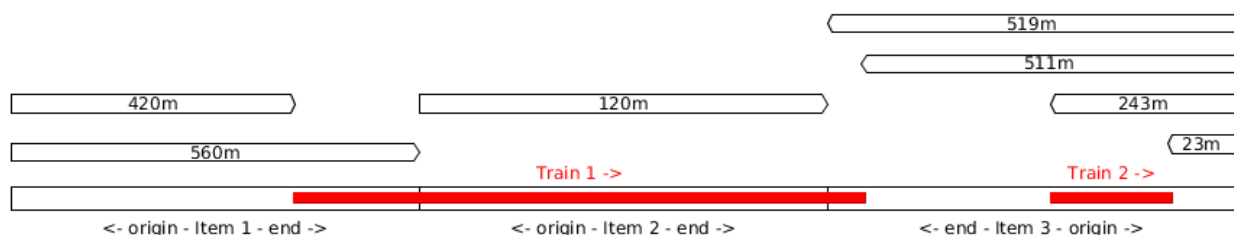
## Technical Attributes

Technical Name	Description
previousTiId	ID of the track item connected to this item at its "origin" (see each item description).  This is computed automatically by the editor.
nextTiId	ID of the track item connected to this item at its "end" (see each item description)  This is computed automatically by the editor.
activeRoute	If a route is set on this item, this value is the ID of that route, otherwise it is null.
activeRoutePreviousItem	If a route is set on this item, this value is the ID of the item before this one in the direction of the route, otherwise it is null.
trainEndsFW	Map of train extremities that are on the "end" side of this item (see each item description).  For example, {"2": 79} means that train with ID "2" has one of its extremity (head or tail) at 79 metres from this items "origin".

Technical Name	Description
<code>trainEndsBK</code>	<p>Map of train extremities that are on the "origin" side of this item (see each item description).</p> <p>For example, <code>{"2": 3}</code> means that train with ID "2" has one of its extremity (head or tail) at 3 metres from this items "origin".</p>

#### Example 1. `trainEndsFW` and `trainEndsBK`

Consider the following figure with 2 trains, going from left to right. **Train 1** spans over 3 track items, while **Train 2** is over a single track item.



In this situation, the `trainEndsBK` and `trainEndsFW` maps are as follow:

Item 1	<code>trainsEndsBK = {"1": 420}</code>  <code>trainsEndsFW = {"1": 560}</code>
Item 2	<code>trainsEndsBK = {"1": 0}</code>  <code>trainsEndsFW = {"1": 120}</code>
Item 3	<code>trainsEndsBK = {"1": 511, "2": 23}</code>  <code>trainsEndsFW = {"1": 519, "2": 243}</code>

#### NOTE

When the `trackCircuitBased` option is true, the `trainEndsBK` and `trainEndsFW` are always with a value of 0 or of the length of the item so that the latter is either completely covered by a train or not at all.

### 3.2.2. Line Items

A line item connects two points on the scenery. One is defined as the "origin" and the other one as the "end" (arbitrarily).



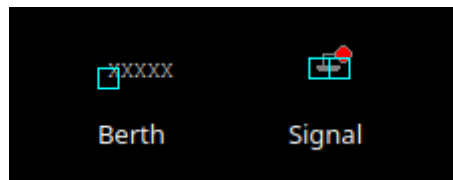
Common attributes `x` and `y` define the position of the "origin", known as "Point 1" in the editor.



Technical Name	Attribute Name in Editor	Description
<code>xf</code>	Point 2	Position of "end" on the x-axis.
<code>yf</code>	Point 2	Position of "end" on the y-axis.  <b>WARNING</b> y-axis increases from top to bottom.
<code>placeCode</code>	Place code	Code of the place item this line item belongs to. The place being a station or a waypoint.
<code>trackCode</code>	Track code	The code of this track as known in the place defined by <code>placeCode</code> . Typically a line or platform number.

### 3.2.3. Signal Items

Signal items are composed of two elements, the signal itself and the "berth" that will hold train descriptors on the layout.



#### Standard Attributes

Common attributes `x` and `y` define the position of entry in the signal which is the left point of the signal itself. Note that when the signal is reversed, then it is the point of the signal on the right.

Technical Name	Attribute Name in Editor	Description
<code>signalType</code>	Signal Type	The code of the type of signal as defined in the signal library (e.g. <code>UK_3_ASPECTS</code> )
<code>reversed</code>	Reverse	If true, then the signal is for train coming from the right of the layout.
<code>xn</code>	Berth Origin	Position of the berth on the x-axis. The position is the bottom left corner of the berth.
<code>yn</code>	Berth Origin	Position of the berth on the y-axis. The position is the bottom left corner of the berth.  <b>WARNING</b> y-axis increases from top to bottom.

#### Technical attributes

Technical Name	Description
<code>activeAspect</code>	Aspect name that this signal shows currently
<code>trainID</code>	ID of the train that has its descriptor on this signal's berth. 0 if the berth is empty.
<code>previousActiveRoute</code>	ID of the route that is set up to this signal. Empty string if none.
<code>nextActiveRoute</code>	ID of the route that is set starting from this signal. Empty string if none.

### Custom properties

Custom properties are defined by the available signal conditions. Each property takes as value a map with signal aspect codes as keys and a list of related object IDs as values, such as:

```
{"UK_CLEAR": ["2", "34", "48"], "UK_CAUTION": ["2", "34"]}
```

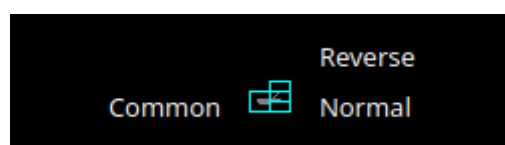
Properties taken into account depend on the signal type. The editor automatically prefills the properties depending on the signal type. The table below lists the properties that are defined by conditions in the current version.

**NOTE** See also [Signal Aspect Resolution](#)

Condition	Property Name in Editor	Related Objects	Description
<code>TRAIN_NOT_PRESENT_ON_ITEMS</code>	No Trains params	Track Items	List of items IDs on which there must not be a train for the aspect to show. If there is a train on a single item of the list, the aspect does not show.
<code>TRAIN_PRESENT_ON_ITEMS</code>	Train Present Params	Track Items	List of items IDs on which there must be a train for the aspect to show. If a train is missing on a single item of the list, the aspect does not show.
<code>ROUTES_SET</code>	Route set params	Routes	List of route IDs which can be activated for the aspect to show. The aspect shows as soon as at least one of the specified route is active.

### 3.2.4. Points Items

Points items are track switches. They have three extremity: the common, normal and reverse ends as shown below.



## Definition Attributes

Common attributes **x** and **y** define the position of the center of points item. Each extremity is at -5 or +5 along x and y axis.

