Template Builder User Guide

Version 3.5

Disclaimer

Vizrt provides this publication “as is” without warranty of any kind, either expressed or implied. This publication may contain technical inaccuracies or typographical errors. While every precaution has been taken in the preparation of this document to ensure that it contains accurate and up-to-date information, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained in this document.

Vizrt’s policy is one of continual development, so the content of this document is periodically subject to be modified without notice. These changes will be incorporated in new editions of the publication. Vizrt may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time. Vizrt may have patents or pending patent applications covering subject matters in this document. The furnishing of this document does not give you any license to these patents.

Antivirus Considerations

Vizrt advises customers to use an AV solution that allows for custom exclusions and granular performance tuning to prevent unnecessary interference with our products. If interference is encountered:

- **Real-Time Scanning:** Keep it enabled, but exclude any performance-sensitive operations involving Vizrt-specific folders, files, and processes. For example:
  - C:\Program Files\[Product Name]
  - C:\ProgramData\[Product Name]
  - Any custom directory where [Product Name] stores data, and any specific process related to [Product Name].
- **Risk Acknowledgment:** Excluding certain folders/processes may improve performance, but also create an attack vector.
- **Scan Scheduling:** Run full system scans during off-peak hours.
- **False Positives:** If behavior-based detection flags a false positive, mark that executable as a trusted application.

Technical Support

For technical support and the latest news of upgrades, documentation, and related products, visit the Vizrt web site at www.vizrt.com.

Created on

2026/04/22

Contents

1 Overview

Template Builder lets you make customized templates using scene import or existing templates from Viz Pilot's Template Wizard. This user guide shows you how to customize templates.

Info: A key feature is that you can add custom HTML panels to templates, giving full control over the template through custom scripting and logic.

1.1 Workflow

A simplified version of the workflow follows below:

- Scenes are made in Viz Artist.
- The scenes are imported into Template Wizard, where templates are made.
- Templates are edited and new templates can be made in Template Builder.
- The template is saved in the Viz Pilot system and is available to newsroom and control room systems for playout.

Note: Changes made to a template in Template Builder are not be available when opening the template in Template Wizard.

1.2 Feedback and Suggestions

We encourage suggestions and feedback about our products and documentation. To give feedback and/or suggestions, please contact your local Vizrt customer support team at www.vizrt.com.

1.3 Support

Support is available at the Vizrt Support Portal.

For more information about all Vizrt products, visit:

- www.vizrt.com
- Vizrt Documentation Center
- Vizrt Training Center
- Vizrt Forum

2 Configuration

This section covers the following topics:

For software and hardware requirements, please check the Viz Pilot Edge User Guide.

2.1 Open Template Builder and Connect to Pilot Data Server

Template Builder opens as a web application in your default browser.

The URL to access Template Builder, if hosted on Pilot Data Server, is: http://pds-hostname:8177/app/templatebuilder/TemplateBuilder.html. Template Builder then tries to connect to a Pilot Data Server on the same host, unless another the Pilot Data Server URL is specified in the *pilot* URL parameter: http://examplehost:8177/app/templatebuilder/TemplateBuilder.html?pilot=http://another-host:8177.

2.2 Specifying Preview Server, Graphic Hub REST and Crop Server

Template Builder connects to Graphic Hub REST to import scenes, to Preview Sever to generate snapshot previews of scenes, and crop server to offer a crop tool for images. All these servers need to run, and their end points need to be specified in the Settings of Pilot Data Server, to which Template Builder connects.

2.2.1 Open, Edit and Save Settings

The Pilot Data Server settings are stored in the Pilot database, and apply to all Pilot Data Server instances connecting to this database.

1. Access the Pilot Data Server Web Interface (http://pds-hostname:8177).
2. Click the **Settings** link.
3. Search for the relevant setting.
4. Add or modify the setting.
5. Click **Save**.

Preview Server (PS)

Preview Server manages one or more Viz Engines, providing frames for thumbnails and snapshots in an ongoing preview process.

Follow the procedure above, and select the *preview_server_uri* setting, and add the URL for the machine on which you installed Preview Server (usually http://previewserver-hostname:21098).

> **Note:** All applications with a connection to the database now have access to Preview Server.

Graphic Hub REST (GH)

The REST service of Graphic Hub used to store your scenes must be specified explicitly in Pilot Data Server.

Follow the procedure above and select the *graphic_hub_url* setting, and add the URL for the machine on which your scenes are stored (usually http://gh-hostname:19398). Note that this setting is pointing to the Graphic Hub REST end point.

Note: This connection needs authentication from the Pre-authenticated Hosts part in the Search Providers settings.

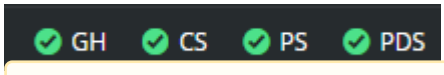
Crop Server (CS)

A crop service is normally run on the same computer as Pilot Data Server. When specified in settings, the Viz Pilot Edge user can open a crop tool to crop images.

Follow the procedure above and select the *crop_service_uri* setting, and add the URL for the machine on which Crop Server is running (usually http://pds-host:8178).

2.3 Monitoring Server Status

Green icons at the bottom of the interface show the server status of the service end points mentioned above, and which database you are currently connected to. Hover over the icons to see the URL and CTRL + Click to open Pilot Data Server Settings. Note that the status does not refresh unless Template Builder is reloaded in the browser.



Note: GH REST status info is based on the *graphic_hub_url* parameter mentioned above, not Graphic Hub's search provider settings.

3 Managing Templates

This section covers the following topics:

3.1 See Also:

- Transition Logic and Combo Templates

3.2 Creating and Opening Templates

All templates created in Template Builder must be based on one or more Viz Artist scenes. Properly organizing and importing scenes into a template is a critical step in the template creation process. These scenes determine the graphics displayed on-air after the journalist enters the required data and the operator triggers playback in the graphics renderer.

The process begins with creating a new concept. Concepts function as organizational containers for content and must be established in the system before any templates can be created. Additional templates can be added to a concept later. Unlike traditional folders, templates in Template Builder can belong to multiple concepts, allowing to create global templates that can be shared across concepts.

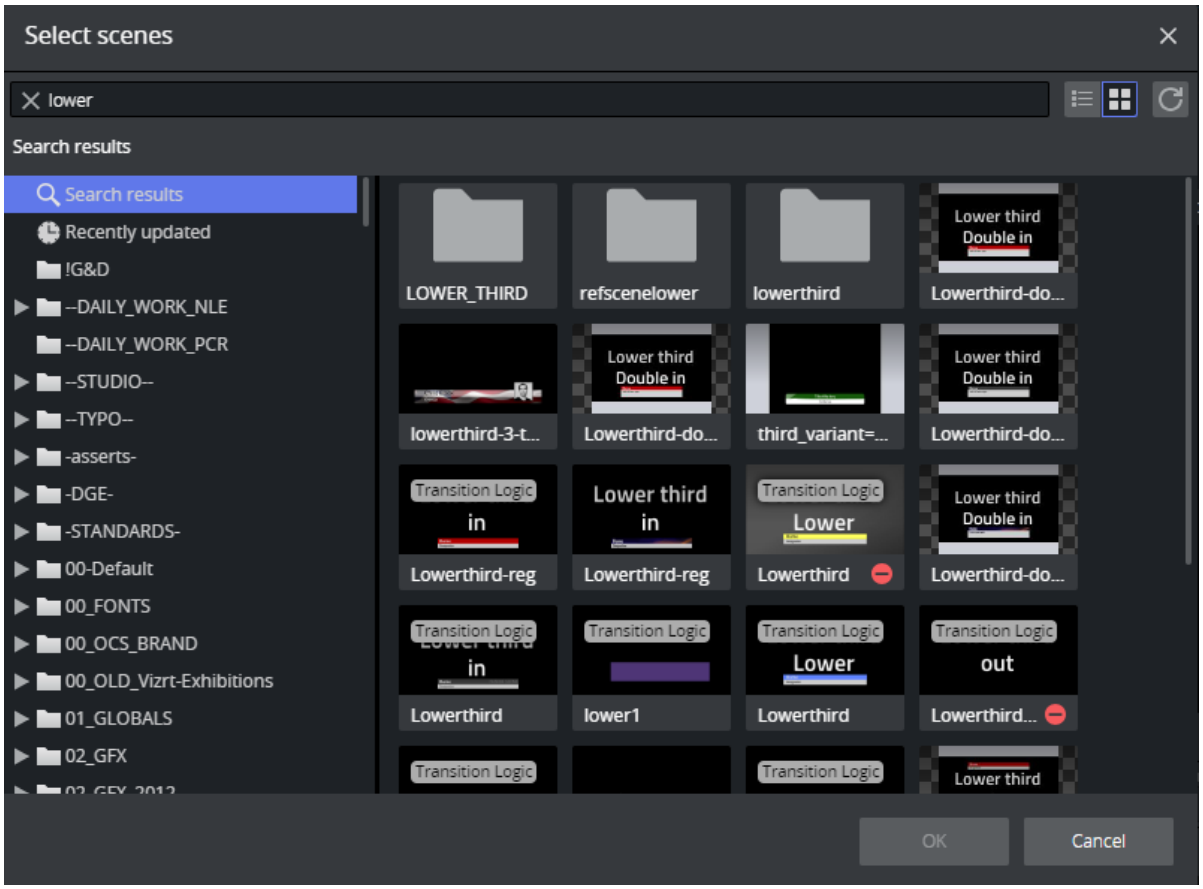
See Concept Manager to learn how to create a new concept.

Once the concept has been created in the system, you can proceed to create templates.

3.2.1 Creating a Template

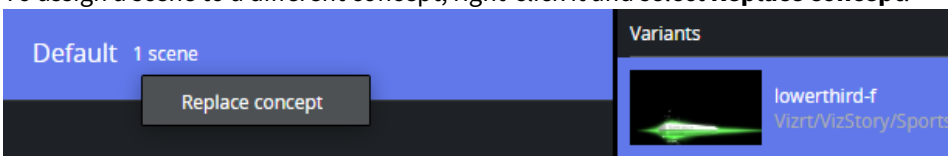
- Click Create new in the upper right corner of Template Builder.
- The Scene Manager window opens.
- Click **+Add Scene**.

The **Scene Browser** appears, displaying all scenes stored in Graphic Hub to which you are connected:



Note: The Graphic Hub containing your scenes is specified through the *graphic_hub_url* setting in Pilot Data Server.

- Enter a search term or browse the folder structure. Once you have selected the correct scene or scenes, press **OK** to add them to the template.
- To assign a scene to a different concept, right-click it and select **Replace concept**:



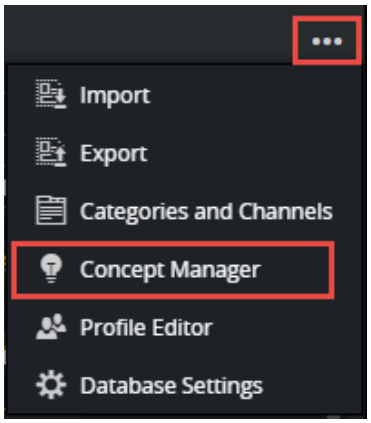
- Select an existing concept and click **OK**.
- Click **Apply** to return to the template with its auto-generated fields.

3.2.2 Opening a Template

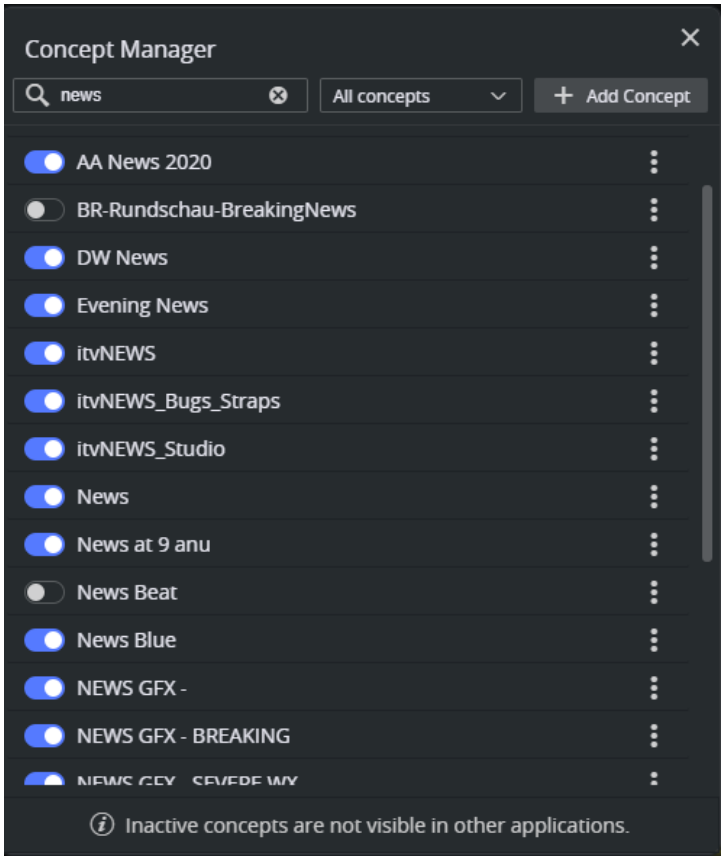
- Click the **Home**  button to show templates available within the Viz Pilot system.
- Use **Concepts** and **Tags** to filter templates. You can also narrow the search by entering the template name in the **Type to search...** field at the top of the dialog.
- Double-click a template to open it.

3.3 Concept Manager

The Concept Manager is accessible by clicking the **Tools** button on the toolbar and selecting **Concept Manager**:



The Concept Manager allows you to add and delete concepts, and mark concepts as active or inactive (draft):



Activating and deactivating concepts is done by clicking the toggle button to the left of the concept name. **Inactive concepts** are not visible in Viz Pilot Edge, and the templates of this concept are also hidden from the Viz Pilot Edge user. Setting a concept as inactive allows it to remain in draft mode for Template Builder users. Once activated, it is visible to Viz Pilot Edge newsroom users.

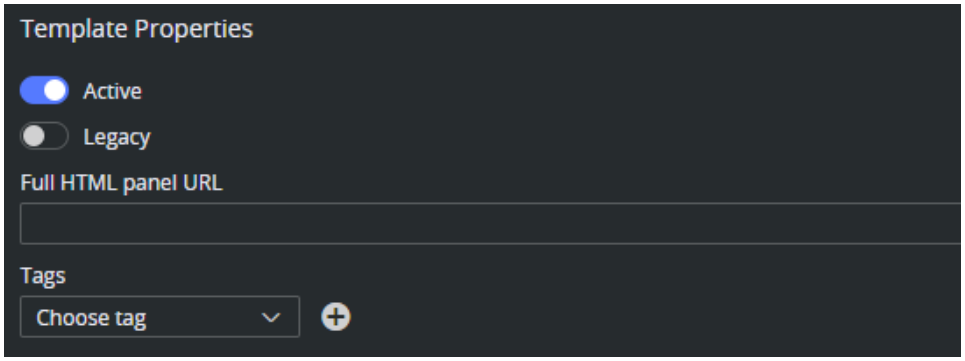
Adding concepts can be done by clicking the **Add Concept** button and entering a name for the new concept. Concept names must be unique.

Deleting concepts can only be done if the concept has no templates. Click the 3 vertical dots to the right of the concept name and select **Delete**.

3.4 Template Properties

3.4.1 Template Properties

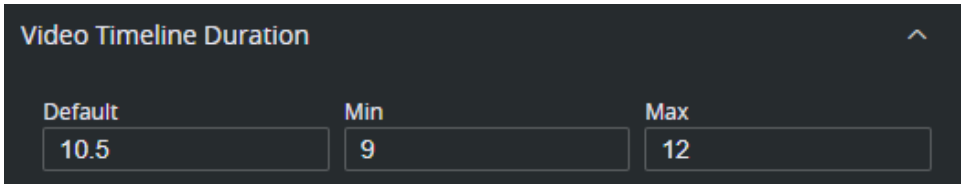
This section controls the most important properties of a template.



- **Active:** A template marked as **Active** is visible in Viz Pilot Edge, otherwise it is not. This setting can be used to treat a template as a draft until it is ready for use in Viz Pilot Edge, or to hide unused templates.
- **Legacy:** Determines how the template behaves in Viz Pilot Director:
    - **Enabled:** In Viz Pilot Director. Legacy templates open in the built-in template window, but with no UI. See below.
    - **Disabled** (default): In Viz Pilot Director 8.6 or later, the template opens in Viz Pilot Edge.
- **Full HTML panel URL:** URL to a custom HTML page that replaces the UI generated in Template Builder. If you add a URL to a custom HTML page, the payloadhosting JavaScript API must be used to handle the fields and data in the template.
- **Tags:** View and edit the **Tags** of the template. These tags can be used to classify and group templates.

For more information about the Legacy setting, see Template Compatibility.

3.4.2 Video Timeline Duration



The **Video Timeline Duration** section allows you to control timing properties for data elements based on this template when they are added to a video timeline as overlay graphics. Typically, these properties can be left blank, as the default duration for overlay graphics is automatically retrieved from the scene when the template is added to

the video timeline. However, in certain cases, it may be useful to override the default duration and define minimum and maximum duration values.

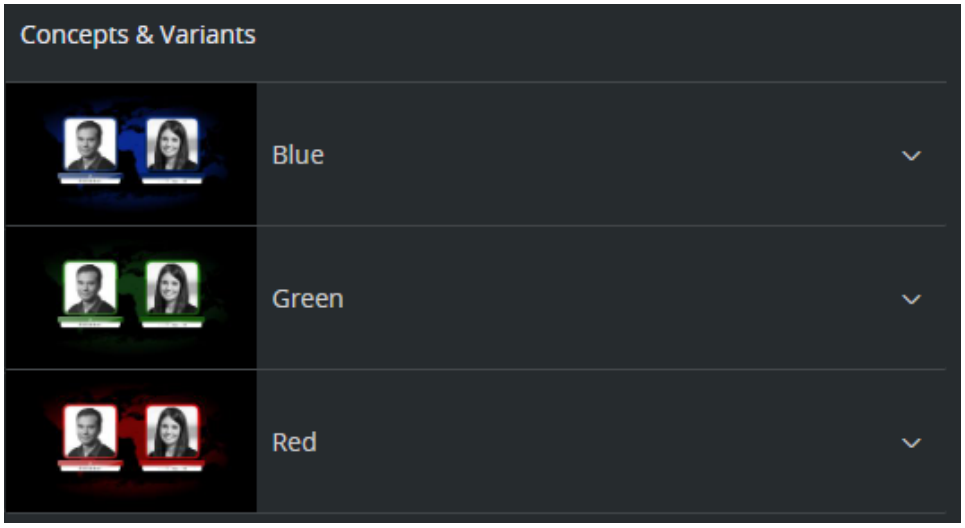
All durations are expressed in **seconds**.

3.4.3 Update Service

Update Service can be used to ensure the latest data is filled in automatically when the data element is taken on air. See Update Service for more information.

3.4.4 Concepts & Variants

The template's concepts are listed on the left side of Template Properties.



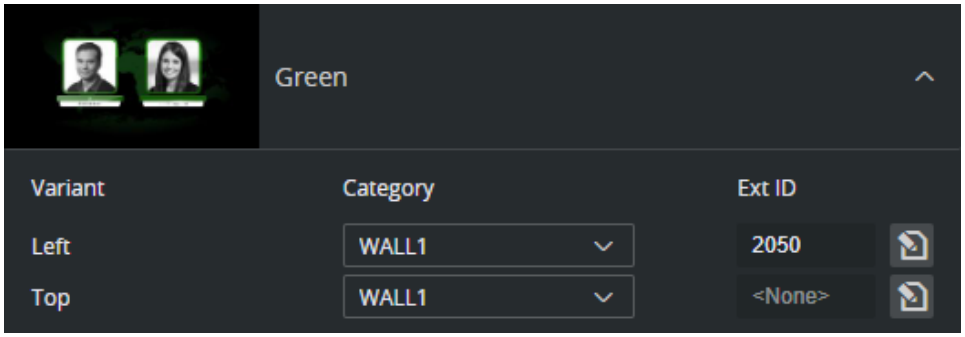
When hovering over the thumbnails, the following functionalities are possible:

- **Upload a custom thumbnail :** Select a file from the local disk. The image is resized to 320 x 180 pixels and stored in the database.
- **Revert to- or refresh the default thumbnail :** If reverting to the default thumbnail, the custom thumbnail is discarded. If refreshing the default thumbnail, a thumbnail snapshot is generated from Preview Server. This thumbnail is not saved back to the database. The default thumbnail is generated automatically by Pilot Data Server.

> **Note:** After saving the template, it may take some time until the default thumbnails for all concepts are refreshed in the browser, as Pilot Data Server does this in a background operation. Additionally, caching in the browser can keep the old thumbnails for a while.

Variant Properties

Expanding each concept reveals the variants for the expanded concept.



Warning: To ensure the template behaves correctly in Viz Pilot Edge and during playout, it is strongly recommended that all concepts contain the full set of variants, and that variants are based on scenes with the same set control fields. This makes it easy to crate templates with a coherent input UI for the user.

The following properties can be set per variant:

- **Category:** A category must be configured to use a specific playout channel. See Categories and Channels for more information.
- **External Id:** When a template has no unsaved changes, each variant can be assigned to a numeric number. Click the **Edit** button to open the **External ID** dialog box, where you can assign a new, unique number to this specific variant and add a description. The External ID can be used by third-party integrators to refer to a template with the specified concept and variant selected, for example, to create data elements based on this template.

3.5 Template Compatibility

In general, it is recommended to create a Viz Pilot Edge based workflow, instead of mixing template compatibility between Viz Pilot and Viz Pilot Edge. Viz Pilot Edge includes features that are not available in Viz Pilot (for example, multiple tags on templates). Additionally, some features in Viz Pilot Edge break backward compatibility, for example, using auxiliary data through scripting. However, it should generally be possible to transition smoothly from Viz Pilot to Viz Pilot Edge by following the workflows described below.

3.5.1 Mixed Workflow

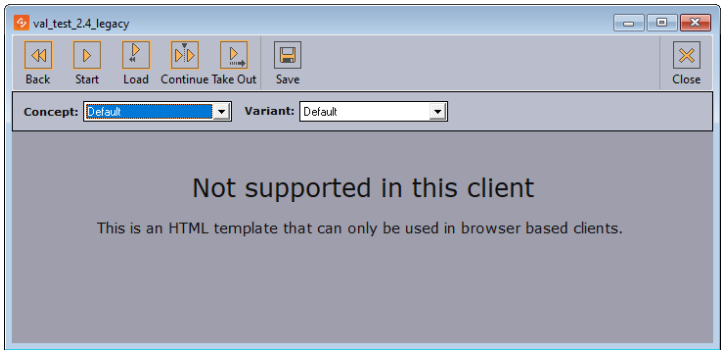
To use a template in both Template Wizard and Template Builder, the template must be created in Template Wizard. It behaves as a regular template with a built in old-style UI in Template Wizard and Viz Pilot Director. The template can also be opened in Template Builder, where an HTML based UI can be added to the template. As a result, the template can have two UI representations, one for classic Viz Pilot, and one for the HTML based Viz Pilot Edge workflow. The template is then automatically marked as **Legacy** in Template Builder.

3.5.2 HTML Based

If a template is created in Template Builder, by default, it is not marked as Legacy. This means the template opens in the Viz Pilot Edge HTML client when accessed from Viz Pilot Director. The template has limited functionality in Director, and can only be used to fill in data and save data elements. Neither playout nor macro commands work on this type of template.

3.5.3 HTML Based Legacy Template

If a template is created in Template Builder, and marked as **Legacy**, the template can be opened in Director in the built in window, but with no auto generated form for the graphics fields and with limited support. The template has an auto generated dummy form that does not contain any of the fields of the scene, and it cannot be saved. The macro commands and playout, work.



Info: It is possible to open this template in Template Wizard, remove the script and the labels with the messages, and manually add the fields of the scene. The template then behaves like the mixed workflow described above.

3.6 Categories and Channels

Occasionally, a user might want specific types of graphics to be played out on a designated output channel. For example, lower thirds on one channel and full screens on another. By adding a specific category to a variant of a template, the category's channel is set as the default output channel when the template's data elements are added to a playlist.

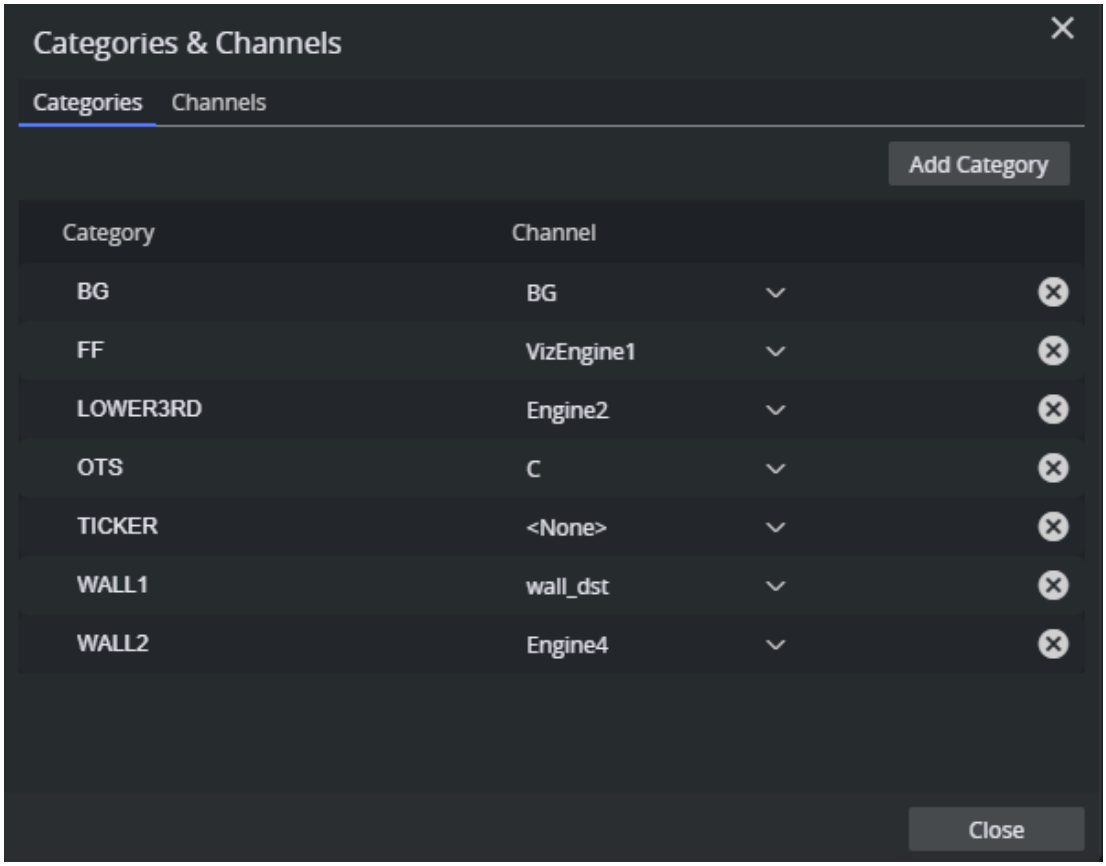
Categories are abstract terms without any specific meaning. If two Viz Pilot backends contain the same categories, they can be mapped to different channels, matching the channels of the Media Sequencer's playout profile in the respective location.

For example, two network stations import a template with the category set to **FULLSCREEN**. The first station maps FULLSCREEN to channel **vizengine01**, while the second station maps the FULLSCREEN category to channel **station-viz2**. When adding data elements based on this template to a playlist, the mapped channel is added to the data element and the graphic is played out on the correct viz engine.

> **Note:** The channel name must exist in the Media Sequencer's active profile, when playing the graphics out.

The same channel can be mapped to multiple categories, but one category can only be mapped to a single channel.

To open the Categories and Channels dialog, click the ••• button on the toolbar and select **Categories & Channels**.



In the **Categories** section, the list on the left displays all categories in the Viz Pilot database, while the Channel list on the right shows the channel each category is mapped to. Select the desired channel for each category with the drop-down.

To apply a category to a variant of a template, open the template and go to **Template Properties**.

3.7 Pilot Collections

Pilot Collections allow you to restrict the templates available to Viz Pilot Edge users. A Pilot Collection can include a list of concepts and tags that define which templates a journalist can access. Only the templates belonging to the specified concepts, and optionally containing the assigned tags, are accessible to the user when a particular Pilot Collection is selected in the interface.

If authentication is enabled, each logged-in user can be restricted to a specific number of collections. This limitation is managed by assigning special roles to the user in the identity provider.

For example, journalists in the main office working with news, might be restricted to using templates from the "News" and "Sports" concepts, which include the tags "bugs," "lower thirds," and "fullscreens." When, for example, the user "John" logs into Viz Pilot Edge, his authorized collection is automatically selected, limiting his access to the two assigned concepts and their appropriately tagged templates.

3.7.1 Organizing Concepts, Templates and Tags

The simplest way to use Pilot Collections to organize templates, is to align the concepts in the Pilot system with the specific needs of user groups. For example, if a group of journalists primarily works on a limited set of TV shows, it may be logical to create a concept for each show and assign the relevant templates to those concepts. If templates are reused across multiple concepts, this poses no issue, these templates appear in all applicable concepts and open in the context of the selected concept. This ensures journalists are guided directly to the appropriate graphics for the show they are working on.

Once a practical set of concepts is established, they can be added to Pilot Collections. Each user can then be authorized to access one or more of these collections.

Using Tags to Restrict Access

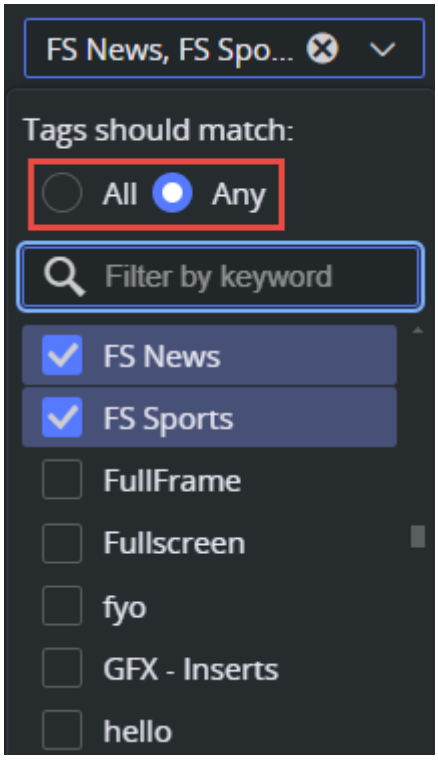
If concepts alone are insufficient to organize templates and finer control over template accessibility is required, **tags** can be used to provide additional granularity. For instance, consider a scenario where two groups of journalists work on the same TV show: one group focuses on editing videos in an NLE workflow, Viz while the other prepares live studio graphics. While creating separate concepts for each group is an option, it can quickly lead to an unmanageable number of concepts. Instead, templates within the shared concept can be tagged with appropriate identifiers, and these tags can be added to Pilot Collections.

