
Designing Templates
MissionX
Flying Adventures in X-Plane

Welcome - Few Words

Hello all,
In Aug 2022 I decided to re-write parts of this document, since it became more complex with edge cases that needed to have better exposure to the designer. As of this writing I'm thinking along the lines of the "Mission-X Designer Guide" document as a reference.
If you have any ideas, please share them with me in snagar.dev@protonmail.com

The document is still a Work In Progress…

!!! Breaking changes since v3.304.x !!!

Remember to also read the "Breaking Changes in the Designer Guide Document".

Although not breaking change, you can distribute your template in a dedicated "Template Mission Folder" like any other mission.
Please check the topic "Template as a mission package".

In Mission-X v3.0.224 I have modified some of the names of key elements or attributes due to concept change and to make the mission concept easier to grasp.
Instead of the "goal=>objectives=>tasks", I decided to change it to:
"leg=>objectives=>tasks". This might seem like a small change, but I'm not using the "goal" anymore to describe the mission, instead I prefer to think of it as a kind of "flight plan". You need to finish each "leg" in the "flight plan" and each leg might have tasks in it that you need to accomplish.
Therefore I had to go over all the XML elements and attributes in the plugin and modified them to reflect this change. Instead of <goals> it is now renamed to <flight_plan> and instead of <goal> it is now called <leg> (short for flight_leg), and the attribute: "next_goal" is now called: "next_leg".

Template version should be: ***301***.

*A good starting point is to **replace** the following strings:*

Old	New
version="300"	version="301"
"goal> "	"leg> " (pay attention there is a space after the last letter).
"<goal "	"<leg " (pay attention there is a space after the last letter).
"/goal>"	"/leg>"
"next_goal"	"next_leg"
"start_goal_message"	"start_leg_message"
"post_goal_message"	"post_leg_message"
"min_valid_goals"	"min_valid_flight_legs"
"is_set_of_goals"	"is_set_of_flight_legs"
"fire_commands_at_goal_begin"	"fire_commands_at_leg_start"
"fire_commands_at_goal_end"	"fire_commands_at_leg_end"

“flag_pick_location_based_on_same_goal_template”	“pick_location_based_on_same_template_b”
Since v3.0.242.x	
“starting_speed_kts”	“starting_speed_mt_sec”
mxpad_label mxpad_label_placement mxpad_label_color	label label_placement label_color
Since v3.0.242.9 - Explicit formatting is a ***requirement*** and not optional any more. The plugin will ignore your values otherwize.	
location_value=”20” or location_value=”hospitals” location_value=”” or location_value=”_”	location_value=”**nm**=20” or location_value=”**nm**=_” or location_value=”**tag**=hospitals”
Since v3.0.254.9	
Osm Web filters must be explicit. Before: ['piste:type'='downhill']['piste:type'='sled']	Osm Web filters must be explicit. After: **way**['piste:type'='downhill']**({{bbox}});**way['piste:type'='sled']({{bbox}})**;out;**
Since v3.0.256.2	
Before: <options> implemented as a sub element of <mission_info>.	After: we have a new root element by the name <REPLACE_OPTIONS> and it holds the sub elements replace rules. See dynamic templates topic.
Since v3.304.9.1	
<rank> was renamed to <scoring>	
Since v3.304.11	
Not exactly breaking behavior but important to note: All <display_object_xxx> elements will be read from <leg > element even if there is a <display_object_set> present. **Previous behavior**: if there was a “display_object_set” then the plugin ignored simple “display_object” elements. ***New behavior**:* All <display_object_xxx> elements will be read from the <leg> element: “display_object_set”, “diplay_object” and “display_object_near_plane” that were defined in the <leg>. Make sure you remove duplicate tags.	
Attribute “post_goal_script” has been deprecated	Use “post_leg_script”
Attribute “pre_goal_script” has been deprecated	Use “pre_leg_script”
Since v3.304.12	
`<dataref_start_cold_and_dark>` now follows the new syntax.	Changed the dataref text syntax. **before:** `sim/cockpit2/controls/yoke_heading_ratio:0,{next dataref=value}...`

Instead of using :":" for equals and "," for delimiter, Use "=" for equality sign and "|" as a delimiter.	**after** `sim/cockpit2/controls/yoke_heading_ratio=0|{next dataref=value}...`
`dataref_to_modify_when_fired` and `dataref_to_modify_when_left`	Both attributes are now using the updated "dataref" syntax guidelines. For more information see my remark on `<dataref_start_cold_and_dark>` above
`Since v25.01.1`	

Future deprecation: `<opt>` needs to be inside `<option_group>`
 	
`<REPLACE_OPTIONS>`
`  <option_group name="">`
`  <opt..>`
`  <opt..>`
`  </option_group>`
 	
`  <option_group name="">`
`  <opt..>`
`  <opt..>`
`  </option_group>`
`</REPLACE_OPTIONS>`
	<option_group> was added in v25.01.1, and it should allow for multi-option group support. Instead of having one big "replace" option, you can now define groups of options with their specialized replace <opt> options. Example: Inventory compatibility can be defined in different option groups, so simmer can pick which layout they define. This is also true for weather and other mission setup.

Future deprecation ???: *add_tasks_from_template* *add_messages_from_template* *add_triggers_from_template* *add_scripts_from_template*	Use the "REPLACE_OPTIONS" to achieve the same. If you think the list on the left should not be deprecated, please send me an e-mail to "snagar.dev@protonmail.com"

The <opt> sub element <info> is not supported anymore.
 	
Not supported:
`<REPLACE_OPTIONS>`
`  <option_group name="">`
`  <opt..>`
`  ~~<info>~~`
	The original use case was to replace the <mission info> attributes based on the "<info>" attributes. Instead, use a simple <opt> to replace the information in your <mission info> element.

Other changes I made are the need for a <content> element in the case of random templates.
If you want in one template to manage more than one type of missions then you must define the <content> element. In the <content> element you could define any set of delivery or medevac flight legs.

What you should not change/modify
In v3.0.242.7, I have added the "`target_marker_b` `[v3.0.242.7]`" attribute to the "<display_object>" element. Check the document for more explanations or read about it in the "designer guide" document.

`custom_hover_time` renamed to “***hover_time_sec_random***” (this is an old change that was not reflected in the documentation. sorry)

`location_value:` must be in format: “attrib=value”. Can be simple or complex: “nm_between=40-120|ramp=H”

Preface

“Generating Random Missions” engine has gone through some evolutions from the first time it was conceived as a Proof Of Concept. It was based on “delivery” or “medevac” templates but as I added more features and capabilities the need for “template type” started to feel restrictive.
Since Mission-X v3.0.222.16 this dependency is no longer valid for custom made templates, Instead we will use the following heuristics and suggestions:

1. We always try to Land.
2. If you don't define a plane type, the default will be “helos”.
3. If you want to be able to hover, you need to define plane_type as “helos” but…
4. Plugin can also decide if to hover based on location characteristics. Example: if the location falls in a water body or steep slope we will probably need to hover.
5. (optional) You should use the <content> element as much as possible to add diversity.
6. According to a poll I did on my site, many simmers would love to have meaningful locations in case of “medevac” missions (make sense if you ask me).

More Flexible

In the first months of the Random Engine the plugin was tuned for “template designers”, but as we progressed in time, new features were added and the latest, as of v3.0.242.8 where:

1. “User Create Template” screen
2. “Template” as a mission package [v3.0.242.10]

- In the case of “*User Create Template*” - the user could build a template from a set of preferences the user picked from (also referred to as an "in memory" template).
  In such cases, we had to make the distinction between “medevac” and “cargo” missions, type of planes, distances, osm and so on.
- In the case of “*Template as a mission package*” - the template designer can create a mission folder and place it as any other mission package with all its resources. The main difference is, that the mission file will be generated from a “template.xml” file located at the root of that folder and the template designer can create several flavors based on the template itself and generate similar experiences but with different content or challenges.
  I’ll explain in more detail later on.

  The plugin will read all “template” files from the local mission folder and from the “custom mission folders” too.

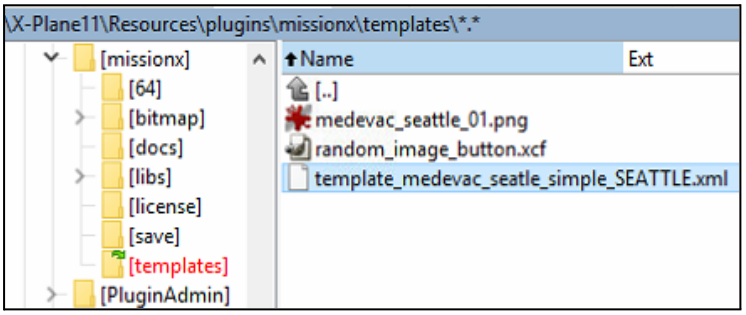
In the following pages I will discuss the Mission Template, its parts, what we should expect from it and what are the limiting factors.

Disclaimer: In order to create a new Mission Template you do not need background knowledge of writing an adventure for Mission-X plugin, but it is strongly advisable to read the "designer guide" to get a good grasp of what is being generated behind the scenes.

At the heart of this "engine", the plugin creates a mission file that you can then load and execute to your heart's content. It always writes to the **same** file and folder, so if you want to keep one of the generated missions, you will have to rename the file first.

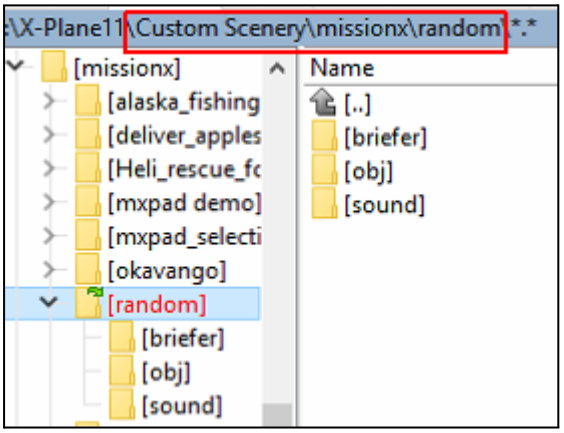
How to setup a Standalone Template

Basically you have two folders, one is the plugin and the other represents the "random" mission folder (just like in any other mission, but its name is "random").



The standalone "template" folder resides in the plugins subfolder: "...plugins/missionx/***templates***".

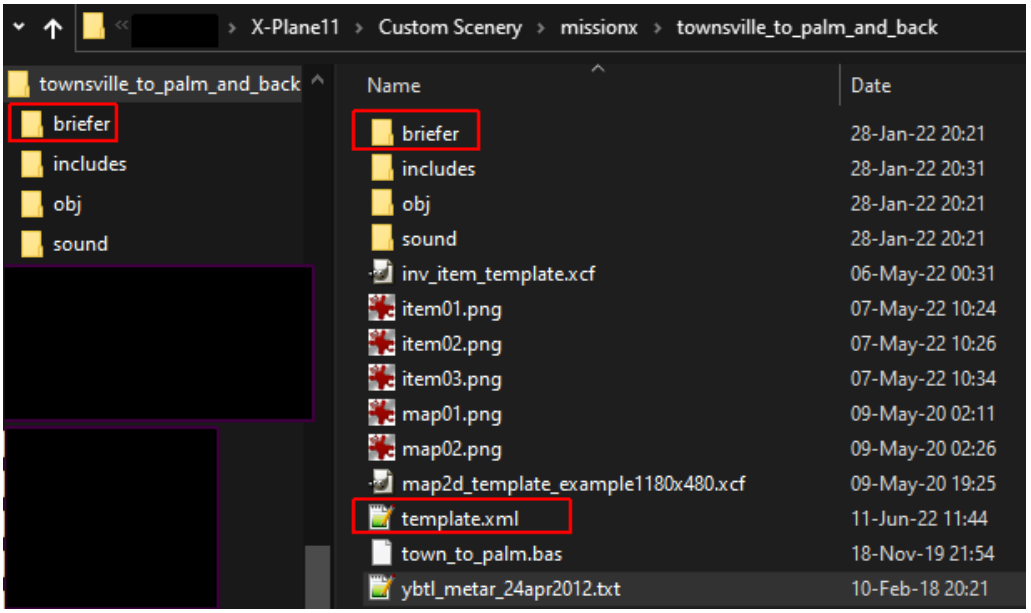
It should have a template name and an image file at the size: [590x440]



Second folder is a mission pack folder, just like any other mission pack. It resides in: "...Custom Scenery/***missionx***/" and its name is "***random***"

You can add special files in this folder pack, like new 3D Objects in the "obj" folder or sound files in the "sound" folder.
In the "briefer" folder you should not touch, for now, since the plugin writes into it and it uses the same image file for all generated missions.

For “Template Package” cases, you will create a new folder in “...Custom Scenery/***missionx***/” by any name you like but make sure you have the same structure as the “random” folder (as a reference):



See below for a detailed explanation.

Templates as a mission package

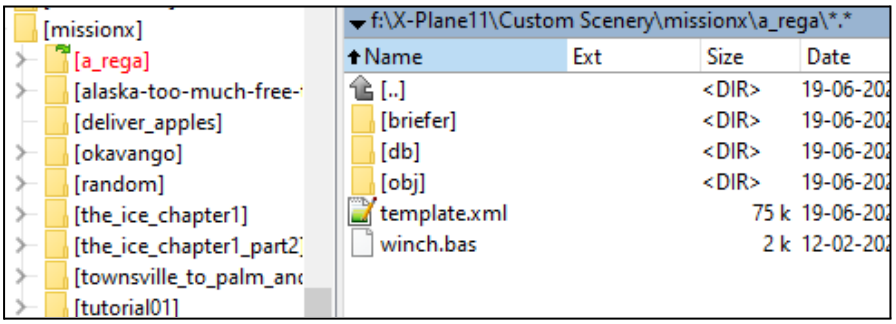
As of v3.0.242.10 you can create your own mission folder with a template at its core. This means that everything in that folder belongs to that template, and should make the template sharing as a package much easier. This folder is like the “random” folder, only unique for your template.