Technical Name	Attribute Name in Editor	Description
<b>xf</b>	Common End	Position of the common extremity along the x-axis. Must be equal to -5, 0 or +5.
<b>yf</b>	Common End	Position of the common extremity along the y-axis. Must be equal to -5, 0 or +5.  <b>WARNING</b>   y-axis increases from top to bottom.
<b>xn</b>	Normal End	Position of the normal extremity along the x-axis. Must be equal to -5, 0 or +5.
<b>yn</b>	Normal End	Position of the normal extremity along the y-axis. Must be equal to -5, 0 or +5.  <b>WARNING</b>   y-axis increases from top to bottom.
<b>xr</b>	Reverse End	Position of the reverse extremity along the x-axis. Must be equal to -5, 0 or +5.
<b>yr</b>	Reverse End	Position of the reverse extremity along the y-axis. Must be equal to -5, 0 or +5.  <b>WARNING</b>   y-axis increases from top to bottom.

### NOTE

In the editor, these attributes are defined by setting the cardinal point of the extremity such as:

- N  $\Rightarrow$  (0, -5)
- SW  $\Rightarrow$  (-5, +5)

## Technical Attributes

Technical Name	Description
<b>reverseTiId</b>	ID of the track item connected to this item at its "reverse" end.  This is computed automatically by the editor.
<b>reverse</b>	true if the points are set to the reverse end, and false if they are set to the normal end.

### 3.2.5. Platform Items

Platform items are mostly decorative. They can be linked to a place and a track code.



Common attributes `x` and `y` define the position of "Point 1".

Technical Name	Attribute Name in Editor	Description
<code>xf</code>	Point 2	Position of Point 2 along the x-axis.
<code>yf</code>	Point 2	Position of Point 1 along the y-axis. <div><b>WARNING</b> y-axis increases from top to bottom.</div>
<code>placeCode</code>	Place code	Code of the place item this platform item belongs to. The place being a station or a waypoint.
<code>trackCode</code>	Track code	The code of this platform as known in the place defined by <code>placeCode</code> . Typically a platform number.

### 3.2.6. End Items

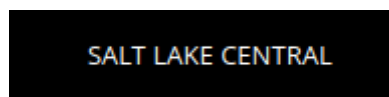
End items are technical items used to connect free extremities of the simulation.



All extremities, including those after a buffer **MUST** be filled with an end item, so that all items are linked to other items.

### 3.2.7. Place Items

Places represent stations or waypoints. They are represented by a text label on the scenery.



The common attribute `name` is the name of the place as displayed.

Technical Name	Attribute Name in Editor	Description
<code>placeCode</code>	Place code	Code that will be used to reference this place in other items.

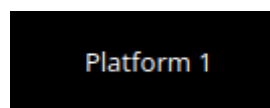
### 3.2.8. InvisibleLink Items

Invisible links work exactly the same way as line items, but are not represented at all on the scenery.



### 3.2.9. Text Items

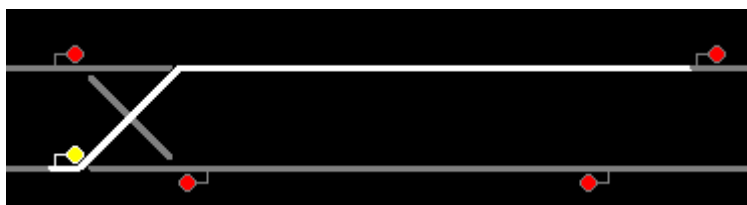
Text items are purely decorative. Use them to add labels which are not station or waypoint names, such as track or platform no.



The caption is set through the `name` attribute.

## 3.3. Routes

A route is a locked path from one signal to another signal. It will lock all the points in the correct position and open the entry signal. The route is locked until either a train passes over, or the signaller cancels the route manually.



A route can be set as "persistent". In this case, it cannot be released by a train and must be cancelled manually.

#### 3.3.1. Definition Attributes

Technical Name	Attribute Name in Editor	Description
<code>id</code>	Route no.	Route ID used to reference this Route. It is set automatically by the editor and cannot be changed.
<code>beginSignal</code>	Begin Signal	ID of the entry signal item on the scenery.
<code>endSignal</code>	End Signal	ID of the exit signal item on the scenery.
<code>initialState</code>	Initial state	State of the route at the beginning of the simulation. Takes a <a href="#">Route State Value</a>

### Route States Values

Value	Description
0	Route is not set
1	Route is set, with automatic release when a train passes over
2	Route is set and persistent

### 3.3.2. Technical Attributes

Technical Name	Attribute Name in Editor	Description
state	Current state	Current state of the route. Takes a <a href="#">Route State</a> Value
directions	Points directions	<p>Map of the points directions along this route. Each key is a points item ID and the value a <a href="#">Points Position</a> value such as <code>{"512":1,"521":1}</code></p> <p>If a points does not appear in the map, either its position is obvious (i.e. route goes from the normal or reverse end to the common end) or it is assumed that it is in the "normal" position.</p>

### Points Positions Values

Value	Description
0	Normal position
1	Reversed position (i.e. diverging)
2	Unknown position, usually because the points are moving  (not implemented yet, reserved for future releases)
3	Points have a failure  (not implemented yet, reserved for future releases)

## 3.4. Train Types

Train types are the different kinds of rolling stock available in the simulation.

Technical Name	Attribute Name in Editor	Description
<code>id</code>	Code	Code used to reference this train type
<code>description</code>	Description	Human readable description of this train type (e.g. "Class 313/2 EMU")
<code>length</code>	Length (m)	Length in metres of this rolling stock
<code>maxSpeed</code>	Max speed (m/s)	Maximum speed in metres per second
<code>stdAccel</code>	Std acceleration (m/s <sup>2</sup> )	Standard acceleration in metres per square second. A train will always speed up on a constant ramp (over time) defined by this value.
<code>stdBraking</code>	Std braking (m/s <sup>2</sup> )	Standard braking in metres per square second. A train will slow down on a constant ramp (over time) defined by this value when it can foresee a speed limit ahead with sufficient prior notice.
<code>emergBraking</code>	Emerg. braking (m/s <sup>2</sup> )	Maximum braking capacity in metres per square second. When a speed limit arises without sufficient prior notice, the train will brake as much as it can with a constant ramp (over time), not exceeding this value.
<code>elements</code>	Elements (codes list)	<p>List of other train type codes this rolling stock is composed of, such as <code>["C313-2", "C313-2"]</code></p> <p>This information will be used in future version for splitting/joining trains.</p>

## 3.5. Services

Services are train schedules. A service is composed of several lines, defined by a place and a time.

### 3.5.1. Service Attributes

Technical Name	Attribute Name in Editor	Description
<code>id</code>	Code	Code used to reference this service. It is the code that will be used as the train descriptor on the layout.
<code>description</code>	Description	Free human readable description of the service.
<code>plannedTrainType</code>	Planned Train Type	The train type code that is expected for this service.
<code>postActions</code>	Next service code / Auto reverse	<p>Actions to be performed automatically by a train when it terminates this service. It must be a list of <a href="#">train actions</a>.</p> <p>e.g.</p> <pre><code>"postActions":[{"actionCode":"SET_SERVICE","actionParam":"WB02"}, {"actionCode":"REVERSE","actionParam":""}]</code></pre>

Technical Name	Attribute Name in Editor	Description
<code>lines</code>		Lines of this service. It is a list of <a href="#">service lines</a> as defined below.