For example, within the "News" concept, templates might be tagged as "NLE." A specific Pilot Collection could then limit a user to accessing only templates from the "News" concept that include the "NLE" tag. By marking this tag as mandatory in the Pilot Collection, the user is restricted to templates with this combination, and they cannot deselect the tag or switch concepts.

Using Tags to Organize Templates

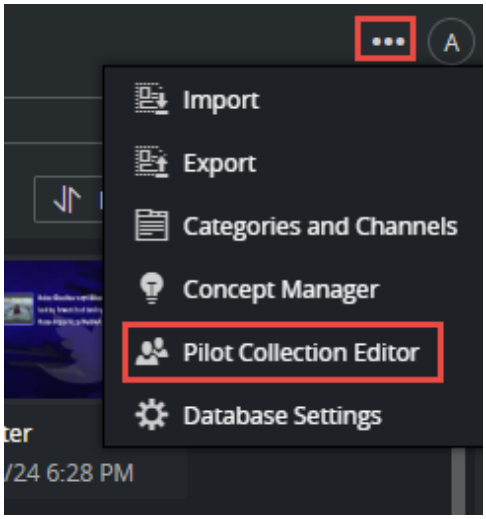
Alternatively, tags can complement concepts without imposing additional restrictions. In this case, concepts define the primary organizational structure, while tags serve to help users find the right template type. For instance, templates can be systematically tagged with labels like "FULL," "L3," "LOGO," and "BUGS." These tags can be included in Pilot Collections to guide users to the appropriate templates without limiting their overall access.

In the Viz Pilot Edge template search interface, users can use the **Any** tag operator to search by tag. By checking a tag in the dropdown, they can quickly filter and view templates that match their selection.



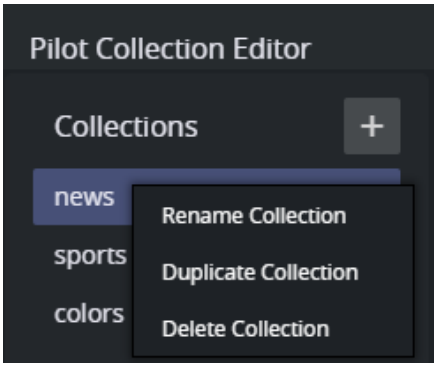
3.7.2 Creating Pilot Collections

To create a new Pilot Collection, open the **Pilot Collections Editor** from the Tools menu on the Template Builder toolbar:



In the Pilot Collection Editor:

1. Click the **plus button** to add a new Pilot Collection.
2. Right-click any existing collection to delete, duplicate, or rename it.

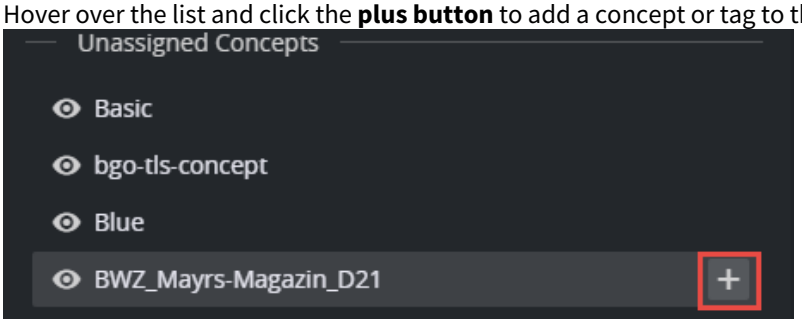


Note: It is important to assign each Pilot Collection a meaningful and persistent name, as this name may be used to match roles within the authentication system.

Adding a Concept or Tag

Once a new Pilot Collection has been created, you can add concepts and/or tags to the collection:

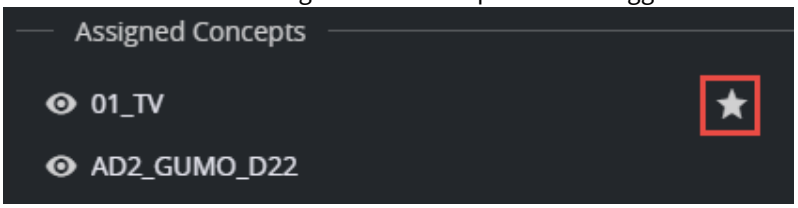
1. Search for the desired concepts or tags in the list of **Unassigned Concepts** or **Unassigned Tags**.
2. Hover over the list and click the **plus button** to add a concept or tag to the Pilot Collection.



Default Concepts

If a Pilot Collection contains more than one concept, you can designate one of them as the default selection for the Viz Pilot Edge user. For instance, this could be useful if one concept contains templates for a daily show, while the other concepts are used less frequently, such as for special events.

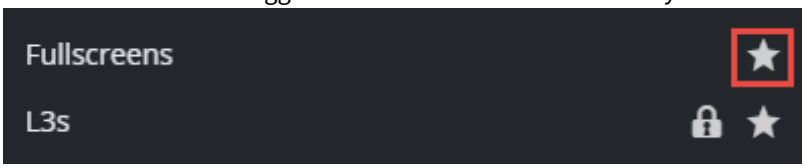
- Click the **star icon** to the right of the concept name to toggle which concept is set as the default:



Default Tags

In the Viz Pilot Edge UI, users can check multiple tags during a template search and choose whether to use the **OR** ("Any Tag") or **AND** ("All Tags") operator. Pilot Collections allow you to define which tags should be selected by default for users in the Viz Pilot Edge UI.

- Click the **star icon** to toggle whether it should be selected by default for the user:



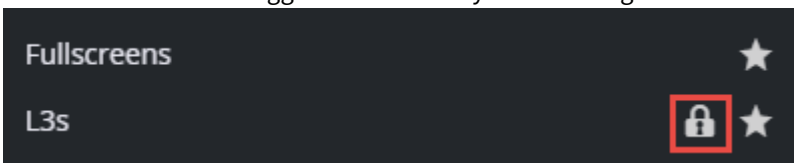
These default settings are intended to guide the user. However, the user can freely check or uncheck tags in the search interface.

Mandatory Tags

If tags are used to restrict the user's accessible template collection, you can mark one or more tags as mandatory.

These tags are always selected for the user and cannot be deselected. Enforce the **AND** ("All Tags") operator, meaning that templates must include all mandatory tags, in addition to any optional tags checked by the user in the search interface.

- Click the **lock icon** to toggle the mandatory state of a tag:



3.7.3 Connecting Pilot Collections to Authentication

In an open system, all journalists have access to all collections in the UI and can choose to view all available content. While this approach can still guide users toward the appropriate templates for their show, a more restrictive and controlled setup can be achieved by connecting Pilot Collections to the Authentication system. In this configuration, a user's profile or group determines which collections they can access, based on role-to-collection name matching.

Role-Based Access Control

These roles already define the user's access level and determine which application they can access:

- pilot-journalist
- pilot-editor
- graphic-designer
- pilot-admin

Access to Pilot Collections is controlled by assigning additional roles to users within the Authentication system. Roles prefixed with **pilot-collection-** are used to match authenticated users to specific collections. For example:

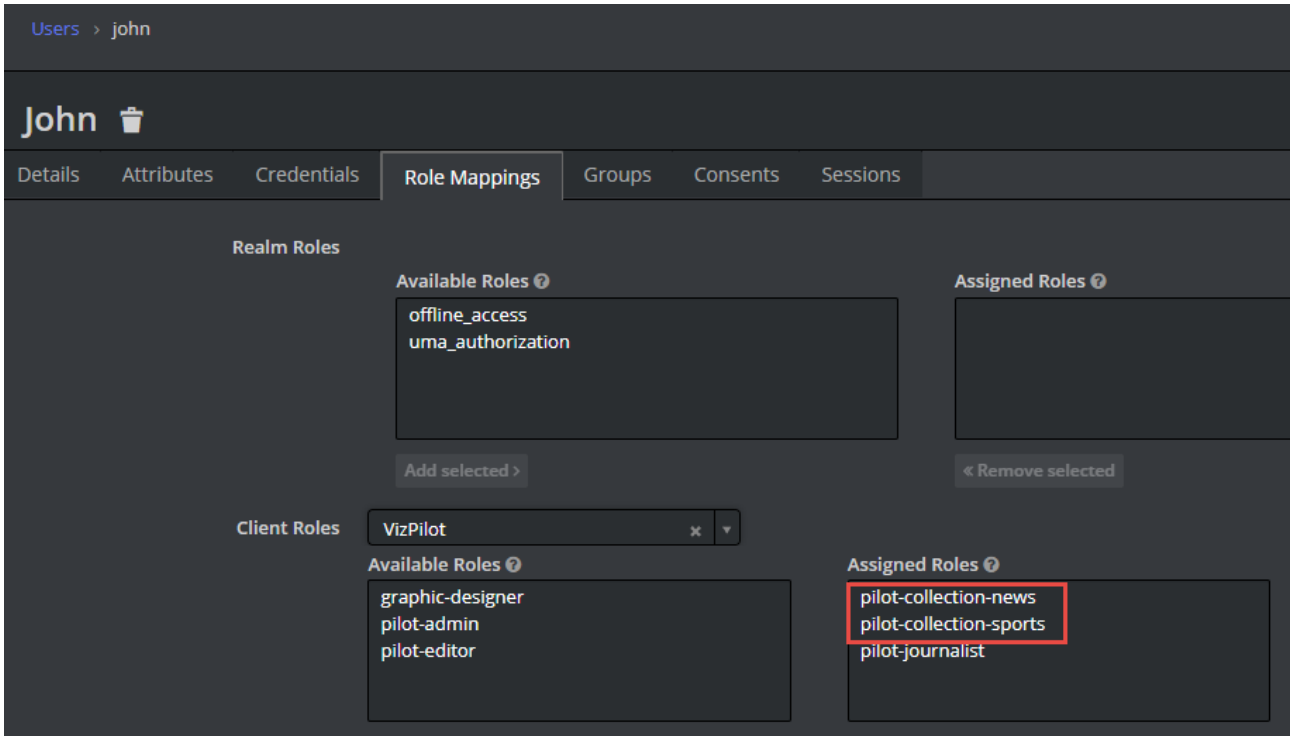
- A user with the roles “pilot-journalist” and “pilot-collection-news”, only has access to the Pilot Collection named **news.**
- Users can have multiple roles, allowing them to access multiple Pilot Collections.

Example Use Case

If a journalist has the following roles in the authentication system:

- pilot-journalist
- pilot-collection-news
- pilot-collection-sports

This user is only able to access Viz Pilot Edge, and both the **news** and **sports** collections are available in the UI, based on these assigned roles.



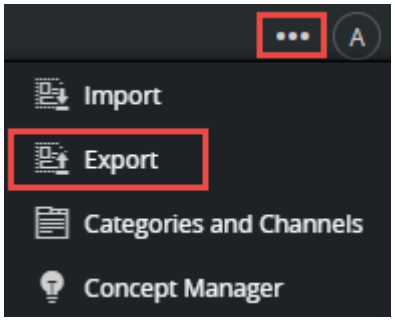
3.8 Import and Export

It is possible to export templates and data elements from the Pilot system and save it to a file. This file can be used to import the content into another Pilot system. The file format is JSON, and this format is not compatible with the XML files exported from Viz Director or Template Wizard.

> **Note:** The Viz Engine scenes are not exported with the templates. They need to be exported explicitly from Viz Artist or Graphic Hub Manager.

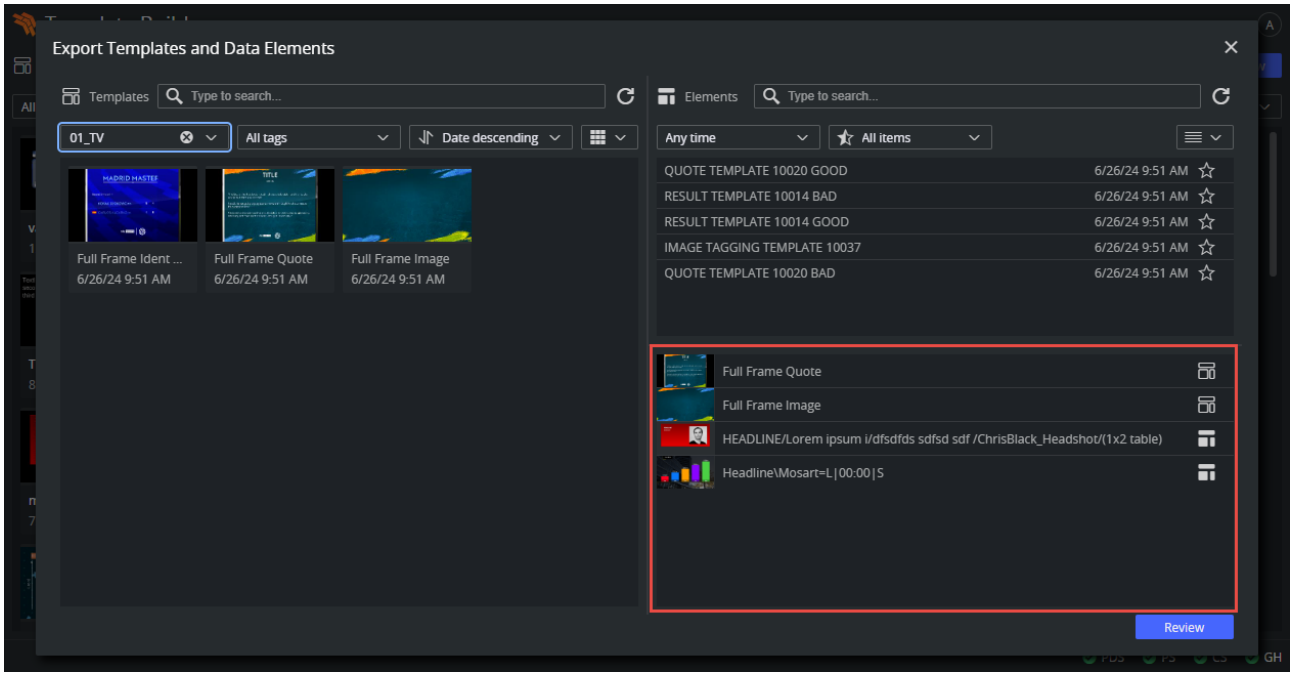
3.8.1 Exporting

Select **Export** from the tools menu in Template Builder:



Step 1 - The Export Window

The export window contains the same user interface as in Template Builder, with the lower right panel as the export panel.



To export data elements and templates:

- Drag templates or data elements into the lower right panel.
- Double click items in the export panel to remove them.
- Click **Review** to proceed.

> **Note:** When exporting a data element, the template and the concept are automatically included in the exported file. If the template has an external ID or a defined Category, these are also exported implicitly.

Step 2 - Review

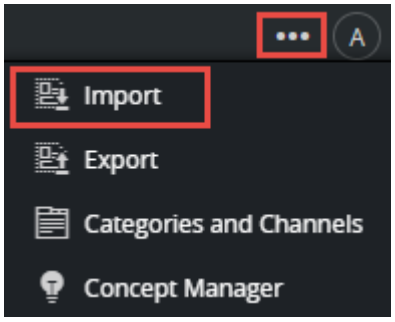
In the review window, a summary of the export is shown, including the data elements, concepts and templates that belong to the exported content.

1. Enter a **file name** for the exported content. The file extension is *.json* and the file is downloaded to the download folder of the browser.
2. Optionally, enter a **description** of the exported content. This is useful to include a change log or similar.
3. Finally click **Export** to download the exported content.

3.8.2 Importing

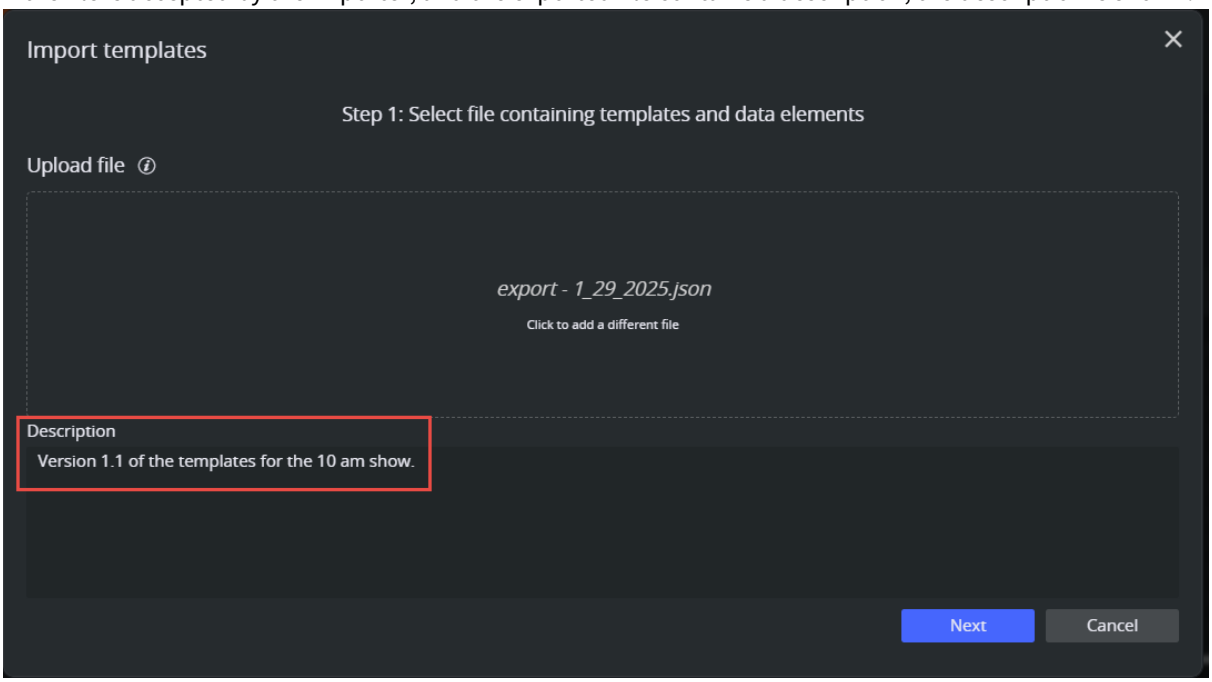
The importer lets you import content from a previously exported JSON file. Note that this format is not compatible with exports from Viz Director or Template Wizard.

Select **Import** from the tools menu in Template Builder:



Step 1 - Upload File

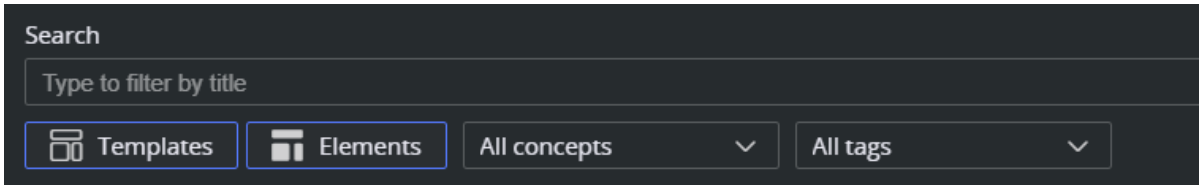
- Drag a file to the import panel or click to browse for a file.
- If the file is accepted by the importer, and the exported file contains a description, the description is shown.



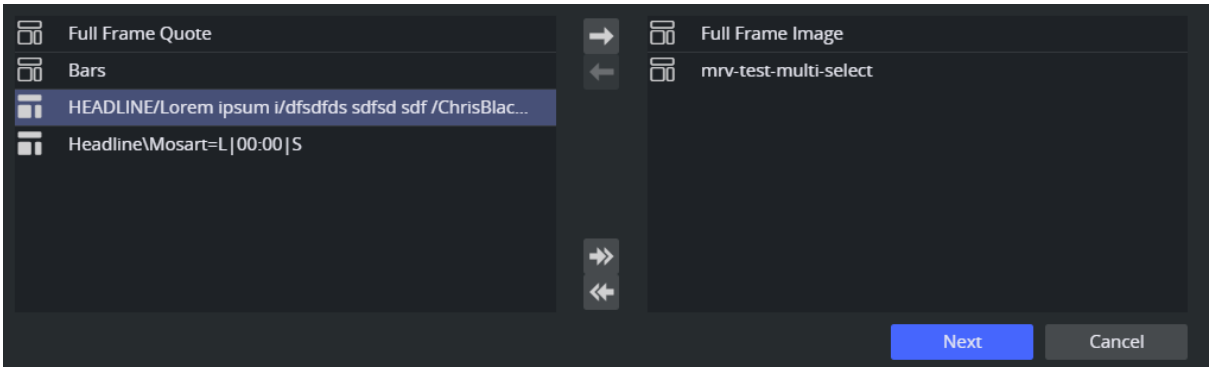
- Click **Next** to proceed.

Step 2 - Select Items

On this step, you can see the list of templates and data elements in the archive, where you can select the items to import. Filter and search the list of items in the archive by using the filter options on the toolbar above the lists:



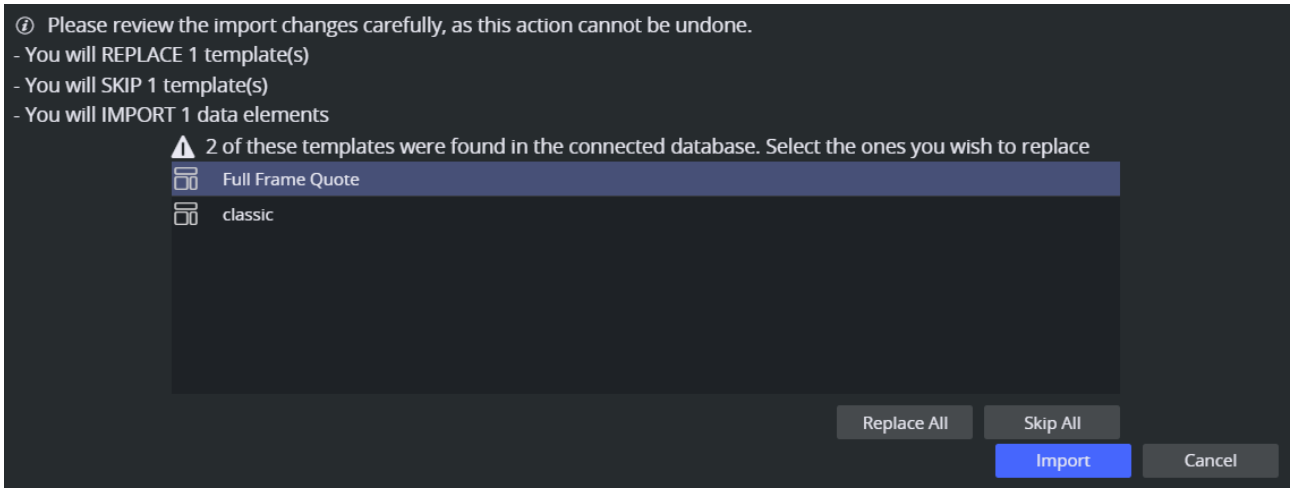
- Select single or multiple items from the list and add them to the list of items being imported:



- Click **Next** to proceed.

Step 3 - Review

The last window does a check of all the items and displays a summary of the items being imported:



Note:

1. Importing a data element also explicitly imports the template of the data element.
2. If the template being imported already exists in the database, you can choose to skip or overwrite the existing one.
3. Data elements are always imported as new, and never overwritten.

4 Working with Templates

For simple use cases, the auto-generated template is often usable out-of-the-box after importing scene(s). Though, in a professional newsroom workflow, advanced customization is needed. There are numerous of ways a template can be customized in Template Builder:

- Template Layout
- Template Fields
- Custom HTML Templates
- Environment Variables
- Custom Execution Logic
- Update Service
- Spell Check
- Testing Templates (Run/Design Mode)

4.1 Template Layout

Editing a template's layout makes it easy to create fill-in forms for journalist. With drag & drop functionality, creating new fill-in forms is quick and easy.

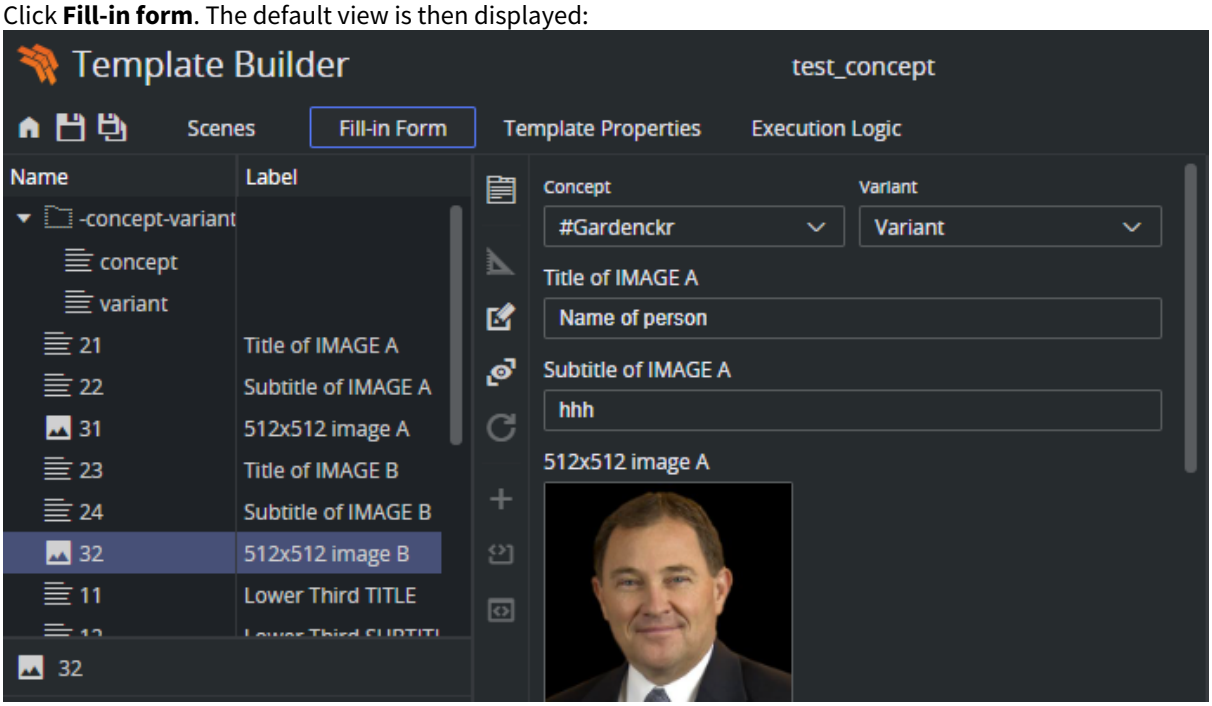
- When a template is created, only the **auto-generated form** is displayed. The layout of this form cannot be modified.
- Adding new tabs enables you to quickly create additional fill-in forms based on selected fields.
- When adding new tabs, the default auto-generated form is accessible in the **All** tab. The auto-generated tab can be hidden from the Pilot Edge user.
- In the additional tabs, it is possible to resize, move, edit, add and delete fields quickly.

This section covers the following topics:

Follow the steps below to get started.

4.1.1 Creating a Template

1. Open or create a new template and add a scene.
2. Click **Fill-in form**. The default view is then displayed:



3. The **toolbar** in the middle has the following functions for the Fill-in-form:
   Add a new Tab (fill-in form). All tabs are visible in Viz Pilot Edge.
   Enter layout edit mode. Only enabled in additional tabs, not enabled for the default auto-generated **All**

form. Click this to be able to move and resize fields.

Allow temporary editing of read-only fields. This is only when working with the template in Template Builder, it is not stored anywhere.

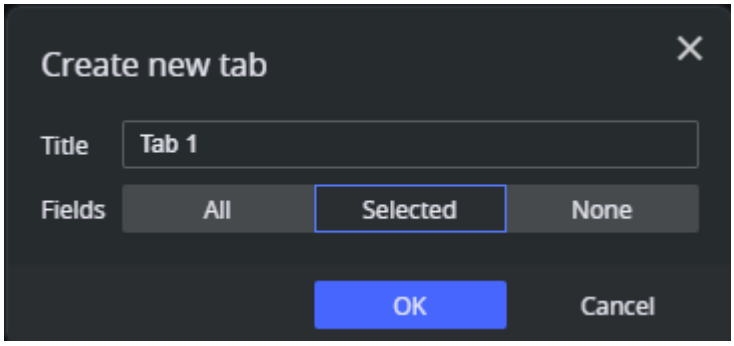
Reveal hidden fields. In the auto generated **All** mode, some system fields are also revealed (for instance the title, the auto generated title, the resolved concept and variant).

Refresh HTML panel(s) in the template or the full HTML panel if the template is represented by a custom HTML panel.

4.1.2 Adding Alternative Layout Forms

To create an additional template representation, click the **Create New Tab** button :

Enter a new title for your new tab and click **OK**:

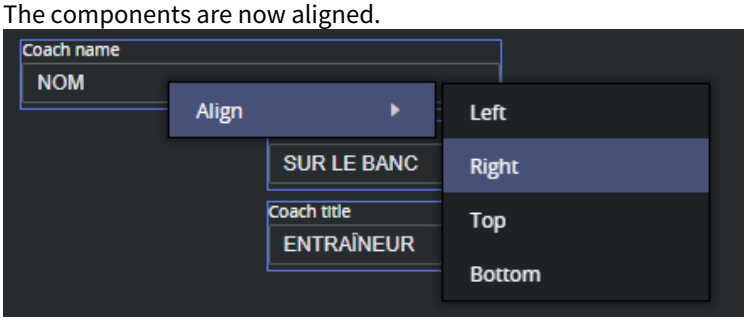


There are 3 ways of adding fields to the new form by selecting one of the following options:

1. The **All** option in the dialog box, then all fields in the auto-generated form are added to the new form.
2. The **Selected** option only includes selected fields in the new form.
3. **None** creates a new empty form.

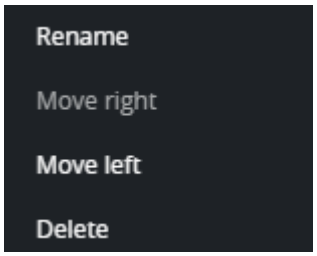
Moving, Resizing and Aligning Fields

1. Click the ruler button to enter edit layout mode
2. Move and resize by grabbing the edges of each field.
3. Align fields by:
    a. Selecting multiple fields and UI components.
    b. Right click and select **Align** > **Left**, **Right**, **Top** or **Bottom**.
    c. The components are now aligned.