How to setup a template mission folder

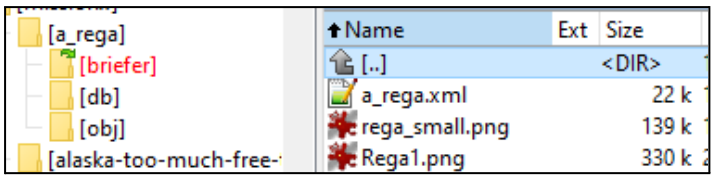
The rules of a “*template mission folder*” are the same as any other “mission folder”. You can use the “random” folder as a baseline, but “template mission folders” should abide to the following 3 rules:

- The template file must reside in the root mission folder.
- The template file name must be: “**tempalte.xml**”
- The template “image” and “mission image” must be located in the “*briefer*” sub folder.

Here is an example:



The outcome of the mission file will be a mission file by the name of the mission folder + ".xml" at the end.



The "**db**" folder, in this example, holds the "*osm database*" files. Osm databases can only be placed in the "db" subfolder". The plugin will first search for any osm database files in that location and only then will search the local plugin folder.
This means that your custom osm database files, the plugin's own "database files", are located locally in the plugin folder.
Please remember that any database file must end with ".db" postfix.

Changes in the way the plugin reads template files:
Before build v3.0.242.10b2 the plugin only read templates from the plugins internal folder "{missionx}/templates".
From v3.0.242.10b2 the plugin will read all "template" files from the local mission folder and from the "custom mission folders" too, so you should see all valid templates files.
When reading from the "custom mission folder" the template file name must be "template.xml"

You should read the OSM topic too to get the full picture.
Extending templates - including external files instead internal placeholder strings
In v3.0.256.4 we introduced the dynamic template.
You basically prepare one main template file with specific "*placeholder strings"* in its body, to be replaced as an outcome of user preference.
The flexibility is achieved by replacing the "*placeholder strings*" in the XML file with the content of an external file.

For more information, see "Dynamic Templates" in the Advanced topics.

The main concepts of Mission Templates

A mission template file should guide the plugin how to construct a mission data file and its “flight legs”. Each “leg” should also define its characteristic: `[land|hover| ...]`
The mission template needs to include these attributes since the plugin needs guidelines on how to build each “flight leg”, and produce plausible messages for the simmer.

The “Language” used in the Template file
We use XML standards to describe a mission template file (same as writing a custom mission file).
A template is divided into two main section and one optional section (for now):

- <TEMPLATE> The mission descriptor.
- <MAPPING> for the plugin - internal use, do not modify unless you know what you are doing.
- **[optional]** <REPLACE_OPTIONS> Allow the designer to define different template outcomes based on predefined content.

The <MAPPING> part is used internally by the plugin and should not be modified, altered or replaced, Doing so might lead to build or parse failure (unless you know what you are doing).

Your main working area is between the <TEMPLATE> element tag. In it you need to define:

- Plane Type (should)
- Briefing information.
- Starting location.
- “Flight Legs”, their template type and their characteristics (what we should see and do).
- Points: predefined locations of interest (optional).
- 3D Objects to display or pick from (optional but highly useful).

Any diversity in the mission will be defined in the <leg> element.
That is a short list of things to define in order to let the plugin build a mission out of it.

There are few demo templates that you can use as a base line or copy over to your own template file.

On the next pages, we will investigate each definition part and explain what is expected from a designer in order to customize his/her own templates.

Template Example

A “medevac” mission template based on a helicopter can contain information for specific regions, but you can also create your own generic “medevac” template that “provides” random target locations (but at your own “risk”).

```
<?xml version="1.0" encoding="ASCII"?>
<TEMPLATE version="301" plane_type="helos" >
  <mission_info template_image_file_name="template_heli_medevac.png"
  written_by="Saar N. (snagar)"
  plane_desc="Helicopter"
  estimate_time="~45 Minutes."
  difficulty="Easy"
  other_settings=""
  weather_settings="By Simmer"
  scenery_settings="Rescue-X, R2 Library, 3D people library, OpenSceneryX and MisterX libraries"
  short_desc="Hello pilot.;The following template ...;;Enjoy" />

  <!-- [plane | xy] {element name that holds points} -->
  <briefer_and_start_location location_type="plane" location_value="" lat="" long="" elev_ft="0" heading_psi="" >
    <![CDATA[Welcome pilot,
    you will have to fly to a location on your GPS to assist in evacuating a person in need.
    You will have to use a helicopter for this mission.
    ]]>
  </briefer_and_start_location>

  <!-- plugin will probe location and will decide if it is water or ground -->
  <!-- location_value = random_pick_target_in_area -->
  <leg template="land" name="goal_1" >
    <expected_location location_type="xy" location_value="nm=25" />
    <display_object_set random_tag="object_sets" set_name="medevac" />
    <special_leg_directives base_on_external_plugin="" add_tasks_from_template=""
                            add_messages_from_template="" add_triggers_from_template="" />
    <timer name="goal_1_timer" min="25" run_until_leg="goal_2"/>

    <desc>Hello Pilot. we received a radio message regarding an injured person.
    Follow the locations in GPS and assist.
    Coordinates are - lat: {navaid_lat}, Long: {navaid_lon}. Estimated distance: ~{distance}.</desc>
  </leg>
  <!--  -->
  <leg template="icao,land,start" name="goal_2" >
    <expected_location location_type="near,near" location_value="nm=_,nm=20" force_template_distances_b="yes" />
    <display_object name="marker" instance_name="marker_goal_2" target_marker_b="yes" replace_elev_above_ground_ft="60" />
    <display_object name="ambulance" instance_name="ambulance_end_target"
                    random_tag="ambulance_select" relative_pos_bearing_deg_distance_mt="180|25" />
    <desc>It is time to fly back. Fly to: {navaid_name}. distance: ~{distance}.</desc>
  </leg>

  <object_templates>
    <!-- markers -->
    <obj3d  name="marker" file_name="marker_five_parts_02.obj" />
    <obj3d  name="marker_q" file_name="marker01_q.obj" />
    <!-- cars -->
    <obj3d  name="ambulance" file_name="" random_tag="ambulance_select"   />
    <obj3d  name="random_thing" file_name="" random_tag="people_at_location"  />
  </object_templates>
```

The following screenshot is from a demo template. In it you can locate the main elements or “tags” that construct a template.
To better understand each part, we will break down the template into its smaller parts and explain their role and meaning.

The Template <TEMPLATE>

```
<TEMPLATE version="301" plane_type="[helos|props|prop_floats|turboprops|jets|heavy]">
```

```
<?xml version="1.0" encoding="ASCII"?>
<TEMPLATE name="" title="" plane_type="helos" version="301" >
  <mission_info template_image_file_name="template_heli_medevac.png"
  written_by="Saar N. (snagar)"
  plane_desc="Helicopter"
  estimate_time="~45 Minutes."
  difficulty="Easy"
  other_settings=""
  weather_settings="By Simmer"
```

The main tag for any template file is the <TEMPLATE> element.
It holds the following information:

Attribute Name	Type	Value	Meaning
version*	int	301	Describes the format of the template file. Currently only version “301” is supported.
plane_type	str	[**helos**|props|prop_floats|turboprops|jets|heavy]	Default is “helos”
mission_file_format v25.03.1	str	301,312..	What should be the generated <mission> file version attribute value. Useful to force compatibility, like Inventory Layout. Example, if the user picks Inventory Layout like XP11, then this attribute could be set to “301” which should force the “XP11” layout.
name	str	File name	Will affect the “saved” progress file name. Useful when you have a template with optional configuration and you want to make sure that saved progress is unique.
title	str		Useful for templates as mission packages. After generating a mission the mission title can be unique. This way you can provide different titles for your dynamic template implementation.

Currently we use “**plane_type**” mainly as a reference to what we expect to fly. In some cases the plugin will change plane_type to “helos” if it was instructed by the template (Example in <point> definition) or if location is not accessible by regular plane (water location).

Please be advised that the current “**plane_type**” options are more or less inline with Laminars apt.dat file.

Mission Information <mission_info>

```
<mission_info template_image_file_name="medevac_seatle_01.png"
mission_image_file_name="medevac_small_01.png" title="{title}"
written_by="{name}" plane_desc="Helicopter" estimate_time="~45 Minutes."
difficulty="Easy" other_settings="" weather_settings="By Simmer"
scenery_settings="OpenSceneryX" short_desc="{some text}" />
```

```
<?xml version="1.0" encoding="ASCII"?>
<TEMPLATE version="301" plane type="helos" >
  <mission_info template_image_file_name="template_heli_medevac.png"
  written_by="Saar N. (snagar)"
  plane_desc="Helicopter"
  estimate_time="~45 Minutes."
  difficulty="Easy"
  other_settings=""
  weather_settings="By Simmer"
  scenery_settings="Rescue-X, R2 Library, 3D people library, OpenSceneryX
  short_desc="Hello pilot.;The following template ...;;Enjoy" />
```

Attribute Name	Type	Value	Meaning
template_image_file_name*	str	Image file name	Mandatory, image file name to display
written_by	str	“Some text”	The template designer name
plane_desc	str	“Some text”	Some text that describes the type of plane to choose from.
estimate _time	str	“Some text”	Estimate time, preferable in minutes.
difficulty	str	“Some text”	Hint about the difficulty of the mission
weather_settings	str	“Some text”	Suggest how to set the weather. Can be blank.
scenery_settings	str	“Some text”	Special scenery that is mandatory or optional to the mission.
other_settings	str	“Some text”	Add more explanation how to setup the mission, if it does not fall into one of the categories above
short_desc	str	“Some text”	Free text where you explain the template goals for the pilot. This is not the briefer though.
Template as a mission package			
mission_image_file_name	str	Image file name	Relevant only in cases where the template is part of a “Template Package” folder and it is not located in the plugin folder..

title [v3.0.256.4.1]	str	“Some text”	optional attribute that will be displayed in the missions picker screen. This way you can provide different titles for your dynamic template implementation.

The *mission_info* element is an informative element that describes the settings and expectations from the simmer. All the information is free text.
The “**template_image_file_name**” will be used as the “templates” image in the Template screen.
The “**mission_image_file_name**” will be used in the mission picker screen, where the images do not have the same size (see below for detailed explanation).
The “**title**”[v3.0.256.4.1] is an optional attribute that will be displayed in the missions picker screen. This way you can provide different titles for your dynamic template implementation if you so choose.

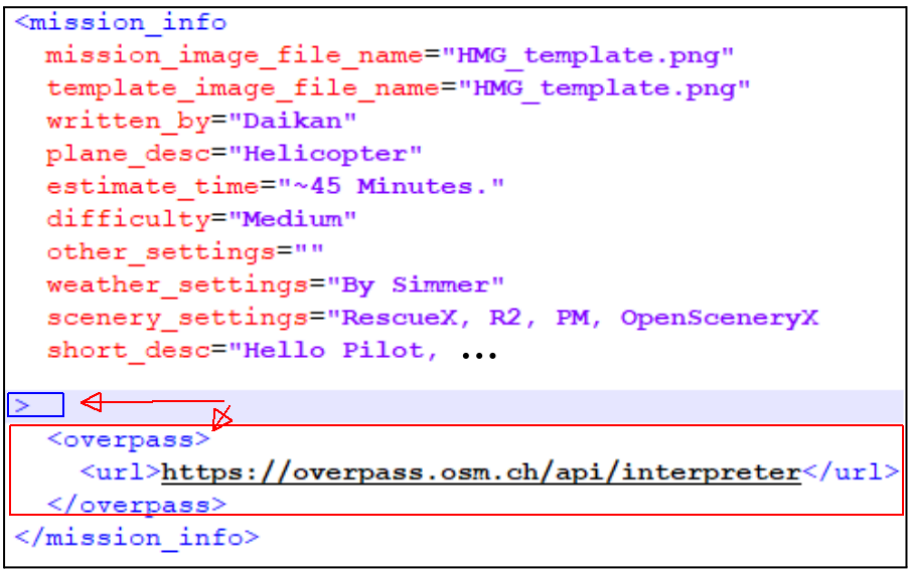
Please make sure not to use characters that may break the XML format, like double quotes or Ampersand “&” in your description.

<overpass> sub element of <mission_info>

As of v3.0.256.4.2 a designer can define an overpass override urls to the plugin default list (defined in the missionx preference file).
The element format is:

```
<mission_info ....>
    <overpass>
      <url>https://overpass.osm.ch/api/interpreter</url>
    </overpass>
</mission_info>
```

This element is optional



<global_settings> and <scoring>

In every mission file there must be a global_settings element, but in “template” mode the plugin will generate it for you from the <MAPPING> root element.
This means that you can define a custom <global_settings> that include elements you want to implement like <scoring>.

Starting v3.304.9, you can define the <scoring> element inside the <TEMPLATE> root element (place it anywhere under it), and it will be added or override any <scoring> element predefined in the <MAPPING> root element in the “template file”.

This will allow you to have flexibility when defining custom ranking for templates with multiple options

I suggest you read the <scoring> topic in the “Mission Designer Guide” document first.