### Train Actions

A train action is a map with two keys:

<code>actionCode</code>	The code of the action to perform (see below).
<code>actionParam</code>	The parameters for the action (if applicable for the given action).

Currently two actions are implemented

Action Code	Action Parameters	Description
<code>SET_SERVICE</code>	service code	Assign the service with the given service code to this train.  In the editor, this action is set by filling in the "Next service code".
<code>REVERSE</code>	None	Reverse the train direction.  In the editor, this action is set by the "Auto reverse" field.

### 3.5.2. Service Line Attributes

Technical Name	Attribute Name in Editor	Description
<code>placeCode</code>	Place code	Code of the place (station or waypoint)
<code>scheduledArrivalTime</code>	Arrival Time	Time at which the train is expected to arrive at the place of this line.  Should be left empty (i.e. "00:00:00") when the train does not stop at this place.
<code>scheduledDepartureTime</code>	Departure Time	Time at which the train is expected to depart (or pass) at the place of this line.
<code>mustStop</code>	Stop	Set to <code>true</code> if the train must stop at this station.
<code>trackCode</code>	Track code	Track or platform no. at which this train is expected to stop (or pass) at this place.



## 3.6. Trains

Trains are the rolling stock instances that run in the Simulation. Most of the time a service is assigned to a train.

### 3.6.1. Definition Attributes

Technical Name	Attribute Name in Editor	Description
<code>id</code>	<code>id</code>	Internal ID of the train, automatically assigned.  <b>NOTE</b> This ID is an integer and can be different in the editor and in the simulation.
<code>serviceCode</code>	Service code	ID of the service assigned to this train
<code>trainTypeCode</code>	Train type	ID of the rolling stock type of this train
<code>appearTime</code>	Entry time	Time at which this train appears on the layout
<code>trainHead</code>	Entry position	Position of the train head. This is a <a href="#">position</a> object.
<code>initialSpeed</code>	Entry speed	Speed of this train when it appears on the scenery
<code>initialDelay</code>	Initial delay	Delay from <code>appearTime</code> that this train will have when entering the area. This field is either an integer (in seconds) or a <a href="#">delay generator</a> .  Set this field to 0 to use the <code>defaultDelayAtEntry</code> value from the <a href="#">options</a>

## Positions

A position object uniquely defines a position and a direction on the scenery.

Attribute Name	Description
<code>trackItem</code>	ID of the item this position is on.
<code>previousTI</code>	ID of one of the connected item defined as "Previous Item" to give the direction.
<code>positionOnTI</code>	Number of metres between this position and the extremity of <code>trackItem</code> that is connected to <code>previousTI</code> .

On the image below, positions `P1` and `P2` are defined as follow:

```
P1 = {"trackItem":"2","previousTI":"1","positionOnTI":73}
```

```
p2 = {"trackItem":"2","previousTI":"3","positionOnTI":98}
```

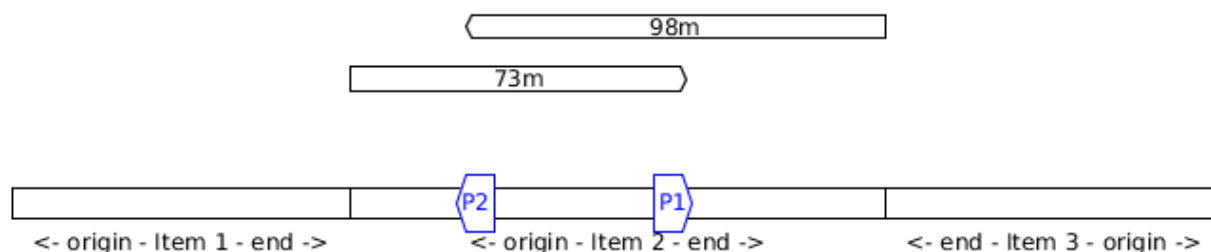


Figure 2. Positions

### 3.6.2. Technical Attributes

Technical Name	Description
<code>status</code>	Current status of the train. See available <a href="#">train status values</a> .
<code>speed</code>	Current speed of the train in metres per second
<code>nextPlaceIndex</code>	Index of the next service line, i.e. the index to the next station or waypoint. Counted from 0.
<code>stoppedTime</code>	The number of seconds the train has stopped at the station. If the train status is not "Stopped", this value has no meaning.

### Train Status Values

Code	Status	Description
0	Inactive	The train has not entered the area yet
10	Running	The train is running with a non zero speed
20	Stopped	The train is stopped at a station
30	Waiting	The train is waiting at a red signal or other unscheduled stop
40	Out	The train exited the area
50	EndOfService	The train has finished its service and has not been assigned a new one

### 3.6.3. Standard Train Behaviour

Train behaviours are defined in compile-time plugins called train managers. TS2 ships by default with a "Standard Manager" which makes the trains behave as described in this section.

The train driver will always try to reach the maximum possible speed limited by :

- The train type's maximum speed
- The speed limit of the line, both the current limit and reduced speed limits ahead
- The distance to the next station the train should stop
- The speed limit imposed by signals

For speed limits ahead (such as reduced line speed or next station or signal aspect), the maximum speed allowed is defined by a constant speed ramp (over time) of `stdBraking` (or `stdAccel`) in order to be at the target speed at the target point.

## 3.7. Signal Library

The Signal Library holds the information about each signal available in the simulation. It is composed of a list of "Signal Aspects" and "Signal Types".

Technical Name	Description
<code>signalAspects</code>	<p>Map of signal aspects. The key is the name of the aspect, the value is a <a href="#">signal aspect object</a>.</p> <p>A signal aspect is the colour of a signal lamp or combination of lamps on one signal. The signal aspect provides an unambiguous message to the driver of a train. In TS2, this message is a list of actions to perform.</p>

Technical Name	Description
<code>signalTypes</code>	<p>Map of signal types. The key is the name of the type, the value is a <a href="#">signal type object</a>.</p> <p>A signal type defines a kind of signal capable of displaying a set of aspects depending on conditions.</p> <div> <div>NOTE</div> <div>The signal type usually differs between the simulation and reality, as a signal type in the simulation can be configured to simulate several real types.</div> </div>

### 3.7.1. Signal Aspects

#### General Attributes

Technical Name	Description									
name	Code of the signal aspect. This code must be unique and is used to reference this aspect in the simulation.									
lineStyle	<div>Defines how the line along the signal must be displayed. Possible values are:</div> <table><tr><th>Code</th><th>Style</th><th>Description</th></tr><tr><td>0</td><td>lineStyle</td><td>Normal signal placed on the side of the line</td></tr><tr><td>1</td><td>bufferStyle</td><td>Buffer</td></tr></table>	Code	Style	Description	0	lineStyle	Normal signal placed on the side of the line	1	bufferStyle	Buffer
Code	Style	Description								
0	lineStyle	Normal signal placed on the side of the line								
1	bufferStyle	Buffer								
actions	<div>List of actions to be done by the train driver when seeing this signal. See <a href="#">signal actions</a></div> <div>Examples</div> <table><tr><td>[[2, 0]]</td><td>Prepare to stop before the next signal.</td></tr><tr><td>[[1, 0, 60], [0, 8.33]]</td><td>Stop before this signal, wait 60 seconds and proceed at 8.33 m/s (30 km/h).</td></tr></table>	[[2, 0]]	Prepare to stop before the next signal.	[[1, 0, 60], [0, 8.33]]	Stop before this signal, wait 60 seconds and proceed at 8.33 m/s (30 km/h).					
[[2, 0]]	Prepare to stop before the next signal.									
[[1, 0, 60], [0, 8.33]]	Stop before this signal, wait 60 seconds and proceed at 8.33 m/s (30 km/h).									