Renaming, Deleting and Reordering Tabs

Right click an additional tab (not the auto-generated **All** tab) to reveal the following functionality:

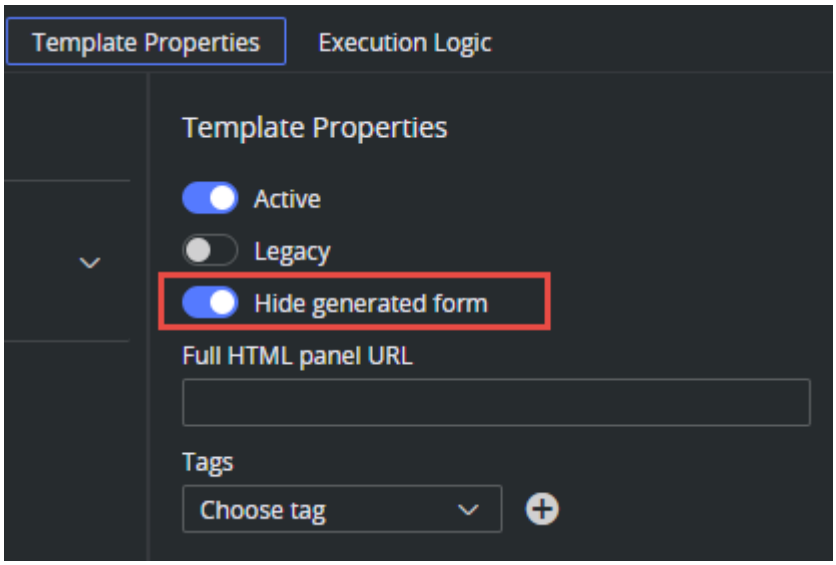


Now you can rename, reorder or delete each tab.

Hiding and Showing the Auto-Generated Tab

You can hide the auto-generated **All** tab from the Viz Pilot Edge user:

- Click **Template Properties**.
- Select **Hide generated form:**



4.2 Template Fields

The left window in Template Builder contains the field tree. The initial fields reflect the exposed Control plugins in the imported scene(s), but it is possible to add fields that are not bound to plugins in the scene. Each field has a type, and numerous properties that alter the behavior of the field.

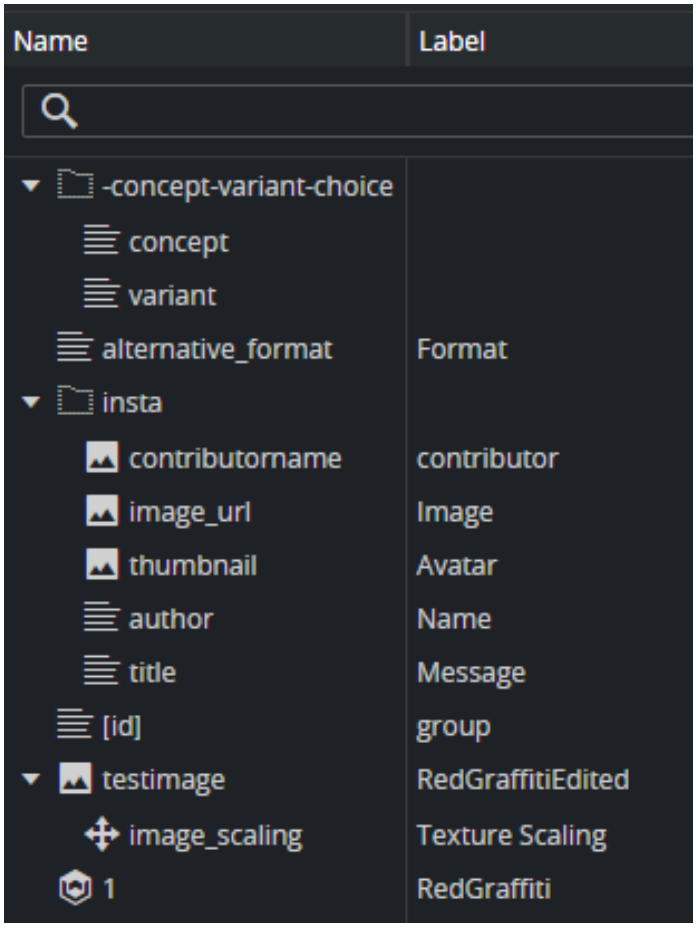
Fields can be restricted: for example, to only include text with a certain amount of characters, numbers within a specific range, or media placeholders for media assets, or be displayed as options in a drop-down list.

The fields can be manipulated in various ways to decide how data is entered into the field. See Data Entry.

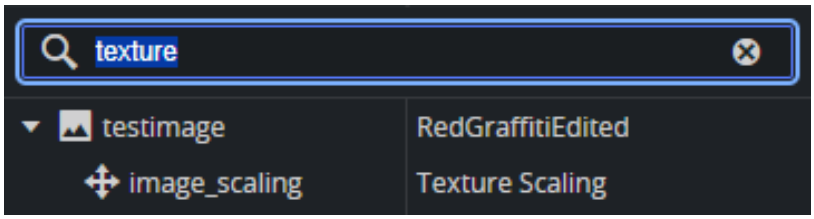
Field representation in the UI can also be replaced with an inline HTML panel.

4.2.1 Field Tree

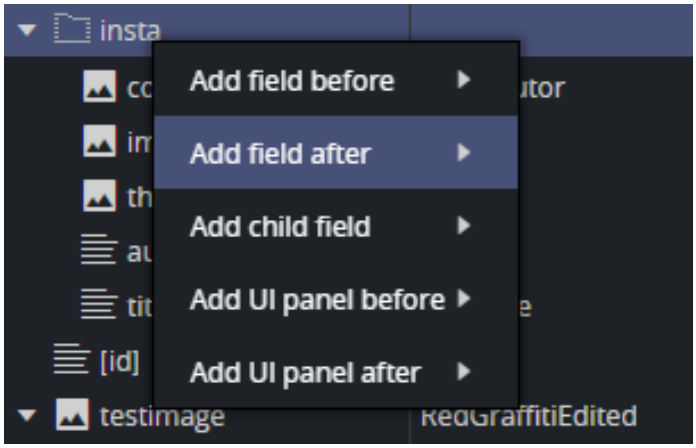
The field tree contains the **Name** (field ID) and **Label**, which are also shown in the Fill-in form. The icon beside each line in the tree indicates the Type of content in the field.



At the top of the field tree there is a filter option. By typing in this box, the field tree is filtered to contain only matching items. The filter searches both in the name (field ID) and the Label columns.



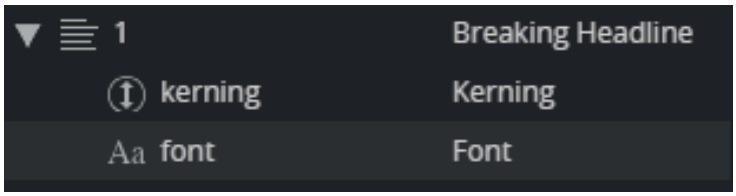
Fields can be rearranged by drag-and-drop within the field tree. Right-click a field to open a menu where additional fields can be added.



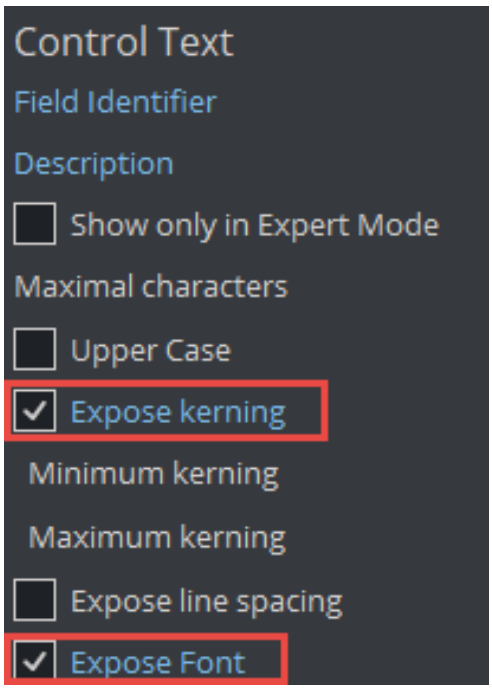
Info: Only fields created in Template Builder can be deleted and given a new Name (field ID). Fields bound to the scene have a fixed name and cannot be deleted, nor can the type be changed.

Sub Fields

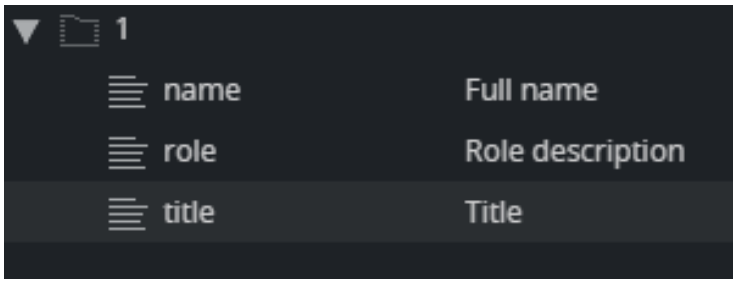
In the Field tree we can find the main fields and the sub fields:



In this case, sub fields are Control Plugin properties of the main field. In Viz Artist, the scene designer has chosen to expose kerning and font for the main text field:

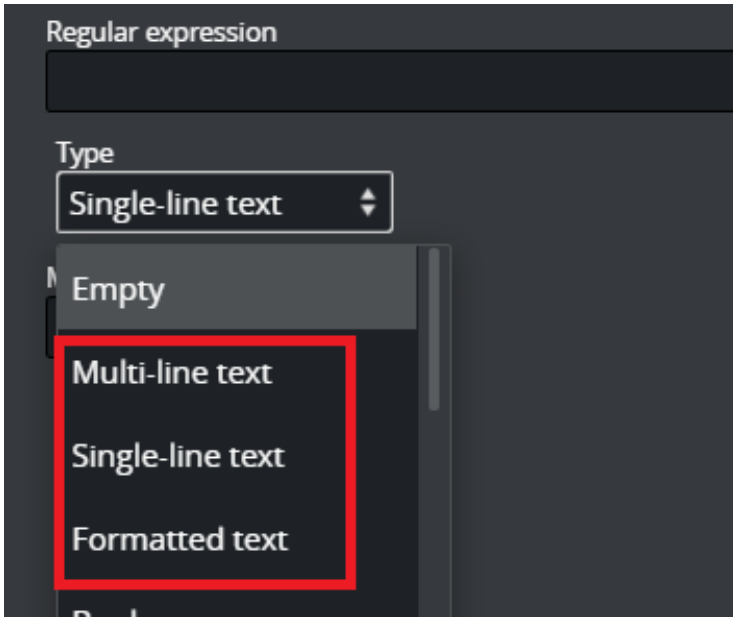


Sub-fields can also be used in Viz Artist and Template Builder to group fields. This is important when using Feed Browser functionality to bind multiple child fields to the parent field:



In Viz Artist, the three Control Plugins above are named **1.name**, **1.role** and **1.title** in the Viz Artist scene. When using the dot naming notation in Viz Artist, Graphic Hub and Template Builder group these fields as parent/children.

Text Fields



- **Multi-line text:** Multi-line text supports standard ASCII characters. It does not support any type of text formatting and does not convert any text. It keeps the text as it was typed.
- **Single-line text:** A single-line text field converts any white-space to space. White-space includes space, tab, newline, etc. Single-line text converts any white-space to the space character you get by pressing the spacebar KEY only. For Template Builder, this is also a text field with a single-line entry, unlike multi-line text.
- **Formatted text:** Formatted text refers to the ability to hold formatted text. For example, a formatted text field can show that some of the texts are bold or italic, for example, when a field has Rich-Text functionality.

  Although such a display is not yet completely supported (no Rich-text support yet) on our payload text field, formatted text is used so that if a field has a formatted type text created in Viz Artist, the field type can also be selected in Template Builder.

4.2.2 Field Properties

The **Field Properties** window is located below the Field Tree window. It displays the properties of a selected fields in the Field Tree.

Multi-selection: If several fields are selected in the Field Tree (CTRL + click), a subset of the field properties is displayed. If the selected fields have different field property values, the Field Properties window displays a multiple values state. Changes made in the Field Properties window are immediately applied to all the selected fields.

Note that the set of properties displayed depends on the Type of the field. The following properties are available:

- **Label:** Specifies the label of the field in the Fill-In Form.
- **Tip:** A tooltip text can be entered to provide more information about the field.
- **Read-only:** The field remains visible, but is grayed out in the Fill-In Form.
- **Hidden:** Hides the field in the Fill-In Form.
- **Publishing variable:** Viz Story specific property. Can be used to link the field to a system field affecting the playout or publishing of the template.
- **Regular expression:** Defines constraints for the value in the field, using **Regex**. See the table below for examples.
- **Preview point link:** If set, clicking on/selecting the field, also shows the given preview point in the Preview.

> **Note:** The preview points in a scene must be placed on the **default director**, which is the director called **Default** in the stage of the scene.

- **Type:** The field type. might be changed here.
- **Max length:** Text fields only, see comments in table below.
- **Single-line:** Formatted text fields only, see comments in table below.
- **Data entry:** Set how data is entered into the field. Drop-down list of all field types. For more information, see Data Entry.
- **Read-only** and **Hidden expression:** Basic Javascript eval expression that decides whether a field should be hidden or read only. This can be used instead of scripting to build conditions based on values in other fields. See Hidden and Read-only Expressions for more information.

> **Info:** A Regular Expression (or **Regex**) is a pattern (or filter) that describes a set of strings that match the pattern. In other words, a regex accepts a certain set of strings and rejects the rest.

Regex	Description	Example
[a-zA-Z]+	Match a word containing only letters.	MyLongWord
^[A-Z][a-z\s]*$	Match a string starting with capital letter containing only lowercase letters and space.	This is a sentence
\d+	Match a sequence of digits.	123489

Regex	Description	Example
\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b	Match a typical email address.	joe@microsoft.com

Upload and Paste Images

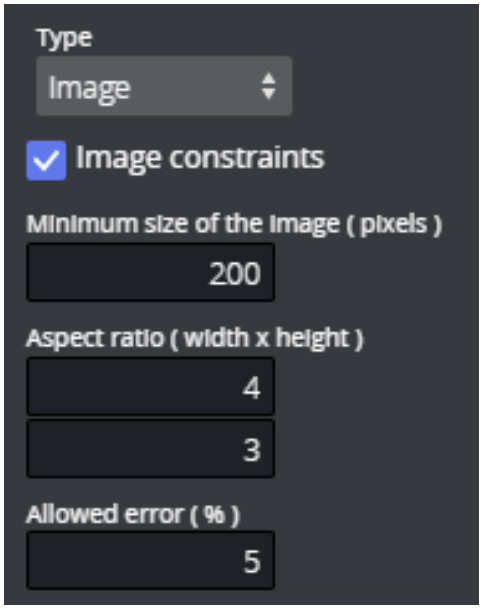
For image fields, users can upload images directly from their local files.

In addition, images can be pasted from the clipboard using **Ctrl + V**. When an image is uploaded or pasted, it is automatically uploaded to Graphic Hub and stored in a folder named *uploads*. This folder is intended for temporary assets and should ideally be cleaned up on a regular basis.

Images uploaded this way do not include metadata and are not indexed or made available through any image provider. They are intended for one-time use in playout only and are not designed for reuse or discovery by other users.

Image Constraints

For fields of type **Image**, it is possible to set image constraints to force the Viz Pilot Edge user to select an image with the correct aspect or a minimum resolution.



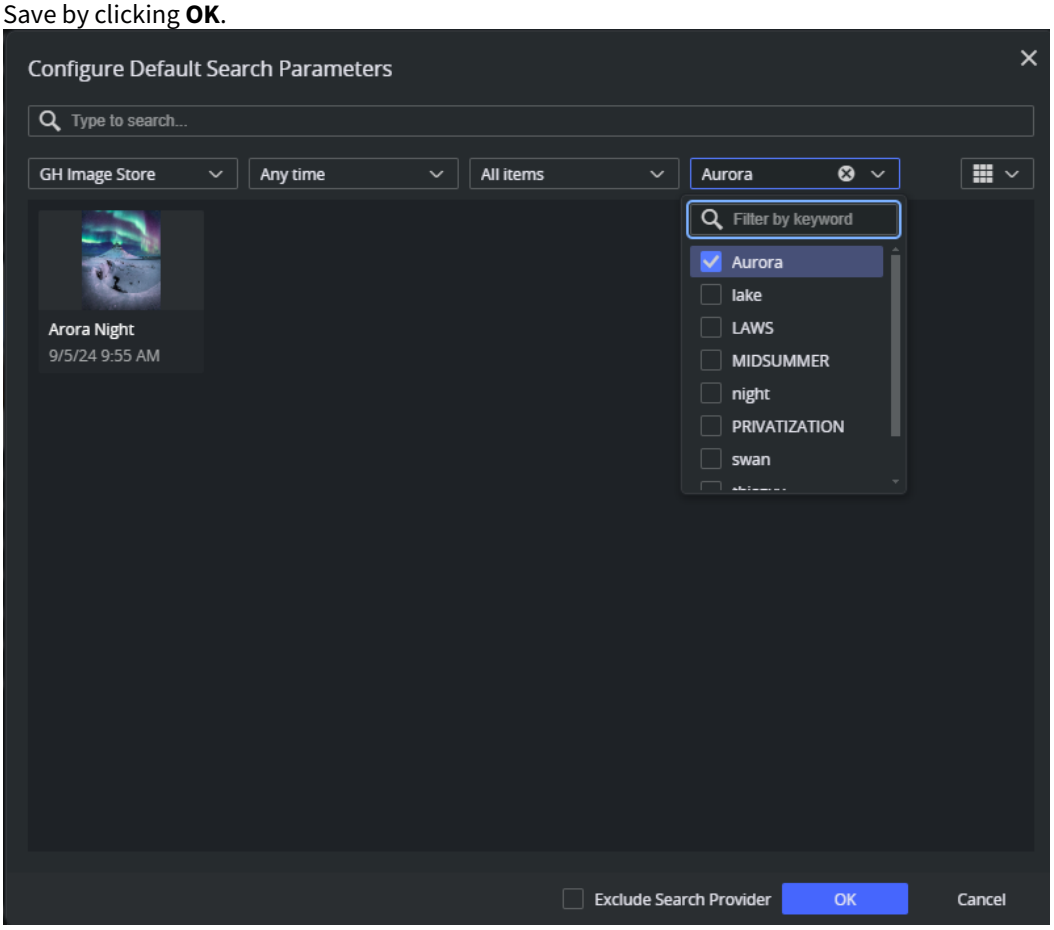
Specify standards for image quality and size, using either or all of the following:

- **Minimum size of the image (pixels):** Number of pixels (width or height), irrespective of aspect ratio.
- **Aspect ratio:** Any number describing the proportional relationship between the image width and height.
- **Allowed error**: Specifies a margin for the constraints when an image is selected.

Default Search Parameters

For the types **Image**, **Video**, **Font**, **Geometry** and **Material**, it is possible to define default search parameters that are used by the media search that is launched when you click on the field:

1. Click the field in the field tree.
2. Click the **Set** button under **Default search parameters** to open a media search window.
3. Select search provider, and/or enter text in the search field, select whether to show all items or to limit by time from the **Show** drop-down list, and/or selecting tags from the **Tags** drop-down.
4. Choose whether the currently selected search provider should be a part of this default search or not. The search provider is included in the default search parameters, meaning that any time the Viz Pilot Edge user clicks the image to search, this search provider is pre-selected. If **Exclude search provider** is checked, the last used search provider is used when the Viz Pilot Edge user clicks an image. This is only useful when the search parameters only contains parameters that are compatible throughout all types of search providers (for example, Any time / Last Month etc).
5. Save by clicking **OK**.



4.2.3 Field Types

The type of content allowed in the field in **Default Values** is set by using the drop-down list under **Type**. Depending on the type selected, different sub-options become available, as specified in the table below.

There are two main field type categories: *scalar* and *list*. Fields of all types apart from the list type are referred to as *scalar fields*. Fields using the list type are referred to as list fields.

The following types are available:

Type	Icon	Media Type (XSD Type)*	Comments
Empty			Makes the field unavailable in the Fill-in form. Typically used as grouping for other fields.
Multi-line text		text/plain (string)	**Max length**: Sets the maximum number of characters allowed in the field.
Single-line text		text/plain (normalizedString)	**Max length**: Sets the maximum number of characters allowed in the field.
Formatted text		application/vnd.vizrt.richtext+xml	**Max length**: Sets the maximum number of characters allowed in the field. **Single-line**: Check this box to specify that the rich-text editor allows one line of text only.
Boolean		text/plain (boolean)	Creates a checkbox that has two states: true and false.
Integer		text/plain (integer)	This field is an integer field. **Minimum**: Sets the minimum value allowed in the field. **Maximum**: Sets the maximum value allowed in the field.
Decimal		text/plain (decimal)	This field allows decimal numbers. **Minimum**: Sets the minimum value allowed in the field. **Maximum**: Sets the maximum value allowed in the field.
Date and time		text/plain (dateTime)	Use the **Date Chooser** in **Default Values** to select date and time in this field.

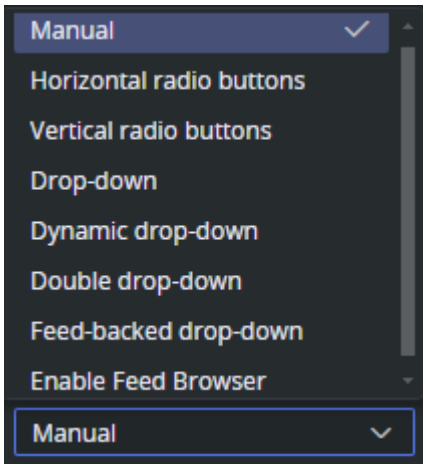
Type	Icon	Media Type (XSD Type)*	Comments
Date		text/plain (date)	Use the **Date Chooser**, or the individual editors for day, month and year in **Default Values**, to select the date in this field.
Two numbers (duplet)		application/vnd.vizrt.duplet	This field allows two numbers (decimal numbers are allowed). **Minimum**: Sets the minimum value allowed for both numbers. **Maximum**: Sets the maximum value allowed for both numbers.
Three numbers (triplet)		application/vnd.vizrt.triplet	This field allows three numbers (decimal numbers are allowed). **Minimum**: Sets the minimum value allowed for all three numbers. **Maximum**: Sets the maximum value allowed for all three numbers.
Image		application/atom+xml;type=entry;media=image	Makes the field available for an image. **Image Constraints**: Enable this option if you want to set constraints on the image. **Minimum Size of the image (pixels)**: Specifies the minimum allowed image dimensions in pixels. Both width and height must be at least this big. **Aspect Ratio (width x height)**: Specifies the aspect ratio of the image. **Allowed Error (%)**: Specifies the maximum stretch limit for both the width and height of the image, in relation to its defined aspect ratio.
Video		application/atom+xml;type=entry;media=video	Makes the field available for a video.
Font		application/vnd.vizrt.viz.font	Makes the field available for a font.
Geometry		application/vnd.vizrt.viz.geom	Makes the field available for a geometry.

Type	Icon	Media Type (XSD Type)*	Comments
Material		application/vnd.vizrt.viz.material	Makes the field available for a material.
Material Definition		application/vnd.vizrt.viz.material_definition	Viz Engine Renderer specific for the new Advanced Material (Phong and PBR).
Map		application/vnd.vizrt.curious.map	Makes the field available to present and edit a map.
List			Lists may be modified by adding and removing columns in the Field Tree. • To add columns to a list - right-click the columns node under the list field node in the Field Tree and select **Add column**. • To remove a column - select the column field in the Field Tree and press **Delete**, or right-click it and select **Delete field**. **Note:** List fields are fundamentally different from scalar fields. It is therefore not possible to change a list type to a scalar type and vice versa. **Minimum number of rows**: Defines the minimum allowed number of rows in the list. **Maximum number of rows**: Defines the maximum allowed number of rows in the list.
Color		text/vnd.vizrt.color	Text (for example: #140E7E or rgba(255, 0, 0, 1)).

* For more information on media types, see: Overview of Media Types.

4.2.4 Data Entry

The Data entry field property specifies how users should fill in field values.



Manual

Selecting **Manual** in the Data entry drop-down list does not give access to any additional settings for the field. The default is for the input to the field to be a text box with manual input.

Radio buttons and Drop-down

Radio buttons (vertical and horizontal) and Drop-down makes it possible to create a list of static options for the user to choose from. See Radio Button and Drop-down.

Double Drop-down

With the Double drop-down it is possible to add a two-level selection, letting you set multiple sub-choices for each primary choice. See Double Drop-down.

Field Linking to a Feed

The following options specifies that the field should get its value from a property of an Atom or RSS feed entry.

- Linking a field value using Feed Browser, see Field Linking with Feed Browser.
- Linking a list field to a feed, see Feed Linking to Tables.
- Using a feed-backed dropdown instead of a Feed Browser, see Feed-backed Drop-down.

Dynamic Drop-down

The **Dynamic drop-down** option allows you to create a dynamic drop-down with items read from the value of another field. Whenever the (hidden) source field is changed, the drop-down items are updated. See Dynamic Drop-down.

Radio Button and Drop-down

Selecting **Drop-down** or one of the **radio button** options, lets you see the content in a static list, which may in some cases make it easier and less error-prone to fill the template with the right content.

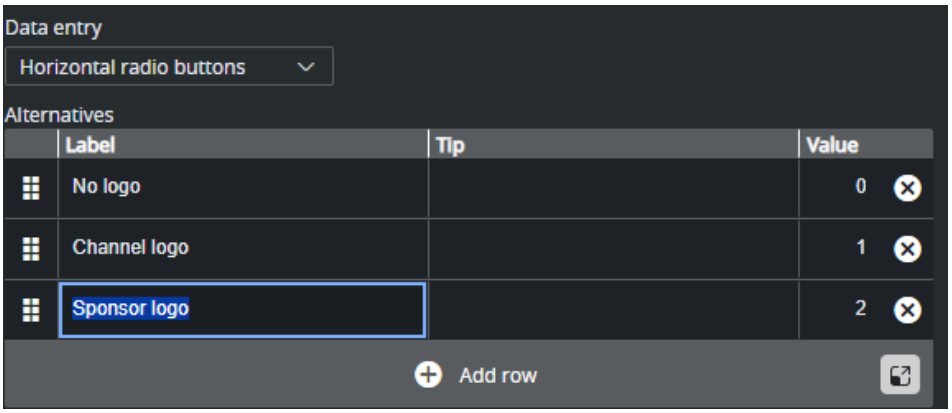
Example: OMO Plugin

When a Control Object moving (Omo) plugin is accessible in the template, scenes using Omo plugins are originally presented as integer values for the different elements in the Fill In Form. The **Drop-down** and **radio buttons** options can assign text to these values to make it easier to select the right element.

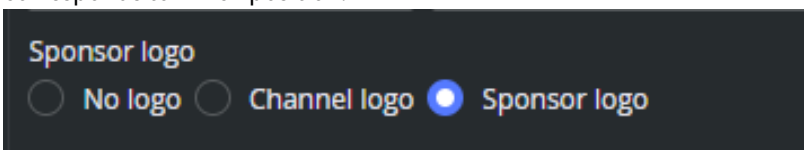
The example below contains a scene with a sponsor logo having different display options in the graphics. For the Omo plugin, these options correspond to the values 0, 1 and 2 respectively.

To assign text to these values:

- Select the **Omo** field in the Field Tree.
- Select **Horizontal radio buttons** in the **Data entry** drop-down list.
- Add alternatives in the inline list editor:



- The **Omo** field in the Fill In Form now contains a drop-down list or radio buttons, containing the alternatives created above as text, as opposed to an integer field where the user had to remember which integer corresponds to which position.



Dynamic Drop-down

The entries of a drop-down can be dynamically populated based on the value from another (hidden) field in the template, or from the temporary $data object that is not stored in the payload.

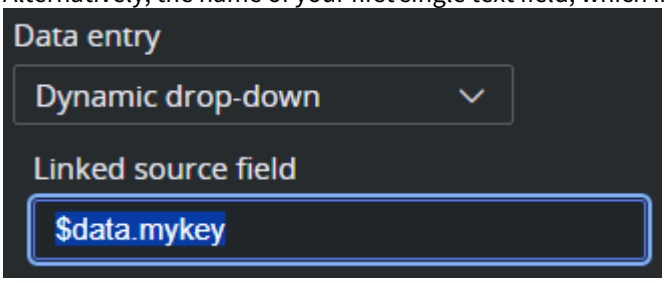
- The drop-down is populated with a JSON string from the source.
- Whenever the source is changed, the drop-down is re-generated.
- Use case: A natural workflow is to use the `onLoad()` event to fetch data and populate the source whenever data elements based on this template is opened in Viz Pilot Edge.

Example - onLoad() Populating a Drop-down

Follow these steps to create a dynamic, data driven drop-down that is populated each time a data element is opened:

> **Note:** In the following example we use the temporary *vizrt.$data* as the source for the drop-down. This object acts like a key-value map where it is possible to add any data as a string, to a key in vizrt.$data, but using a hidden source field in the model is also possible in the example below. The source data is then stored in the payload.

1. If using a field as a source for the drop-down, create a new single-line text field that may be called *source*. This acts as the source of the drop-down item (this field can be hidden).
2. Create a single-line text field, for instance called *drop-down*. This is where your drop-down is displayed.
3. In the drop-down single-line text field, click on it to have the properties displayed.
4. In the **Data entry** property, click on the **Dynamic drop-down** alternative.
5. Once the alternative is added, a new text field right below called **Linked source field** appears.
6. Fill the **Linked source field** with your own key in the custom data object, for instance `$data.mykey`. Alternatively, the name of your first single text field, which in this example is *source*.



7. Within the **source** field, you can add data, either manually through the source field directly, or from the script editor like shown below.
8. The data must be in JSON array format: [{"label": "my label1", "value": "my value1"}, {"label": "my label2", "value": "my value2"}, {"label": "my label3", "value": "my value3"}]
    a. The **label** property is what is displayed in the drop-down.
    b. The **value** property is what is being stored in the data element (this is eventually sent to Viz Engine).