Example:

```
<TEMPLATE>
...
  <scoring>
    <pitch best="-10|15|0.5" good="-15|18|0.4" average="-20|20|0.25" />
    <roll best="-25|25|0.5" good="-30|30|0.4" average="-35|35|0.25" />
    <gforce best="-0.5|1.55|0.5" good="1.55|1.82|0.4" average="1.82|1.95|0.25" />
    <center_line best="-1.0|1.0|0.5" good="-1.8|1.8|0.35" average="-2.3|2.3|0.25" />
  </scoring>
...
</TEMPLATE>
```

<briefer_and_start_location>

```
<!-- [plane | xy] {element name that holds points} -->
<briefer_and_start_location location_type="plane" location_value="" lat="" long="" elev_ft="0" heading_psi="" >
  <![CDATA[Welcome pilot,
  you will have to fly to a location on your GPS to assist in evacuating a person in need.
  You will have to use a helicopter for this mission.
  ]]>
</briefer_and_start_location>
```

Attribute Name	Type	Value	Meaning
location_type*	str	[xy, **plane**]	Start location. “xy” is used for random predefined locations too. “Plane” - Current plane coordinates will be stored.
location_value ~~??adhoc_locations_tag??~~	str	{tag name}	Used in conjunction with “xy” location type. Provide an internal “tag name” that holds predefined starting locations to randomly pick from. This will limit the missions to a specific area.
Lat / long	float	47.439491	Coordinates values for the starting location, used with “xy” location type.
starting_icao	str	KSEA | **“”**	If your template revolved around specific icao, you can set it using this attribute in addition to the lat/long attributes
heading_psi	float	0..359	Heading degree between 0 and 359.
start_cold_and_dark	bool	[true | **false**]	If defined then you will have to add the <datarefs_start_cold_and_dark> tag to the template with correct settings.
elev_ft	int	**0**	Don’t see the reason to use it at this time. We should always start on the ground. “0” = on ground.

When planning a template, we need to decide if we want a specific starting location or randomly picked from a list of predefined locations or wherever your plane is located.
In addition you can use the attribute “`start_cold_and_dark`” but you will have to add: “<datarefs_start_cold_and_dark>...</datarefs_start_cold_and_dark>” element at the template level.
For Ad-Hoc locations we use *location_type*=”xy” and define “lat/long” attributes.
Example:

- `location_type="xy"` .
- `lat="47.439491" long="-122.315522"` - this is an exact starting location to place your plane in.

For location type “plane” as your starting location, we just define: location_type=”plane”, and for multiple options to pick from we should define: *location_type=”xy” and location_value=”{tag_name}“.*

Here are some examples to a briefer and starting point definitions:
Example 1: Exact location:

```
<briefer_and_start_location location_type="xy" lat="47.439491"
long="-122.315522" elev_ft="0" heading_psi="60.82" start_cold_and_dark="yes" >
    <![CDATA[{Welcome pilot... you will have to ... location on your GPS...}]]>
</briefer_and_start_location>
```

Example 2: Based on predefined list of locations:

```
<briefer_and_start_location location_type="xy" location_value="start_loc"
elev_ft="0" heading_psi="60.82" start_cold_and_dark="" position_pref="" >
    <![CDATA[{Welcome pilot... you will have to ... location on your GPS...}]]>
</briefer_and_start_location>

<start_loc>
  <!-- define starting location based on icao code -->
  <icao name="CYZT" />
  <icao name="CYBL" />
  <icao name="CYPW" />
  <!-- or define exact points -->
  <point lat="52.520879"  long="-125.386485" />
  <point lat="50.941883"  long="-122.977645" />
</start_loc>
```

Example 3: Other option is to use: “`plane`”

```
<briefer_and_start_location location_type="plane" >
    <![CDATA[{Welcome pilot... you will have to ... location on your GPS...}]]>
</briefer_and_start_location>
```

The last thing we should add is a generic and short mission description so simmer will understand what this mission is all about.

- `<![CDATA[ some text ]]>`

We write our generic briefing text inside the “CDATA” brackets.

As of Mission-X v3.0.302 you can use: “`position_pref`” with the values: “11” or “10” (“11” is the default). The first will use the position logic of XP11 and “10” will use the logic from XP10. The benefit of the latter is that it does not force the engine to start after positioning the plane but it also does not always load DSF immediately and there might be other issues with it, so test it thoroughly before picking the older option..

<start cold and dark>

Warning, when using "start cold and dark" we need to make sure that the datarefs being used works as expected. Try not to add too much datarefs just to be on the safe side since it might cause the plane or other planes to fail to start (including helis).

If you decided to set the <**briefer_and_start_location ...>** with the "start_cold_and_dark="yes"" then you should also set the tag "<datarefs_start_cold_and_dark>" in the <TEMPLATE> root element as a child with a set off text that represent datarefs you would like to set on mission start, divided with commas.. You can read more about it in the "*designer guide*" document.

Example:

```
<TEMPLATE>
...
...
<datarefs_start_cold_and_dark>

sim/cockpit2/controls/yoke_heading_ratio=0|
sim/cockpit2/engine/actuators/prop_pitch_deg[0]=-4.9|sim/cockpit/radios/transponder_mode
=0|sim/cockpit/switches/pitot_heat_on=0|sim/cockpit2/controls/flap_ratio=0.0|
sim/cockpit/switches/pitot_heat_on2=0|sim/cockpit2/engine/indicators/prop_speed_rpm=156|
sim/cockpit/electrical/taxi_light_on=0|sim/cockpit/electrical/nav_lights_on=0|sim/cockpi
t/electrical/beacon_lights_on=0|sim/cockpit/electrical/landing_lights_on=0|sim/cockpit/e
lectrical/strobe_lights_on=0|sim/cockpit2/engine/actuators/ignition_key=0|sim/cockpit2/e
ngine/actuators/ignition_on=0|sim/cockpit2/autopilot/autopilot_on=0|sim/cockpit2/electri
cal/battery_on=0|sim/cockpit2/electrical/generator_on=0|
sim/multiplayer/controls/engine_prop_request=0

</datarefs_start_cold_and_dark>
...
...
</TEMPLATE>
```

Once we define the introduction of our "Template" based mission we can then design the "flight legs" in it

Flight Legs

The mission generator engine needs to construct a plausible “flight plan” for the simmer. Each “leg” can handle only one location and in each “leg” you can define one or more 3D objects to display (if you want).

```
<!-- plugin will probe location and will decide if it is water or ground -->
<!-- location_value = random_pick_target_in_area -->
<leg template="land" name="goal_1" >
  <expected_location location_type="xy" location_value="nm=25" />
  <display_object_set random_tag="object_sets" set_name="medevac" />
  <special_leg_directives base_on_external_plugin="" add_tasks_from_template=""
                          add_messages_from_template="" add_triggers_from_template="" />
  <timer name="goal_1_timer" min="25" run_until_leg="goal_2"/>

  <desc>Hello Pilot. we received a radio message regarding an injured person.
  Follow the locations in GPS and assist.
  Coordinates are - lat: {navaid_lat}, Long: {navaid_lon}. Estimated distance: ~{distance}.</desc>
</leg>
```

```
<leg template="land" name="goal_1" >
  <expected_location location_type="xy" location_value="nm=25" />
  <display_object_set random_tag="object_sets" set_name="medevac" />
  <display_object_near_plane name="crate01" instance_name="crate01_01"
                    relative_pos_bearing_deg_distance_mt="{acf_psi}+90|{wing_span}+1"/>
  <special_leg_directives base_on_external_plugin=""
                          add_tasks_from_template=""
                          add_messages_from_template=""
                          add_triggers_from_template="" />
  <timer name="goal_1_timer" min="25" run_until_leg="goal_2"/>

  <desc>Hello Pilot. ...Coordinates are - lat: {navaid_lat}, Long:
        {navaid_lon}. Estimated distance: ~{distance}.</desc>
</leg>
```

In the next pages we will explain the main <leg> element and its sub elements.

<leg> attributes and sub elements

Attribute Name	Type	Value	Meaning
template*	str	“land,start,hover”	Describes this flight leg main task to be done.
name	str	[unique name]	Optional attribute. If empty then the plugin provides a name of its own.
pick_location_based_on_same_template_b	bool	[yes|**no**] or [true|**false**]	Force the plugin to pick a random point with the same template type.
disable_auto_message_b	bool	[yes|**no**] or [true|**false**]	Disable auto plugin messages and construct your own, if any.

<desc>

<desc>{some text}</desc>

<**desc**>
The <desc> sub element is an optional element without attributes. It allows you to define custom description text and override the default plugin description. I suggest checking the default generated descriptions first, before writing your own. Some of the benefits of the default generated description is the location awareness that hints on what is to come like: “skewed” location or “water bodies”

You can read some more about the description and special text replacements further in this document.

<**expected_location**> attributes

Attribute Name	Type	Value	Meaning
location_type*	str	“xy”,”near”,”start”,”osm”, ”webosm” {tag name}	Target type to construct. You can define multiple types
location_value* v25.02.1: In the future it will be named: location_properties	str	In the format: {attrib=val} At least enter: “nm=20” or some other number	Complex value depends on the location type. See example and explain in next pages. You can define multiple values, relative to the number of types. **Minimal attribute is: “nm”** More on that can be found below
force_sloped_terrain v25.02.1: In the future it might be **deprecated.**	int	{**0**..10}	This attribute only forces the plugin to pick an area that is sloped enough so the simmer will have to hover. The value tells the plugin how many times to search for a valid location. **[0]** means ignore. If it fails after N times it flags the “flight leg” as failed.
force_leveled_terrain	int	{**0**..10}	Force the plugin to search an area that is leveled for landing.

v25.02.1: In the future it will be **deprecated.**			The plugin checks relative to “max slope allowed” (for helos, default is 6 degrees). **[0]** means to ignore this attribute. As a designer provide the number of times to search for leveled terrain before failing the “flight leg”
max_slope_to_land v25.02.1: In the future it will be **deprecated.**	float	**6.0**, 10.0 Nnn	Define the max slope helos can land. Default is **6.0** degrees and you can modify it to force another slope value. Minimum slope is 1.0.
radius_mt	int	50, 120 Must be >= 50	Used this only if you want to change the plugin defaults Target radius. It overrides the plugin's default radius (except <point>, see below). ***Value must be greater or equal to 50.***
force_template_distances_b	bool	[true | **false**]	If the user picks “expected distance” in the UI setup screen, the designer can still override and force their preferences per <expected_location> element, not globally.
override_trigger_name	str	trig_leg1_dyn	Plugin will use the provided name as the target trigger’s name. More...
override_task_name	str	task_leg1_dyn	Plugin will use the provided name as the task’s target name (for the mandatory task). More...

force_template_distances_b: Try not to use this attribute too much since it will make the “expected distance” setup option obsolete . Use it only when it matters

<fire_commands_at_leg_start> and <fire_commands_at_leg_end>

Attribute Name	Type	Value	Meaning
commands	str	"412/buttons/PATIENT_off"	Execute a sequence of commands at the beginning of a flight leg and at the end of it.

<timer />

One liner element.

Name	type	Values	Meaning
name *	str	“timer_leg1”	Unique timer name
min *	float	**0.0**, 55.0, NN.nn	How many **minutes** to count down. Default is “zero”, which means - ignore the timer.
run_until_leg	str	“”, {some flight leg name}	Provide a flight leg name where the failed timers should stop running (means success for simmer).

we can add one <timer> element to each flight leg or the “global_settings”.
The simmer will only see the lowest timer that currently runs although few timers can run at the same time.
Please remember to define “run_until_leg” the timer should run, not doing so means it won’t stop when the flight leg is transitioned.
Time it correctly to not frustrate the simmer.

```
<timer name="leg_01_timer_30min" min="30" run_until_leg="leg_02"  />
```

<display_object />

One liner element.

```
<display_object name="ambulance" instance_name="ambulance_01"
                random_tag="ambulance_select"
                random_water_tag="water_replace_ambulance"
                relative_pos_bearing_deg_distance_mt="360|10”
                replace_elev_above_ground_ft="60"
/>
```

“display_object” attributes

Attribute Name	Type	Value	Meaning
name*	str	“metar01”	A mandatory object name from <object_template> that our instance will be created from.
instance_name*	str	“into_marker_01”	Unique name for the 3D Object.
relative_pos_bearing_deg_distance_mt	str	"360|10”	Location relative to **target** coordinate: enter degrees and distance in meters from target position. When used in <display_object_near_plane> the location is relative to plane position at the time the flight_leg started.
replace_{attrib_name}	str	rplace_*keep_until_leg, replace_lat, replace_long …*	When we define a new instance we might need to modify some of its attributes, since the 3D object template might not be correct to the target position. For example - location information. The attributes you can add “reaplce_” to them are: ***“lat”, “long”, “elev_ft”, “keep_until_leg”, “elev_above_ground_ft”, “heading_psi”, “pitch”, “roll”, “distance_to_display_nm”***
target_marker_b	bool	[true,**false**]	A flag that tells the plugin if the marker is above a target.

<weather />

(v3.304.12)

```
<weather>
sim/weather/cloud_coverage[2]=1.|sim/weather/use_real_weather_bool=0|sim
/weather/cloud_base_msl_m[2]=10668.|sim/weather/shear_direction_degt[2]=
0|sim/weather/barometer_current_inhg=29.89|sim/weather/cloud_coverage[0]
=0.</weather>
```

The weather element "text" value holds the datarefs to update X-Plane. You can set any dataref you want, but it was designed mainly to focus on the weather dataref aspect. It is being used as a "one time dataref modification" at the beginning of the flight leg or global settings, depending where you set it.
For more information check the "designer guide" documentation in the <leg> topic.

<display_object_near_plane />

One liner element.
An explicit use of <display_object> that will be copied as is to the "random.xml" file and renamed to "<display_object>".
It has same attributes as a <display_object> but the Random engine does not try to parse it, it will only set a dummy "*lat/lon*" attributes with the value "1.0"

The position of the object is determined only when the flight leg is being activated, so it is best to be on ground and not in the air.

Snippet from the designer guide:
"Display object near plane" is a special case of "<display_object>". You can use it exactly like the <display_object> but you must set the "`relative_pos_bearing_deg_distance_mt`" attribute.

keywords to use with "relative_pos_bearing_deg_distance_mt" attribute:
If you use the "`relative_pos_bearing_deg_distance_mt`" attribute, you can use the following predefined keywords to dynamically calculate the bearing or position of the object relative to plane location **or** you can use one of your custom number <*dataref*> elements you defined in the <*xpdata*> root element.