## Signal Actions

A signal action is a triplet with, in order:

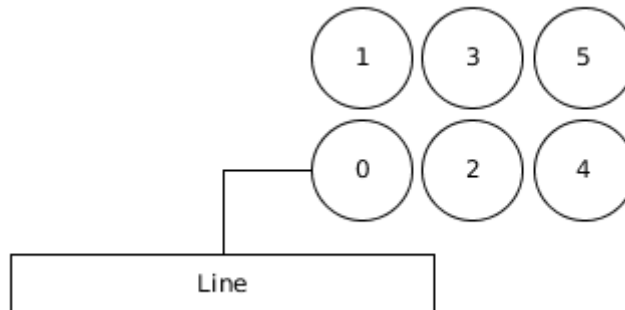
- A [target](#)
- A speed limit (in m/s) to respect at target
- A delay in seconds before executing the next action (optional if there is no next action)

Table 1. Signal Action Targets

Code	Target	Description
0	ASAP	The target speed should be applied as soon as possible
1	BeforeThisSignal	The target speed should be applied before the train reaches this signal
2	BeforeNextSignal	The target speed should be applied before the train reaches the signal after this one.

## Display Attributes

Signal aspects in TS2 can show up to 6 lamps at the same time (numbered 0 to 5) that are arranged like this:
















Attributes in the table below are lists. Each item refer to a lamp based on its index (counted from 0). Clients are expected to render signals as explained below.

Technical Name	Description
<code>outerShapes</code>	List of outer shapes. Outer shapes should be drawn first. Each item of the list must be <a href="#">shape code</a> .
<code>outerColors</code>	List of colors to fill the outer shapes. Each item of the list must be a <code>#RRGGBB</code> color string.
<code>shapes</code>	List of shapes. Shapes should be drawn in front of <code>outerShapes</code> without transparency. Each item of the list must be <a href="#">shape code</a> .
<code>shapesColors</code>	List of colors to fill the shapes. Each item of the list must be a <code>#RRGGBB</code> color string.

Technical Name	Description
<b>blink</b>	List of boolean values. If the value is true, the corresponding lamp should be displayed as flashing.

### Shape Codes

Code	Shape	Image
0	<b>none</b>	Nothing should be drawn at the position of the corresponding lamp.
1	<b>circle</b>	
2	<b>square</b>	
10	<b>quarterSW</b>	
11	<b>quarterNW</b>	
12	<b>quarterNE</b>	
13	<b>quarterSE</b>	
20	<b>barNS</b>	
21	<b>barEW</b>	
22	<b>barSWNE</b>	
23	<b>barNWSE</b>	
31	<b>poleNS</b>	

Code	Shape	Image
32	poleNSW	
33	poleSW	
34	poleNE	
35	poleNSE	

#### NOTE

Cardinal points in the Shape and images code should be understood **with the signal head up** (i.e. N to the right of the screen, or to the left is signal is reversed).



```
{
  "__type__": "SignalAspect",
  "actions": [[1, 0]],
  "blink": [false, false, false, false, false, false],
  "lineStyle": 0,
  "outerColors": ["#000000", "#000000", "#000000",
"#000000", "#000000", "#000000"],
  "outerShapes": [0, 0, 0, 0, 0, 0],
  "shapes": [1, 0, 0, 0, 0, 0],
  "shapesColors": ["#00FF00", "#000000", "#000000",
"#000000", "#000000", "#000000"]
}
```



```
{
  "__type__": "SignalAspect",
  "actions": [[0, 999]],
  "blink": [false, false, false, false, false, false],
  "lineStyle": 0,
  "outerColors": ["#FFFFFF", "#000000", "#000000",
"#000000", "#000000", "#000000"],
  "outerShapes": [2, 0, 0, 0, 0, 0],
  "shapes": [1, 0, 0, 0, 0, 0],
  "shapesColors": ["#FF0000", "#000000", "#000000",
"#000000", "#000000", "#000000"]
}
```



```
{
  "__type__": "SignalAspect",
  "actions": [[1, 0]],
  "blink": [false, false, false, false, false, false],
  "lineStyle": 0,
  "outerColors": ["#000000", "#000000", "#000000",
"#FFFF00", "#000000", "#000000"],
  "outerShapes": [0, 0, 0, 1, 0, 0],
  "shapes": [1, 0, 32, 22, 12, 0],
  "shapesColors": ["#FF0000", "#000000", "#000000",
"#FFFFFF", "#FF00FF", "#000000"]
}
```

**NOTE**      **quarterNE** (12) is rendered on the image as a triangle instead of a quarter.



### 3.7.2. Signal Types

A signal type defines a signal that can display several aspects depending on conditions.

Technical Name	Description
<code>states</code>	Ordered list of signal states.

#### Signal States

A Signal state is the combination of a signal aspect and conditions to have it displayed.

Technical Name	Description
<code>aspectName</code>	Name of the signal aspect attached to this state
<code>conditions</code>	Map of conditions to be met for this signal aspect to be displayed. Keys are condition names and values are lists of parameters (depending on the condition).  See also <a href="#">available conditions</a> .

#### Conditions

The table below describes the different conditions that exist to define a signal type.

Condition Name	Parameters	Description
<code>NEXT_ROUTE_ACTIVE</code>	<code>[]</code>	Met if a route is set starting from this signal.
<code>PREVIOUS_ROUTE_ACTIVE</code>	<code>[]</code>	Met if a route is set ending at this signal.
<code>ROUTE_SET_ACROSS</code>	<code>[]</code>	Met if a route is active across this signal, in the same direction but neither starting nor ending at this signal (e.g. an intermediate shunting signal).
<code>TRAIN_NOT_PRESENT_ON_NEXT_ROUTE</code>	<code>[]</code>	Met if a route is active starting from this signal and no trains are present on this route.  If no route is active from this signal, the condition is met if no trains are found until the next signal on the line.
<code>TRAIN_NOT_PRESENT_ON_ITEMS</code>	<code>[]</code> *	Met if none of the items defined in the signal's <code>customProperties</code> for this signal type and aspect have a train on them.
<code>TRAIN_PRESENT_ON_ITEMS</code>	<code>[]</code> *	Met if all of the items defined in the signal's <code>customProperties</code> for this signal type and aspect have a train on them.