Populate Drop-down Source Field in onLoad() Event

```
// onLoad() is executed each time a data element based on this
// template is loaded in Pilot Edge
vizrt.onLoad = () => {
```

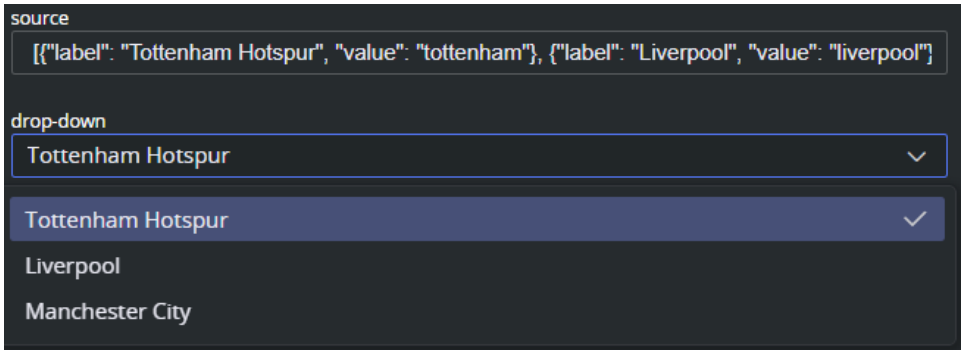
```
    fetch("http://myhost/app/teams.json")
        .then(r => r.text())
        .then(result => {
            vizrt.$data.mykey= result
            console.log("Fetched: ", result)
        }
    ).catch(e => console.log("Error: ",e))
}
```

In the example above, the *teams.json* file looks like this:

```
 [{"label": "Tottenham Hotspur", "value": "tottenham"}, {"label": "Liverpool",
"value": "liverpool"}, {"label": "Manchester City", "value": "mancity"}]
```

The format is a JSON array. This line can be pasted directly into the drop-down source field for testing purposes.

The result looks like this (with the source field visible):

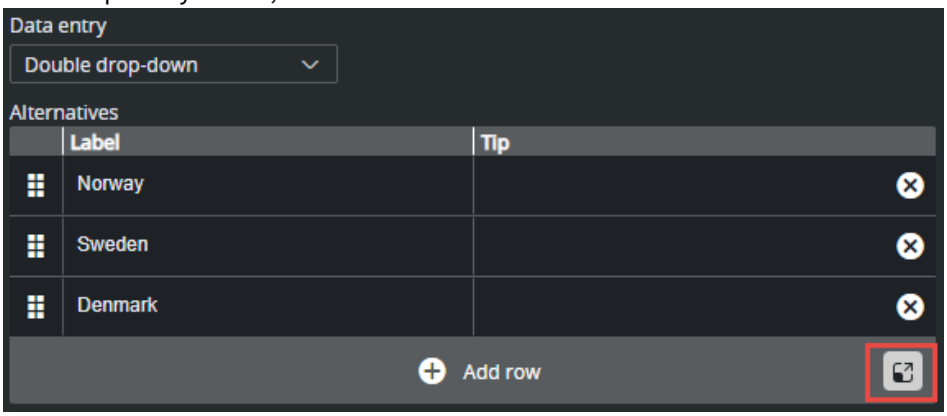


Double Drop-down

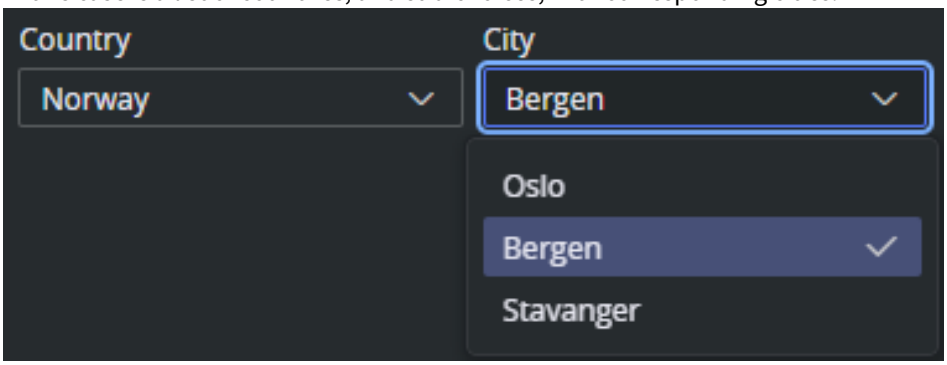
With the **Double drop-down** it is possible to add a two-level selection, letting you set multiple sub-choices for each primary choice.

For example, if the choices list different countries, sub-choices could list cities in each country.

- Mark the desired Field ID in the Field Tree.
- Select **Double drop-down** in the **Data entry** field.
- Fill in the primary choices in the inline table.
- For each primary choice, enter the table editor and add sub choices.



- A new table for sub alternatives appear. Fill the table and click back  when complete.
- Instead of a text field in the Fill In Form, the field now contains two drop-down lists: the main choices, which in this case is a list of countries, and sub-choices, with corresponding cities.



Field Linking with Feed Browser

Using Feed Browser

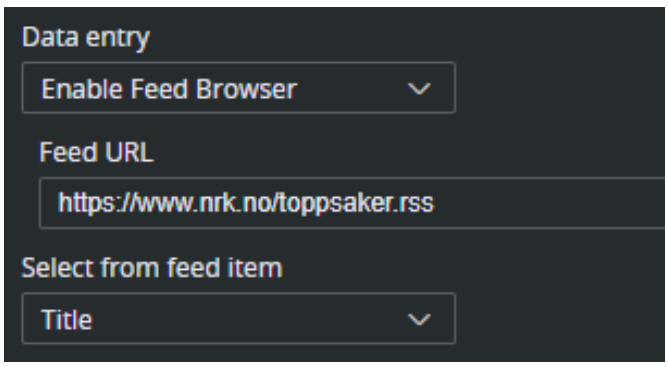
Feed Browser specifies that the field should get its value from a property of an Atom or RSS feed entry. If the field is a sub-field of another field that has enabled feed browser, the option is named *Parent feed browser*. Otherwise, it is named *Enable feed browser*.

- If the **Enable Feed Browser** option is selected, a **Browse** button appears next to the field in the fill-in form.
- Click **Browse** to open the Feed Browser dialog.
- In the Feed Browser, the items of the feed are presented (with thumbnails, if available), and one of the entries can be selected.
- Information from the selected item is used to fill in the feed browser enabling field and its sub-fields.
- Alternatively, if **Feed-backed drop-down** is selected, the feed is presented as a drop-down.

> **Note:** To be able to fill in multiple fields from a single selection in the feed browser, fields must be sub-fields of the field that enables the feed browser.

Feed URL

Specify the feed URL for the field. The URL must be accessible from the Viz Pilot Edge browser and lead to a valid Atom XML or RSS feed:

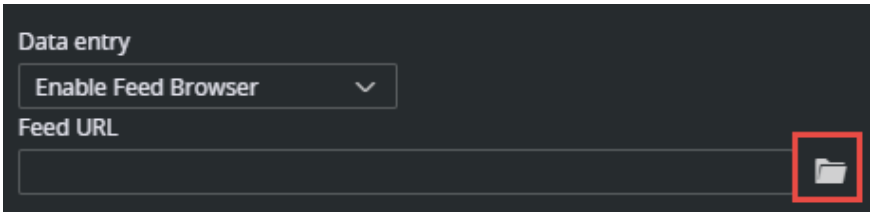


> **Note:** Internet servers can have strict CORS policies denying access to their feed from within Viz Pilot Edge.

Browse a Graphic Hub Feed

The Feed Browser can be used to browse Graphics Hub for images, material, geoms and fonts. To quickly build a valid URL to a folder in Graphic Hub, follow these steps:

- For an image field, select **Enable Feed Browser** as a data entry.
- Click the folder icon to browse the configured Graphic Hub:



- After browsing to the target folder and clicking Ok, the Feed URL points to this folder, for instance:

```
http://examplehost:19398/folder/221DEABE-B112-2F49-BC3C-5C3CD64A8AFE/
```

- The *{$GH}* environment variable can be used to build the base URL part. This variable resolves to the configured Graphic Hub on Pilot Data Server (for example, http://gh-host:19398/). If the Graphic Hub configuration changes in the Pilot Data Server settings, the template points to the new configuration automatically:

```
{$GH}/folder/221DEABE-B112-2F49-BC3C-5C3CD64A8AFE/
```

- The folder UUID part can be built dynamically based on either a field value or a key in the *$data* scripting object. This can be handy if the feed browser must open in different locations based on another user's choice in the template. For instance:

```
{$GH}/folder/{myfield}/
{$GH}/folder/{$data.currentFolder}/
```

Select from Feed Item

This option binds the element of a feed item (title, link, content etc.) to the value of the current field. This is a 1:1 relation between the feed item and the field value, but it is fully possible to bind a feed item to multiple fields in the template. For example, if you select a story from a feed and title, content, author and image are applied to the template. To accomplish this, the fields in the template need to be grouped under a parent field. See Sub Fields for more information on this.

> **Note:** The options available for a given field depend on the type of the field (the atom namespace prefix represents the http://www.w3.org/2005/Atom namespace, and the media namespace represents the http://search.yahoo.com/mrss/ namespace).

These are the fields in Atom/RSS that can be linked to:

- **<Not linked>:** Not linked to the feed item, and must be filled in manually.
- **Content:** Linked to the content of the *atom:content* element in the atom entry.
- **Title:** Linked to the content of the *atom:title* element in the atom entry.
- **Link:** Linked to the *href* attribute of the *atom:link* element in the atom entry. The link entry to pick depends on the *Link-rel in atom entry* property and the type of the field (the first link with a correct rel attribute and a type that matches the type of the field is chosen).
- **Entry:** Linked to the atom entry itself. This option should be used to link an image field to a feed entry from a Graphic Hub feed.

- **Author name:** Linked to the content of the *atom:name* element inside the relevant *atom:author* element, if the entry itself contains an *atom:author* element that is used. Otherwise, the *atom:author* element of the feed is used.
- **Author e-mail:** Linked to the content of the *atom:email* element inside the relevant *atom:author* element, if the entry itself contains an *atom:author* element, that is used. Otherwise, the *atom:author* element of the feed is used.
- **Author URI:** Linked to the content of the *atom:uri* element inside the relevant *atom:author* element, if the entry itself contains an *atom:author* element, that is used. Otherwise, the *atom:author* element of the feed is used.
- **Contributor name:** Linked to the content of the *atom:name* element inside the *atom:contributor* element in the atom entry.
- **Published:** Linked to the content of the *atom:published* element in the atom entry.
- **Updated:** Linked to the content of the *atom:updated* element in the atom entry.
- **Thumbnail:** Linked to the *url* attribute of the *media:thumbnail* element in the atom entry.
- **Summary:** Linked to the content of the *atom:summary* element in the atom entry.
- **Link-rel in Atom Entry:** Only available if **Link** is selected in the Select from atom entry property. It specifies the `rel` attribute of the link element in the atom entry.

Note: A linked field may also be filled in manually if it is not hidden or read-only.

Apply Sub Fields

Instead of linking individual Atom or RSS elements in an item to separate fields in the template, it is possible to link a VDF payload contained within the Atom `<content>` element to a parent field in the template. The field structure defined in the payload can then be mapped to a corresponding substructure within the template field.

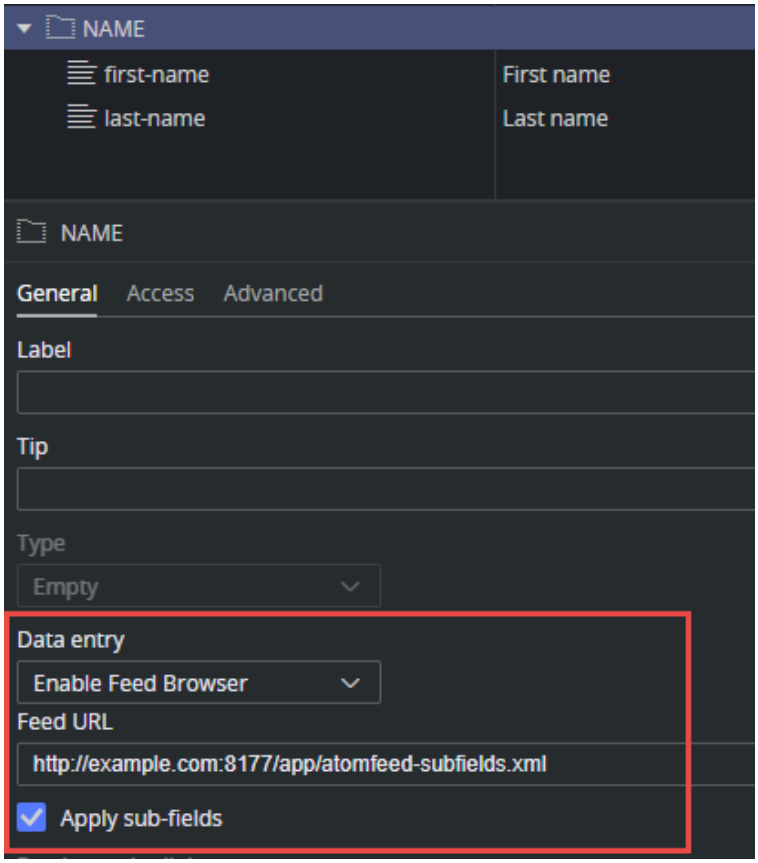
For example, a payload containing two fields (`first-name` and `last-name`) can be linked to a parent field in the template that has these two fields as its children. The field values from the payload are applied through straightforward name matching.

Example feed with two Atom entries containing a `<content>` section with a payload of two fields:

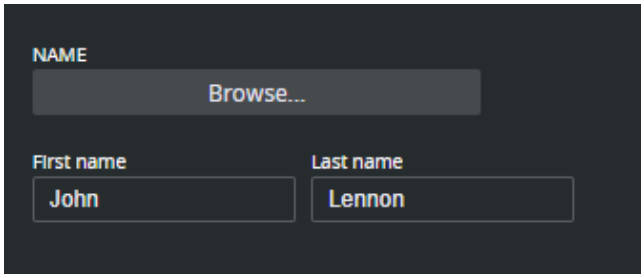
```
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:media="http://search.yahoo.com/
mrss/">
    <title type="text">Sample feed</title>
    <id>17f9d24c-7eab-4845-a346-ae047309999</id>
    <updated>2024-11-02T10:00:00Z</updated>
    <entry xmlns="http://www.w3.org/2005/Atom" xmlns:vaext="http://www.vizrt.com/
atom-ext">
        <id>id:1</id>
        <link rel="self" type="application/atom+xml;type=entry" href="http://
example.com:8177/app/1.xml"/>
        <title>John Lennon</title>
        <updated>2012-10-31T15:08:45Z</updated>
        <media:thumbnail url="https://upload.wikimedia.org/wikipedia/commons/thumb/
8/85/John_Lennon_1969_%28cropped%29.jpg/250px-John_Lennon_1969_%28cropped%29.jpg" />
        <content type="application/vnd.vizrt.payload+xml">
```

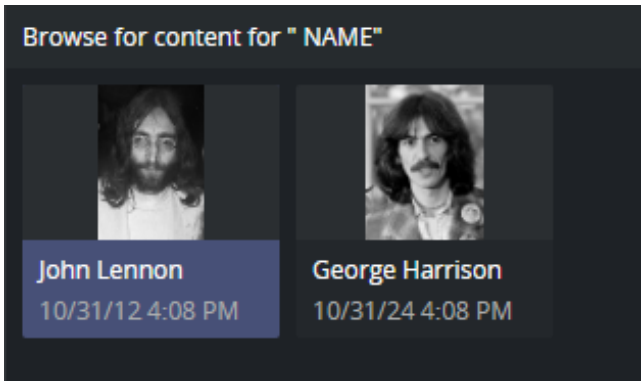
```
            <payload xmlns="http://www.vizrt.com/types">
                <field name="first-name">
                    <value>John</value>
                </field>
                <field name="last-name">
                    <value>Lennon</value>
                </field>
            </payload>
        </content>
    </entry>
    <entry xmlns="http://www.w3.org/2005/Atom" xmlns:vaext="http://www.vizrt.com/
atom-ext">
        <id>id:2</id>
        <link rel="self" type="application/atom+xml;type=entry" href="http://
example.com:8177/app/2.xml"/>
        <title>George Harrison</title>
        <updated>2024-10-31T15:08:45Z</updated>
        <media:thumbnail url="https://upload.wikimedia.org/wikipedia/commons/4/45/
George_Harrison_1974_%28cropped%29.jpg" />
        <content type="application/vnd.vizrt.payload+xml">
            <payload xmlns="http://www.vizrt.com/types">
                <field name="first-name">
                    <value>George</value>
                </field>
                <field name="last-name">
                    <value>Harrison</value>
                </field>
            </payload>
        </content>
    </entry>
</feed>
```

In Template Builder, this feed can be linked to a field having the same sub structure as the payload, for instance:



This enables the user in Viz Pilot Edge to browse the feed and select a person, or select from a drop-down if the option **Feed-backed drop-down** is set as Data entry.





RSS Mapping

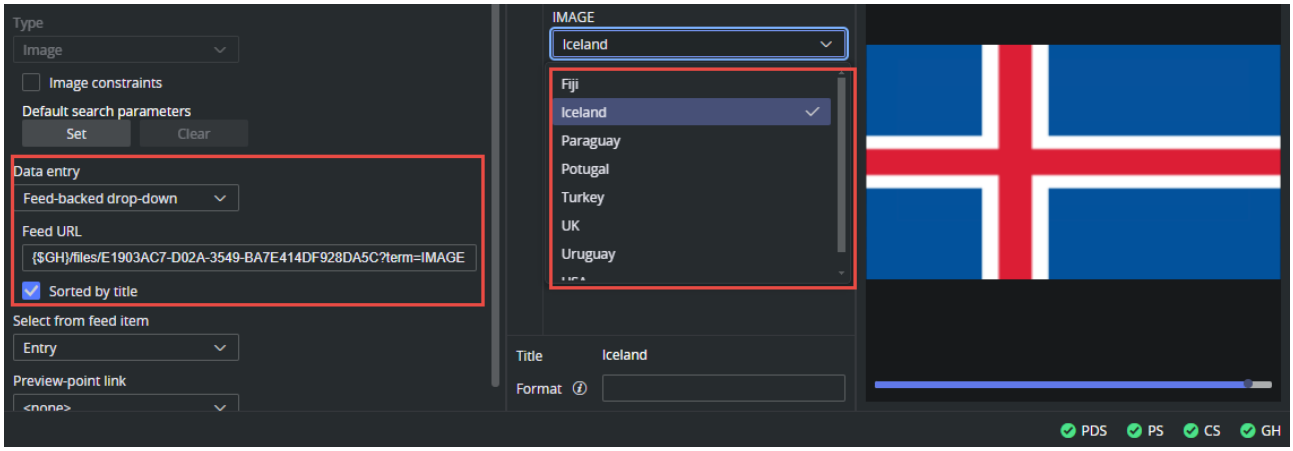
The feed browser in Template Builder works with Atom XML, but also supports a minimal mapping from standard RSS items according to the RSS 2.0 specification: *https://www.rssboard.org/rss-specification*.

The media namespace *xmlns:media="http://search.yahoo.com/mrss/* is specified in *https://www.rssboard.org/media-rss*.

RSS	Template Builder	Comment
<title>	Title	
<description>	Summary	
<pubDate>	Updated, Published	RSS has only one field when the item is published.
<author>	Author e-mail	In RSS, the <author> element is strictly specified as an e-mail address.
<enclosure type =" image/jpeg ">	Thumbnail, Link rel="enclosure"	Both Thumbnail and Link with link relation "enclosure" map to the <enclosure> element in RSS.
<media:content type="image/jpeg">	Thumbnail, Link rel="content"	Both Thumbnail and Link with link relation "content" map to the <media:content> element in RSS.
<media:thumbnail>	Thumbnail	

Feed-backed Drop-down

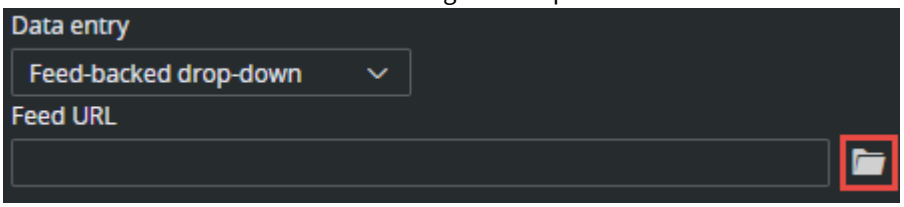
The **Feed-backed drop-down** option works exactly like the Enable feed browser option, but displays the results in a drop-down instead of in the feed browser window:



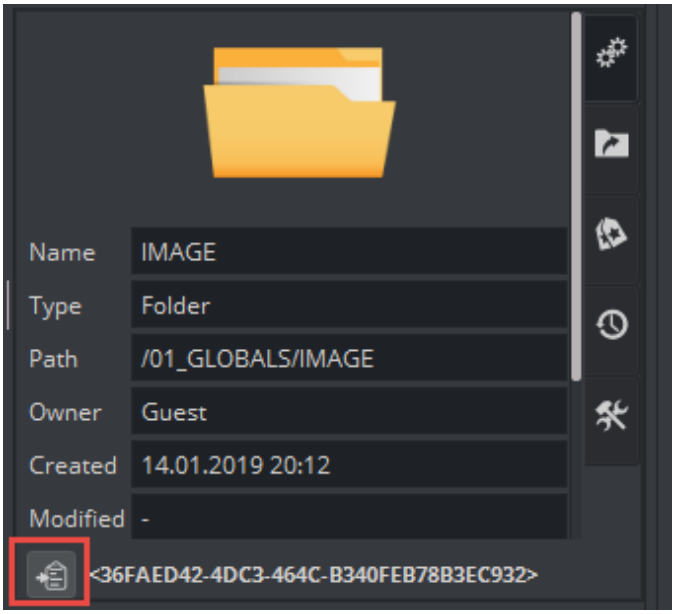
Example: Link Image Field to Graphic Hub Folder

In this example we set up a link from an image field in the template to a Graphic Hub folder, and present the result in a sorted drop-down.

- For the image field in the template, choose **Feed-backed drop-down** from **Data entry**.
- The **Feed URL** should point to a Graphic Hub folder, and we added a search term parameter to display only images. The URL is part of the Graphic Hub REST API.
  To quickly build a valid URL to a folder in Graphic Hub, follow these steps:
  - Click the folder icon to browse the configured Graphic Hub:

  - The *{$GH}* environment variable can be used to build the base URL part. This variable resolves to the configured graphic hub on Pilot Data Server (for example, http://gh-host:19398/).
  - Alternatively, to build the URL manually, add */files/* to the base URL of Graphic Hub, and add the *UUID* of the folder to the URL. This value can be copied from Viz Artist when browsing in Asset view:



- Add the *?term=IMAGE* to the URL. For instance:

```
{$GH}/files/BAA340C1-C71E-0949-BCF6-F7E043856030/?term=IMAGE
```

- In the **Select from feed item** drop-down, select **Entry**. This certifies the complete Atom entry is put into the field, making it playable for Viz Engine.
- Click **Sort by title** to make the entries sorted.
- The result should be a sorted drop-down with the images in the Graphic Hub folder.

Feed Linking to Tables

The Viz Artist plugin **ControlList** is commonly used to create tables with rows and columns. Filling data manually to a table can be a tedious task, and there are numerous ways of integrating data into the Viz Artist scene. To allow a Viz Pilot Edge user to select a data set and fill in the table automatically, link the ControlList field in the template to an Atom feed where the content of the feed is a CSV formatted text.

The data in this feed can also automatically update data on air, through the Update Service's Auto Update function. For instance, polling data changing live as the poll takes place. The latest numbers from the feed can automatically be played out by the control clients, and can also update live on air.

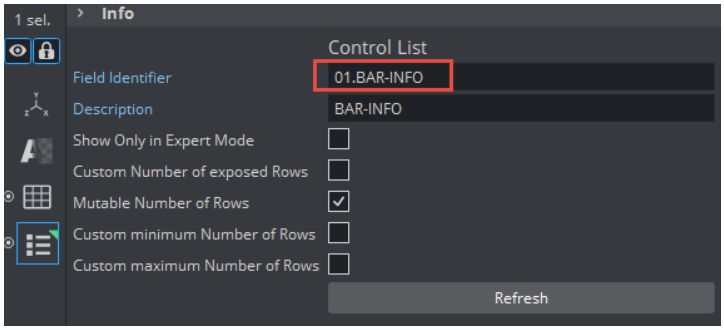
To accomplish this, consider the following example:

- Reporters in 3 cities execute a live poll of the citizen's favorite winter activities. Whenever the poll graphics is played in the Viz Engine, the name of the current city and the latest numbers should be used. We can accomplish this through the following steps.

Design the Scene with a ControlList and Hierarchical Fields

In Viz Artist, create a scene with the ControlList plugin for the table with the data, and a ControlText plugin on the title field. These fields live independently and have to be filled in separately. But in Template Builder, we want to fill in both fields when the Viz Pilot Edge user selects a feed entry. Therefore, they need to be connected in an hierarchical way. This can be done by naming the field with the "dot notation":

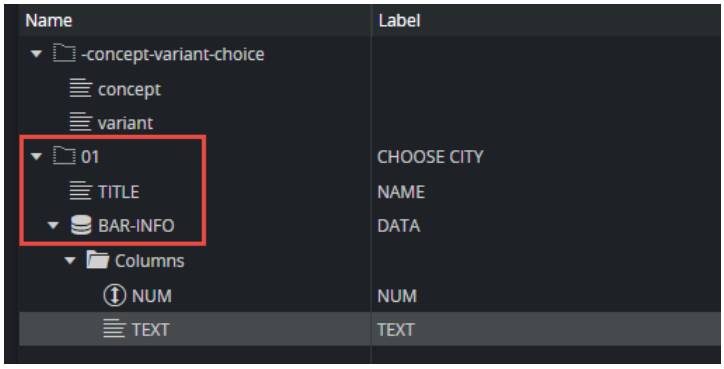
01.TITLE
01.BAR-INFO



By giving them the same prefix, the fields have a common parent field called "01". This is the field we link to the feed.

Create a Template based on the Scene

In Template Builder, create a new scene and select the newly created scene. In the field tree of the new template, the **TITLE** and **BAR-INFO** fields are children of 01:



This template can now be saved, and if opened in Viz Pilot Edge, the user can fill in the title and data for the table manually. Though, in this example we want to simplify the data input and let the user choose from a data feed.

Serve Out a Feed with the Data

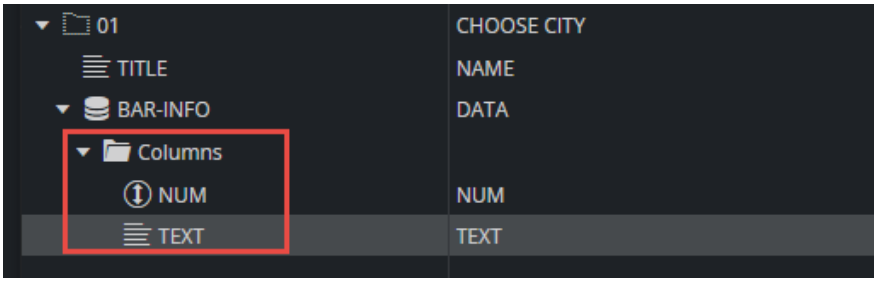
To select among data entries, an Atom Feed needs to be served out from a web server. It is fully possible, but not recommended for production, to serve out files from the Pilot Data Server's "/app" folder. The file with the Atom Feed needs to be accessible via an URL from the Viz Pilot Edge web browser. Below is an example with one feed entry:

```
<feed xmlns="http://www.w3.org/2005/Atom">
    <entry>
        <title>Bergen</title>
        <id>entry-bergen</id>
        <updated>2025-01-10T12:05:00Z</updated>
        <content type="text/csv">
NUM,TEXT
40,Hiking
30,Skiing
15,Ice skating
15,Snowboarding
        </content>
        <published>2025-01-10T14:26:17.290Z</published>
        <link rel="self" type="application/atom+xml;type=entry" href="http://bgo-
eddie-vm:1305/data/csv/bergen.entry.xml"/>
    </entry>
</feed>
```

Note that the <content> of the feed entry is a regular CSV-formatted string with type text/csv.

To map the CSV string to the fields in the table, the following assumptions are made:

- The first row in the CSV file contains the column names for the columns in the list field.
- The row names are mapped to column names in the list fields. In our example there are tow columns in the CSV file called NUM and TEXT, and these names must exist as columns in the list field:

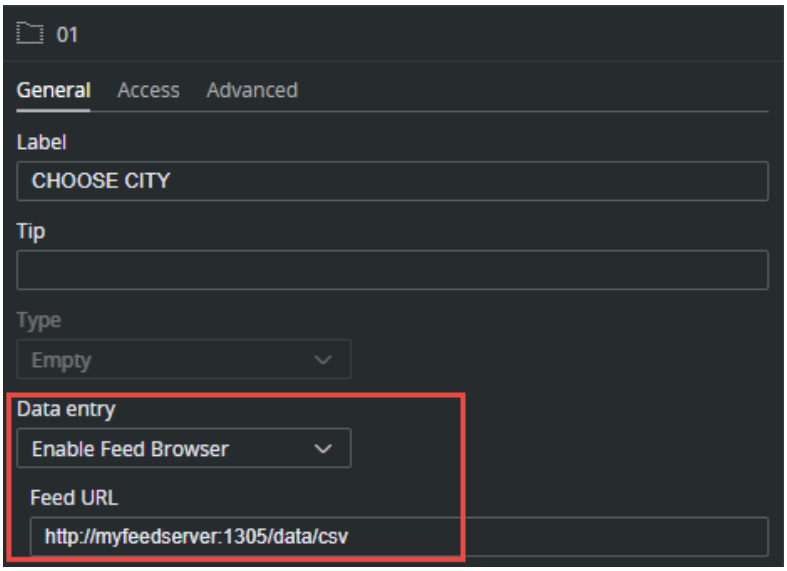


- The mechanism does not allow binding the list fields with children.
- The supported field types for the columns are string (single and multi-line), integer, decimal, integer and boolean.
- Use CSV cells as written by Excel.
  - Cells containing commas (`,`) or double quotes (`"`) are enclosed in double quotes (`"`).
  - Any double quote within a cell is represented by two consecutive double quotes (`""`).

Link the Columns to the Feed

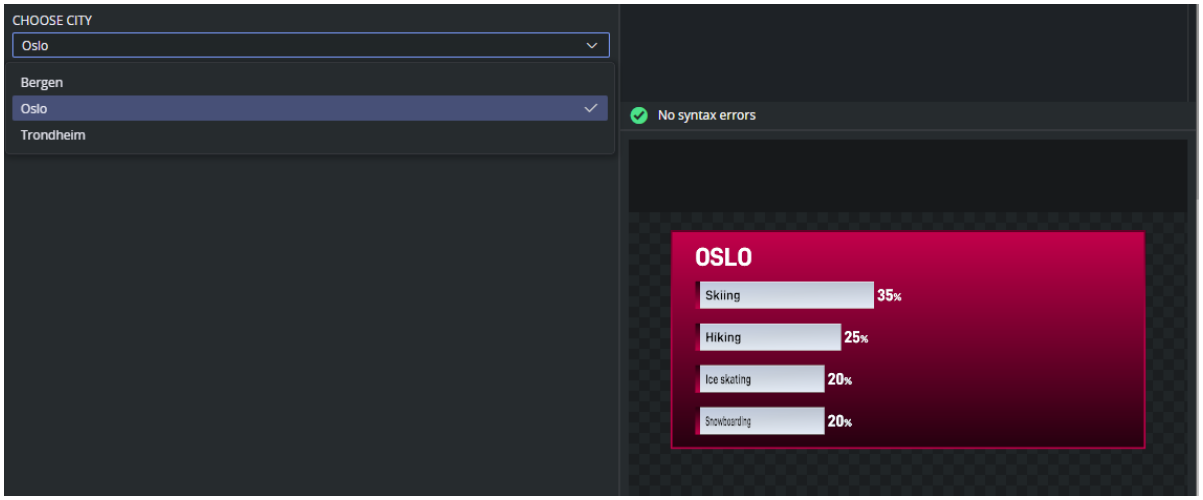
In our example we want to link the TITLE field in the template to the <title> of the feed entries, and we want to link the columns of our BAR-INFO list field to the <content> of the feed entries. In the Viz Artist scene design, the two fields were organized under a common parent field called 01.