Pre Defined Keyword	Supported	Explanation
{acf_psi}	xp11/12	The bearing of the plane
{wing_span}	xp11/12	The length of the wing (1 side of the wing)

You can write a simple mathematical expression as follow:

```
<display_object_near_plane name="crate01" instance_name="crate01_01"
relative_pos_bearing_deg_distance_mt="{acf_psi}+90|{wing_span}+1"
replace_lat="1" replace_long="1" replace_distance_to_display_nm="10.00"
target_marker_b="no" replace_elev_ft="" />
```

In this example we want to place the object 90 degrees from the plane bearing and position it one meter after wingtip (The position depends on the bearing).

Use this feature carefully, It is best used when the plane is on ground and at the start of a flight leg.

In the following example, we use few 3D objects that we place in the missions object folder: ("Custom Scenery/missionx/random/**obj**")

Here is an example for "*medevac*" flight leg:

```
<leg template="land,hover" name="goal_1" pick_location_based_on_same_template_b="yes">
 <expected_location location_type="xy" location_value="tag=goal_1_landing_locations" />
 <fire_commands_at_leg_start commands="412/buttons/PATIENT_off"/>
 <fire_commands_at_leg_end commands="412/buttons/PATIENT_on_board"/>

 <display_object name="marker"    target_marker_b="true" instance_name="marker_1"
                 replace_elev_above_ground_ft="60" />
 <display_object name="ambulance" instance_name="ambulance_01"  />
 <display_object name="car01"     instance_name="car01_1"
                                  relative_pos_bearing_deg_distance_mt="360|10" />

 [optional]<desc>{your custom flight leg description}</desc>
 [optional]<timer name="leg_01_timer_30min" min="30" run_until_leg="leg_02"  />

</leg>
```

This is an example of a single flight leg definition.
From it we can see that:

- Plugin should pick between template: "hover" and "land" (`template="land,hover"`)
- The landing location is based on coordinate: (`location_type="xy"`). The "`location_value`" will determine where we will place the target location.
  If we use the "`nm`" option with a number in "`location_value`" [`location_value="nm=30"`] the Plugin will pick a random lat/long target in that area. But, if we provide a "tag name", [`location_value="tag=goal_1_landing_locations"]` the plugin will translate it to: "*find another element in the template by the name* - "`goal_1_landing_locations`", *and randomly pick a* <point> *element from it. Use its lat/long values*" *to construct a target*.
  Behind the scenes the Plugin will create a radius based trigger.
- "`pick_location_based_on_same_template_b`"`="yes"` When picking a ***random point***, from another element, pick one with the same "template type".

Here is an example of the "`goal_1_landing_locations`" element.

```
<goal_1_landing_locations>
...
 <point lat="47.305596" long="-122.564817" template="hover" radius_mt="100" loc_desc="Shore...beach" />
 <point lat="47.281373" long="-122.560225" template=""      radius_mt="100" loc_desc="Road...Airport" />
 <point lat="47.195262" long="-122.664787" template="land"  radius_mt="100" loc_desc="NcNeil..." />
...
</goal_1_landing_locations>
```

This is a very simple element, all it holds is a list of points and some hints regarding them.
A <point> only needs "lat/long" attributes to be valid, but for the "Mission Generating Engine"

I suggest to also add hints like:

1. "radius_mt": Radius of effect in meters. Default value is 40 meters.
2. "template": This tells the Plugin which template is best for that location.
   If the template is empty, then the Plugin can use any template type. But, if it has value in it then It will override the "leg" template.
   Example: Assume that we have a flight "leg" template with the value "land", and the plugin picks a point that its template is "hover".
   The plugin will replace the flight "leg" template attribute with the one it picked from the <point> element.

3. "loc_desc": location description - is added to the generated descriptions or messages.
   This gives a meaningful description regarding the location the simmer needs to reach.

Although these attributes are ***optional***, they will assist to better define a diversity of missions in the mission template. Some can be harder due to smaller radius or the need to hover, and some simpler.

More explanation can be found below.

Understanding location_value attribute syntax and meaning

Directive	Meaning	Example
nm	Radius of effect in nautical miles. Used in cases of: 1. search icao in certain area 2. place a target randomly 3. filter between points	"nm=20" "nm=_" means default distances
tag	In XML, an element string is also referred to as a "tag name". Provide a tag name of an element to randomly pick a sub element from it. Example: random hospital location.	tag=hospitals, plugin will search for: "<hospitals>" element.
nm_between	no dependency on other tags. Randomly pick location between distances `location_value=` **`"tag=hospitals|nm_between=10-20" or "nm_between=35-67"`**	nm_between=10-20 Can co-exists with the "tag" element
Additional directives for better osm and template mission folder handling:		
dbfile	Force specific file to be used in this "template mission folder" Special Case: nm=0|dbfile={file name}	dbfile=osm-liechtenstein.db nm=0 **Should come with "dbfile"**

	Tells the plugin to pick any location in the database file. This is mainly useful for small areas which will fail in most distance tests.	
Control “osm” query results		
keyname	Default plugin key values are: name Tell plugin which “key” field value to use as the name instead of the default.	keyname=piste:name
keyid	Default plugin key values are: icao, faa Tell plugin which “key” field value to use as the ID name instead of the defaults. Useful for navaids.	keyid=icao
keydesc	Default plugin key values are: amenity Tell plugin which “key” field value to use as description instead of the defaults. Mainly useful for navaids.	keydesc=piste:name Use the name of the field as the description too.
Post “osm” query directives		
desc	This is a short free text. Don’t use special characters in it. Use it to tell the plugin, if the navaid picked has an empty description, then use my description (***do not override***)	desc=ski piste
descforce	This is a short free text. Don’t use special characters in it. Use it to ***override*** the navaid description.	descforce=ski piste

Example:

- Pick a random location in ~25nm radius. Target radius 55 meters (for hover cases)

<expected_location location_type="xy" location_value="nm=25" radius_mt="55" />

- Pick between 2 locations near the plane location. The first option will look for the closest ICAO, the second will look for the closest in 20nm. If you don't find any, the plugin will pick the closest.

<expected_location location_type="near,near" location_value="nm=_,nm=20" />

- Pick a random location from a predefined list of <point> elements. Root point tag element name is: “goal_1_landing_locations”

<expected_location location_type="xy" location_value="goal_1_landing_locations" />

- Pick a random location but the plugin will choose between 3 option types.
  1. xy, “nm=20” = any location in 20nm radius.
  2. “xy”, “tag=goal_2_landing_locations” = random point picked from “goal_2_landing_locations” element.
  3. “near”, “nm=_” = Pick icao closest to plane, relative to last target location.

<expected_location location_type="xy,xy,near"
location_value="nm=20,tag=goal_2_landing_locations,nm=_" />

- Pick a random location from a specific osm database file even if it is a small one.
  1. “type=osm”: Pick from “osm database” file. Plugin will fallback to “xy” if it doesn't find a database file.
  2. “nm=0”: special directive that tells the plugin to pick from a database file only. Useful only in small osm database areas, like “Liechtenstein” data. Don’t use it for big areas.
  3. “dbfile=osm-liechtenstein.db”: provide the database file name to use. Ignore the rest of the database files.
  4. “tag=osm_sql_piste”: Holds partial sql query to assist in filtering the fetched data.
  5. “keyname=piste:name”: Tell plugin which “key field” holds the name of the osm segment to use as name in messages.
  6. “keydesc=piste:name”: Same as “keyname” but here we use the value as the description of the osm segment

<expected_location location_type="osm" location_value="nm=0|tag=osm_sql_piste|dbfile=osm-liechtenstein.db|keyname=piste:name|keydesc=piste:name" force_template_distances_b="yes"/>

The different directives allow us to better control the query from the database and which fields we would like to store to affect the messages.
You will need to do some trial and error before figuring out which fields you would like to use and when. In most cases, the default fields the plugin uses are sufficient for most segments, but in the case of “piste” we will have to use the “keyname” and “keydesc” to get the result we want.

Points Of Interest

In some cases you might have a scenery from a third party, especially from an X-Plane.org contributor. In such cases there might be many points of interest that are published by the contributor with some airfields not far from them. My suggestion is to use the “poi_tag” (point of interest tag) attribute to add these locations to the GPS, but they won’t be part of the mandatory flight_plan.
Benefits: The main flight “leg” will be the landing area, the point of interest will be an optional location that you will see on your GPS as an additional location to visit.
The simmer can ignore them or fly to them.

How to add “point of interest”?
Currently it is only supported in the “point” element.

```
<point lat="47.195262" long="-122.664787" radius_mt="100" poi_tag="poi_xx01" loc_desc="NcNeil..." />

<poi_xx01>
 <point lat="52.748321"  long="-125.530781"  />
</poi_xx01>
```

Let’s assume that plugin picked the point: `lat="47.195262" long="-122.664787"`
The plugin will check if the attribute “poi_tag” is present and will add “all” points in the “`poi_xx01`” tag to the GPS.

I have used this technique with “towhee” scenery: “XP Activities: BC Discovery”

ICAO locations as points

Providing ICAO to choose from can have a great benefit in cases where we want simmer to fly into a specific area and return to specific locations. For example, SEATTLE medevac missions can have two “legs” in the flight plan, one is a random location where you need to assist an injured person, and the second a hospital location.
To make it easier, you can provide a list of <point> locations but instead of writing the exact “lat/long” you could define “icao” locations you picked from the map and the plugin will translate them to “coordinates” using x-plane XPSDK (X-Plane SDK).
Let’s look at an example of a random location element that holds points and icaos

```
<goal_1_landing_locations>
  <icao name="60WA" />
  <icao name="3WA7" />
  <point lat="47.437840" long="-122.462416" template="" radius_mt="100" loc_desc="Vashon Island, near
              99th Eve"/>
  <point lat="47.379917" long="-122.427036" template="" radius_mt="200" loc_desc="Maury Island"/>
...
</goal_1_landing_locations>
```

As you can see, you can mix between “point” elements and “icao” ones.
The format of the “icao” is simple, you use the tag “icao” and provide the name of the airport. as listed in the X-Plane map.

Cons:
When using <icao> the location picked is not a ramp location, rather a location XPSDK returns.
The plugin will try to find a ramp, if and only if, the user generated optimized apt.dat files (mission-x menu/setup screen) and even then there might not be a suitable ramp for the plane type defined in the mission template.

The "<display_object>" element

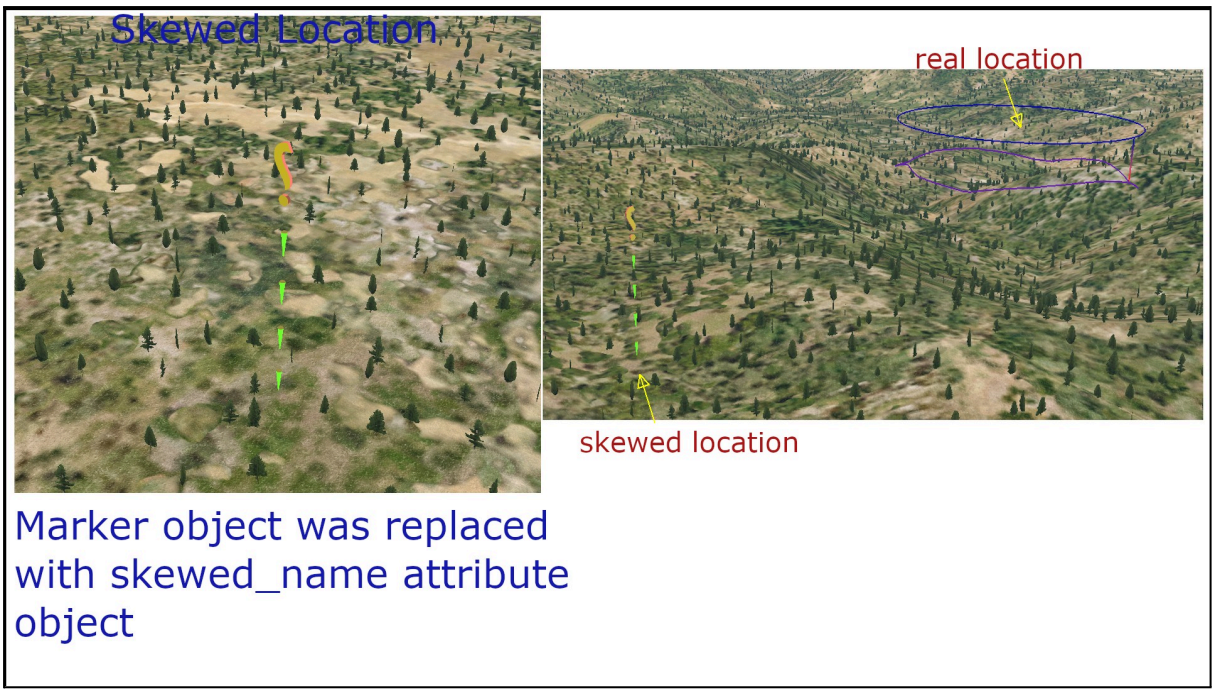
```
<display_object name="marker" skewed_name="marker_q" target_marker_b="true"
                instance_name="marker_1" replace_elev_above_ground_ft="60" />
<display_object name="ambulance" instance_name="ambulance_01"  />
<display_object name="car01"     instance_name="car01_1" relative_pos_bearing_deg_distance_mt="360|10" />
<display_object name="beacon"
                file_name="../../../The_Handy_Objects_Library/Road_Works_Light_On_Group.obj"
                relative_pos_bearing_deg_distance_mt="30|10" />
```

The plugin will display four Objects:

- A "***marker***" (`name="marker"`) its instance name is "marker_1" (`instance_name="marker_1"`) and we want to place this instance 60 feet above ground.
  (`replace_elev_above_ground_ft="60"`). Default is on ground ("0" zero).
  `target_marger_b="true" is needed by the designer if he/she wants the simmer to be able to hide/show these specific instances.`
  The "***skewed_name***" attribute allows designers to define different marker obj files in cases where simmer picks the setup option: "place marker near target".
- An "***ambulance***" (`name="ambulance"`) on terrain elevation, and
- A ***car*** (`name="car01"`) which we will position relative to the target point. It will be positioned 360 degrees and 10 meters away from target point
  (**`relative_pos`**`_bearing_deg_distance_mt="360|10"`)

  The `"car01"` object syntax allows us to place an object in the scene and place it next to each other.
  The best approach would have been to create an object with few elements in it, but this is a work around when there aren't complex scenery objects.