Condition Name	Parameters	Description
ROUTES_SET	[]*	Met if at least one of the route defined in the signal's <code>customProperties</code> for this signal type and aspect is active.
NEXT_SIGNAL_ASPECTS	List of signal aspect names	Met if the next signal shows one of given aspect. The next signal is the exit signal of the route starting at this signal if any. Otherwise it is the next signal on the line.

\*: These conditions parameters are empty in the signal library as they take their parameters from the signal's `customProperties`

### 3.7.3. Signal Aspect Resolution

When a signal is given a signal type, signal aspect resolution can take place. The `states` list is taken in order and the conditions are checked for each state. The first state that have all its conditions met is taken into account: its signal aspect is displayed and any further state is discarded.

Thus, the last state of a signal type should be the most restrictive aspect with no condition.

#### Example

##### Signal Type Definition

```
"US_INTERLOCK": {
  "states": [
    {
      "aspectName": "US_DIVERGING_CLEAR",
      "conditions": {
        "ROUTES_SET": [],
        "TRAIN_NOT_PRESENT_ON_NEXT_ROUTE": [],
        "NEXT_SIGNAL_ASPECTS": [
          "US_CLEAR",
          "US_DIVERGING_CLEAR",
          "US_APPROACH",
          "US_DIVERGING_APPROACH"
        ]
      }
    },
    {
      "__type__": "SignalState",
      "aspectName": "US_CLEAR",
      "conditions": {
        "NEXT_ROUTE_ACTIVE": [],
        "TRAIN_NOT_PRESENT_ON_NEXT_ROUTE": [],
        "NEXT_SIGNAL_ASPECTS": [
          "US_CLEAR",
          "US_DIVERGING_CLEAR",
          "US_APPROACH",
          "US_DIVERGING_APPROACH"
        ]
      }
    }
  ]
}
```

```

    }
  },
  {
    "__type__": "SignalState",
    "aspectName": "US_DIVERGING_APPROACH",
    "conditions": {
      "ROUTES_SET": [],
      "TRAIN_NOT_PRESENT_ON_NEXT_ROUTE": [],
      "NEXT_SIGNAL_ASPECTS": [
        "US_STOP",
        "US_RESTRICTING",
        "BUFFER"
      ]
    }
  },
  {
    "__type__": "SignalState",
    "aspectName": "US_APPROACH",
    "conditions": {
      "NEXT_ROUTE_ACTIVE": [],
      "TRAIN_NOT_PRESENT_ON_NEXT_ROUTE": [],
      "NEXT_SIGNAL_ASPECTS": [
        "US_STOP",
        "US_RESTRICTING",
        "BUFFER"
      ]
    }
  },
  {
    "__type__": "SignalState",
    "aspectName": "US_STOP",
    "conditions": {}
  }
]
}

```

## Signal Definition

```
"22": {
  "conflictTiId": null,
  "customProperties": {
    "ROUTES_SET": {
      "US_DIVERGING_APPROACH": [
        "6"
      ],
      "US_DIVERGING_CLEAR": [
        "6"
      ]
    },
    "TRAIN_NOT_PRESENT_ON_ITEMS": {},
    "TRAIN_PRESENT_ON_ITEMS": {}
  },
  "maxSpeed": 0.0,
  "name": "22",
  "nextTiId": "27",
  "previousTiId": "21",
  "reverse": false,
  "signalType": "US_INTERLOCK",
  "tiId": "22",
  "x": 675.0,
  "xn": 630.0,
  "y": 270.0,
  "yn": 275.0
}
```

	Situation	Aspect Shown
1	<ul style="list-style-type: none"><li>Route "6" is not set</li><li>There are no trains anywhere</li><li>Next signal shows <b>US_CLEAR</b></li></ul>	<b>US_CLEAR</b>
2	<ul style="list-style-type: none"><li>Route "6" is set</li><li>There are no trains anywhere</li><li>Next signal shows <b>US_STOP</b></li></ul>	<b>US_DIVERGING_APPROACH</b>
3	<ul style="list-style-type: none"><li>Route "6" is not set</li><li>There is a train just after this signal</li><li>Next signal shows <b>US_CLEAR</b></li></ul>	<b>US_STOP</b>

Now let's explain each case:

### Case no. 1

- First state (for `US_DIVERGING_CLEAR` aspect) does not meet condition for `ROUTES_SET` because route "6" is not active.

This is defined in the signal's `customProperties`: for `ROUTES_SET` and `US_DIVERGING_CLEAR` aspect, we should have route "6" active.

- The second state (for `US_CLEAR` aspect) conditions are all met. This aspect is shown and any further states are discarded.

#### Case no. 2

- First state (for `US_DIVERGING_CLEAR` aspect) fails for the `NEXT_SIGNAL_ASPECTS` condition as `US_STOP` is not in the list.
- Second state (for `US_CLEAR`) also fails for the `NEXT_SIGNAL_ASPECTS` condition.
- Third state (for `US_DIVERGING_APPROACH`) conditions are all met. This aspect is displayed.

#### Case no. 3

- The first four states fail on the `TRAIN_NOT_PRESENT_ON_NEXT_ROUTE` condition
- The last state (for `US_STOP`) has no condition and acts as a fallback

## 3.8. Message Logger

The message logger of the simulation has a single attribute `messages` which is a list of message objects.

### 3.8.1. Messages

Technical Name	Description
<code>msgType</code>	Code of the <a href="#">type of message</a>
<code>msgText</code>	Text of the message

#### Message Types

Code	Type	Description
0	<code>softwareMsg</code>	Message logged by the server (e.g. Simulation loading)
1	<code>playerWarningMsg</code>	Message logged following a user manipulation error (e.g. route does not exist). Not used by the server.
2	<code>simulationMsg</code>	Message logged by the simulation (e.g. Train XXXX entered the area)

## 4. Writing a simulation

This section gives a few hints on how to create a simulation with the editor.

## 4.1. Load the Signal Library

Before starting your simulation, make sure that the signal library that you will need is loaded. You can check in the editor scenery tab: put a signal onto the layout and check in the **type** property that you can select signal type's of your signal library.

TS2 ships by default signal libraries for UK, France and USA. These libraries can be downloaded in the "Open" dialog, by clicking the "Download" button.

You can also add custom signal libraries by directly putting the **.tsl** file in the **~/.ts2/data** directory. **.tsl** files are JSON files with the definition a Signal Library as described [here](#).

**NOTE** | You need to restart TS2 for the new Signal Libraries to be taken into account.

## 4.2. Setup the layout

The first thing to do when writing a simulation is to create the layout in the scenery tab.

The scenery has two states:

### Unlocked

You can modify the layout, so it might not be valid. A valid scenery is when all the items are linked.