1. As Data Entry for the field 01, specify **Enable Feed Browser** or **Feed-backed Drop-down**.
2. Enter the feed URL.



3. When a parent field is linked to a feed, the child fields can be linked as well. This means that we can link TITLE and BAR-INFO to the feed in parent 01.
4. For the TITLE field, select **Title** from the **Select from feed item** drop-down. Now the TITLE field is linked to the <title> of the selected feed entry.
5. For the BAR-INFO field (the list field), select **Content** from the **Select from feed item** drop-down. Now the columns of the BAR-INFO field are linked to the CSV <content> of the selected feed entry.

The expected result is for the Viz Pilot Edge user to open the template, and select from any entry in the field, filling in the title and the table automatically. For Feed-backed Drop-down, the result should look like this:



Info: When data is bound to the template with feed linking, **Pilot Update Service** can be enabled in “Auto update” mode for this template. When Media Sequencer takes data on air, the latest content from the feed entry is used. This is useful for sports results, polls, weather data or stocks (data that changes live). Media Sequencer can also update data while on air.

JSON-backed Drop-down

The **JSON-backed drop-down** is a Data Entry option that populates a field's drop-down from a remote REST endpoint returning JSON. Unlike the dynamic drop-down (which reads options from another field's value), this option fetches structured option data from a URL at runtime. This results in a searchable drop-down automatically filled with data.

How It Works

When a field is configured with a JSON-backed drop-down, Viz Pilot Edge makes a `GET` request to the configured URL when the template is opened. The response must be a JSON array of objects, each containing a label and value property. These become the selectable options in the drop-down.

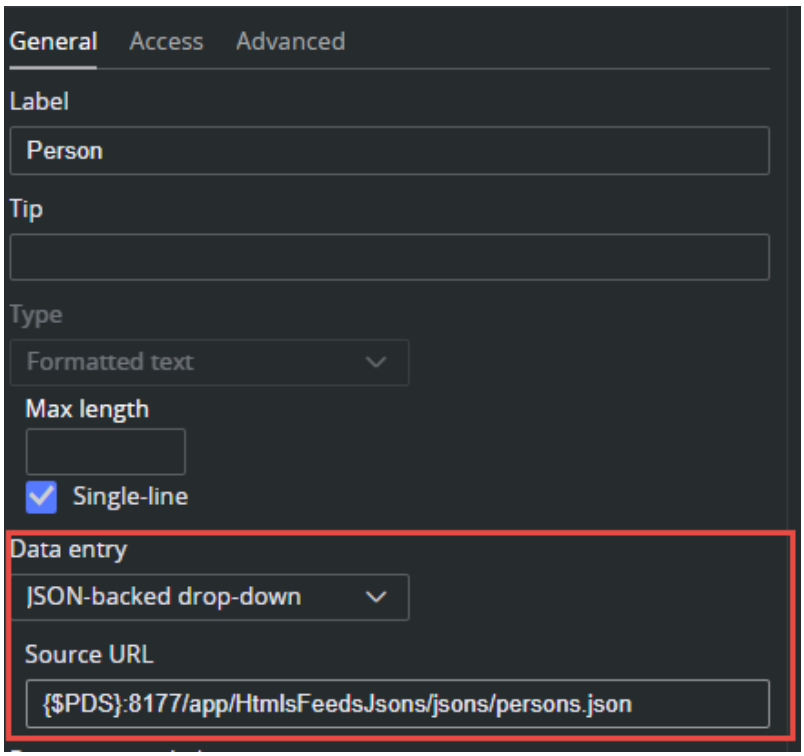
The URL supports **environment variable substitution** using the `{$VAR_NAME}` placeholder syntax, allowing the endpoint to be configured per deployment without changing the template.

Expected JSON Format

The REST endpoint must return a JSON array where each element has at least a label and value:

```
[
  { "label": "Option A", "value": "a" },
  { "label": "Option B", "value": "b" },
  { "label": "Option C", "value": "c" }
]
```

Configuring in Template Builder



1. Select the field.
2. In the **General** panel, locate the **Data Entry** drop-down.
3. Select **JSON-backed drop-down** from the list.
4. A **Source URL** text field appears, enter the URL to your REST endpoint.

The URL field supports environment variable placeholders. For example: `{$PDS}/app/HtmlsFeedsJsons/jsons/persons.json` resolves to `http://pds-host:8177/app/HtmlsFeedsJsons/jsons/persons.json`.

This resolves at runtime using the environment variables defined for the template, so the same template can point to different servers in different environments.

It is also possible to define custom environment variables through the optional settings (prefixed with `env-` in DataServerConfig settings) or via URL parameters (for example, ?$APP_SERVER=value).

See also:

- Environment Variables.

4.2.5 Inline HTML Fragment

In the auto generated form and in the custom layout tabs, it is possible to add an inline HTML fragment. The HTML content of this fragment is restricted to selected tags and attributes (see below). It can be used to insert custom UI elements into the template. The HTML button can also be linked to a click-handler in internal scripting, enabling the possibility to add special functionality when the user clicks the button. The HTML fragments are stored inside the template, so no need to host these fragments on a web server.

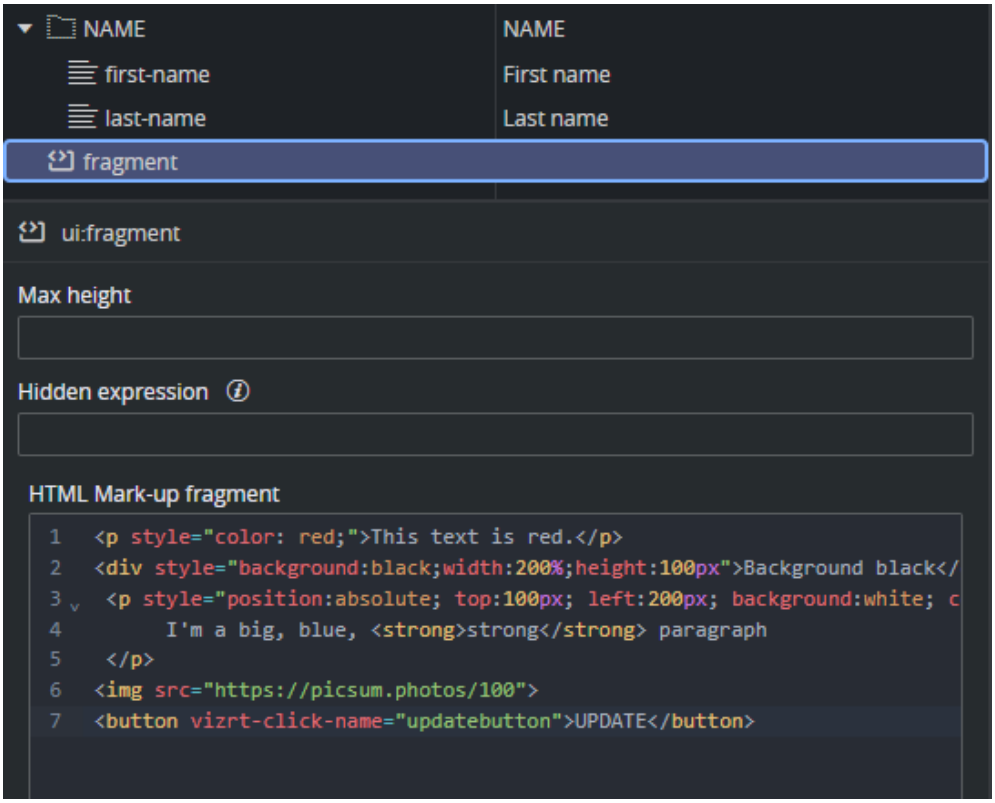
Note that external CSS styling is not supported. All styles need to be inline.

Adding an HTML Fragment

The inline HTML fragment can be added both to the auto-generated All tab, and to the custom layout tabs.

Adding an HTML Fragment to the Auto-generated All Tab

1. Right-click the field tree.
2. Select **Add UI panel** > **HTMLfragment**.
3. Add an ID to the new field. The HTML fragment is now a part of the field tree for the All tab.



In the field editor for HTML fragments the maximum height can be set and the HTML can be entered. See below for HTML tag and attribute limitations for the HTML entered into the box. It is also possible to enter a Hidden- or Read-Only expression for this panel. See Hidden and Read-only Expressions.

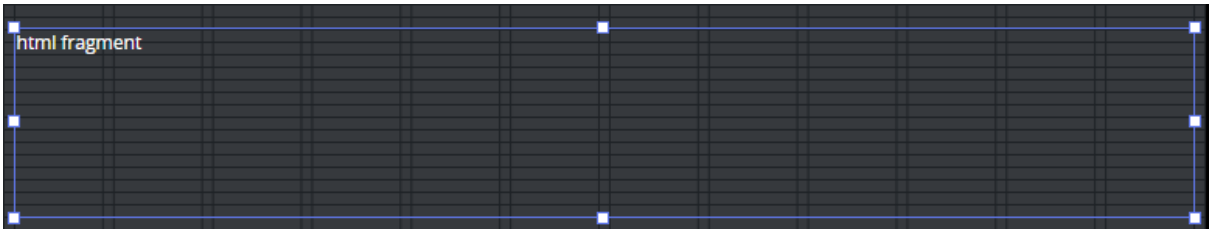
Adding an HTML Fragment to the Custom Layout Tabs

For the Custom Layout tabs, the UI components like HTML panels and HTML fragments, are not part of the field tree on the left. They float inside the custom layout tabs and can only be reached by clicking them in the UI.

To add an inline HTML fragment into a custom layout tab, follow these steps:

1. Create or select a custom layout tab.
2. In the middle toolbar, click the **Add HTML fragment to current view** button.
3. Enter an ID for the fragment.

A new HTML fragment is now added to the form.



4. Click the ruler to toggle edit and view mode:  .

Z-order

In the current version of Template Builder, the only way of specifying the Z-order of the components, is to add them to the form in the right order. The first component added to the form is in the back layer, then each component added has a higher Z-index, and the last component added is on top.

Use Inline Bindings

From inside HTML fragments, it is possible to use values from fields and environment variables. Examples:

HTML fragment	Note
`Headline: {{01-HEADLINE}}`	Display the value from the field called 01-HEADLINE.
`<img src="{{$PDS}}/pilotedge-3.0/vizPilotEdgeLogo.svg">`	Use the value of the built-in environment variable “PDS” and use it in a URL.
`Version: {{$version}}`	Use the user-defined environment variable “version”. This variable can be defined in a URL parameter to Viz Pilot Edge or Template Builder, for instance: *https://pds-host:7373/app/templatebuilder/templatebuilder.html?$version=4-0*

Info: The field binding is supported for the following field types: **Single-** and **multiline**, **numbers**, **boolean** and **date**.

Use Input Bindings

In HTML fragments it is possible to two-way bind data between HTML input fields and internal scripting:

Use the `vizrt-input-name` attribute to bind the control:

Example 1 - Bind to Field

```
<input vizrt-input-name="title" type="text" value="MATCH DAY">
```

This binds the value of the input box to the field “title” in the template.

Example 2 - Bind to Scripting

```
<input vizrt-input-name="$data.home_score" type="number" value="0">
```

See Action Panels for more information.

Adding a Clickable Button

Inside an HTML fragment, it is possible to add an HTML button. Scripting inside HTML fragments is not allowed, but there is a way to add an event handler to the button to enable internal template scripting on click.

1. Add a <button> in the HTML fragment.
2. Add a special attribute *vizrt-click-name* to the button, specifying how a click on this button can be identified in internal scripting.
3. Add an internal script event handler for *onClick* and check for the value of the attribute above.

In the HTML fragment:

```
<button vizrt-click-name="updatebutton">UPDATE</button>
```

The name can then be checked for internal template scripting:

```
vizrt.onClick = (name: string) => {
  if (name === "updatebutton") {
    // Do something
  }
}
```

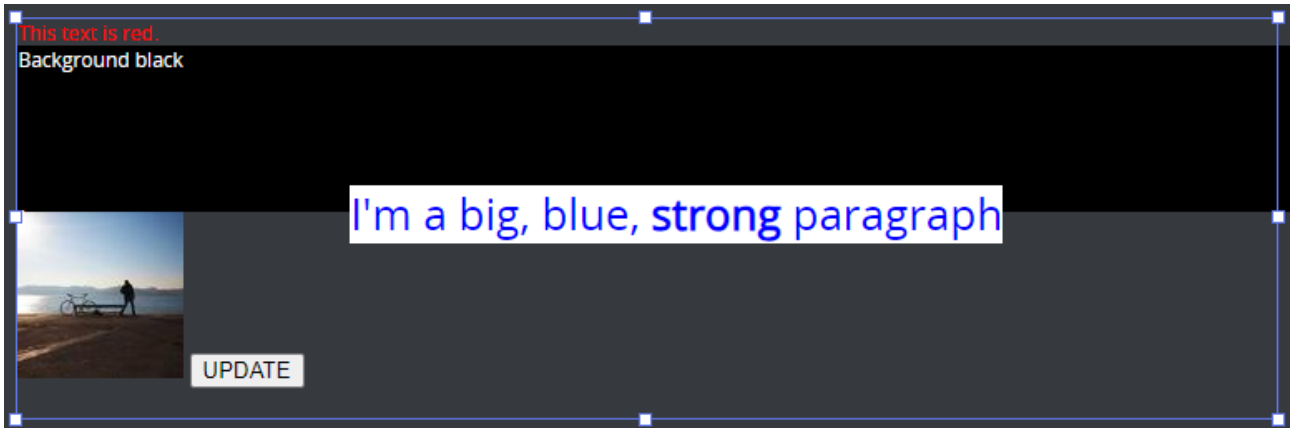
Example

Inside the HTML fragment, you can basically use HTML tags without scripting nor links, and with inline CSS styling. This is due to security and layout reasons, but there is quite a lot of flexibility.

```
<p style="color: red;">This text is red.</p>
<div style="background:black;width:200%;height:100px">Background black</div>
 <p style="position:absolute; top:100px; left:200px; background:white;
color:blue;font-size:26px;">
      I'm a big, blue, <strong>strong</strong> paragraph
 </p>
<img src="https://picsum.photos/100">
```

```
<button vizrt-click-name="updatebutton">UPDATE</button>
```

You then have the following panel:



Allowed HTML Tags and Attributes

Note that styling must be inline. Fragments do not support external CSS with the <STYLE> tag.

White-listed Tags

ABBR	DETAILS	I	Q	THEAD
ADDRESS	DFN	IMG	RP	TIME
AUDIO	DIV	INS	RT	TR
ARTICLE	DL	KBD	RUBY	U
ASIDE	DT	LABEL	S	UL
B	EM	LEGEND	SAMP	VAR
BDI	FIELDSET	LI	SEARCH	VIDEO
BDO	FIGCAPTION	MAIN	SECTION	WBR
BLOCKQUOTE	FIGURE	MAP	SELECT	
BR	FOOTER	MARK	SMALL	
BUTTON	FORM	MENU	SOURCE	
CAPTION	H1	METER	SPAN	
CITE	H2	OL	STRONG	
CODE	H3	OPTGROUP	SUB	
COL	H4	OPTION	SUMMARY	
COLGROUP	H5	OUTPUT	SUP	
DATA	H6	P	TABLE	
DATALIST	HEADER	PICTURE	TBODY	
DD	HGROUP	PRE	TD	
DEL	HR	PROGRESS	TFOOT	
			TH	

Black-listed Tags

These tags cannot be used in inline HTML fragments, either because they expose a security risk, they conflict with the application, they need to be bound to script, or they make no sense inside the <BODY> of an HTML fragment.

AREA A BASE CANVAS DIALOG	EMBED HTML IFRAME INPUT LINK	NAV NOSCRIPT OBJECT PARAM SCRIPT	STYLE SVG TEMPLATE TEXTAREA	TITLE

White-listed Attributes

These are the allowed attributes inside tags in an HTML fragment.

The attributes must be in lower case: **alt, datetime, height, kind, label, name, src, srclang, style, title, type, width**.

4.2.6 Inline HTML Panel

An HTML panel can be added to the template as part of the template customization workflow, giving you full control through custom scripting and logic when building the template. The panel is displayed inside an **iframe** and needs to be hosted on an external web server. Inside the custom HTML panel, the use of the **PayloadHosting API** connects your panel to the fields of the template.

See *samples/html_panels/README.html* under the Template Builder installation folder for samples and more details.

Adding an HTML Panel

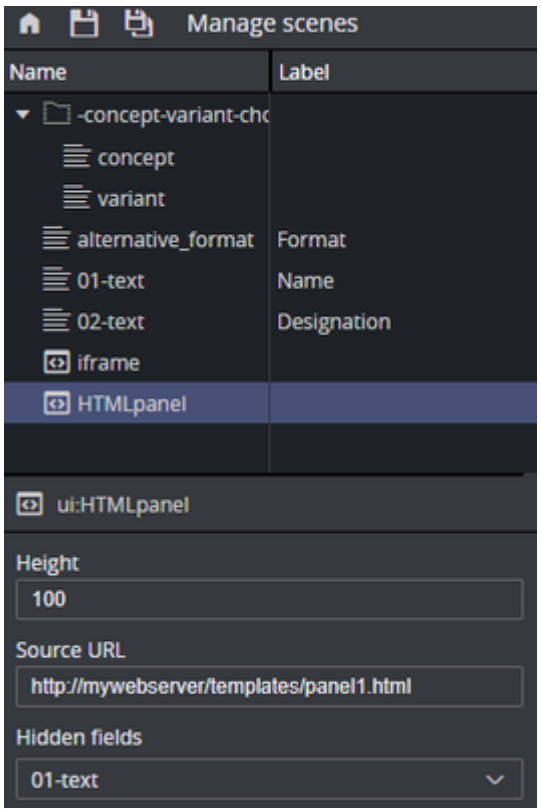
The HTML panel can be added both to the auto-generated All tab, and to the custom layout tabs.

Adding an HTML Panel to the Auto-generated All Tab

1. Right-click the field tree.
2. Select **Add UI panel** > **HTML panel**.
3. Alternatively, right-click a field in the field tree and select **Add HTML panel before/after**.
4. Add a field ID to the new UI field.

The HTML panel is now a part of the field tree for the All tab.

Select the new field ID in the field tree to show its properties in the Field Properties window:



- Adjust the size of the HTML panel shown in the fill-in form using the **Height** field.
- In **Source URL**, enter the web address.

- The **Hidden fields** drop-down list allows you to hide available fields in the fill-in form.

Adding a HTML Panel to the Custom Layout Tabs

For the Custom Layout tabs, the UI components like HTML panels and HTML fragments, are not part of the field tree on the left. They float inside the custom layout tabs and can only be reached by clicking them in the UI.

To add an inline HTML panel into a custom layout tab, follow these steps:

1. Create or select a custom layout tab.
2. In the middle toolbar, click the **Add iframe panel to current view** button.
3. Enter an ID for the panel.

Using Environment Variables in URLs

When adding URLs to a template, either a full page custom HTML template URL or an URL to an HTML panel inside an iframe, the URL can contain *environment variables*. These variables can be set as URL parameters to the application, or picked up from the application's built in variables. For example, if the URL to the application is *http://mypds:8177/app/pilotedge/pilotedge.html&$version=v1*, then the value of URL parameters starting with the dollar sign are available inside the application, and can be used for instance when specifying URLs to HTML panels like this: *{$}http://mywebserver/templatepanels/{$version}/mypanel.html*.

Therefore, the full URL used by the application is http://mywebserver/templatepanels/v1/mypanel.html. This way, the Viz Pilot templates using HTML panels or full custom HTML pages can for instance switch to another version by changing the URL parameter to the application.

See Using Environment Variables for full description of this mechanism.

Browser Caching

You may experience browser caching behavior when trying to update and display changes in the custom HTML template in Template Builder, this is standard behavior. Template Builder does not control caching resources included in the HTML file itself.

To prevent caching:

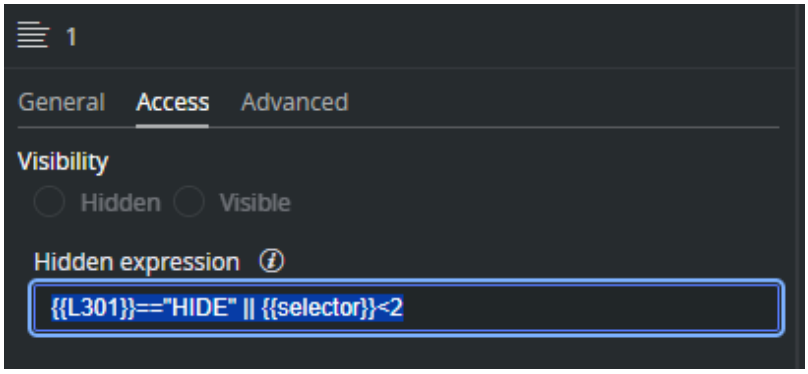
1. Ensure the URLs to the resources are unique upon reload.
2. Optionally configure the web server serving the resources to send Expiry headers set to 0.
3. Disable caching on the browser side.

See also

- Custom HTML Templates with examples of how to use custom HTML templates.
- Environment Variables

4.2.7 Hidden and Read-only Expressions

Instead of writing internal script code to determine the visibility or read-only properties of fields and UI panels, you can add a JavaScript expression in the **Read-only** or **Hidden expression**, in the properties for a field or UI panel.



The **Read-only** and **Hidden-expression** are basic JavaScript `eval` expressions, that decide whether a field should be hidden or read-only. See the table below for examples.

> **Info:** Read-only and Hidden expressions support JavaScript notation to evaluate an expression. They support field lookup, arithmetic and logical operators.

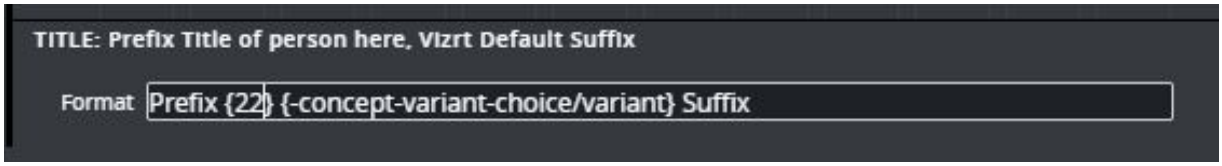
- Field references must be enclosed in double curly brackets, for example `{{field01}}`.
- Text fields referenced in the expression must be of type **Single Line**.

Expression	Description
`{{L301}}=="HIDE" || {{selector}}<2`	Match if the `L301` field value is `"HIDE"` and the numeric field `selector` is less than 2.
`({{A}}+{{B}}) > 2`	Match if `A` (which is 1) *plus* `B` (which is 2) larger than 2
`{{A}}-{{B}} < 2`	Match if `A` (which is 2) *minus* `B` (which is 1) less than 2.
`{{A}}*{{B}} == 4`	Match if `A` (which is 2) *multiplied* with `B` (which is 2) equals 4
`{{A}}*0.5 == 2`	Match if `A` (which is 4) *multiplied* by `0.5` (instead of "/" by 2) equals 2. The reason we use multiplication instead of divide, is that the division character is not supported in the expression.

Expression	Description
`{{$APP}} ==="TemplateBuilder"{{ field_checkbox}} ==false`	Match if the environment variable `$APP` equals `“TemplateBuilder”`, and the field called `field_checkbox` is false.

4.2.8 Auto-generated Title

The **Title** setting provides an auto-generation of the title. The title can be plain text or it can be a placeholder for one or several field values, or it can be a combination of these. The placeholder is the *{Field ID}*, the example below shows a combination of plain text, field name, and sub-field name:



A template title can be auto-generated by combining one or several of these options:

- **Normal text:** Plain text (red).
- **{Field ID}:** Substituted with the value of the field (green).
- **{Field ID/subfield ID}:** Substituted with the value of the subfield (purple).
- **{listfieldname/#index/cellname}:** Substituted with the value of the field in a row in a list. Note that the index is zero-based.

Warning: The auto-generated title's length is not shortened in Vizrt web clients. However, if the title is longer than 128 characters, it is reduced when dragging out the MOS XML file due to size constraints. This affects the element title in the newsroom system.

Note: When the template uses **custom HTML** representation, it is highly recommended to set a pattern for Auto-generated title.

Use Case: Enable Auto Generated Title on Placeholder Elements

A common workflow, is to create library elements and use them as placeholders in a rundown. To easily recognize these placeholders, they are often given special names, such as *PLACEHOLDER – LOWER THIRD*. These elements are typically inserted into the rundown in advance by the person preparing the rundown skeleton.

When a journalist later opens such placeholder item, they fill in the relevant fields and click **Save**. At this point, the suggested save name is still the overridden library title (for example, *PLACEHOLDER – LOWER THIRD*), because the auto-generated title was replaced by the placeholder title. In practice, however, the placeholder name should never appear in the rundown. The auto-generated title should be used instead.

A journalist can manually delete the suggested title in the save dialog, to restore the auto-generated title. Alternatively, the template script can automatically enforce this, by including the following code:

```
vizrt.fields.$1.onChanged = () => {
    const titleField = vizrt.fields["$-title"];
    if (titleField.value?.startsWith("PLACEHOLDER")) {
        titleField.value = "";
    }
};
```

In this example, whenever the field **1** is changed by the user, the script checks the current title. If the title begins with *PLACEHOLDER*, it is cleared. This ensures the auto-generated title is used as the suggested save name, rather than the placeholder name.

4.3 Custom HTML Templates

PayloadHosting is a JavaScript library that enables custom HTML interfaces (both full custom HTML templates and inline HTML field editors) to interact with Template Builder and Viz Pilot Edge. It provides access to the VDF (Viz Data Format) payload of a template, allowing you to create specialized UIs to edit template data, such as headlines, names and images for a news story.

This section contains the following pages:

- Configure Custom HTML Pages and Panels
- Overview of Key Mechanisms
- Quick Start Code Samples

> ⚠ **Note:** The full API documentation is distributed with Template Builder under the location: */app/templatebuilder/js/@vizrt/payloadhosting/dist/docs/index.html*.

4.3.1 Use PayloadHosting with NPM

The `payloadhosting` module is available as an NPM package, making it easier to integrate into modern JavaScript and TypeScript projects. This method simplifies dependency management, improves development workflows, and enhances compatibility with build tools.

- **Install** the package using `npm install "c:\...\js\@vizrt\payloadhosting"`.
- **Module-Based Import:** Use ES module syntax for better structure:

```
import { payloadhosting } from "@vizrt/payloadhosting";
```

- **Type Support:** The package includes `payloadhosting.d.ts`, enabling IDE autocompletion and type checking.
- **Maintainability:** Updates and version control are handled via `package.json`.
- **Integration:** Works well with modern JavaScript tooling such as Webpack, Vite, and Rollup.

4.3.2 Manually Include PayloadHosting in HTML

> ⚠ **Note:** `payloadhosting.js` can be found under the `./js` folder of the Template Builder installation path.

For projects that do not use NPM, `payloadhosting.js` can be included manually in an HTML file by adding a script tag:

```
<!doctype html>
<html>
  <head>
    <script src="./payloadhosting.js"></script>
```

```
  </head>
  <body>...</body>
</html>
```

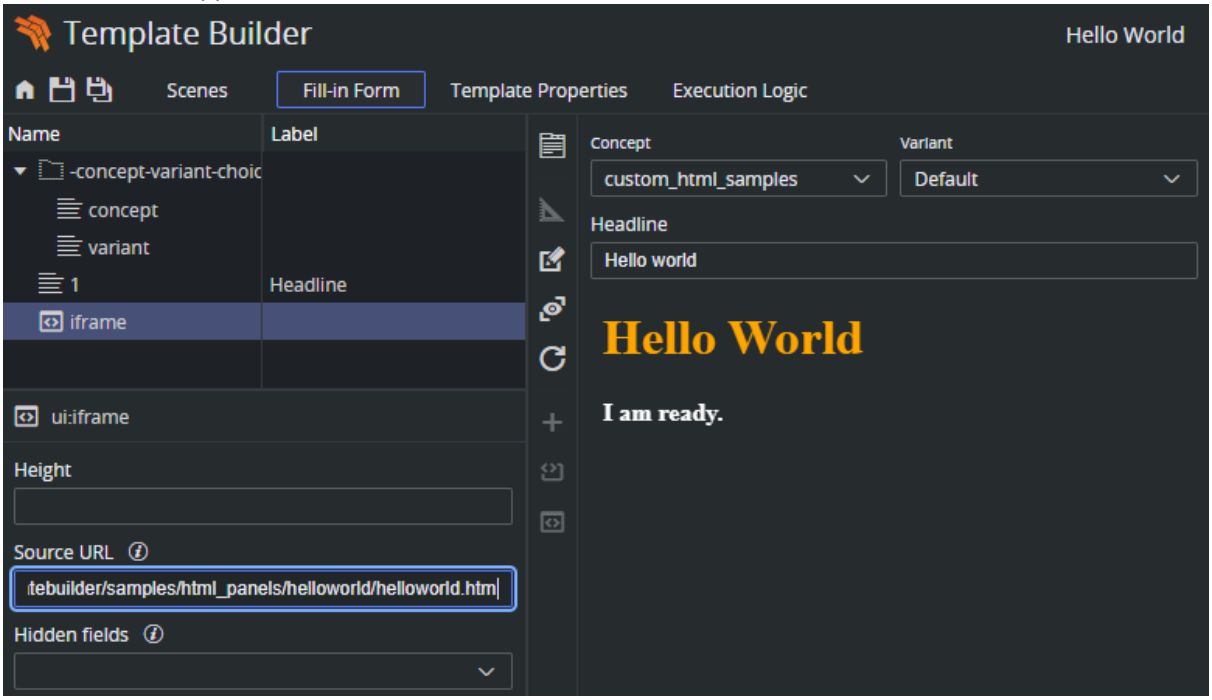
This approach ensures `window.vizrt.payloadhosting` is globally available to use in your scripts.