- [v3.0.256.1] A "***beacon***" - this 3D Object was not defined ahead of time in the *<object_templates>* and therefore it will be added automatically since it has the "file_name" attribute set.
  Although the plugin supports this feature I suggest to prepare this definition ahead of time in the *<object_templates>* part of the template (add one if there is none).



The element "**display_object**" can use 3d object files that are listed in the "**object_templates**" element.
The "**object_templates**" holds file information for each object name, so the plugin will be able to load the object during run time.
Here is an example of an "**object_templates**" element that we use in our example.

```
<object_templates>
  <!-- Ambulance -->
  <obj3d  name="ambulance" file_name="" random_tag="ambulance_select" />
  <obj3d  name="car01"  file_name="../../../OpenSceneryX/objects/vehicles/.../ford_mustang/object.obj" />
  <obj3d  name="marker" file_name="marker_five_parts_02.obj" />
  <obj3d  name="marker_q" file_name="marker01_q.obj" />
</object_templates>
```

You can see that some file locations have no folder or path and some do. When there is a path, it means location relative to the "Custom Scenery/missionx/random/obj" folder. This allows you to use 3D Objects from other libraries (Please add this fact to your mission description to recognise their work).
You can always add more 3D Objects and use them in your flight "leg" definition or the "Custom Scenery/missionx/random/obj" folder.

Please check <display_object_set> for more "3D objects" display options.

Here is an example for flight "leg" in a "*delivery*" template file:

```
<leg template="land" name="goal_1" optional="15%" >
  <expected_location location_type="icao" location_value="nm=60" />
  <display_object name="marker" instance_name="marker_goal_2_1"
                  replace_elev_above_ground_ft="60" />
</leg>
```

Locations in the "*delivery*" template are mainly based on "icao" since the "Plugin" searches for pick/delivery locations for you based on NavAids. The important values in this case are: "`location_value`" and "`optional`" elements.

- The "`optional`" attribute means that the flight "leg" is not mandatory, while the value of optional represents the chance as "percentage (%)" to create it, smaller numbers means lower probability.
- The "`location_value`" is translated by the plugin to "area radius to pick an **icao** from".

Example:

```
<leg template="land" name="goal_2" optional="55%" >
```

You can use the "%" symbol for easier readability or just write a number.

For each "leg" you can assign a percentage number or leave it blank. The plugin "will decide" if to generate or not that location. That way we create some diversity for each time a mission is generated.
I strongly suggest defining at least "***one***" flight "leg" without the "optional" attribute (the first one) so it will represent the minimal locations in your mission.

The <special_leg_directives> in flight "leg" element

This element holds special instruction and info on the flight "leg". It is being populated by the plugin, but you can manually define:

Attribute Name	Type	Value	Meaning
hover_time_sec_random	str	"10-30"	Define the time in seconds to pick hover time (randomly). You can provide numbers in the following formats: "20" - means hover for 20 seconds. "20-30" - means between 20 and 30.
base_on_external_plugin	bool	[true|**false**]	yes, then the first mandatory task that is based on trigger will be overridden and will evaluate the dataref: "xpshared/target/status" for success or failure
The following attributes below should be deprecated. I suggest using the <REPLACE_OPTIONS> instead. Please replace the following attributes with the <REPACE_OPTION> element since they should be deprecated by Q2 2025.			
add_tasks_from_template	str	"add_tasks_1"	add custom tasks residing in the element with tag name: "add_tasks_1".
add_triggers_from_template	str	"add_triggers_1"	add all triggers in element with tag name: "add_triggers_1"
add_messages_from_template	str	"add_messages_1"	add all messages in element with tag name: "add_messages_1"

base_on_external_plugin attribute.
If it is set to yes, then the first mandatory task that is based on trigger will be overridden and will evaluate the dataref: "xpshared/target/status" for success or failure. An external plugin would have to assign the values "1,-1" for success or failure. Anything else is ignored, which means the flight_plan won't progress since the "leg" won't be concluded.
In the case of X-Tridents SAR plugin, the plugin won't automatically set the status for us, the simmer would have to manually press the: "OK or KO" buttons which affects Mission-X task state.

```
Please replace the following attributes with the <REPACE_OPTION> element since they
should be deprecated by Q2 2025.
<leg ...>
 <special_leg_directives add_tasks_from_template="add_tasks_goal_1"
                         add_triggers_from_template="add_triggers_goal_1"
                         add_messages_from_template="add_messages"
                         base_on_external_plugin="" />

</leg>
```

The "<desc>" element in flight "leg"

The "random engine" will try to build a plausible message if you do not provide a custom description. But, if you want to show your message, you will need to add the <desc>" element to your <leg> definition.
To customize your flight "leg" description, you can add a simple "<desc>" element. Write the text between "<desc>your text</desc>"

```
<leg ...>
 ...
 <desc>Fly to {navaid_name}</desc>
 ...
</leg>
```

We have a set of *keywords* that we can place inside the text and the plugin will try and replace them with NavAid data.
The list of keywords:

Keyword	Value
{navaid_name}	Airfield name
{navaid_icao}	Airfield ICAO
{navaid_lat}	Target Latitude
{navaid_lon}	Target Longitude.
{navaid_elev}	Target elevation in feet
{navaid_loc_desc}	Useful when you want to display the custom location description you defined in <point> locations. Example: <point loc_desc="{text}" />
{distance}	Distance between prev and current targets.Number in nautical miles. Will use a "skewed" position if present.
{bearing_target}	Degrees to the target at hand relative to North

Based on the <leg> definition the "Plugin" will generate valid mission elements with an objective and task in it.
The beauty is that you just need to define the rules.

The benefits of not defining <desc> element, is the changing messages based on flight leg position and target characteristics (like water recognition)

The start flight "leg" type

```
<leg template="start" name="goal_2" >
  <display_object name="marker" instance_name="marker_goal_2" replace_elev_above_ground_ft="60" />
</leg>
```

The flight leg that is "start" template type is a much shorter and simpler flight "leg" element definition. The element "*leg*" has the following attributes:

- template is: ***start***, "`template="start"`". Means it should use the same location definitions as defined in the starting point (briefer).
- The "`display_object`" element, means we will display an object above ground in that location.

This type of "flight leg" is useful for "round trip" missions. If you prefer this kind of mission then you should use this template type as the last one in your <TEMPLATE> element.

Using the attribute template in the "leg" element and with different values, we can build simple missions with some diversity whether it is a "round-trip" mission or "one-way" mission.
In many cases that should be entertaining enough.

> ***In some cases you can use Ad-Hoc locations. For challenging missions, we can use the*** `location_type="xy"` ***and*** `location_value="tag={tag name}"` ***so the "plugin" will be able to randomly pick coordinates from the*** `"tag name"` ***element.***
>
> ***For full "medevac" diversity, you can use:*** `location_type="xy"` ***and*** `location_value="nm={number}"` ***. The plugin will pick random coordinates but without the promise that the location is plausible or not.***

Advance Topics

<dynamic_message>

Some template designers might prefer to disable the auto messages that are generated during mission creation. Messages such as:

- Distance from target.
- Arrival to target/inventory area.

You should be able to ***disable*** the auto messages (for distance messages, so far) and add your custom messages. To do that, we have the "*<dynamic_message>*" element.
The "*<dynamic_message>*" allows us to use existing messages or define a new one relative to a certain location. The location is represented by a trigger name or a task name that is mapped to a trigger.
We can construct "*<dynamic_message>*" in two ways:

Option 1: Define a message relative to location, either trigger or task, and the message text you want to send.
Its format should look as follows:

```
<dynamic_message [relative_to_trigger="trig_name" | relative_to_task="task name"]  length_mt= "" >{Message Text}</dynamic_message>
```

Attribute	Comment
relative_to_trigger	Message is coupled with a trigger.
relative_to_task	Message is coupled with a task
length_mt	The radius of the message area, used with triggers
message_name_to_call	Name of a predefined message
override_task_name	Custom task name, forces the plugin to use this name for the mandatory <task> element
override_trigger_name	Custom trigger name, forces the plugin to use this name for the trigger used with the dynamic message
message_name_when_entering_physical_area (v3.306.1b)	Which message to call if we entered the physical area even if not all conditions were met. See designer guide for more information.
Message text if we are not basing it on a predefined message	
{Message Text}	Free text, make sure not to use problematic characters that might break the XML file.
Used with the	

sound_file_name	Provide a sound file name f

Explanation: The plugin needs to create a new “radius based trigger” relative to a known target location. The best way to pick a target location is to provide its trigger name, if you are unsure you can provide the task name, as long as it is based on a trigger.
The plugin will create a new message and a trigger. The trigger radius will be copied from the “`length_mt`” attribute.

All this “Hocus Pocus” is being built only when the simmer loads the mission (not while the mission is being generated), and you should be able to see it in the “Log.txt” file.

How to distinguish dynamic messages:

- The message name will start with the prefix: “*msg_4_dyn_message_*” and
- The trigger name will start with the prefix: “*trig_dynamic_message_*”.

There is **one** major drawback when creating a random mission using a template:
We do not always know the trigger or task names ahead of time.

To solve this issue, we have two attributes for the “*<expected_location>*” element. You should use just the one you need. For example:

```
override_trigger_name="my_trig_for_{task_name}_{seq}"
or
override_task_name="task_name1"
```

Final element should look something like:

```
<expected_location location_type="xy" location_value="nm=50"
                   override_trigger_name="my_trig_for_task1_in_goal1"  />
```

This will cause the target’s trigger name of goal1 to be: “`my_trig_for_task1_in_goal1`”, that way the plugin could find the trigger later and fetch the coordinates to construct the “dynamic_message” trigger.
But wait, something is missing, we need to disable the leg auto messages. For that we have another <leg> attribute: “`disable_auto_message_b`”

Here is a full <leg> example from a template:

```
<leg template="land" name="goal_1"   disable_auto_message_b="true">

      <expected_location location_type="xy" location_value="nm=50" override_task_name="task_01"
                         override_trigger_name="custom_trig_goal_1_for_task_01"  />

      <display_object name="marker" instance_name="marker_goal_2_1" replace_lat="" replace_long=""
```

```
                    replace_elev_above_ground_ft="60" />
        <desc>Hello Pilot. Please fly to {navaid_name} (elevation: ~{navaid_elev} ft)</desc>

        <dynamic_message relative_to_trigger="custom_trig_goal_1_for_task_01" length_mt="1000">This is a
                    test of dynamic_message trigger</dynamic_message>
</leg>
```

The "override_task_name" and "override_trigger_name" are just an example. We use the custom trigger name in the "dynamic_message"

And here is a snippet from the template and the generated random.xml file (zoom in):

```
<!-- From Tempalte -->
<leg template="land" name="goal_1"   disable_auto_message_b="true">
  <expected_location location_type="xy" location_value="nm=50" override_task_name="task_01" override_trigger_name="custom_trig_goal_1_for_task_01" />
  <display_object name="marker" instance_name="marker_goal_2_1" replace_lat="" replace_long="" replace_elev_above_ground_ft="60" />
  <desc>Hello Pilot. Please fly to {navaid_name} (elevation: ~{navaid_elev} ft)</desc>
  <dynamic_message message_name_to_call="" relative_to_trigger="custom_trig_goal_1_for_task_01" length_mt="1000">You are near "{navaid_name}</dynamic_message>
</leg>

<!-- Snippet from random.xml file -->
<leg name="goal_1" title="" next_leg="goal_4" >
  <start_leg_message name="leg_goal_1_start_message"/>
  <link_to_objective name="obj_goal_1"/>
  <post_leg_message name=""/>
  <special_leg_directives name="goal_1" title="" next_leg="" shared_flight_leg_template="land" task_trigger_name="custom_trig_goal_1_for_task_01"
                          length_mt="100" distance_nm="19.932" shared_template_type="" target_pos="-19.58514404|146.78019714|89.92" custom_flight_leg_desc_fla
                          "yes"/>
  <display_object name="marker" instance_name="marker_goal_2_1" replace_lat="-19.58514404" replace_long="146.78019714" replace_elev_above_ground_ft="60"/>
  <desc>
    <![CDATA[Hello Pilot. Please fly to STARKE FIELD (elevation: ~294.999 ft)]]>
  </desc>
  <dynamic_message message_name_to_call="" relative_to_trigger="custom_trig_goal_1_for_task_01" length_mt="1000">You are near "STARKE FIELD</dynamic_message>
</leg>
<!-- ------- -->

<objective name="obj_goal_1" title="">
  <task name="task_01" base_on_trigger="custom_trig_goal_1_for_task_01" base_on_script="" eval_success_for_n_sec="10" mandatory="true" force_evaluation=""
                       base_on_external_plugin="false"/>
</objective>
```

You can see that we now have the "*dynamic_message*" in the first leg. The objective's target task name is "task_01" just like the value of "override_task_name".
You should also notice that there are no automatic "distance messages" and "triggers" in the *<leg>*.
Lastly, ***You do not have to use override_task_name*** if you used the "***override_trigger_name***", this was done to show the effect and for convenience.

```
Trigger "trig_dyn_message_goal_1_on_trigger_custom_trig_goal_1_for_task_01_4" - is standalone.
properties information:
Has special elevation settings: no.
["commands_to_exec_when_fired",""]  ["commands_to_exec_when_left",""]
["cond_script",""]                  ["cumulative_timer_flag","false"]
["dataref_to_modify_when_fired",""] ["dataref_to_modify_when_left",""]
["enabled","true"]                  ["eval_success_for_n_sec","10"]
["is linked","false"]               ["length_mt","1000"]
["message_name_when_fired","msg_4_dyn_message_goal_1_4"]  ["message_name_when_left",""]
["name","trig_dyn_message_goal_1_on_trigger_custom_trig_goal_1_for_task_01_4"]  ["plane_on_ground",""]
["post_script",""]                  ["rearm","false"]
["script_name_when_fired",""]   ["script_name_when_left",""]
["target_pos","-19.58514404|146.78019714|0.00"] ["trig_elev_type","0"]
["type","rad"]
Points defined: yes, List of stored points:
point 1: lat: -19.58514404, lon: 146.78019714

[Center] lat: -19.58514404, lon: 146.78019714, elev_ft: 0 [Defined: yes]
```

And here is our dynamic trigger and message that were created during load time. This can be extracted from the Log.txt file.
Unfortunately we can't see the message due to it not being part of the messages list when they were written to the Log file. I might change this to allow the "dynamic_messages" to also be displayed.