### Validated

The layout is locked and the scenery is valid. It must be in this state before setting routes.

### 4.2.1. Adding a new item

To add a new item, first check that the scenery is unlocked.

Click and drag an item from the **tools** pane onto the layout.

### 4.2.2. Editing an item

Click on the item to edit: it should turn pink. Edit the properties in the **properties** pane.

The following properties can be edited directly from the layout:

Position	Click and drag the item to change the value automatically.
Point 1	Click on the extremity of the item and drag it to change the value automatically.
Point 2	
Berth Origin	Click on the berth of the signal and drag it to change the value automatically.
Reverse	Right click on a signal to reverse its direction.

### 4.2.3. Mass editing items

You can set properties to several items at once.

To select multiple items, you can:

- Select an item by clicking on it, then hold **ctrl** key and click on other items
- In the **Edit** menu select **Selection tool**. Then draw a rectangle on the layout to select all items inside it.

When the items are selected, you can edit any properties that is common to all of them. In particular, you can move them at once by dragging them.

### 4.2.4. Deleting an item

Select one or several items to delete. Press the **del** key on your keyboard.

### 4.2.5. CSV Export / Import

You can export all items as a CSV list, edit it and import it again.

#### WARNING

When you import a CSV file, it will delete all existing items. Make sure that you have all the items in the imported file.

### 4.2.6. Layout tips

#### Connect ALL items

All items should be connected at each of its end for the simulation to validate.

There is a special "End Item" that is used to connect free extremities of the layout. This includes buffers and tracks leading out of the area.

For items and points, make sure you did not connect twice the same extremity, leaving the other one not connected, as this is not visible at first sight.

#### Buffers are signals

Don't look for a buffer item, it does not exist. Buffers are actually "Always red" signals.

Place a signal at the end of the line, and select the "BUFFER" type. Don't forget to add an "End item" on the other side to connect the free extremity.

#### 0, 45 or 90

Have line always horizontal, vertical or at 45° angle whenever possible.



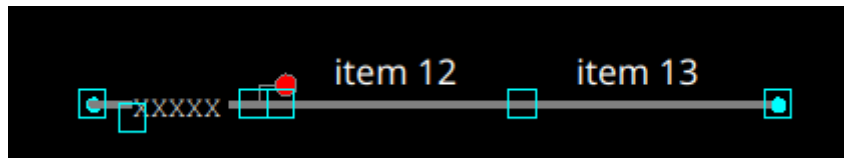
While having other angles is possible, it will make strange line breaks, especially with points.



## Last signal

The last signal on the extremity of the layout where the train will exit the area has no next signal to determine its state. To handle this situation, the standard signal libraries have special signal types ending with **\_TP**, meaning "Train Presence". These signal types will typically show their "Clear" and "Caution" aspects when there are no trains on the following items.

Typical construction is as follows:



For **UK\_3\_ASPECTS\_TP** signal type, for instance, the "Not train params" would be set to

```
{'UK_CLEAR': ["12", "13"], 'UK_CAUTION': ["12"]}
```

so that it shows :

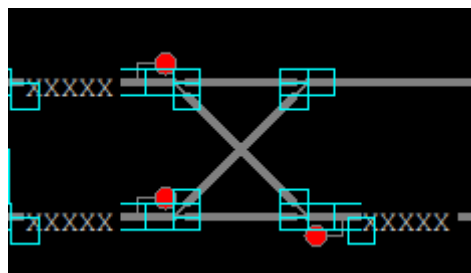
- "Clear" if there are no trains on item 12 and item 13,
- "Caution" if there are no trains on item 12.

### NOTE

A variant of this construction is to have item 13 as an invisible link, giving the impression that the signal turns to "Caution" when the train leaves the area and to "Clear" after a while.

## Level crossover

This is a construction like in the "drain" demo simulation:

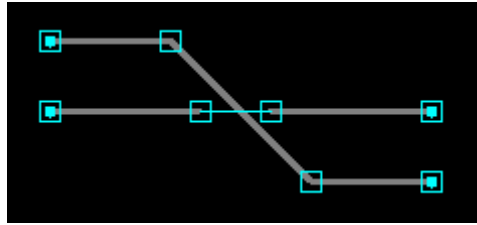


Set the "Conflict Item" on each item of the cross to the other one to prevent a route top-left/bottom-right to be set at the same time as a bottom-left/top-right route although they do not share any item.

## Bridge crossover

Use invisible links to nicely represent tracks going over or under another one:





This will render like this:



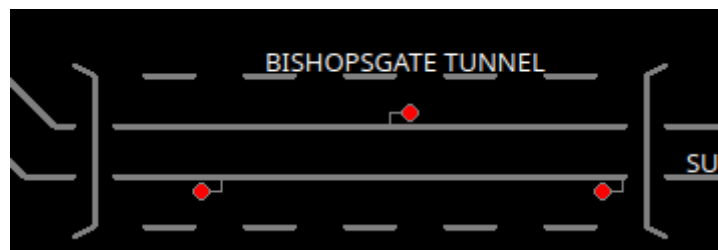
### Use lines for decoration

Sometimes, you need to add decorative graphics. You can use line items for this purpose.

This is the case in the "London Liverpool Street" simulation to represent "Bishopsgate tunnel".



This will render like this:



## 4.3. Define Routes

Routes are defined in the "Routes" tab.

### 4.3.1. Adding a route

To define a route, follow these steps:

1. Click on each point on the route to change their position and have all points correct.
2. Click on the entry signal. Its line should turn blue.
3. Click on the exit signal. The route should highlight. If it doesn't, it means that TS2 could not connect the entry signal to the exit signal with the points in their current position.
4. Click on "Add route" to register the route. If the route already exists, nothing happens and a

message is displayed in the status bar at the bottom of the screen.

**NOTE** Right click on the entry signal of a highlighted route to abort route definition.

### 4.3.2. Deleting a route

When you select a route in the route's table, it will be highlighted on the layout above.

Find the route you want to delete in the route's table, then click on "Delete Route".

### 4.3.3. CSV Export / Import

You can export all the routes as a CSV list, edit it and import it again.

**WARNING** When you import a CSV file, it will delete all existing routes. Make sure that you have all the routes in the imported file.

## 4.4. Define Train Types

Managing train types is straightforward. Refer to [Train Types](#) section for the explanation of each attribute.

You can export all the train types as a CSV list, edit it and import it again.

**WARNING** When you import a CSV file, it will delete all existing train types. Make sure that you have all the train types in the imported file.

## 4.5. Define Services

Defining a service is also straightforward. A service is a train schedule, with its main attributes displayed in the top table. Each service contains several lines referring to a station or a waypoint. When you click on a service, you can see/edit its lines in the bottom table.

You can export all the services as a CSV list, edit it and import it again.

**NOTE** The service lines are represented by the repetition of the last columns. We strongly advise you to create a service with three lines in the editor and export it to understand the schema.