4.3.3 Configure Custom HTML Pages and Panels

Custom HTML pages are hosted within an iframe in Template Builder and Viz Pilot Edge. They must be served from a web server and accessible via a URL. These pages can function either as a full-screen replacement for a Template Builder template or as an inline panel within a template.

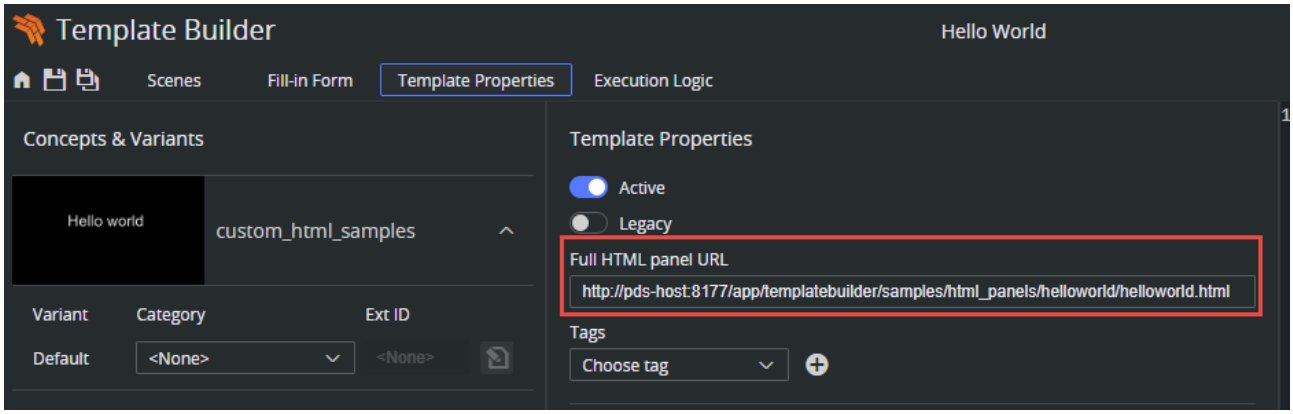
Specify an Inline HTML Panel

- Open a template and add an HTML panel in the auto generated form, or as a UI component in a custom view, as described in Inline HTML Panel.
- In the URL field, enter the URL of the custom HTML template. In this example, the URL from Hello World sample is: *http://pdshost:8177/app/templatebuilder/samples/html_panels/helloworld/helloworld.html*
- *Hello world* now appears in the Fill-in-Form:

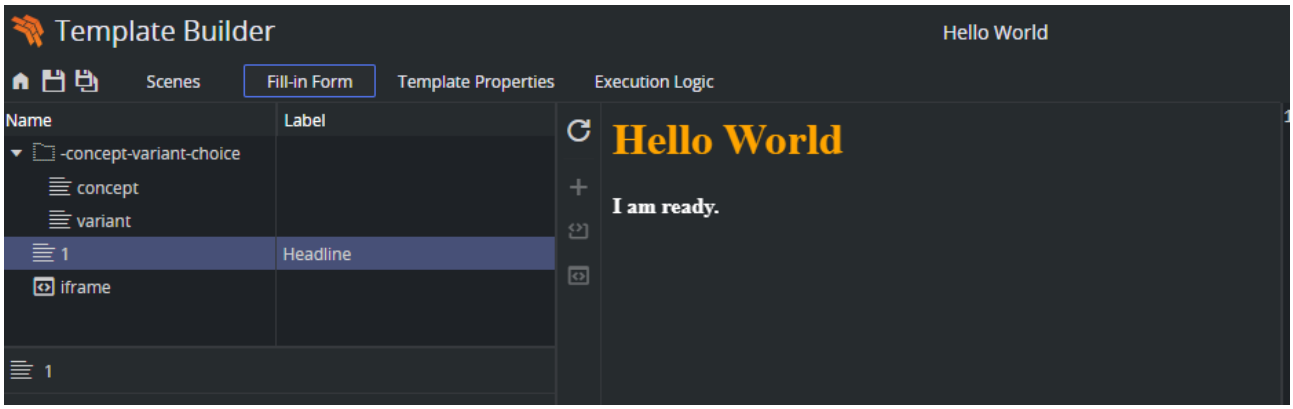


Use a Full Screen Custom HTML Page

With a full screen HTML, it is possible to replace the built-in template with a custom HTML representation that replaces the whole template. The URL to the custom HTML page must be set in **Template Properties**:



The custom HTML page replaces the whole UI:



4.3.4 Overview of Key Mechanisms

The `payloadhosting.js` script acts as a bridge between a custom-built UI and the underlying VDF model, handling data exchange, automatic field bindings, and event-driven updates.

> ⚠ **Note:** The full API documentation is distributed with Template Builder under the location: */app/templatebuilder/js/@vizrt/payloadhosting/dist/docs/index.html*.

Initialization

- The `PayloadHosting` object is the core of the script.
- The `vizrt.PayloadHosting.initialize()` function connects the HTML document to the Template Builder host and fetches the current template payload.
- If a callback function is provided, it is executed once the payload is ready.
- The script automatically binds HTML elements to payload fields using element IDs with a `field_` prefix , for instance `<input id="field_headline>` , has automatically a bi-directional connection to the field `headline` .

Accessing the VDF Payload (Data)

- The **VDF payload** holds the actual data entered into the template (for example, headlines, images, and other content).
- The script allows reading and modifying payload values but **does not modify the model structure**.
- Key methods:
  - `getFieldText(fieldPath)` : Retrieves a text value from the payload.
  - `getFieldXml(fieldPath)` : Retrieves XML content from the payload.
  - `setFieldText(fieldPath, value)` : Updates a field's text value.
  - `setFieldXml(fieldPath, xmlString)` : Updates XML content in a field.
- While the script reads **field definitions** from the model (for example, checking if a field exists or is a list), it does **not** modify the model schema.
- Example: `isListField(fieldPath)` checks if a field is a list, but does not create or change the list definition.

Event-Driven Updates

The script listens for changes in the payload and updates the UI dynamically.

- Event listeners can be registered using:

```
vizrt.payloadhosting.addEventListener("payloadchange", callbackFunction);
```

- This allows custom UI components to react to external changes in real time.

Automatic and Manual Bindings

By default, `payloadhosting.js` automatically maps input elements (`<input>` , `<textarea>` , `<select>`) to payload fields.

This can be disabled if manual control is required:

```
vizrt.payloadhosting.setUsesAutomaticBindings(false);
```

Handling Lists and Complex Fields (Tables)

The script supports **list fields**, allowing users to manage repeating elements in a template.

- Methods for managing lists:
  - `addListFieldItem(fieldPath)` : Adds an item to a list.
  - `removeListFieldItem(fieldPath, index)` : Removes an item from a list.
  - `getListFieldLength(fieldPath)` : Returns the number of items in a list.
- A `fieldPath` is a string that describes the location of a field in the payload.
  - **Top-level field**: `"myList"` (if `myList` is a list field).
  - **Nested field**: `"container/myList"` (if `myList` is inside `container`).
  - **List item reference**: `"myList/#0"` (first item in `myList`).
  - **Field inside a list item**: `"myList/#2/name"` (the `name` field inside the third item of `myList`).
- Lists in `payloadhosting.js` **can effectively be used as tables** because each list item can contain multiple sub-fields, similar to rows and columns in a table.
  - Each **list item** is a **row**.
  - Each **sub-field** inside a list item is a **column**.

Invoking Native Field Editors

The `payloadhosting.editField(fieldPath, editRequestParameters?)` method allows a custom HTML UI to trigger the built-in Template Builder editor for a specific field in the payload. This is especially useful for editing complex data types, such as images, videos, or formatted text, using Template Builder's native editor instead of standard HTML inputs.

Syntax:

```
vizrt.payloadhosting.editField(fieldPath, editRequestParameters);
```

Parameters:

- `fieldPath` **(string, required)**: The path to the field in the payload that should be edited.
- `editRequestParameters` **(optional)**: An object that provides additional options for the edit request, such as predefined values, constraints, or configurations for the editor.

Communication with the Host

The `payloadhosting.js` script communicates with the host application (Template Builder and Viz Pilot Edge) using the `postMessage()` API. Messages are exchanged between the custom UI and the host to update and retrieve data.

- **Sending messages to the host:** When changes occur in the payload, `payloadhosting.js` sends updates to the host using:

```
this._host.postMessage({ type: "data_changed", changes: [...] },
getHostOrigin());
```

- **Receiving messages from the host:** The script listens for messages, such as new payload data, using an event listener:

```
window.addEventListener("message", (event) => {
    if (event.data.type === "set_payload") {
        console.log("New payload received:", event.data.xml);
    }
});
```

- **Types of Messages**
  - `set_payload` : The host sends updated payload data to the custom UI.
  - `data_changed` : The UI notifies the host of modifications made to the payload.
  - `request_model_info` : The UI requests the model schema from the host.
  - `provide_model_info` : The host provides the requested model schema.

Security and Data Validation

The `payloadhosting.js` script includes multiple security measures to ensure safe data handling and prevent unauthorized modifications:

- **Safe Media Handling**

    - It ensures URLs for media assets (for example, images and videos) are correctly formatted and not externally manipulated.
    - The function `isSafeMediaType(type)` checks whether a given media type is permitted.
- **Field Access Control**
    - The script restricts modifications to registered fields only.
    - Any attempt to update a non-existent or unauthorized field results in an error.
    - Functions like `fieldExists(fieldPath)` ensure a field exists before attempting modifications.
- **Cross-Origin Communication Restrictions**
    - The script ensures messages are only exchanged with the expected host origin using `getHostOrigin()`.
    - Unauthorized sources attempting to send messages are ignored.
- **Preventing Invalid Data**
    - Input values are validated before being stored in the payload.
    - XML content for structured fields is parsed and checked before insertion using `setFieldValueAsParsedXml()`.

4.3.5 Quick Start Code Samples

Note: More samples can be found at */app/templatebuilder/samples/html_panels/README.html.*

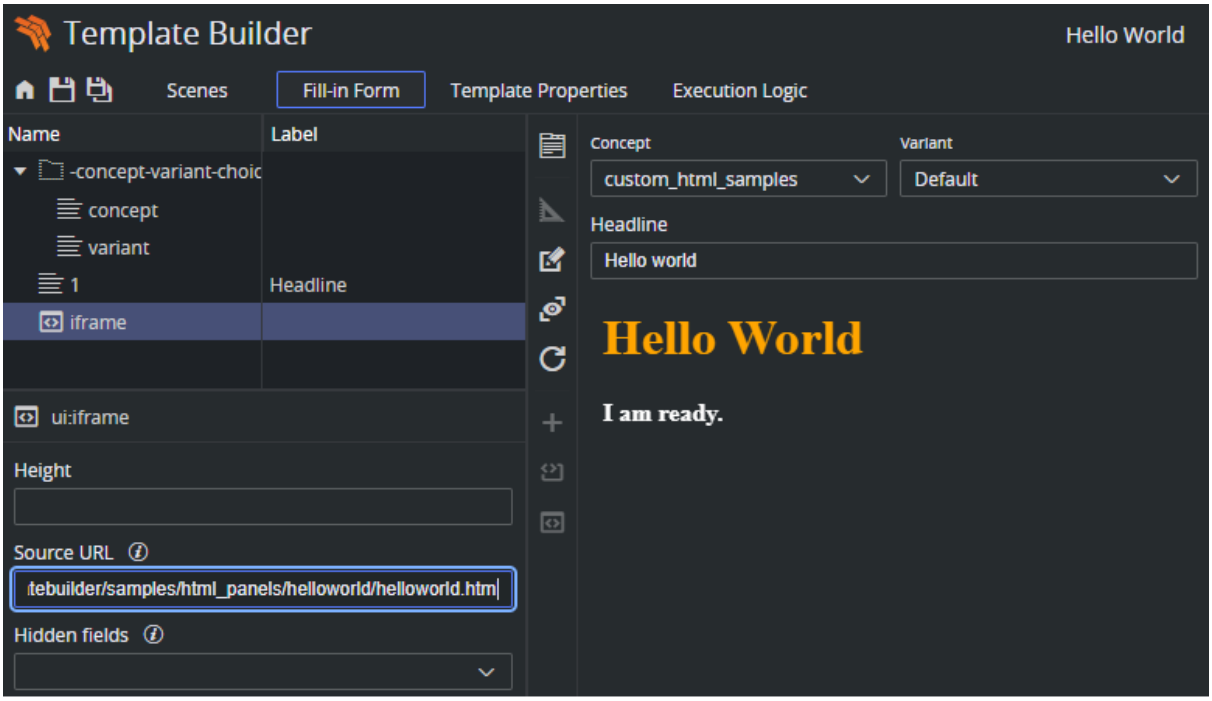
Hello World

Info: See *samples/html_panels/helloworld/helloworld.html.*

The example below uses a template that shows the message **Hello world** when opened in a browser.

```
<html>
<head>
    <script src="../payloadhosting.js"></script>
</head>
<body onload="vizrt.payloadhosting.initialize(callback)">
    <h1 style="color: orange;">Hello World</h1>
    <div style="color: white;" id="hello"><div>

    <script>
    function callback(_initType, data) {
        document.getElementById("hello").innerHTML = "<b>I am ready.</b>";
    }
    </script>

</body>
</html>
```

- When the HTML page is loaded, the `initialize` function is called with a given callback.
- When Template Builder or Viz Pilot Edge has loaded the payload and the data model is ready, the callback is called.



Connecting to Fields

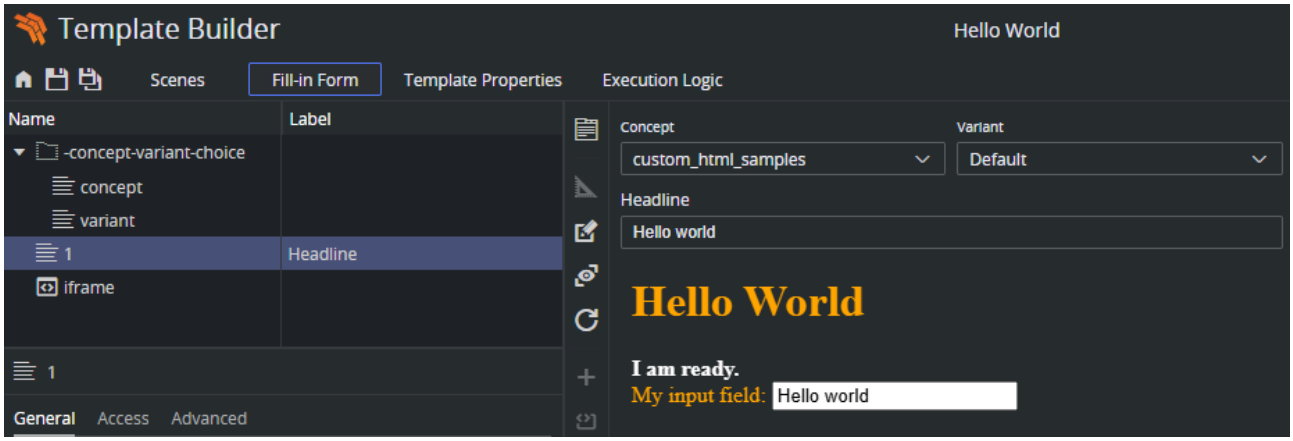
Following the example above, we can establish a two-way communication, or *bind fields*, between the HTML template and the opened pilot template. This provides a simple way of setting up a binding field. Add a new field to the template:

- Make sure your template has a field called “1”, that is, right-click in the HTML panel field, choose **Add field before/after** and select **Single-line text.**
- Assign the ID **1**.

This must be added to the <body> block:

```
<div>
  <label style="color: orange;"  for="field_1">My input field:</label>
  <input name="field_1" type="text" id="field_1">
</div>
```

Saving the HTML file and clicking **Refresh HTML panels** reloads the custom HTML template with the changes just made. A bi-directional connection between the custom template and pilot template has now been established. If you now type inside either the template or the field with ID 1, both fields are updated at the same time.

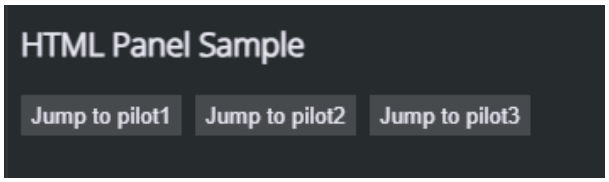


Note: This way of binding fields works for any HTML fields that have *value* support, typically `<input>` types and `<textarea>`.

The JavaScript file automatically seeks input elements in the HTML that match the ID of fields inside the template. Adding the `id="field_1"` to the <input> element inside the HTML template is all that is needed for the two-way communication to be set up since a field with ID 50 was added above. An unlimited number of these binding fields can be established in the exact same way, since they are mapped via ID.

Tip: Use the **Hidden fields** setting inside the HTML panel settings to prevent two editors for the same field being visible at the same time.

Jump to Preview Point



Info: See *samples/html_panels/jump_preview/jump_preview.html.*

From custom HTML panels and full HTML templates, it is possible to force the preview to refresh by jumping to a given preview point. In payloadhosting.js this is done by the method:

```
vizrt.payloadhosting.jumpToPreviewPoint(preview)
```

The *preview* parameter is a string with the name of the stop point or tag in the scene. Usually, the default preview point is called “pilot1”, and if an empty string is given to the function, preview jumps to this point.

The Loaded and Created States in Initialize

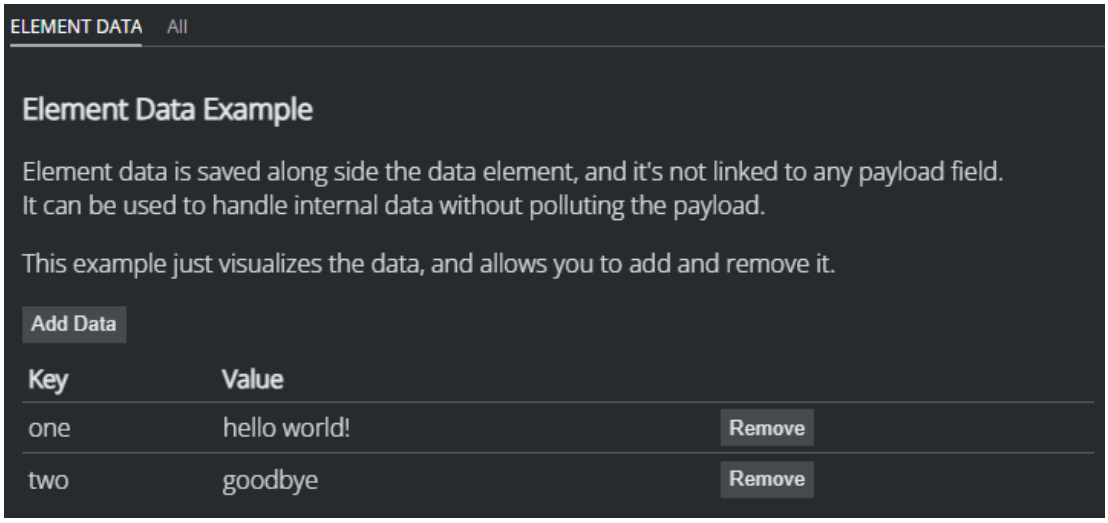
When setting up the connection between the custom HTML template and Viz Pilot Edge, the *vizrt.payloadhosting.initialize()* method needs to be called. In the callback function given in the parameter to this

method, the payload is loaded ready to be accessed by custom HTML scripting. It is useful to know whether the recently opened payload is a newly created data element, or a reopened data element. This is given in the state parameter given to the callback. This is an example of the usage of initialize states:

```
<html>
<body onload="vizrt.payloadhosting.initialize(callback)">
  <script
    src="./payloadhosting.js"></script>
  <pre id="logs"></pre>
  <script>
    function callback(state, elementData) {
      document.querySelector("#logs").innerText += "Payload ready: " + state
    }
  </script>
</body>
</html>
```

In Template Builder the state is **undefined** as these events/states do not apply to templates in Template Builder, they only apply to payloads of data elements.
In Viz Pilot Edge, the state is **created** for newly created data elements, and **loaded** for re-opened data elements.

Storing Element Data



Info: See */samples/html_panels/element_data/element_data.html*.

Occasionally, custom HTML templates need to store auxiliary data that should remain private to the data element and not be included in the payload itself. This might involve UI states that need to be reinstated when the data element is reopened. This data is not sent to the preview server or the playout system.

A special associated data object is stored in the database, alongside the data element and can be accessed using the `initialize` callback:

```
<html>
```

```
<body onload="vizrt.payloadhosting.initialize(callback)">
  <script
    src="./payloadhosting.js"></script>
  <pre id="logs"></pre>
  <script>
    function callback(state, elementData) {
      document.querySelector("#logs").innerText += "My custom data: " +
elementData.mykey
    }
  </script>
</body>
</html>
```

The `elementData` parameter is a standard JavaScript object with string properties used as key-value pairs. This allows flexible storage and retrieval of custom data tied to the data element.

Setting Element Data

To assign values to this data object, use the following method:

```
vizrt.payloadhosting.setElementData("mykey", "myvalue")
```

Connecting a Custom HTML Template to a Viz Pilot Template - Advanced

The example below goes into more detail than Connecting a Custom HTML Template to a Viz Pilot Template, and uses more scripting, to give you 100% control over the template. The three files mentioned in the Setup a Simple Custom HTML Template example are also used here.

Creating a List of Functions Where You Can Bind Fields

By adding the following above the *document.ready()* function in the *customTemplate_sample.js* file:

```
// Will be called when the field with id "50" changes
function on50Changed(value) {

}
```

And the following inside the $(document).ready function:

```
var pl = vizrt.payloadhosting;
pl.initialize();
pl.setFieldValueCallbacks({ "50": on50Changed });
```

You set up a way for a custom JavaScript function to be called upon detecting a change. When field_50 receives a change from the host, the function is called with its new value as a parameter.

Some changes are made to the HTML file below to demonstrate that we can use custom HTML/JavaScript to do something with these values.

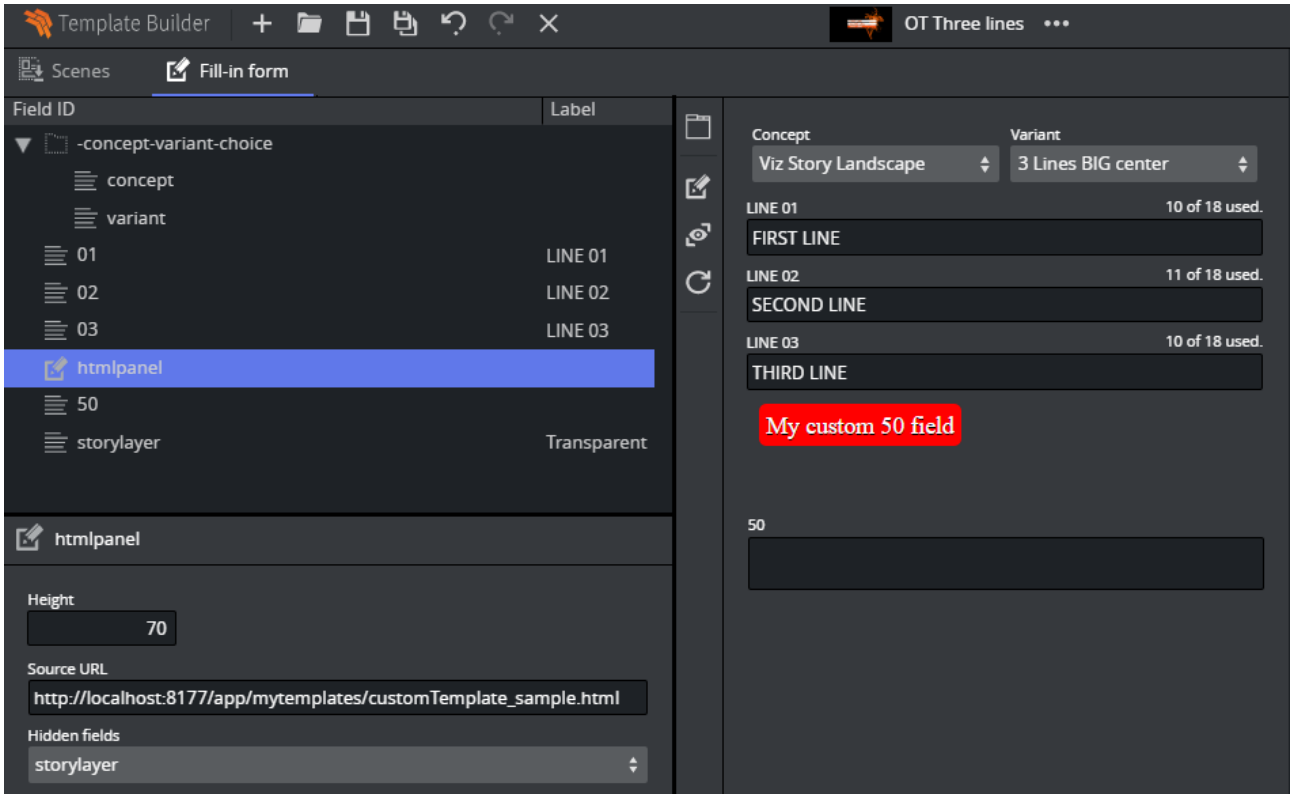
Inside the HTML file, the entire body is replaced with:

```
<body>
    <span id="myfield" class="sample red">My custom 50 field
</body>
```

To add some CSS to style the text, add the style tag after the closing </head> tag and before the <body> tag:

```
<style>
     .sample {
          padding:5px;
          color:white;
          border-radius:5px;
          text-shadow:0 1px 0 black;
          background:red;
    }
   .green {
          background:green;
    }
</style>
```

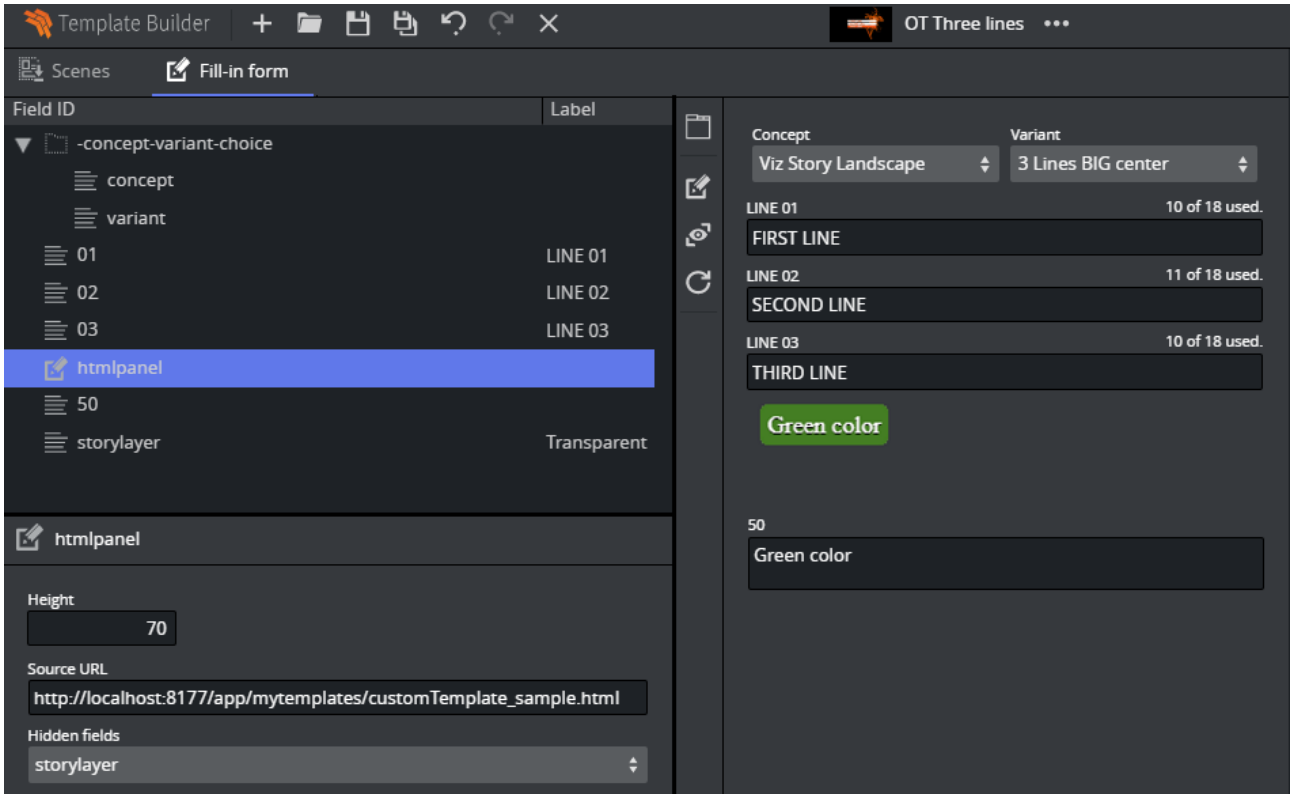
This provides the following output in Template Builder:



Adding a bit more custom logic, making the background color green when there is a text value that is longer than five or shorter than 20 characters. The function is expanded by adding the following function:

```
function on50Changed(value) {
     var myField = $("#myfield");
     myField.text(value);
     if (value.length > 5 && value.length < 20) {
              myField.addClass("green");
     } else {
              myField.removeClass("green");
     }
}
```

After refreshing the HTML panel, the background color should change to green dynamically when typing.

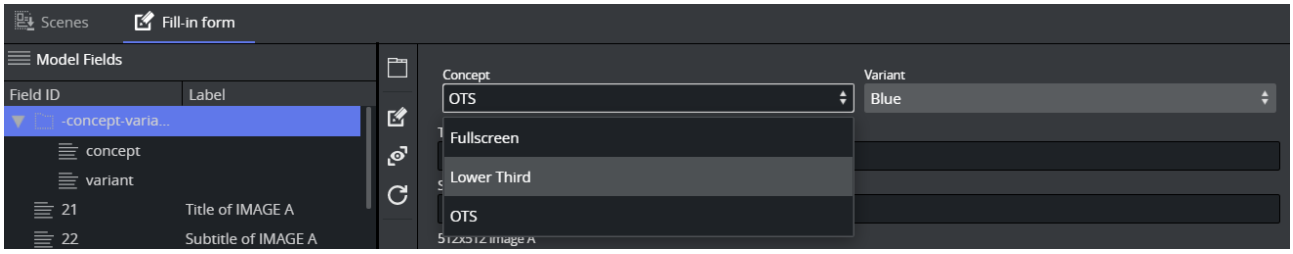


Redesigning Concept/Variant Fields

This example shows how to present the concepts and variants in a template in a different way.