Summary

If you want to add or replace the generic messages, you should:

1. Define *disable_auto_message_b="true"* at the *<leg>* element.
2. Add *override_trigger_name* or *override_task_name* or both to the `expected_location` element (so we can predict the source coordinate name).
3. Add the element `dynamic_message` and point to the *trigger* or *task* names
   Do not forget to define the "length_mt" for the trigger radius.
4. Please do remember that the inventory area location is at the target trigger and its radius is 100 meters. Any message to simmer should take this into consideration.

In some templates the trigger/tasks are pre-defined so you can just use them and ignore the second step. *Just remember that the task must be trigger based*.

Option 2:

Define a message relative to location, either task or trigger and a predefined message name from "*<message_templates>*".
Its format should look as follows:

```
<dynamic_message [relative_to_trigger="trig_name" | relative_to_task="task name"] message_name_to_call="new_message_goal_1" length_mt= "2000" />
```

The second option does not need a text message since it uses a predefined message name. You will have to add the <*message_templates*> element to the <*TEMPLATE*> (you can read about <*message_templates*> in the designer guide).
The idea is to somewhat simplify the way a designer can construct his/her messages. The outcome will be the same as the first option.

OSM and WEBOSM (OVERPASS) Integration

OSM Web: Overpass

As of Mission-X v3.0.254.4 you can now fetch information from the overpass turbo website but as of v3.0.254.9 ***you also need to be explicit regarding the syntax***.
This also means that the feature is good as long as the site is active.
The benefit of this is that you can now design templates that should work all over the world (if there is data for that) and hopefully enjoy a diverse missions.

I strongly suggest to read about the implementation behind this feature to also understand the limitations and optimization part of it and thus when to ask the plugin to ignore optimizations (there is an example for this below).

In order to implement the overpass support you need to be aware of few concepts in overpass:

- The filter format should closely follow “overpass turbo” syntax.
  Since v3.0.254.9 you **must** provide a valid and explicit filter syntax
  Example:

```
(way['piste:type'='downhill']({{bbox}});way['piste:type'='sled']({{bbox}}));out;
```

  This syntax differs from the old one since the plugin stops adding the missing parts in the filter.

- The area coordinate is in the format of: “bottomLeft” and “topRight” coordinates.
- If you do not define a bound area (in location_value), then you have to at least define nm=”...” or nm_between=”...”. This will allow the plugin to define a boxed area with min/max distances.
- “Overpass” always fetches information based on a bounded boxed area, so you should always have to add the “**({{bbox}})**” to your filter string.
- Instead of spaces use: “**%20**” or “**+**”.
  I recommend using the DEBUG build to test the outcome in the Log.txt file.
- If you use the “`center`” in the print directive, ie:“`;out%20center;`”, the plugin will prioritise the element over the <nd> ones.
- It is a good practice to enclose your “key”,“value” strings with ***apostrophe*** marks.
  Example: `['piste:type'='downhill']`
  Do not use spaces in your filter rules, replace spaces with “**%20**” or “**+**”
- If you have more than one filter then use “**;**” between the filters.
- If you want a few filters to construct a search (a logical “*and*”) then do not use “**;**” between brackets.
- You have to enclose the filter correctly with parentheses:
  `(way[..]..way[..]({{bbox}}););out;`
- Test your filter as a URL string using “overpass turbo” or just check what Mission-X

sends to “overpass turbo” in the Log.txt file (use debug build to see additional outputs).

- The “***out;***” basically asks to print out the result. I do not suggest to use “***skel***” in your print format since it will remove tags that the plugin might use.

- Multiple “key=value” options will be translated to “union” of the data, which means, more information to search from, you have to take this into consideration when deciding if to ignore area optimization or not.

Template new attributes that supports **webosm**:

- location_type: *webosm*
- *Location_value* attributes:
  bounds=47.2966235,9.4304927,47.2966235,9.6530452
  The bounds is an *optional* attribute but you have to have at least one of:
  nm, nm_between or bounds.
- Location value attribute:
  webosm_optimize=”[y|n]”.
  Use the “webosm_optimize=n” flag **only** if you expect and know that the amount of data that will return is small for the whole area.
  This will force the plugin to ignore the area optimization and retrieve all nodes in the area.
  It is important that designers will test their work and check build times, in some cases like the one I provided below (for landing near hospitals) using area optimization is much slower than the whole area query.

Example 1: from “leict template_webosm” template:

```
<goal_medevac_piste_webosm template="land" name="goal_1" >
    <expected_location location_type="webosm"
location_value="nm=25|tag=webosm_piste_filter|bounds=46.9688169,9.4304927,47.296
6235,9.6530452|keyname=piste:name|keydesc=piste:name"
force_template_distances_b="yes"/>
  ...
  </goal_medevac_piste_webosm>
```

This directive will fetch a “piste” node from an overpass site, but we will also expect the plugin to optimize the area of search (default behavior).
We also defined both attributes: nm=”25” and bounds=”47.2966235...” the plugin will prioritize the “bounds” attribute over the “nm” one, but this is not an error. The benefit of using the “bounds” attribute is to make sure you only fetch information in a certain area and the use of “nm” is good practice since failing to have bounds nor nm will fetch empty data.

Here is an example to the filtering tag of example 1:

```
<webosm_piste_filter>
  <filter>
(way['piste:type'='downhill']{{bbox}};way['piste:type'='sled']{{bbox}};);out;
  </filter>
</webosm_piste_filter>
```

Another example is when we search for a landing location at the end of the mission. Let's assume we only want information from “overpass” and not from X-Plane.
The main element will look like:

```
  <goal_land template="icao,land" name="goal_2" >
    <expected_location location_type="webosm"
location_value="nm=25|tag=webosm_helipads|webosm_optimize=n"
force_template_distances_b="yes" />
 ...
  </goal_land>
```

And its filter:

```
<webosm_helipads>
<filter>
(way['amenity'='hospital']{{bbox}};way['aeroway'='heliport']{{bbox}};);out;
</filter>
</webosm_helipads>
```

In this example we are searching for hospitals and heliports in a radius of “25” nautical miles (`nm=25`). Since the expected information is quite small, we force the plugin to ignore optimization search on the area at hand (`webosm_optimize=n`).

The plugin basically ignores the usual “`icao`”template and just uses the webosm data. This is important since if you want icao searches, then you will have to add “`location_type`” other than “webosm”

The benefit of using OVERPASS api is tremendous since you only need to enter the base filter you are interested in and the rest will be done by the API. It also saves the hassle of preparing database files ahead of time.
In both cases - database or overpass, you will have to prepare the filters correctly and test them.

The limitations of the current implementation of the filters are:

- You can’t use more than one “bounded” area in your filters.
- You must provide explicit filters.

Useful elements to assist in manipulating plane state/condition:

<fire_commands_at_leg_start /> <fire_commands_at_leg_end />

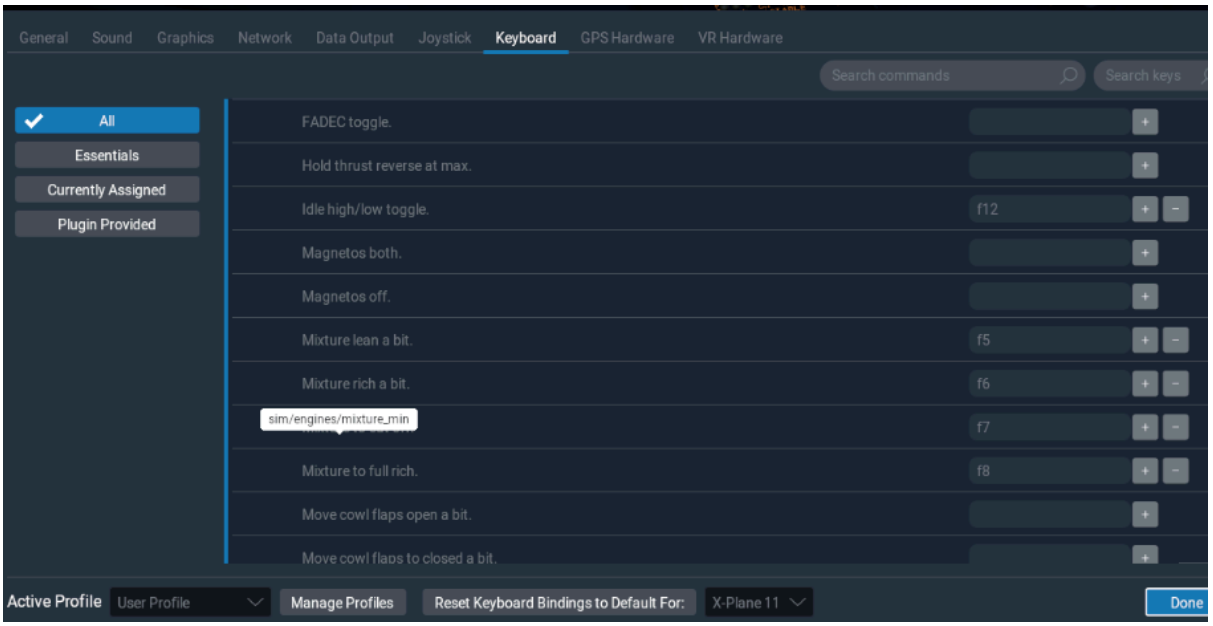
These two elements assist the designer to manipulate the plane commands at the beginning or at the end of a flight “leg”. You can use X-Plane native commands or custom commands, like the Bell-412 by X-Trident:
Example:

```
<leg ....>
<fire_commands_at_leg_start commands="412/buttons/PATIENT_off" />

<fire_commands_at_leg_end commands="412/buttons/PATIENT_on_board" />
</leg>
```

You can concatenate more commands using comma delimiter (“,”) between them.
You can even define a cold and dark set of commands, but I suggest using the dataref approach, since it manipulates the system directly.

You can get the commands path by hovering above the desired command in X-Plane’s keyboard binding screen.



The tooltip holds the commands path.

<datarefs_start_cold_and_dark>
This element should be defined in the “template” part as a stand alone element and it will be added to the mission file as is.
By using this element you can define a set of “datarefs” and their values (only one value for each one), to be set at mission start.
This can assist in bringing the plane to a cold and dark state, or other desired states, as long as the “dataref” is available to the plugin.

For more information, consult the “designer guide” document.

Translating “Leg” template type and “Location” Attributes

Translation of *<leg>* ***template*** vs ***location_type*** and its ***location_value*** attributes.

Plane Type	**<leg template=”” >**	**Location type**	**Location value**	**How plugin translates**
helos	land/hover	xy	nm={number}	*Pick a random coordinates location in radius {number}* *(only if not last flight leg)*
helos	any	osm *[v3.0.242.10]*	nm={number}|tag=”osm_land”	*Pick a random location in the radius of {number} nm and try to pick them first from the OSM database.*
helos	any	osm *[v3.0.242.10b3]*	nm=10|tag=”osm_land”|dbfile={file.db}	*Pick a random location in the radius of {number} nm and try to pick them only from the OSM database file {file.db}. Ignore any other database file.*
helos	any	osm *[v3.0.242.10b3]*	nm=0|tag=”osm_land”|dbfile={file.db}	*Pay attention: “nm=0” and “dbfile={file.db}”* *Pick **any** location in the OSM database file {file.db}. Ignore any other database file.* *Use in small osm database areas that will fail to find segments even in small distances, like: “Liechtenstein”*
any	land/hover/near	near, xy	~~“” empty or “_” or~~ “nm=_”	*[v3.0.220.4]Pick nearest ICAO .* *If the last flight leg in the mission then the <leg template=””> attribute will be overridden with the “land” value.* The underscore “_” represents an empty option. Use it when there are multiple options to pick and one of them should be blank.
plane	land/near	xy	nm={number}	*Pick ICAO in the radius of {number}.* *If “near” was picked, then pick the nearest ICAO.*
any	land/hover	xy	tag={tag name}	Picks a random point from “element” with the same tag name
plane	land	xy, icao	nm={number} and tag={tag name}	*Pick ICAO in the radius of {number}.*
any	land/hover	xy, icao	tag=”{tag_name} and nm_between=20-40	*Pick between <points> in {tag_name} that are between “20-40” nm. If none was find, then pick the closest one.*
	start	Any (not empty)	ignores	Will pick the starting location coordinates.
	{anything}	start	{anything}	*[v3.0.220.6]* Plugin will override <leg template=””> with the “start” value and will do the same for location_value.