**WARNING** When you import a CSV file, it will delete all existing services. Make sure that you have all the services in the imported file.

## 4.6. Define Trains

### 4.6.1. Adding trains from services

The usual and quickest way to add trains is to click on "Setup trains from services". This will create one train per service that is not following another one.

**WARNING** Clicking on "Setup trains from services" will remove all existing trains.

Then you can modify each train manually.

### 4.6.2. Adding a new train manually

Click on "Add new" button to create a new train. The train will appear at the bottom of the list. You can now modify its attributes. See [train attributes](#) for more details.

**TIP** You can set the position of the train head by selecting a train and click on the layout above. A yellow arrow will show the train head position. You can reverse it by clicking on "Reverse direction".

### 4.6.3. Deleting a train

Select the train you want to delete and click the "Delete" button.

## 5. Websocket API

### 5.1. URI

The TS2 Simulation Server exposes 2 endpoints:

- Websocket endpoint at `ws://<SERVER>:2222/ws`
- HTTP Web client endpoint at `http://<SERVER>:2222`

Where `<SERVER>` is the hostname or the IP of the server (e.g. `localhost` if you started the server on your computer).

### 5.2. Initializing a websocket connection

1. Open a connection to the websocket endpoint.
2. The first request to the server MUST be a valid login request. Otherwise, the connection will be shut down by the server. A login request has the following format:

```
{
  "object": "server",
  "action": "register",
  "params": {
    "type": "client",
    "token": "<TOKEN>"
  }
}
```

Where **<TOKEN>** is the simulation's **clientToken** defined in the **options**. It defaults to **client-secret** if it has not been customized.

3. The server will return a **status message** with **OK** result if the login request succeeded.

## 5.3. Requesting data from the server

### 5.3.1. Request format

You can retrieve data from the server at any time by sending a request message over the websocket connection. Request messages have the following syntax:

```
{
  "id": <ID>,
  "object": "<OBJECT>",
  "action": "<ACTION>",
  "params": <PARAMS>
}
```

The **"id"** parameter is optional, but highly recommended in production. When set, **<ID>** is an optional integer that will be returned by the server in the response so that the client can match the response with the request.

The **"params"** attribute format depends on the requested actions and is optional if the action has no parameters.

The tables in the following sections shows all possible actions and their parameters that can be requested from the server.

### 5.3.2. Response format

All messages received from the server have the following format:

```
{
  "msgType": "<MSG_TYPE>",
  "id": <ID>,
  "data": <PAYLOAD>
}
```

<MSG\_TYPE> is either:

- **response** if it is a direct response to a client request. In this case <ID> is the ID sent in the request or 0 if there where none.
- **notification** if it is a message sent by the server following a fired event. In this case, the "id" attribute is not sent.

<PAYLOAD> is the actual data of the response. Its format depends on the request

### Status Message

A status message is returned by the server to acknowledge a request that does not require the server to send data. Its format is the following:

```
{
  "msgType": "response",
  "id": <ID>,
  "data": {
    "status": "<STATUS>",
    "message": "<MSG>"
  }
}
```

- <STATUS> is either **OK** or **KO**
- <MSG> is a human readable message explaining the situation.

### 5.3.3. server Object

Action	Params	Returned payload	Description
register	{ "type": "client", "token": "<TOKEN>" }	Status Message	Register this client in the simulation. See <a href="#">websocket connection</a> .  <TOKEN> is the simulation's <b>clientToken</b> defined in the <a href="#">options</a>

Action	Params	Returned payload	Description
addListener	{"event": "<EVENT>", "ids": [<IDS>]}	Status Message	<p>Add a listener to the given event, to get notified each time this event is fired.</p> <p>&lt;EVENT&gt; is the name of a simulation event. &lt;IDS&gt; is a list of object ids that we listen (strings except trains which have integer ids). If no ids are given, then the listener is added for all objects concerned by the event.</p>
removeListener	{"event": "<EVENT>"}	Status Message	<p>Removes all listeners to the given event.</p> <p>&lt;EVENT&gt; is the name of a simulation event.</p>
renotify	{}	Status Message	<p>Ask the simulation to be immediately notified of the last event of each kind that we listen to.</p> <p>The server will simply return an <b>OK</b> status message and the actual notifications will be pushed as normal notifications, independently from this request.</p>

#### 5.3.4. simulation Object

Action	Params	Returned payload	Description
start	{}	Status Message	Start the simulation.
pause	{}	Status Message	Pause the simulation.
isStarted	{}	true or false	<p>Request the simulation state.</p> <p>Returns <b>true</b> if the simulation is started and <b>false</b> if it is paused.</p>
dump	{}	Simulation object	<p>Request the simulation data.</p> <p>Returns a complete dump of the simulation at the current state.</p>

#### 5.3.5. option Object

Action	Params	Returned payload	Description
list	{}	Map of options	Returns the current options values
set	{"name": <OPTION_KEY>, "value": <VALUE>}	Status Message	Set the option given by <OPTION_KEY> to the given <VALUE>.

### 5.3.6. route Object

Action	Params	Returned payload	Description
list	{}	Map of route objects indexed by their id.	Returns all the routes of the simulation.
show	{"ids": [<IDs>]}	Map of route objects indexed by their id.	Returns the routes of the simulation with the given string <IDs>.
activate	{"id": "<ID>"}	Status Message	Request activation of the route with the given <ID>.
deactivate	{"id": "<ID>"}	Status Message	Request deactivation of the route with the given <ID>.

### 5.3.7. train Object

Action	Params	Returned payload	Description
list	{}	List of train objects.	Returns all the trains of the simulation.
show	{"ids": [<IDs>]}	List of route objects.	Returns the trains of the simulation with the given integer <IDs>.
reverse	{"id": <ID>}	Status Message	Request that the train with the given integer <ID> reverses. The train will reverse only if it is stopped.
proceed	{"id": <ID>}	Status Message	Request that the train with the given integer <ID> proceeds with caution. If the train is stopped in front of a red signal, this instructs it to pass the signal.
setService	{"id": <ID>, "service": "<SERVICE_CODE>"}	Status Message	Assign the service with the given <SERVICE_CODE> to the train with the given integer <ID>.
resetService	{"id": <ID>}	Status Message	Restart the current service on the train with the given integer <ID>. The train will now expect to stop at the station of the first line of the service.

### 5.3.8. `trackItem` Object

Action	Params	Returned payload	Description
<code>list</code>	<code>{}</code>	Map of <code>track items objects</code> indexed by their <code>id</code> .	Returns all the items of the simulation.
<code>show</code>	<code>{"ids": [&lt;IDs&gt;]}</code>	Map of <code>track items objects</code> indexed by their <code>id</code> .	Returns the items of the simulation with the given string <code>&lt;IDs&gt;</code> .

### 5.3.9. `place` Object

Action	Params	Returned payload	Description
<code>list</code>	<code>{}</code>	Map of <code>place objects</code> indexed by their <code>placeCode</code> .	Returns all the places of the simulation.
<code>show</code>	<code>{"ids": [&lt;PLACE_CODES&gt;]}</code>	Map of <code>place objects</code> indexed by their <code>placeCode</code> .	Returns the place of the simulation with the given string <code>&lt;PLACE_CODES&gt;</code> .