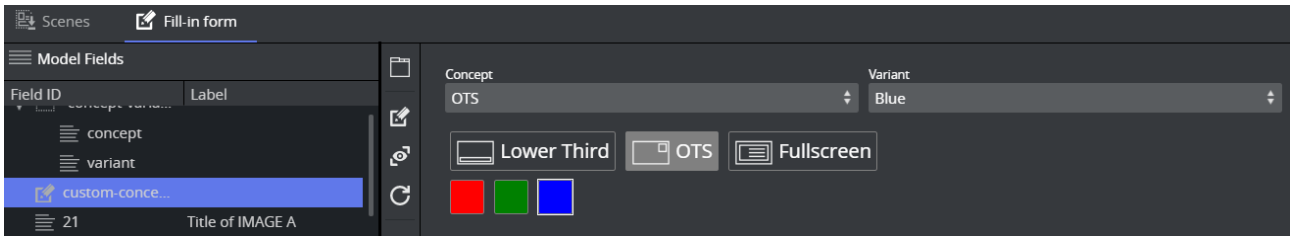
The full HTML / JavaScript code is available at *http://<pilotdataserverhost>:8177/app/templatebuilder/samples/html_panels/concept_variant.*

Consider a template with concepts **Fullscreen**, **Lower Third**, **OTS** and variants **Red**, **Green**, **Blue** available as drop-down lists in the Fill In Form:

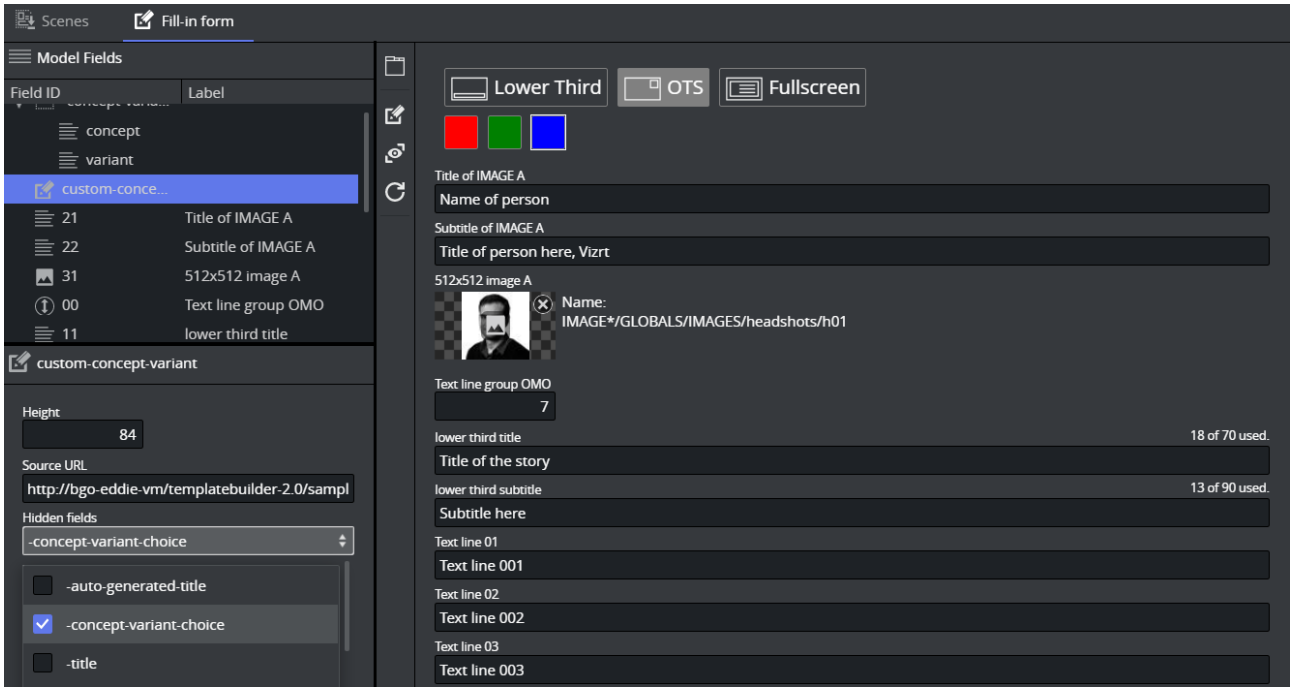


The field **-concept-variant-choice** actually contains 2 subfields, **concept** and **variant**.
You can access their value using slash "/" to navigate in the list. For example, to access the concept use **-concept-variant-choice/concept**.

By setting up the HTML panel hosted at *http://<pilotdataserverhost>:8177/app/templatebuilder/samples/html_panels/concept_variant*, the concepts and variants are now presented as buttons.
This example has mutual binding support for both concept and variant. Clicking on the new buttons updates the original drop-downs and vice versa:



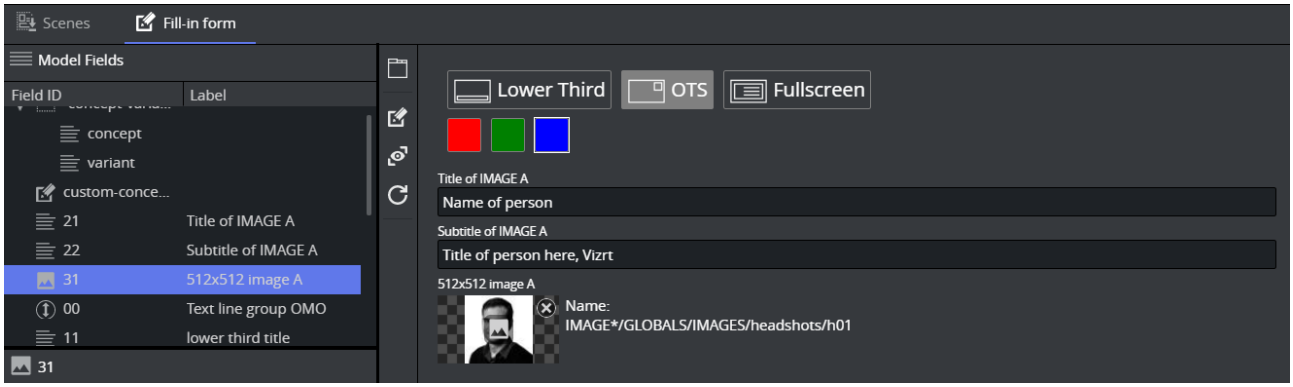
The drop-down lists are no longer needed and can be set as a **Hidden field** in the HTML panel properties window:



Visibility and Read-Only

It is possible to dynamically set visibility and read-only attributes, so you can filter the auto-generated form based on the custom HTML template. In the following example, the **31** image field should only be visible when the **Fullscreen** or **OTS** concept is active:

> **Note:** Visibility and read-only properties of fields can also be controlled from the Read-only and Hidden-expressions in the field properties.



In the JavaScript used in the example, there is a function called *updateActiveConcept* which is called when the concept changes.

Adding the following line inside the updateActiveConcept method block, it checks which concept is chosen. If it isn't **Lower Third**, it displays the field with ID **31** in the Fill In Form:

```
pl.setFieldVisibility("31", conceptValue != "Lower Third");
```

If you now click on the **Lower Third**, the image field with ID **31** disappears, but is displayed if the **OTS** or **Fullscreen** concept is selected.

Sub Fields

Sub fields can be addressed by using a " / ", which is read as **field/subfield**.

When using duplets or triplets, the example below shows how you are able to control the value of *image_scaling* by using *1/image_scaling*.

```
vizrt.payloadhosting.setFieldText("1/image_scaling", "1 2 3")
```

A `fieldPath` is a string that describes the location of a field in the payload.

- **Top-level field**: `"myList"` (if `myList` is a list field).
- **Nested field**: `"container/myList"` (if `myList` is inside `container`).
- **List item reference**: `"myList/#0"` (first item in `myList`).
- **Field inside a list item**: `"myList/#2/name"` (the `name` field inside the third item of `myList`).

4.4 Environment Variables

In Template Builder there are some built-in system environment variables, and it is also possible to define your own environment variables through URL parameters to the application, or specifying these in optional settings in DataServerConfig. If defined by URL parameters, these must also be added to the Viz Pilot Edge URL when templates and data elements are using these variables. If a variable is used, but not defined, it is possible to set a default value.

4.4.1 Defining Environment Variables

The application defines the following **application specific environment variables**:

Variable	Value
$APP	"TemplateBuilder" or "PilotEdge".
$PDS	The base URL of Pilot Data Server that the application is connected to.
$GH	URL to the configured Graphic Hub for scenes.

URL Parameters

It is possible to define **custom environment variables** to be used in the templates. These variables can be specified as URL parameters to the application. Environment variable in the URL must use the format *$var=value*, where the name of the variable must start with a dollar sign and the value is set after the equal sign. There can be multiple environment values specified. For example, if the variable *version* can be specified like this in the URL to the application http://mypds:8177/app/templatebuilder/templatebuilder.html?$version=v1, then the value of the variable *version* can be used when specifying URLs to HTML panels, in feed URLs, expressions and auto title formatting.

> **Info:** This is an example of how to define a variable called *version* in your URL: http://mypds:8177/app/templatebuilder/templatebuilder.html?**$version=v1**.

Optional Setting in DataServerConfig

Custom environment variables can also be configured in **DataServerConfig**, by adding optional settings fields with names prefixed by `env-`. For example, a setting named `env-myserver` with value `https://api.example.com` is available as the environment variable `myserver` (usable in URL placeholders as `{$myserver}`). These are handy for configuring spell checker dictionary base path and language.

4.4.2 Using Environment Variables

In general, the notation for using environment variables is *{$var|default}*, where *var* is the name of the variable and *default* is the default value if the variable is not defined.

When using an environment variable in a URL, the URL must be prepended with "{$}" if it does not start with an environment variable.

See the examples below:

Scenario	Example	Description
URL to HTML panel or full HTML page	**{$}** http://mywebserver/templatepanels/**{$version|v1}**/mypanel.html	The full URL used by the application is http://mywebserver/templatepanels/v1/mypanel.html if the environment variable *version* is not defined in a URL parameter.
URL to a feed of objects from graphic hub	**{$GH}** /files/BAA340C1-C71E-0949-BCF6-F7E043856030/	The {$GH} environment variable contains the full URL to the Graphic Hub and certifies that if the system is configured to point to another Graphic Hub, the link still works.
Inside HTML fragments	<img src="**{{$PDS}}**/pilotedge-3.0/vizPilotEdgeLogo.svg">	Note the double brackets around the variable.
Inside HTML fragments	{{$APP}} {{$PDS}} {{$GH}}	Writes out the application specific environment variables.
Auto generated title format	{field1} / {field2} (**{$version|DEV}**)	Concatenates the values of field1 and field 2 and the word DEV, if *version* is not defined in a URL parameter.
In expressions	{{$APP}} ==="TemplateBuilder"	Note the double brackets. This can hide a field if the template is opened in Template Builder, when added to the "Hidden expression" for a field in the field editor.
In scripting	`const value = vizrt.$env["headline"]`	Example URL: http://pds-host/pilotedge.html/?%24headline=Spurs+wins+Premier+League&template=25480

```
// Minimal scripting example. This code will open a template in Pilot Edge and fill
the
// headline field from the "headline" environment variable in a URL like
// http://pds-host/pilotedge.html/?$headline=Spurs+wins+Premier+League&template=25480
vizrt.onCreate = () => {
    fillHeadline()
}

function fillHeadline (): void {
```

```
    const value = vizrt.$env["headline"]
    if (value !== undefined) {
        vizrt.fields.$headline.value = createRichText(value)
    } else {
        console.log("No value set for headline");
    }
}
```

4.5 Custom Execution Logic

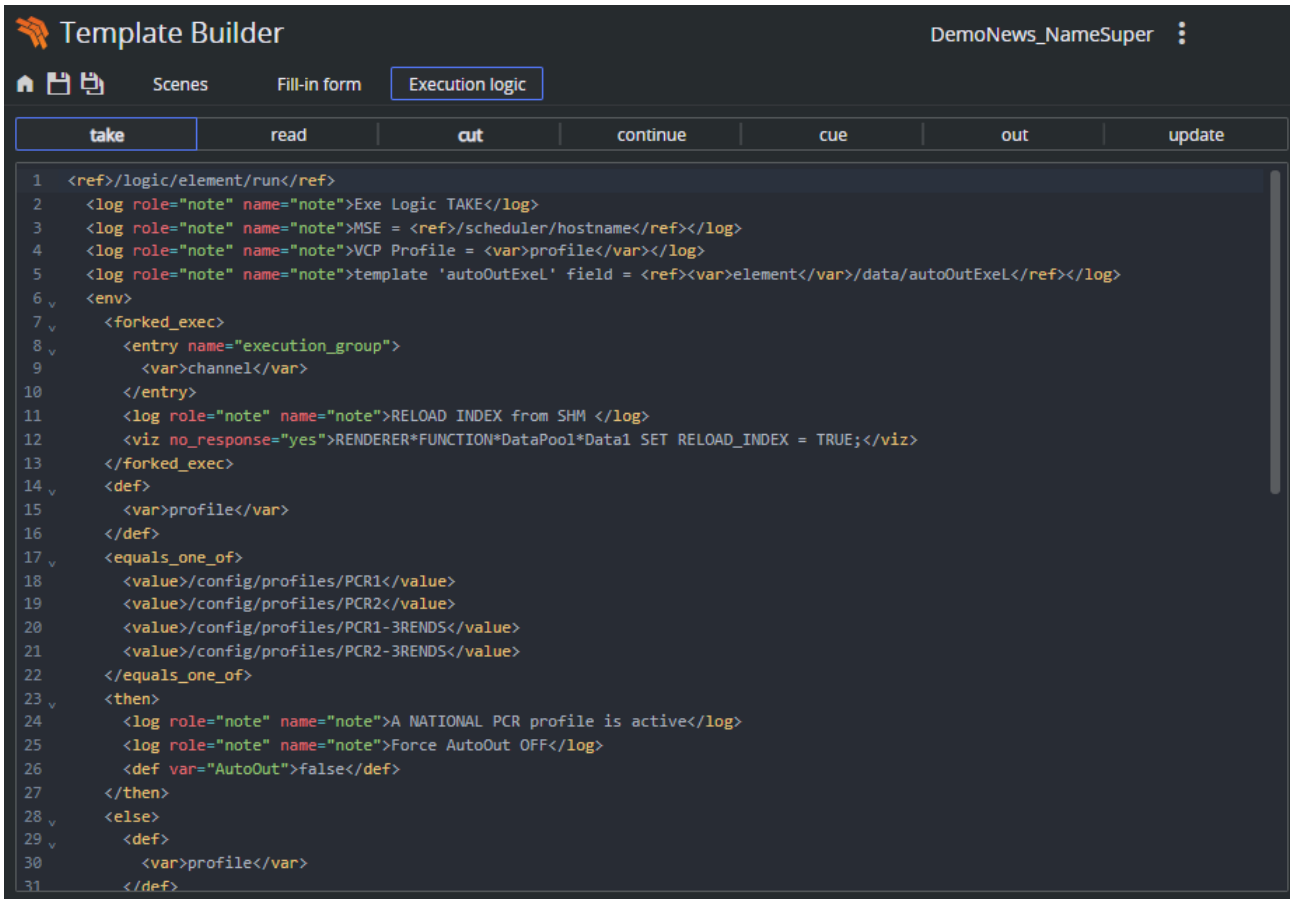
Custom Execution Logic is a way to specify Media Sequencer behavior for different actions (take, read, out, continue, cue, out and update). The "script" logic is expressed in the Media Sequencer's own internal VDOM logic through a simplified XML format.

> **Note:** Custom Execution Logic is a powerful, but also potentially complex mechanism that requires **expert level** understanding of Media Sequencer's internal working.

Execution Logic commands are saved as part of a template. This allows data elements based on the template to use the same execution logic commands. For example, adding an execution logic script to Media Sequencer's command *Take*, replaces the *Take* command for all data elements based on that template. A benefit of using Execution Logic is that the script can be run without the need for external services or clients to be open. It is possible to use a limited set of commands, or any Viz Engine command, straight on Media Sequencer to issue instructions.

This section contains the following topics:

4.5.1 Execution Logic Editor



The Execution Logic editor consists of two parts, a list of commands and an XML editor for the MSE commands. In addition there is a status bar displaying syntax errors.

- **Command list:** The top toolbar displays the available commands. If a command is written in bold, it has custom logic inside. It is possible to implement some commands but not all. The commands left empty behave as normal. It can also be useful to prevent some commands, then inserting a log line only logs and does not execute the default action. For instance `<log>CUE not supported.</log>`
- **Command editor:** Displays the currently selected command and its execution logic. Available context menu option is *Insert Default Action*.
  - **Insert default action:** Inserts the default command `<ref>/logic/element/run</ref>`.

4.5.2 Working with Execution Logic

This section contains the following topics:

- Send basic commands
- Example - Play, Continue, Take Out
- Example - Forked Execution
- Example - Commands generated by a template

Send Basic Commands

The `<viz>` handler is used to send commands to Viz Engine.

The following example sends the `RENDERER SET_OBJECT SCENE*...` command to channel “A” in the current profile:

```
<env viz="A">
    <viz>RENDERER SET_OBJECT SCENE*...</viz>
</env>
```

Multiple commands can be sent by separating each command with `<br/>`, for example:

```
<viz>RENDERER SET_OBJECT ...<br/>RENDERER*STAGE START</viz>
```

Instead of setting the command directly, a more powerful approach is to use the contents of a field in the template. The field (a hidden textbox for instance) can then be filled with the Viz commands that need to be sent.

This example shows how the contents of a data field in a data element can be retrieved by using the `<ref>` construct (`"field_01"` is the ControlObjectName of the data field):

```
<env viz="A">
    <ref><var>element</var>/data/field_01</ref>
</env>
```

To send commands to several channels, duplicate the command:

```
<env viz="A">
    <ref><var>element</var>/data/field_01</ref>
</env>
<env viz="B">
    <ref><var>element</var>/data/field_02</ref>
</env>
```

Note: Forked execution is required when a channel contains multiple engines, otherwise the commands only applies to the first engine in a channel.

Note: Execution logic does not work with non-control object based templates.

Example - Play, Continue, Take Out

This example shows how execution logic can be used to play an element, do a *Continue* after five (5) seconds, and then a *Take Out* after ten (10) seconds.

In the Execution Logic Editor, select the “take” command, add the logic into the editor (right pane). This means that when a “take” is issued on a data element based on this template, Media Sequencer executes the logic.

The commands are modified to do a `"take","continue"` and then an `"out"`. The timecode for each operation must be set.

```
<relative>
    <env command="take" timecode="00:00:00:00">
        <ref>/logic/element/run</ref>
    </env>
    <env command="continue" timecode="00:00:05:00">
        <ref>/logic/element/run</ref>
    </env>
    <env command="out" timecode="00:00:10:00">
        <ref>/logic/element/run</ref>
    </env>
</relative>
```

Note: The running/outer context takes precedence over attributes on the “ref”. Instead of adding attributes on the “ref” node, you use an “env” node as in the example above.

Example - Forked Execution

These examples show how the "take" command can be modified to make the template override the standard logic and instead send `RENDERER*STAGE START`.

Here, the command is sent to the channel assigned to the data element:

```
<forked_exec>
    <entry name="execution_group"><var>channel</var></entry>
        <viz>RENDERER*STAGE START</viz>
</forked_exec>
```

To send commands to a specific channel in the current profile, replace `<var>channel</var>` with the name of the channel you want to send to, as follows:

```
<forked_exec>
    <entry name="execution_group">MY_CHANNEL</entry>
        <viz>RENDERER*STAGE START</viz>
</forked_exec>
```

Example - Commands Generated by a Template

By using the information from the preceding examples, we can create logic that sends custom Viz commands that are generated by the template.

One way would be to add a Value Control Component (see **Viz Pilot > Template Wizard Components > Viz Control Components**) to the template, and set the ControlObjectName to `"vizcmds"`. Then create a regular script that sets the ControlValue of the TWValueControl to whatever command needs to be sent.

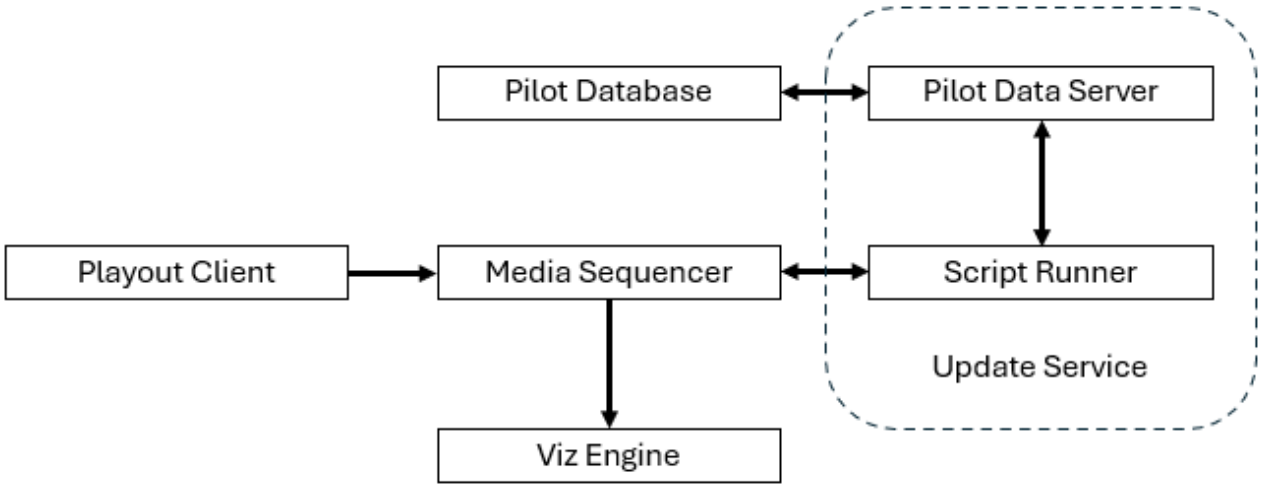
Alternatively, use a standard memo box, and set the ControlObjectName to `"vizcmds"`. Then enter the Viz commands (or script what the contents should be). The memo box's visibility can be set to false so the user can't see it. In the Execution Logic you can then add the following:

```
<forked_exec>
    <entry name="execution_group"><var>channel</var></entry>
    <viz>
        <ref><var>element</var>/data/vizcmds</ref>
    </viz>
</forked_exec>
```

4.6 Update Service

Update Service allows you to update fields in a data element right before the graphics is taken on air, or during on air time. This process occurs on the server side, and is typically used for updating scores, stock prices, statistics, etc. The script being run is invoked by **Media Sequencer** on different actions: **take**, **read**, **cue** and **update**. The script being invoked receives a copy of the fields in the data element (the payload), it can manipulate the fields and return a modified payload. The modified payload is then used by Media Sequencer when sending playout commands to the Viz Engine. The modified payload is not saved in the Pilot database. There are three ways to invoke an update service:

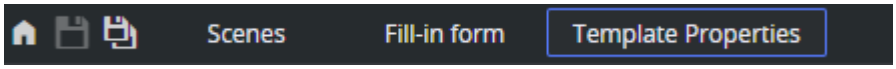
1. Using the Vizrt Pilot Update Service to execute JavaScript templates written in Template Builder.
2. Implementing custom script runner on an external web server.
3. Using the **VBScript** script runner in Pilot (for templates created in Template Wizard).



This section contains the following topics:

4.6.1 Enabling Update Service in a Template

Enter the Template Properties page:



The **Update Service** section has four available options:

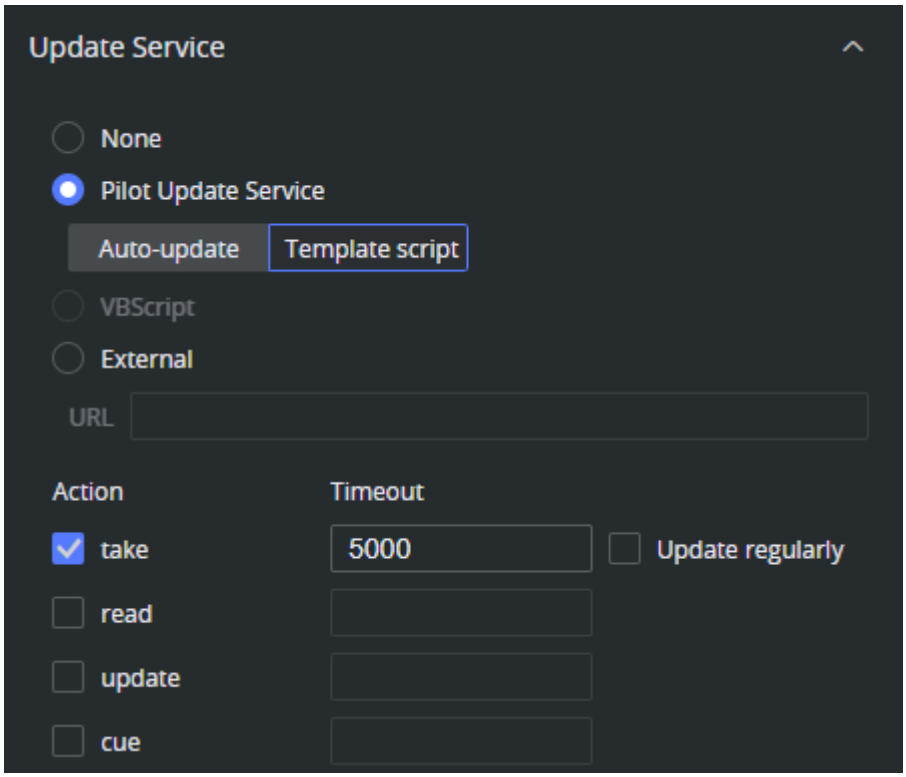
- **None:** No update service is used.
- **Pilot Update Service:** Lets the Pilot Update Service either **auto-update** the template, or run the **onUpdate** event handler in the internal template script (see below). In this case, the JavaScript in the onUpdate event is executed on the server side by the Viz Pilot Edge Script Runner.
- **VBScript** (legacy): Enabled if the template is created with Template Wizard and the template contains a VBScript.
- **External:** Use this to specify a URL to an external update service.

4.6.2 Pilot Update Service with JavaScript in Template Builder

When an update script is enabled for a template, the Media Sequencer representation (VDOM) of the master template contains a *<live_update>* node. This node contains URLs to the update service responsible for updating the payload. The result is put into VDOM and sent to the Viz Engine, but not saved back to the Viz Pilot database.

Viz Pilot Update Service

The Viz Pilot Update Service is automatically installed together with Pilot Data Server. Make sure this service runs and is responsive and reachable from Media Sequencer. The default port is 1991 for HTTP and 1992 for HTTPS. This service executes the **onUpdate** part of the template script. To determine when the script runner should be invoked by Media Sequencer and control the behavior, the following properties can be set in the Update Script section of Template Properties:



- **Pilot Update Service:** Determines that the Pilot Update Service should be used to update data.
  - **Auto-update:** Lets the Update Service refresh all field links to external data with the latest content. No script is executed.

    - **Template script:** The onUpdate() in the internal template script should be executed by the Update Service.
- **Action:** Determines on which Media Sequencer actions the script should be invoked.
    - **take:** To get the graphics on air.
    - **read:** Used to preview the graphics on an external Viz Engine (Viz Trio for example).
    - **update:** Explicitly update data in an element being on air in the renderer, without starting in-animations.
    - **cue:** Setting graphics in the first frame of the animation in the renderer.
- **Timeout:** For all actions, specifies the time in milliseconds that Media Sequencer waits for a response from the Update Service, before it continues with the operation.
- **Update regularly:** For the take action, the Update Service can also be invoked on regular intervals while the graphics are on air. The interval is decided by the *live_update_interval* database parameter. The default and minimum value is 2 seconds.

> ⚠ **Note:** If **Update regularly** is checked, the update script action runs with the **take** action on subsequent updates. However, if **update** is also checked, the action script is called with update as an argument. This means you can differentiate the script behavior based on whether the Media Sequencer action is **take** or **update**.

Auto Update

To auto update the data without writing scripts, the fields to update need to be connected to a feed entry through the field linking. The workflow is:

- The Template Designer links fields in the template to content, through the feed browser or feed-backed dropdown.
- The journalist selects a feed entry and saves a snapshot of the data from the feed.
- On playout, Media Sequencer calls the auto update mechanism in the Update Service, which follows the **self link** in the Atom entry, to download the latest data from the feed.

Example of an Atom Feed Entry with a self link used by Update Service:

```
<entry>
    <id>1832050330902700035</id>
    <link href="https://twitter.com/luna_corgi/status/1832050330902700035"
rel="alternate" type="text/html"/>
    <title type="text">It's finally Friday frens!</title>
    <content type="text">It's finally Friday frens! I'm going to enjoy this beautiful
day. What are your plans for today?</content>
    <summary type="text">twitter</summary>
    <media:thumbnail url="http://myserver:9090/data/media/v1/https/flowics-
server.com/_vXH9hj3mlVUK4VowdEqA5CyjYA_C3iWkuc_Di26JEg.jpg"/>
    <author>
        <name>Luna the Corgi@luna_corgi</name>
        <link href="http://myserver:9090/data/media/v1/https/flowics-server.com/
_vXH9hj3mlVUK4VowdEqA5CyjYA_C3iWkuc_Di26JEg.jpg" rel="image"/>
    </author>
    <contributor>
```

```
        <name>twitter</name>
    </contributor>
    <link href="http://myserver:9090/data/media/v1/https/pbs.twimg.com/
ZNEhXojq54WQHSbxjd6pl8uXD4U4CKryXl8zE1tPTkg.jpg" rel="image" type="image/jpeg"/>
    <published>2024-09-06T13:36:51.000+00:00</published>
    <updated>2024-12-17T10:46:07.164Z</updated>
    <link rel="self" type="application/atom+xml;type=entry" href="http://my-feed-
server:1305/data/images/1.entry.xml"/>
</entry>
```

See also Data Entry - Using Feed Browsing.

Adding a Script in Template Builder

To add a script that is invoked by the Media Sequencer's Update Service mechanism, a special event needs to be implemented in the internal template script editor: **onUpdate**. This event is never called from within Template Builder or Viz Pilot Edge, it is only executed by the Viz Pilot Edge Script Runner service when Media Sequencer invokes it. The mechanism is invoked by Media Sequencer only if it is enabled in Template Settings. Make sure the **Pilot Update Service / Template script** option is enabled in the Update Service section in Template Settings (see above). In its simplest form, an update script can look like this:

```
vizrt.onUpdate = (fields, action) => {
  fields["$02-Designation"].value = `${fields["$01-Name"].value} live from Svalbard`
}
```

The fields object contains a list of the fields of the payload and can be accessed as one normally accesses this object in Template scripting. Only the field values are available and can be manipulated (and not UI properties like visibility for instance). The (manipulated) fields object is automatically returned to Media Sequencer by the Viz Pilot Edge Script Runner.