We can mix location types and values

Plane Type	<leg template="" >	Location type	Location value	How plugin translates
any	land,hover	xy,near	"tag={ name},nm=_"	1. Pick one of the <leg> template types. 2. Pick one of the "location_type"s: xy or near. Respectively pick location value - If Picked "xy" then will use {tag name} If picked "near" then will use "_" which represents empty string = pick closest icao.
any	land,hover	xy,near	"tag={ name}" This time we only have one value option	1. Pick one of the <leg> template types. 2. Pick one of the "location_type"s: xy or near. Use: "{tag name}" with either "xy" or "near" location types.
any	land,hover	xy,xy,near	"nm=20, tag={ name},nm=_"	1. Pick one of the <leg> template types. 2. Pick one of the "location_type"s: xy, xy and near. Respectively pick location value - If Picked first "xy" then will use 20 as area to pick icao/coordinate. If Picked second "xy" then will use {tag name} to pick random point and If picked "near" then will use "_" (empty string) = Pick closest icao,

When *location_value* has more than one option, then the plugin tries to give each location_type its respective *location_value*.
If there are few "*location_types*" but only one "*location_value*" then all "*location_type*" attributes will use the same value.

3D Objects Diversity

```
<object_templates>
  <!-- Ambulance -->
  <obj3d  name="ambulance" file_name="" random_tag="ambulance_select" />
  <obj3d  name="car01" file_name="../../../OpenSceneryX/.../ford_mustang/object.obj" />
  <obj3d  name="marker" file_name="marker_five_parts_02.obj" />
</object_templates>
```

We define 3D Objects in the "**object_templates**".
If we want to have a random "Ambulance" then we can add the "random_tag" attribute with another "element name". The plugin will pick a 3D Object from that element and replace it with our "file_name" value.

Here is an example of a random ambulance element:

```
<ambulance_select>
  <obj3d  name="ambulance" file_name="USAmbulance.obj" />
  <obj3d  name="ambulance2" file_name="car_ambulance.obj" />
</ambulance_select>
```

The plugin will only replace the attribute "file_name" value in the "obj3d" element with the one in the <ambulance_select> one. This means that an Ambulance instance might be "USAmbulance" or "car_ambulance". We can extend this to any object as long as we have them available.

In **v3.304.14** the "is_virtual_b" was deprecated since the plugin always searches for a file firstly in the "virtual path" space, and then as a "physical" file.
Example:
Instead of defining a 3D Object with its physical location:

```
<obj3d  name="car01" file_name="../../../OpenSceneryX/.../ford_mustang/object.obj" />
```

We can write the following (copied from opensceneryx library.txt file):

```
<obj3d  name="car01"
file_name="opensceneryx/objects/vehicles/cars/coupe/ford_mustang.obj" ~~is_virtual_b="yes"~~
/>
```

This will minimize the failure in reading the 3D object file due to entering the wrong path or wrong folder name.

```
<display_object name="car01" instance_name="car01_1"
relative_pos_bearing_deg_distance_mt="360|10" replace_lat="" replace_long=""  />
```

In the following example we see the usage of the attribute:
"relative_pos_bearing_deg_distance_mt".
This long attributes name means: Relative Position an object in bearing (degrees) and distance in meters ("relative_pos_bearing_deg_distance_mt")
The values: "360|10" hints where to place the 3D Object. The 360 represents "heading" while

the “**10**” represents distance in meters from the target.
Meaning: place 3D Object 360 deg relative to target and at 10 meters away from it.

The importance of that attribute is the ability to place objects relative to target position. That way we can place a few objects next to each other in our generated mission and create some clutter.

Handling water bodies locations

More options in case of "fully random" missions

In missions that are fully randomized, we do not know where the flight "leg" location would be. Would it be on land or in water? What does it mean from handling 3D Objects?
Introducing: "*random_water_tag*" attribute in the <display_object> element.
The plugin uses "*random_tag*" and "*random_water_tag*" attributes in the <display_object > element.
Using "random_water_tag" in the "<display_object>" will allow for granular control over what will be randomly placed on water instead of the original "on land" object.
Let's look at an example:

```
<leg template="hover,land" name="goal_1" >
  <expected_location location_type="xy" location_value="25" />
  <display_object name="marker" instance_name="marker_goal_2_1"
                  replace_elev_above_ground_ft="60" />
  <display_object name="ambulance" instance_name="ambulance_01"
                  random_tag="ambulance_select"
                  random_water_tag="water_replace_ambulance" />
  <display_object name="random_thing" instance_name="something near ambulance"
                  relative_pos_bearing_deg_distance_mt="360|10"
                  random_water_tag="something_in_water" />
</leg>
```

There are two random attributes we can use in the *<display_object>* level, "random_tag" is mainly for regular 3D Objects (I refer to them as land objects) while the second attribute is the "random_water_tag" for specific cases where the flight leg ends in the water.
The technique is exactly the same, only the attribute: "*random_water_tag*" will be used by the plugin when the location will be in water. If the attribute is not present, then it will use what was defined originally.

3D Object Set

The “display_object_set” element allows you to define combinations of <*display_object*> and <display_object_near_plane> elements for certain cases and also define their rules. The benefit is: the plugin will pick one of the object sets and will add **all** <*display_objects*> and <display_object_near_plane> elements to the flight <leg> being built.

```
<leg template="hover,land" name="goal_1" >
  <expected_location location_type="xy" location_value="nm=25"  />
  <display_object_set random_tag="object_sets" set_name="medevac" />
</leg>
```

The element needs 2 attributes:

1. “*random_tag*”: tag name that holds the set of display objects and
2. “*set_name*”: the name of tag element to pick from.

Let’s look at an example of an element that holds some display sets:

```
<object_sets>
  <medevac>
    <display_object  name="marker"           replace_elev_above_ground_ft="60" />
    <display_object  name="random_ambulance" random_tag="ambulance_select"
                     random_water_tag="water_replace_ambulance"    limit_to_terrain_slope="20" />
    <display_object  name="random_car"       random_tag="crashed_car"
                     relative_pos_bearing_deg_distance_mt="360|10" random_water_tag="blank"
                     limit_to_terrain_slope="20" />
    <display_object  name="random_sign"      random_tag="clutter" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="180|10" />
  </medevac>

  <medevac>
    <display_object  name="marker"           replace_elev_above_ground_ft="60" />
    <display_object  name="random_police_car" random_tag="police_car"
                     random_water_tag="water_replace_ambulance"    limit_to_terrain_slope="20" />
    <display_object  name="random_car"       random_tag="crashed_car" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="360|10" limit_to_terrain_slope="20" />
    <display_object  name="random_police01"   random_tag="policeman" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="360|5" limit_to_terrain_slope="" />
    <display_object  name="random_police02"   random_tag="policeman" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="345|5" limit_to_terrain_slope=""/>
    <display_object  name="random_person1"    random_tag="people_at_location" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="10|10" limit_to_terrain_slope="" />
    <display_object  name="random_person2"    random_tag="people_at_location" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="15|10"  limit_to_terrain_slope="" />
    <display_object  name="random_sign"       random_tag="clutter" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="180|10" />
  </medevac>
```

```
 <medevac_slope>
    <display_object  name="marker"          replace_elev_above_ground_ft="60" />
    <display_object  name="random_person1"  random_tag="people_at_location" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="10|10"  limit_to_terrain_slope="" />
    <display_object  name="random_person2"  random_tag="people_at_location" random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="15|10"  limit_to_terrain_slope=""
                     optional="70%" />
    <display_object  name="random_sign"     random_tag="clutter"  random_water_tag="blank"
                     relative_pos_bearing_deg_distance_mt="180|10" optional="50%" />
 </medevac_slope>

</object_sets>
```

In this example, we have sets that are “*medevac*” and “*medevac_slope*”.
You can define any name to the set of “display_objects” you want, I used names relevant to a template.

If you define your own set names, and you want to handle slope cases, you should add the “`limit_to_terrain_slope`” attribute for each 3D object you want to restrict.

The <Content> element:
A New way to present random missions

A <content> element can assist in the way we present a mission.
Prior to v3.0.222.6 we defined flight “legs” as medevac or delivery and if they are optional or not. Since v3.0.222.6 there is a new element by the name: “<content>.

<content>
The content allows you to define a high level mission flow. It should provide from which “set” of elements it should pick the “flight <leg>” information to build the mission. That is also what makes it flexible and the best way to construct your mission upon.
When we define the “<content>” element we also define list elements that will assist to produce a mission. The name of the elements in the list are free text and defined by the designer (I suggest they will be meaningful).
Here is a short snippet that describe a content element:

```
<content>
  <medevac list="goal_medevac,goal_land" min_valid_flight_legs="2"
           plane_type="helos" >some text
  </medevac>
...
</content>
```

You can create any element name in content but make sure you support it later on.
If you want a specific type of plane, you should use the “plane_type” attribute,the plugin won’t guess and will default to “helos”.
You can define the “plane_type” at template level or content level. Default is “helos”.

The format of the sub elements of <content> element should be:

```
<element_name list="tag_name|{option}={value}|....,tag_name,..."
              plane_type="" min_valid_flight_legs="" > text </element_name>
```

- **element_name:** designer defines element name (I suggest to use meaningful names, like medevac or delivery).
- **List attribute format:**
  - **First value** in the "list" attribute, **must** represent a *tag name* in the template to randomly pick from.
  - [optional] Second value that are concatenated to the first value using pipe character "**|**" must be in the format: "key=value" (for easier readability).
  - We use comma ("**,**") as a delimiter between tag names in the list.
  - [optional] "plane_type" should be defined if it is different from the template or you did not define it at "template" element level. Default is "helos" but you should not rely on defaults for this attribute.
- The text between the element tag: "`<medevac ... > text </medevac>`" is used in the mission briefer description.

Here is an example:

```
<content>
  <medevac list="goal_medevac,goal_land" min_valid_flight_legs="2" >medevac text 1</medevac>
  <medevac list="goal_medevac,goal_land,goal_delivery" min_valid_flight_legs="2">medevac text 2</medevac>

  <medevac_hover list="goal_medevac_hover,goal_land" min_valid_flight_legs="2" >
                 medevac with hover text</medevac>

  <delivery list="deliver_medevac_set, goal_delivery|optional=50%,goal_delivery"
             min_valid_flight_legs="1">delivery text 2</delivery>
  <delivery list="goal_delivery,deliver_medevac_set,goal_delivery,goal_back_to_start">
            delivery text 3</delivery>
</content>
```

- Our content options are: "`<delivery` ", "`<medevac` " and "`<medevac_hover`".
  You can provide any name you want just to make their name meaningful, for example: <medevac_hover> or "<mountain_rescue>".

- The "`list`" attribute defines a <u>set</u> of element tags that the plugin will have to pick and build flight legs from it.

- You can create many different *element_tag_names* for specific cases and add them to one of the lists to create diversity.
  In the first example I used the element tag name: "`goal_medevac`", "`goal_land`".
  In the last example I used: "`goal_delivery`", "`deliver_medevac_set`", "`goal_back_to_start`".

- `min_valid_flight_legs=""` allows you to define minimum expected "legs" to flag the mission is valid to fly in. For example, a medevac mission might need at least 2 "legs"

according to your definition, in other cases it might need 3 “legs”. In that case we can just change it to 3.
This means that the plugin will fail the mission creation with a message similar to:
“Failed to create a mission based on <content>. Generated: 2 ‘flight legs’, but needs minimum: 3 ‘legs’. Aborting mission creation.”

Flow of evaluation from plugin side in relation to content element:

1. Pick a random element from <content> element.
2. Read the “*plane_type*” attribute if it was set in the picked element (strongly suggested)..
3. Read the “*min_valid_flight_legs*” attribute if it was set.
4. Split the “*list*” attribute
   Example: list="*goal_delivery*,*goal_delivery*|*optional=50%*,*goal_land*”
   a. Each split *variable* in the list represents a “*tag name*”.
   b. For each tag name, split its *attached rules* - if any (example: goal_delivery|optional=50%).
5. Go over each tag name in the list and randomly pick an element with the same “tag name” we read from our list.
   a. If the tag represents a simple flight “leg” then build one based on it.
   b. If the tag represents “set of flight legs” then add all flight <leg>” sub elements to the new mission (see below).

The *content* element allows the template designer to define a set of “flight legs” just like we define a set of 3D Objects. Each time a mission is created it might have a different number set of flight “legs” and different template for the “legs”. This let’s you add diversity to the generated “flight plan” in only one file.

What is special in the "`deliver_medevac_set`" element tag name ?
The short answer - nothing is special about it.
The longer answer is: you need to check the attributes of the element "`deliver_medevac_set`".
Let's take a closer look:

```
<deliver_medevac_set is_set_of_flight_legs="yes" copy_leg_as_is_b=”no” >
  <leg template="land" name="goal_2_1" >
    <expected_location location_type="icao" location_value="20"   />
    <display_object name="marker" instance_name="marker_goal_2" replace_elev_above_ground_ft="60" />
  </leg>
  <leg template="land" name="goal_2_2" >
    <expected_location location_type="icao" location_value="20"   />
    <display_object name="marker" instance_name="marker_goal_2" replace_elev_above_ground_ft="60" />
  </leg>
</deliver_medevac_set>
```

When using the "<content>" element in a template, we can also build a set of "flight legs", just like sets of 3D Objects in a scene.
In order for the Mission Generator to distinguish between sub elements that it needs to pick only one element or pick all elements, we need to use a special attribute:
`is_set_of_flight_legs="yes"`

The "`copy_leg_as_is_b`" is new to Mission-X v3.0.304. It tells the Random engine to not generate or evaluate the <leg> element but to copy it as a whole. This is in cases when you want hardcoded flight legs especially when you convert a mission to a template and just want to add some options.

> FYI: The plugin ***does not*** evaluate the *value* of the attribute, It just checks for the presence of the attribute and if it is not empty.

Once the Mission Generator "sees" this attribute, it adds ***all*** sub elements with the name "`<leg>`" to the mission being constructed.
You can see that the sub element names are "`<leg>`" and not something randomly picked like "`goal_delivery`".