### 5.3.10. `trainType` Object

Action	Params	Returned payload	Description
<code>list</code>	<code>{}</code>	Map of <code>train type objects</code> indexed by their <code>id</code> .	Returns all the items of the simulation.
<code>show</code>	<code>{"ids": [&lt;IDs&gt;]}</code>	Map of <code>train type objects</code> indexed by their <code>id</code> .	Returns the train types of the simulation with the given string <code>&lt;IDs&gt;</code> .

### 5.3.11. `service` Object

Action	Params	Returned payload	Description
<code>list</code>	<code>{}</code>	Map of <code>service objects</code> indexed by their <code>id</code> .	Returns all the services of the simulation.
<code>show</code>	<code>{"ids": [&lt;IDs&gt;]}</code>	Map of <code>service objects</code> indexed by their <code>id</code> .	Returns the services of the simulation with the given string <code>&lt;IDs&gt;</code> .

## 5.4. Server Event Notifications

Clients can add a listener to a simulation event to be notified when this event is fired.



See [server Object](#) for adding a listener.

When an event is fired, the client receives a notification with the following format:

```
{
  "msgType": "notification",
  "data": {
    "name": "<EVENT>",
    "object": <PAYLOAD>
  }
}
```

- **<EVENT>** is the name of the event fired.
- **<PAYLOAD>** depends on the event and is usually the modified object with its new attributes.

The table below lists all available events with the payload it sends with its notification.

Event	Returned Payload	Description
Clock	Current Time string	Fired each time the clock changes, i.e. every 500ms.
StateChanged	{"value": true false}	Fired when the simulation is started or paused.  The returned value is <b>true</b> if the simulation is now running and <b>false</b> if it is now paused.
OptionsChanged	Options map	Fired when an option is changed (except <b>currentTime</b> ). The whole options map is sent with the notification.
RouteActivated	Route object	Fired when a route is activated.  Returns the activated route.
RouteDeactivated	Route object	Fired when a route is deactivated.  Returns the deactivated route.
TrainStoppedAtStation	Train object	Fired when a train stops at a scheduled station.  Returns the train that stopped.
TrainDepartedFromStation	Train object	Fired when a train departs from a scheduled station.  Returns the train that departed.
TrainChanged	Train object	Fired when a train sees one of its attribute changed. If the train is running, the event will be fired every 500ms as its position would have changed.  Returns the modified train.

Event	Returned Payload	Description
SignalAspect Changed	Signal Object	<p>Fired when the aspect of a signal changes.</p> <p>Returns the signal which changed its aspect.</p>
TrackItemChanged	Track Item Object	<p>Fired when a track item sees one of its attribute changed. This can be for any reasons, such as a route set, a train entering the item, an internal state changed, etc.</p> <p>Returns the modified item.</p>
MessageReceived	Message Object	<p>Fired when a message is added on the message logger.</p> <p>Returns the new message.</p>

## 6. Developing a Client

This section presents the way the standard python client is developed as guidelines for other client developers.

### NOTE

In this section, calls to the websocket API are represented in the `object.action(params)` form, such as `train.proceed(id=10)`

### 6.1. Getting the simulation dump

When a clients connects to a simulation server, the first action after registering is getting a simulation dump to be able to render the layout, schedules and train tables.

This is done by calling `simulation.dump()`. Collect all the needed information from the dump and build up your UI with it.

A typical UI would include the following:

- A layout
- A train table, with train details when a train is clicked
- A service table, with service details and service lines when a service is clicked
- A message logger

### 6.2. Register listeners

Register your client to the following listeners (for all ids) and connect them to your client's internal actions as follow:

Event	Client Internal Action
<code>trackItemChanged</code>	Redraw the given item.
<code>clock</code>	Change the clock display with the new time
<code>routeActivated</code>	In standard client, we use this only to check if the route is persistent or not, to display it on the layout.
<code>trainChanged</code>	Refresh train data in the train's table. Refresh of the layout is done by <code>trackItemChanged</code> event.
<code>messageReceived</code>	Notify the user of the new message, typically by adding it to the logger.
<code>optionsChanged</code>	Update display of options values when applicable (title, score, etc.).
<code>stateChanged</code>	Update the UI to show running or paused simulation.

This way, after initial setup the UI updates will be done by notifications only.

## 6.3. Get renotified to sync the UI

As simulation time passed between the first call to `dump()`, the UI rendering and the connection of the listeners, the UI is certainly out of sync with the simulation server by now.

To solve this issue, you can call `server.renotify()` to get a new notification of the last event of every type for every item that you are listening to. Thanks to the connection of the events with the internal actions, this should bring the UI fully synced again.

## 6.4. Drawing trains and activated routes

The track item information, whether received from `simulation.dump()`, `trackItem.list()`, `trackItem.show(...)` or sent through a `trackItemChanged` notification includes:

- Trains that are currently present on the item (See ["trainEndsFW" and "trainEndsBK" maps](#)).
- If there is a route set on the item and which one (See `activeRoute` in [Track Items](#) and `nextActiveRoute` and `previousActiveRoute` on [Signal Items](#)).

You should primarily use this information to display trains and routes on the scenery and not try to recompute the data from trains or route information.

## 6.5. Interact with the simulation

There are only a few ways to interact with the simulation:

### 6.5.1. Change options

Options can be changed at runtime with `option.set(...)`.

You should not assume the option has been changed just because you received an OK response, since another client might have changed it at the same time. Instead add a listener to

`optionsChanged` and wait for the notification.

### 6.5.2. Start/Pause the simulation

Start/Pause the simulation with `simulation.start()` and `simulation.pause()`.

You should not assume that the simulation is started or paused just because you received an OK response, as the simulation status may have been changed by another client. Instead add a listener to `stateChanged` and wait for the notification.

### 6.5.3. Set or unset a route

Route can be set with `route.activate(...)` and unset with `route.deactivate()`.

These actions take a route ID as parameter: it the responsibility of the client to provide the correct route ID from the user interaction on the UI. For example, in the standard client, a route is set by clicking on the entry signal, then on the exit signal. The standard client converts these clicks to a route ID which it sends to the server. The route ID is computed using the information returned by the `simulation.dump()` call.

You should not assume that the route is set or unset just because you received an OK response, as the route may have been set/unset by another client. Instead add a listener to `routeActivated` and/or `routeDeactivated` and wait for the notification.

### 6.5.4. Give instructions to the train driver

The following instructions can be sent to the train driver:

- `train.reverse(...)`
- `train.proceed(...)`
- `train.setService(...)`
- `train.resetService(...)`

See [train object](#) for the meaning of each instruction.

You should not assume that the instruction has been applied since another client may have sent a different instruction in the meantime. Instead add a listener to `trainChanged` and wait for the notification.