<u>To Summarize the <content> element.</u>

The "content" element brings another dimension to the Random Mission Generator. It should allow the designer to better create a mission with some logical flow and options.
In my opinion the <content> element is the way to go forward when generating missions, My reasoning:

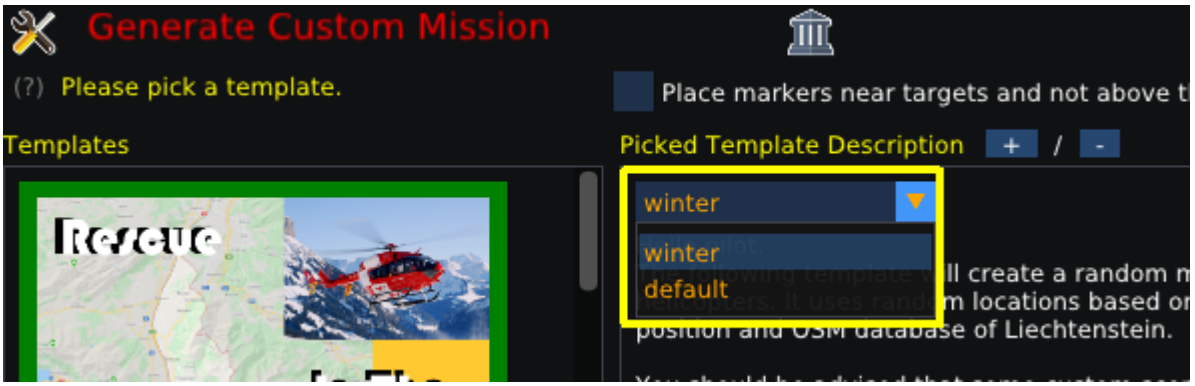
- It allows to use features like: set of flight legs (ad-hoc "flight legs")
- Define "optional" "legs" which makes for some diversity.
- Define description for the mission.

I suggest checking a template that was already defined by someone to better understand the flow. It should not take more than 10min to be able to decipher how the template should work, it is almost like a regular template, only we added goal picking as an option and not as a rule (using the “list” attribute).

Dynamic Template

By Dynamic templates we mean that the designer can allow the simmer to change the content of the original template as a result of picking from an option combo.
For that we have a new element, the: *<REPLACE_OPTIONS>*
The *<REPLACE_OPTIONS>* element is a new root element in the template file.
The idea behind this element is to allow the designer to build only ***one*** template but with different contents in specific locations of the XML page.
The replacing content might be located in external files relative to the location of the template at hand or as a simple "text" to replace some string in the template.
For example, let's assume you want to create a template that its 3D objects will be replaced by the options the user will pick from a list. Now let's assume there are two options: "*winter*" and "*default*".
Instead of writing two different templates, you can write ***one*** template with two different external files that will have different 3d-object elements, one for winter and one for the rest of the seasons.

In the following example, you can see the two options: "*winter*" and "*default*"



[WILL BE DEPRECATE]
Please read the <REPLACE_OPTIONS> topic after the "striked" paragraph.

~~In the template file it will look as follow:~~

```
<REPLACE_OPTIONS>
  <opt name="winter">
    <find_replace find="%obj_people%" replace_with="winter_people.txt" />
    <info difficulty="Medium" weather_settings="Set to broken or low visibility"/>
  </opt>
  <opt name="default">
    <find_replace find="%obj_people%" replace_with="default_people.txt" />
  </opt>
</REPLACE_OPTIONS>
```

~~The **<REPLACE_OPTIONS>** is now a root element.~~
~~The **<opt>** sub element defines the text the end-user will see and pick from.~~

~~The *<find_replace>* sub-element defines which string to replace in the current "template" file and with which file content or other string to replace with (the plugin replaces all occurrences).~~
~~You can define one or more *<find_replace>* elements.~~
~~The **<info>** element is a new sub-element that allows you to change the <mission_info> template sub-element but with some limitations. The rule is that any attribute you define in info will override the same attribute in <mission_info> element. It has precedence over the "find_replace" sub-element since it is run last in the code.~~

<REPLACE_OPTIONS>

As of "**v25.01.1**" the plugin supports "multi-options", so the simmer does not need to only pick from "*one*" set of options, but it can choose from different ones. This is dependent on what the designer defined to tailor the mission.
In order to support this, we should use the <option_group> sub element.
See example:

```
<REPLACE_OPTIONS>
    <option_group name="Weather XP12">
        <opt name="Calm weather">
            <find_replace find="%weather%" replace_with="includes/weather_calm_12.txt"/>
        </opt>
        <opt name="Rainy weather">
            <find_replace find="%weather%" replace_with="includes/weather_rainy_12.txt"/>
        </opt>
        <opt name="Windy weather">
            <find_replace find="%weather%" replace_with="includes/weather_windy_12.txt"/>
        </opt>
    </option_group>

    <option_group name="Inventory Layout">
        <opt name="X-Plane 12">
            <find_replace find="%comp%" replace_with="12"/>
        </opt>
        <opt name="X-Plane 11">
            <find_replace find="%comp%" replace_with="11"/>
        </opt>
    </option_group>
</REPLACE_OPTIONS>
```

Here is an example of "multi option selection".

Converting Suggestion

Mission-X v25.01.1 will still support the "old" and "new" formats, so they can both co-exist. But please do remember that in the near future the "old" format support will be dropped.

To convert the "old" format to the "new" format you just need to enclose your "<opt>" nodes, with <option_group name="short description"> element.

<option_group>

Attribute Name	Type	Value	Meaning
name*	str	"some text"	Short description of what the group holds. It will also be used as the "**label**" for the "drop down list", so make sure to not have a long name.

This is the main element that encloses all the options sub elements.

<opt>

Attribute Name	Type	Value	Meaning
name	str	"some text"	Short description of what the Option does. This will be listed in the "drop down list".
<find_replace> Sub element to <opt>			
find	str	"%some text%"	Provide a text to replace. I suggest delimiting the text with "%".
replace_with	str	Text	The text can represent the "value" to replace with, or a "relative path" to a file that holds the "replacing text".

There are some limitations for some attributes:
If you modify the "`template_image_file_name`" or "`mission_image_file_name`" attributes, the image won't be loaded until the next time you reload X-Plane.

Here is an example of a *before* and *after*:

Before:

```
  <people_aiders>
    %obj_people%
  </people_aiders>
```

After:

```
<people_aiders>
 <obj3d name="person01" file_name="../../../3D_people_library/policeman_1.obj"/>
 <obj3d name="person01" file_name="../../../3D_people_library/policeman_2.obj"/>
 <obj3d name="person01" file_name="../../../3D_people_library/policeman3.obj"/>
</people_aiders>
```

Remember:
It is up on the designer to validate that the “injected” text from the external file will not break the XML validity.
You should see an error in the plugin screen and in the Log.txt file if that is the case.

Appendix - A

OSM database

A more practical way is to use the OSMWEB solution instead of the database one.

One of the nice features introduced in v3.0.242.9 was the inclusion of OSM support, based on sqlite database but only in the “user create template” screen (osm data needs to be manually converted to sqlite database using the provided script).
The OSM database is supported in both the “plugins template folder” (“Resources/plugins/missionx/templates”) and in a “template mission folder” (“Custom Scenery/missionx/{mission folder})”.
Unfortunately to fully implement it you need to have some knowledge of SQL, fortunately this is quite simple and the main queries to use are already defined so you just need to tinker with the filters. Check any template with OSM support and try to mimic it and extend it to your liking (in your template).

About the database structure

The OSM database has 2 main tables that we base all of our queries:
“***way_street_node_data***” (alias name: wsnd) and “***way_tag_data***” (alias name: wtd).
The “wsnd” table holds all the coordinates.
The “wtd” holds the metadata of each coordinate segment.
The field that joins these two tables is the “***id***” field.

How to add the OSM support to your template

In order to be backwards compatible with the template syntax, we will use the following elements + attributes in order to tell the plugin to use the OSM data:

- "*location_type*" and "*location_value*" will flag the target as OSM based and will provide filtering information.
  This means you can mix and match other "location types" like "land,icao" and others.

Example:

```
<expected_location location_type="osm" location_value="nm=25|tag=osm_sql" />
<osm_sql>
  <filter_sql>
    and key_attrib='highway' and val_attrib in ('primary', 'secondary', 'tertiary',
    'residential', 'service', 'living_street', 'track')
  </filter_sql>
  <filter_sql>
    and key_attrib='highway' and val_attrib in ('primary', 'secondary', 'living_street')
  </filter_sql>
</osm_sql>
```

The "*location_type*" has a new supported value and it is the "**osm**". This tells the plugin that the OSM databases should be used.
The "*location_value*" holds the expected radius and the root tag name that holds all the possible filters (`tag=osm_sql`).
In this example we have two optional filters but you can have as many as you want.
The name of the sub element is not important, you just need to have a valid "fragment sql text" in it.

The plugin builds an SQL query from a predefined string and your filter. Here is an example:
Part 1 of the query (plugin internal):

```
  select id, distance from (
         select distinct wtd.id, round( 3440 * acos(cos(radians(?1)) *
cos(radians(wsnd.lat)) * cos(radians(?2) - radians(wsnd.lon)) + sin(radians(?3)) *
sin(radians(wsnd.lat))) ) as distance_nm
         from way_street_node_data wsnd, way_tag_data wtd
         where 1 = 1
         and wsnd.id = wtd.id
         and wsnd.lat between ?4 and ?5
         and wsnd.lon between ?6 and ?7
```

Part 2 of the query is picked from the filter element:

```
and key_attrib='highway' and val_attrib in ('primary', 'secondary', 'living_street')
```

Part 3 of the query (plugin internal) - appended by the plugin to make it complete:

```
 ) where distance_nm between ?8 and ?9
```

The full statement will look something as follows:

```
  select id, distance_nm from (
        select distinct wtd.id, round( 3440 * acos(cos(radians(?1)) *
cos(radians(wsnd.lat)) * cos(radians(?2) - radians(wsnd.lon)) + sin(radians(?3)) *
sin(radians(wsnd.lat))) ) as distance_nm
        from way_street_node_data wsnd, way_tag_data wtd
        where 1 = 1
        and wsnd.id = wtd.id
        and wsnd.lat between ?4 and ?5
        and wsnd.lon between ?6 and ?7
and key_attrib='highway' and val_attrib in ('primary', 'secondary', 'living_street')
 ) where distance between ?8 and ?9
```

This specific query can only be executed from the plugin itself, since some of the SQL functions I use here are not built into the vanilla SQLite API (highlighted in yellow).
If I find a solution for that, I'll share it with you.

What happens if there is no OSM data in the area

In such cases, when there is no OSM data to pick from, the plugin will fallback to original behavior and will pick randomly a point in the area at hand.

Limitation

This feature is currently supported in "medevac" with "helos". Doesn't really make sense in other cases.

Rules regarding OSM databases

All of the sqlite OSM databases should be placed in "{missionx}/db" folder or in "Custom Scenery/missionx/{template mission folder}/db".
You can have many files with different names but all have to be based on the sqlite v3.x version of the database.
When the Random Engine search an osm location, it will do the following:

1. Search in all the "database files" data that is in the expected lat/long boundaries of your current position.
2. Plugin reads the table "bounds" in each file and picks the file with at least 3 coordinates in the boundaries.
3. Once a file was found, it will build the base statement and filters based on the [**tag=**"{some name}"] or based on predefined ones (see the "missionx/libs/sql/sql.xml" file).
   These rules could be modified externally in the "missionx/libs/sql/sql.xml" file but at your own risk (they are subject to change though).
   You are more than welcome to share your queries if you think they produce more diversity or desirable outcomes.

4. Plugin executes the “query” on the database and retrieves all the location “id”s we should pick from.
5. After picking a random location, we fetch its metadata and store the desired attributes values into our “new NavAid”.
6. If you are using “osm” in your last leg, then the location found will be tested against X-Plane own database and the plugin will use X-Plane’s own coordinates to be on the safe side.

If we fail to fetch information the plugin will fallback and try to pick information from X-Plane (as before). In worst case scenarios the simmer will have to rerun the mission generator.

Revision Table

Revision No.	Date + Modifier	What was changes
0.46		* Added “mission_file_format” attribute to the <TEMPLATE> element, so you could define the preferred “mission file format” version when the mission file is being generated.
0.45	24-feb-2025	* Added some "deprecation" warnings due to the new "land hover" implementation which makes the "slope" and similar attributes moot. Search for "v25.02"
0.44	28-jan-2025	* Extended <REPLACE_OPTION> element. Added <option_group> to support "multi-option" to better tailor the mission. * Added a list of "future deprecation" attributes and setup. Please check the "Breaking Changes" for "v25.01.1".
0.43	02-nov-2023	* Added “`message_name_when_entering_physical_area`” attribute to dynamic_message.
0.42	23-june-2023	* Removed the need of the “is_virtual_b” attribute.
0.41	03-feb-2022	* Check breaking changes for 3.304.12 * Added <weather> support for <leg> elements.
0.40	04-dec-2022	* Added “<display_object_near_plane>” and “relative_pos_bearing_deg_distance_mt” usage * Added the new behavior of how the plugin reads <display_object_xxx> elements.
0.39	02-nov-2022	* Renamed <rank> to <scoring> (breaking change)
0.38	08-oct-2022	* Added <rank> element
0.37	20-sep-2022	* Added “`starting_icao`” to “<briefer_and_start_location>”
0.36		* reorganize parts of the documents to have a similar look as the “designer guide”.
0.35	13-feb-2022	* Added explanation on: “copy_leg_as_is_b”
0.34	19-jan-2022	* Added “`position_pref” attribute.`
0.33	20-nov-2021	* Added missing <REPLACE_OPTION> breaking change from v3.0.256.4.1 and v3.0.256.4.2
0.32	29-oct-2021	* Added <overpass> sub element to the <mission_info> to force different url lists to overpass search.
0.31	17-oct-2021	* Moved <replace_options> as new root element, also renamed to capital letters: <REPLACE_OPTIONS>. * Added the “title” attribute to the <mission_info>