Script Technology

The script language in the template editor is TypeScript. All JavaScript is valid Typescript, so you can choose what you want inside the editor, but the syntax highlighting and the error checking expects Typescript. The script inside the editor is automatically compiled into JavaScript and stored both as Typescript and JavaScript inside the template. When a template script normally executes in the browser, it is executed through the browser's Web API.

For Update Service scripts, it is slightly different. The script code executed inside the onUpdate method has a different execution context than the rest of the template script. While the rest of the template script executes in the web browser's JavaScript environment, the onUpdate part of the script executes on the server side through a Node.js process. Both execution environments support the JavaScript Core Language, but they serve different purposes and have different sets of APIs.

Warning: Even if syntax highlighting inside **onUpdate** reports no errors in the template script editor, the code can still cause errors when executed on the server side due to being executed inside a node.js environment, and not through the browser's JavaScript API. Make sure to configure Viz Pilot Edge script runner and test the script through the Update Service.

Example - Async Fetch from REST Source

A more realistic example is where the script fetches some data from an external source and uses that to update the fields. A typical example is updating live scores and sports results. This can be done with a JavaScript fetch command and making the method marked as async. Here done only on *take* action in Media Sequencer (right before the graphics go on air):

```
interface Team {
  name: string;
  score: string;
}

interface GameData {
  teams: Team[];
  current_quarter: string;
  game_clock: string;
}

// doUpdate() can be called both from the template itself and from
// Pilot Update Service.
async function doUpdate(fields: Payload, action: string) : Promise<void> {
  try {
    const response = await fetch('http://myscore-host:3000/app/score.json');

    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    const data: GameData = await response.json()

    // Log the JSON object as a string
    console.log(JSON.stringify(data, null, 2));

    processData(data, fields)

  } catch (error) {
    // Handle errors (e.g., network issues, JSON parsing errors)
    console.error('Error fetching data:', (error as Error).message)
  }
}

vizrt.onUpdate = async (fields, action) => {
  console.log("onUpdate is called")
  await doUpdate(fields, action)
}

// Method to process JSON data
function processData(data: GameData, fields: Payload) {
  console.log("processData is called")

  // Extract the current quarter and game clock
```

```
  const currentQuarter = data.current_quarter
  const gameClock = data.game_clock

  // Extract the scores for each team
  const [homeTeam, awayTeam] = data.teams
  fields["$102"].value = homeTeam.score
  fields["$202"].value = awayTeam.score
  fields["$002"].value = currentQuarter + " - " + gameClock
}

// Simulate the "take" action update script in the browser
vizrt.onClick = async (name: string) => {
  await doUpdate(vizrt.fields, "take")
}
```

The code above expects this JSON structure:

```
{
  "teams": [
    { "name": "CSU", "score": "21" },
    { "name": "Longwood", "score": "10" }
  ],
  "current_quarter": "2nd quarter",
  "game_clock": "3:45"
}
```

Interesting points about the code above:

- In the code example, the **OnUpdate** script can be tested inside the browser (client side only) as we added a button and a click event to manually execute the **doUpdate** method. This requires the code inside onUpdate to run in the browser and in node.js. To add a button to test the script:
    - Add a new template Tab.
    - Add an inline HTML fragment.
    - Add the following HTML:

      ```
      <button vizrt-click-name="updatebutton">TEST UPDATE</button>
      ```

- We use the async **fetch** method to get data from an external host, and we convert it to JSON. We can use fetch because both the browser's **Web API** and **node.js**, contain this method and the signature is the same. But this is not something that can be taken for granted. When using functionality outside the JavaScript Core language, please check that both the browser's Web API and node.js implement the functionality.
- A benefit of using fetch in the onUpdate run by the server side in node.js, is that it handles headers differently (for example, **CORS** issues do not occur).
- If the **onUpdate** method contains API calls that are not compatible with the browser's Web API but only with node.js, the update script needs to be tested by a proper server side execution.
- Pay special attention to the **async** method declaration and use of **await**. An async function returns a promise, and using await pauses execution until the promise resolves. This makes it possible to write script code that executes synchronously and returns with a value. The onUpdate code executed by the Pilot Update Service depends on processing everything before returning with the new payload, and forgetting an

await if the method contains asynchronous code like a **fetch** leads to undefined behavior, as the script returns with data before the processing is done.

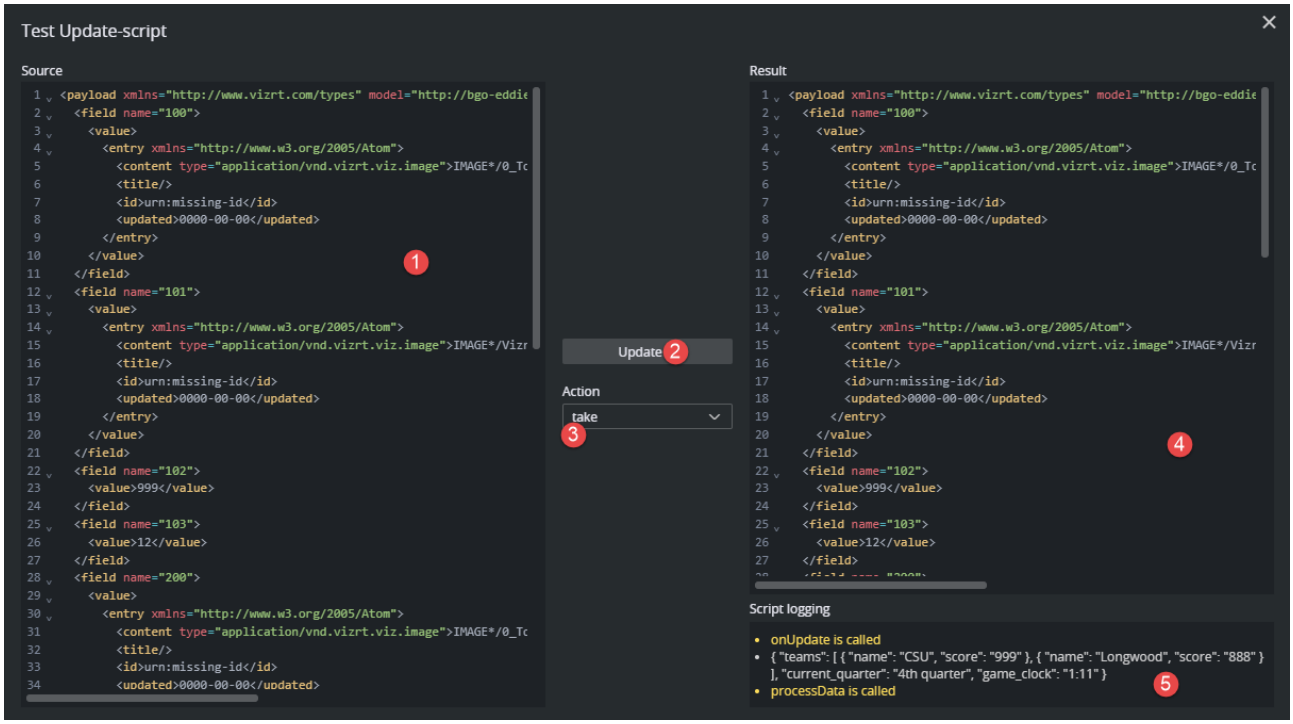
- In onUpdate, the event handler works with a copy of the fields in the payload. Notice that the code executed in onUpdate does not change the template in Template Builder, nor invoke any onchanged events.

Testing the Script

To test a proper server side execution of the update script:

- Save the template.
- Click the test update script-button in Template Builder  .

A new window opens, allowing you to test the script by doing a POST to the script runner URL with the source data on the left, and the returned VDF payload on the right.



1. The left window contains the VDF payload of the current data element.
2. When clicking **Update** the REST end point in Pilot Update Service (as configured in the Pilot Data Server Launcher) is called with the URL to the current data element as a parameter. Pilot Update Service then loads the template script from the Pilot Data Server, executes onUpdate and returns with a new, updated payload.
3. The action parameter simulates different actions performed by a control client on the data element. This action is sent as an argument to the update script and makes it possible to differentiate between actions in the script.
4. The right panel contains the response from the Pilot Update Service. To check whether it performed the desired updates, look for the field values it was supposed to manipulate inside the payload. Note that the current data element in Template Builder is not updated, nor is the preview. The only result of the server side script execution is the VDF payload in this window.
5. If the script contains logging to console or exceptions, these log messages are returned from the Pilot Script Runner along with the VDF payload and displayed here. This is useful for debugging and testing. There are 4 log types available when logging in script, they are each displayed with different colors in the log window:

```
console.log("LOG")
console.info("INFO")
console.warn("WARN")
console.error("ERROR")
```

4.6.3 External Update Service

It is fully possible to write a custom update service and omit the usage of the template script and Viz Pilot Edge Script Runner. In Template Builder, enter Template Settings for the template and enable the **External** option under **Update Service**.

- Specify the URL to the external update service.
- Specify the Media Sequencer actions that invoke the update service. Only one URL can be specified for all actions.

Implementing a Custom Server

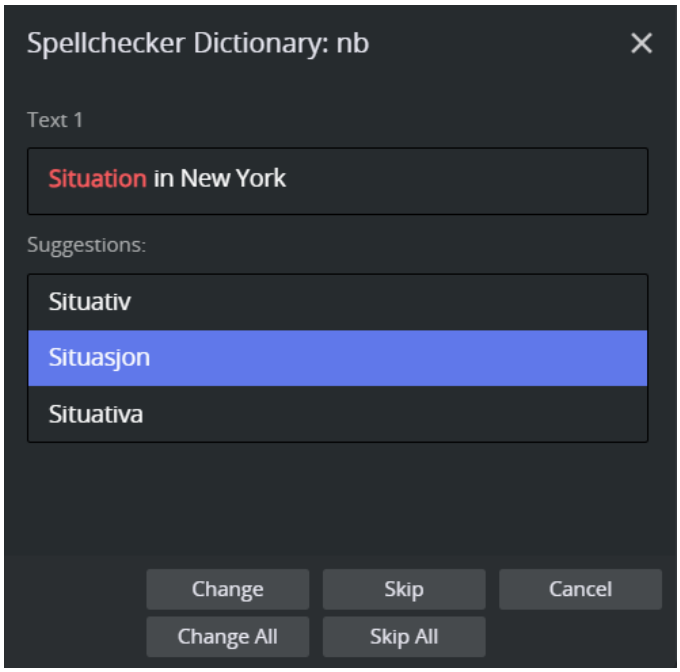
On the server side, to implement a custom Update Service, a web service should respond to the URL specified in the template. The service must accept an HTTP(S) POST request on a given endpoint with a Content-Type of *application/vnd.vizrt.payload+xml*. The service should only modify field values, and not add or remove fields. The service must return an updated payload, or the same payload that was posted if there are no changes. In order to implement an external update service, you need to parse and return a Viz Data Format (VDF) document to Media Sequencer. The document is posted to the external update service by Media Sequencer.

See Also

- For a detailed description on how Media Sequencer handles Update Service scripts, see **Live Update Support** in the Media Sequencer Administrator Guide (go to http://localhost:8580/mse_manual/dispatch_element%20actor.html#live-update-support on the host where Media Sequencer is installed). For a description of the Viz Data Format (VDF), go to the web interface of Pilot Data Server and locate the link to the specification.

4.7 Spell Check

The Viz Pilot Edge payload editor, includes a built-in spell check feature allowing template scripts to check text fields for misspellings. When invoked, a dialog guides the user through each misspelled word, offering suggestions, changes or skips field by field.



4.7.1 Prerequisites: Dictionary Files

Spellchecker uses **Hunspell** dictionaries, the same format used by LibreOffice, Firefox, and many other applications. Each language requires two files with the same base name, but different extensions:

- `.aff` : affix rules file (grammar/morphology).
- `.dic` : word list file.

Where to Obtain

- LibreOffice dictionaries on GitHub: Comprehensive, actively maintained collection and organized by language code.

- Hunspell dictionaries on CGIT: Original Hunspell project.

The language code in the filename must match what you pass to the `spellCheck()` script method (for example, `en_US`, `nb_NO`, `de_DE`).

Placing Dictionary Files

By default, Viz Pilot Edge looks for dictionary files in a folder named *dictionaries* served at the same path as the application:

```
dictionaries/
  en_US.aff
  en_US.dic
  nb_NO.aff
  nb_NO.dic
```

To use a different location, set the environment variable `vizrt-spellcheck-path` to the desired base path or URL. For example:

- *https://cdn.example.com/pilot-dicts*: Absolute URL to a remote dictionary server.
- *./my-dicts*: Relative path on the same server.

The application fetchs `{vizrt-spellcheck-path}/en_US.aff` and `{vizrt-spellcheck-path}/en_US.dic` at runtime.

Environment Variables

Variable	Description	Default	Example
`vizrt-spellcheck-path`	Base URL or path where *.aff*/*.dic* dictionary files are served.	`dictionaries` (relative to the app page)	• *https://cdn.example.com/pilot-dicts*
`vizrt-spellcheck-language`	Default language code used when none is passed to `spellCheck()`.	`en_US`	• *./my-dicts*

These environment variables can either be set in URL parameters to the Viz Pilot Edge plugin, or they can be specified in the Pilot Data Server settings, as optional parameters. For more information, see Environment Variables.

To set these environment variables in Pilot Data Server settings, follow these steps:

- Open **DataServerConfig** (default URL http://pds-host:8177/app/DataServerConfig/DataServerConfig.html).
- Click **Add New Optional Parameter**.
- Add `env-vizrt-spellcheck-path` and `env-vizrt-spellcheck-language`.

4.7.2 Script API

Spell check is available in template scripts via the `pilot.spellCheck()` method.

```
vizrt.spellCheck(languageCode?, fields?)
```

Parameter	Type	Required	Description
`languageCode`	`string`	No	The Hunspell language code to use, for example `"en_US"` or `"nb_NO"`. If omitted, the value from the `vizrt-spellcheck-language` environment variable is used. Falls back to `"en_US"` if neither is set.
`fields`	`string[]`	No	An array of field paths to check. If omitted, all applicable plain-text fields in the template are checked.

Return value: `Promise<boolean>` resolves to `true` when the user completes the spell check, or `false` if the user cancels the dialog.

4.7.3 Examples

Check all text fields using the default language:

```
const completed = await vizrt.spellCheck()
```

Check all text fields in Norwegian:

```
const completed = await vizrt.spellCheck("nb_NO")
```

Only check specific fields:

```
const completed = await vizrt.spellCheck("en_US", ["title", "subtitle"])
```

Respond to cancellation:

```
vizrt.onBeforeSave = async () => {

   const result = await vizrt.spellCheck("nb")
   if (result) {
```

```
     console.info("Spell check completed, changes applied")
    } else {
     console.info("Spell check cancelled by user")
    }
    return result
}
```

4.7.4 Behavior

- The dictionary is **fetched once** per language and cached for the duration of the session. Subsequent spell check invocations for the same language do not re-fetch the files.
- If a dictionary fails to load, an error is shown in the dialog and the spell check cannot proceed for that language.
- Only **plain-text fields** are spell checked. Image fields, numeric fields, and other non-text types are skipped automatically.
- If an explicit list of `fields` is provided and a field path does not correspond to a text field, it is silently ignored.
- The payload is only updated after the user completes the dialog. If the user cancels, no changes are applied.

4.7.5 Custom Dictionary

Organizations can supplement any language dictionary with a custom word list by placing a file named *custom.txt* in the same dictionary folder as the *.aff* and *.dic* files. When a language dictionary is loaded, the spell checker automatically attempts to fetch *custom.txt* from the same base path and adds any words found to the dictionary. The file is fetched once and cached for the session, and it applies to all language dictionaries loaded from the same base path.

File format: Plain UTF-8 text, one word per line.

- Leading and trailing whitespace on each line is ignored.
- Empty lines are ignored.
- Lines beginning with # are treated as comments.

custom.txt example:

```
# Organization-specific terminology
Vizrt
MOS
NRCS
Mosart

# Names
Bjørn
Tullevold
Sigve
Sturegutt
```

If *custom.txt* is not present, the spell checker proceeds normally with no error.

4.8 Testing Templates (Run/Design Mode)

Template Builder operates in two distinct modes, **Design mode** and **Run mode**. Understanding the difference between these modes is essential for building, testing and previewing templates effectively.

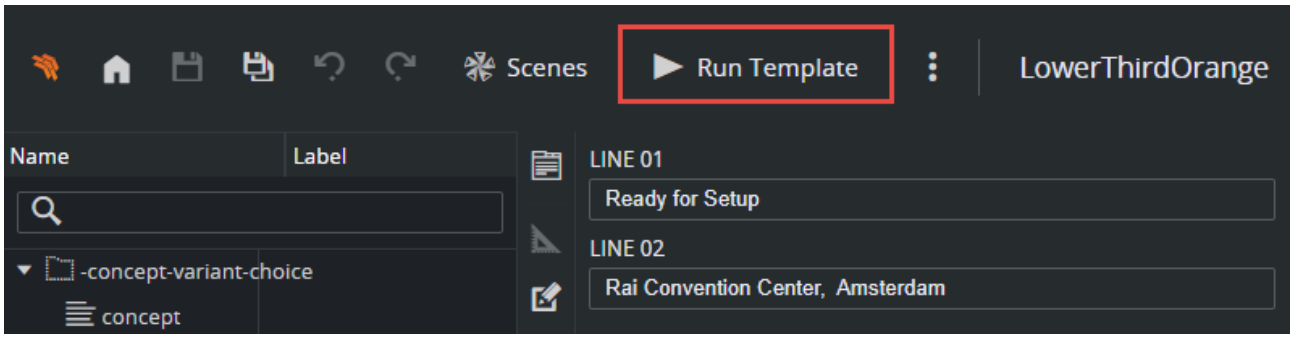
This section covers the following topics:

4.8.1 Overview

Templates in Template Builder often contain **template scripts**, TypeScript code that runs lifecycle events such as `onCreate`, `onLoad`, `onBeforeSave` and `onAfterSave`, that can manipulate field values, control visibility and perform validation.

Run and **design mode** clearly separates the editing phase from the execution/testing phase:

- **Design mode:** The default editing mode. The model, fields and script can be modified without the script running automatically.
- **Run mode:** A simulation mode where the script is active, lifecycle events can be triggered and playout commands can be tested against a real Viz Engine, through Media Sequencer.



4.8.2 Design Mode

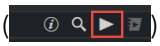
Design mode is the default state when a template is opened. In this mode, full access is available to:

- **Model Editor:** Add, remove and configure fields in the VDF model.
- **Script Editor:** Write and edit template script (TypeScript).
- **Graphics Preview:** A live preview of the template with the current model defaults.
- **Save, Undo, Redo:** All editing operations are available.
- **Template settings**, tag management, scene editing and other configuration.

While in Design mode, the template script is **not running**. The preview shows the template with the model's default values, but no script logic is executed.

> **Note:** The script is still being transpiled in the background during editing (for syntax checking and error reporting), but it is not executed against the model.

Running the Script in Design Mode

The Script Editor panel includes a **Run Script** button () allowing to execute the script *without* leaving Design mode. When activated, the button changes to **Stop Script** (pause icon).

What Happens when Run Script is Clicked

1. The current TypeScript is compiled to JavaScript.
2. The compiled script is inlined into the model (as a data-URI link).
3. The Model Editor and preview now operate with the script active (script-driven changes to field values are applied to the **model itself**).

When to Use Run Script in Design Mode

This is useful when a script manipulates **default values** in the VDF model (for example, setting initial field values, configuring visibility rules, or populating dropdown options). Because Design mode edits the model directly, any changes the script makes to field values become the new **stored defaults** that are saved with the template.

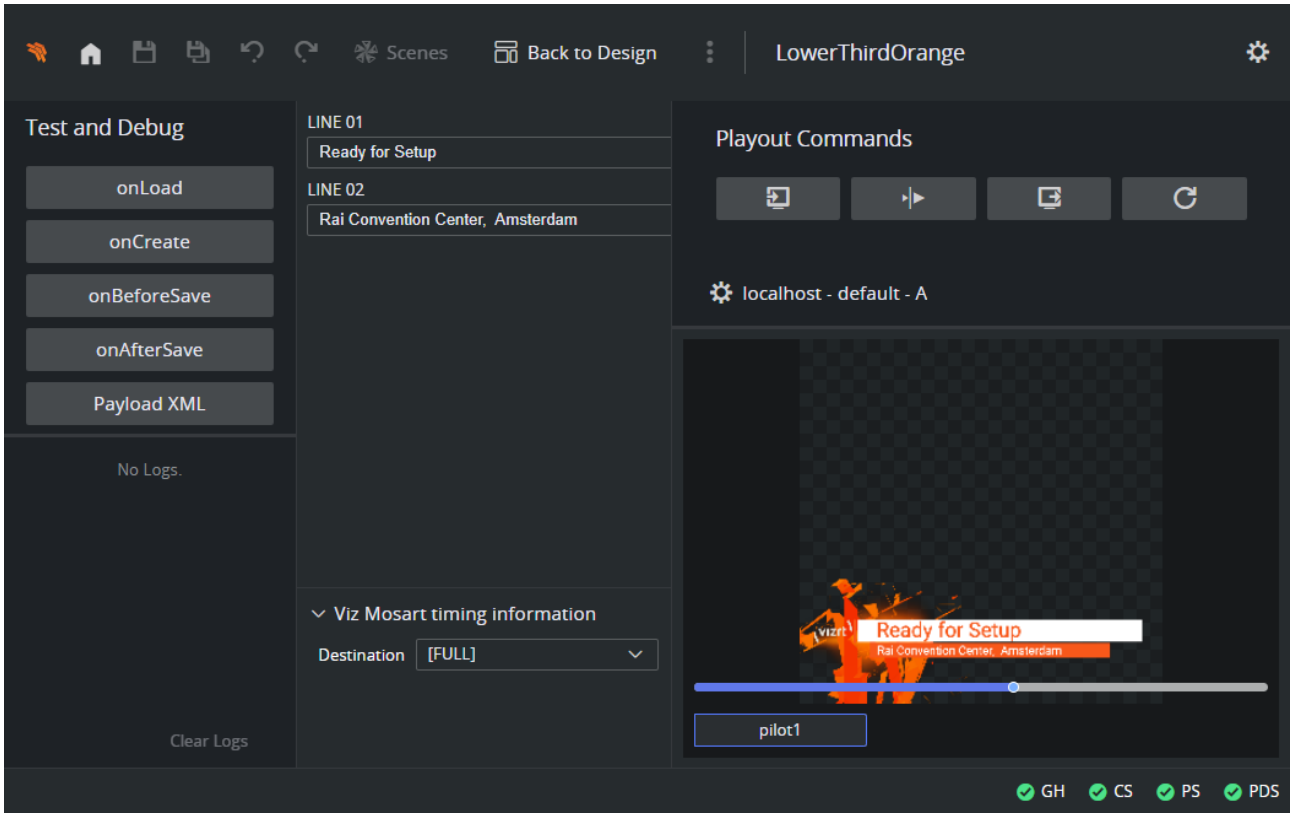
> **Tip:** Use *Run Script* in Design mode when the script should set up default values that persist in the template. Use Run mode to test runtime behavior without altering the saved template.

Auto-stop Behavior

The script is automatically stopped if the model structure is modified or the script source is edited. This prevents unexpected interactions between manual edits and running script logic.

4.8.3 Run Mode

Clicking **Run Template** in the toolbar, switches the template into Run mode. The toolbar button changes to **Back to Design**.



What Happens when Entering Run Mode

1. The template script is compiled.
2. A **payload** is generated from the model's default values.
3. The UI switches from the Model Editor to the **Payload Editor**, the same editor that users interact with in Viz Pilot Edge.
4. The **Test and Debug** panel appears on the left.
5. The **Playout Commands** section appears on the right.

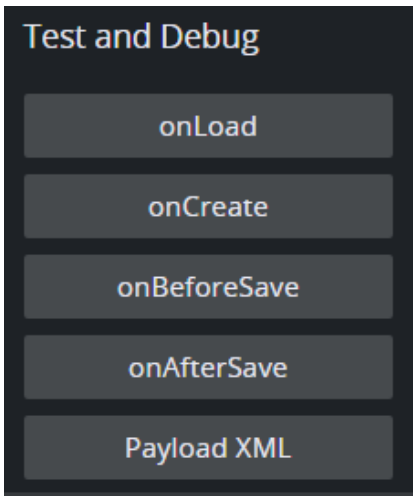
Key Characteristics of Run Mode

- **Payload-centric:** The interaction is with a working payload, just as an operator would see it. Script logic runs against the payload, not the model.
- **Non-destructive:** Changes made in Run mode do *not* modify the saved template model. When returning to Design mode, the model is unchanged.
- **Editing disabled:** Save, Undo, Redo, scene editing and template title editing are disabled to prevent accidental modifications during testing.
- **Custom data is reset:** Transient script data (`$data`) is cleared when entering and leaving Run mode to avoid stale values between sessions.

Important: Any values entered or changes the script makes in Run mode, are **not saved** back to the template. Run mode is purely for testing and simulation. To persist script-driven default values, use the *Run Script* button in Design mode instead.

Test and Debug Panel

The Test and Debug panel is displayed in the left pane when in Run mode. It provides buttons to manually trigger script lifecycle events and utilities to inspect the current state.



Trigger Buttons

- **onLoad:** Triggers the `vizrt.onLoad` handler, simulating a template being loaded from an existing data element. Useful for testing how the script initializes fields when a previously saved element is opened.
- **onCreate:** Triggers the `vizrt.onCreate` handler, simulating a new data element being created from the template. Useful for testing first-time initialization logic.
- **onBeforeSave:** Triggers the `vizrt.onBeforeSave` handler, simulating a save operation. The handler can return `true` (allow save) or `false` (cancel save). A notification appears indicating whether saving would proceed or be cancelled.
- **onAfterSave:** Triggers the `vizrt.onAfterSave` handler. Useful for testing any post-save logic in the script.

Utility Buttons

- **Payload XML:** Copies the current payload as XML to the clipboard. Useful for inspecting the exact payload structure or pasting it into other tools for debugging.
- **Model XML:** Copies the model (with inlined script) as XML to the clipboard.

> **Tip:** The XML copy buttons are helpful when reporting issues or verifying the data structure that would be sent to MSE during playout.

Script Log

The lower half of the Test and Debug panel contains the **Script Log**, a live log that captures all `console.log`, `console.info`, `console.warn` and `console.error` calls made by the template script during execution.

- Log entries are color-coded by severity (info, warning, error).
- Object values are displayed as JSON strings for easy inspection.
- Clicking **Clear Logs** resets the log pane.

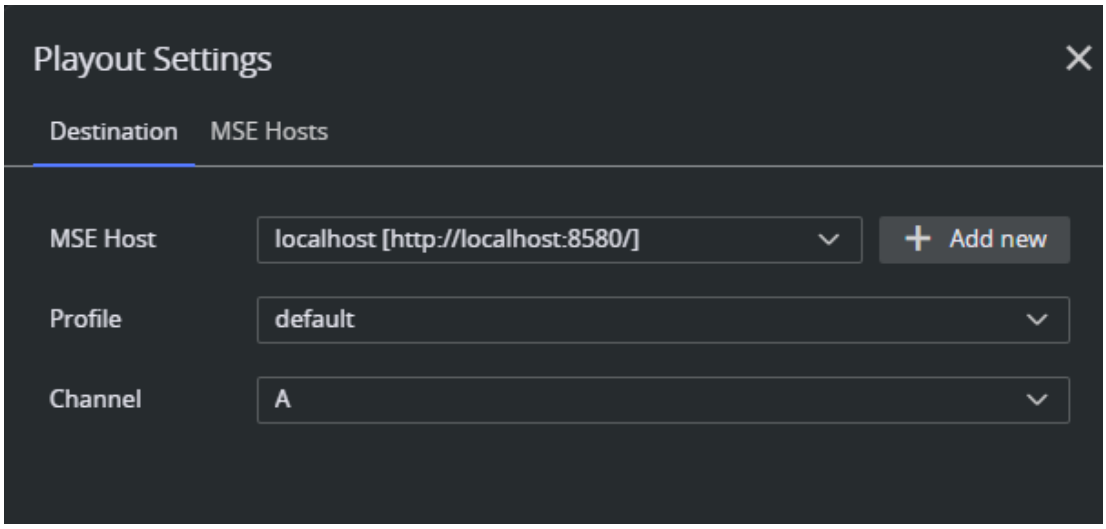
The Script Log is the primary way to debug template scripts. Adding `console.info()` statements in a script, triggering events from the panel, and observing the output makes it straightforward to trace script behavior.

Playout Commands

The Playout Commands section in Run mode allows sending the current template to a Viz Engine via Media Sequencer for playout. This enables testing the template in a production-like environment without leaving Template Builder.

Configuring a Playout Destination

Before using playout commands, a destination must be configured. Clicking the settings button at the bottom of the Playout Commands section opens the **Playout Settings** dialog.



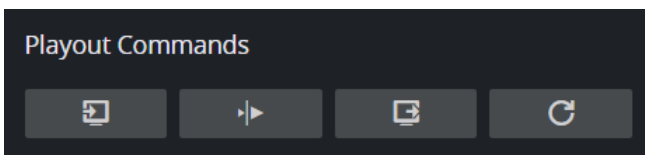
The dialog has two parts:

1. **Host management:** Add, edit, or remove MSE hosts. Each host has a name and a URL, and the connection can be validated to confirm the URL is reachable.
2. **Destination selection:** Select a host, then choose a **profile** and **channel** from the drop-downs. The available profiles and channels are fetched from the selected MSE host automatically.

> **Note:** The configured destinations are persisted in the browser's local storage.

Command Buttons

Once a destination is configured and the template is saved, the following commands are available:



- **Take:** Takes the template on-air on the configured channel. The current payload and model are sent to MSE.
- **Continue:** Sends a continue command to advance the graphic to the next state (for example, the next animation keyframe).
- **Out:** Takes the graphic off-air on the configured channel.
- **Update:** Sends an update to the on-air graphic with the current payload values, without taking it off-air first.

Important: The template must be saved before entering Run mode for playout commands to work. Unsaved changes mean the system cannot construct a valid reference to the template's model on the server, which is required for playout.

4.8.4 Design Mode vs Run Mode

	Design Mode	Run Mode
Primary Editor	Model Editor	Payload Editor
What is Edited	The template model (VDF model), field definitions, structure, defaults.	A working payload generated from the model's defaults.
Script Execution	Off by default, can be toggled with "Run Script".	Always active (script is inlined into the model).
Changes Saved	Yes, all edits (including script-driven) are saved with the template.	No, payload changes are discarded when leaving Run mode.
Save / Undo / Redo	Available	Disabled
Lifecycle Event Testing	Not available	Available via Test and Debug panel.
Playout Commands	Not available	Available (requires saved template + configured destination).
Script Log	Not available	Available, captures console output from the script.
XML Copy Utilities	Not available	Available, copy payload or model as XML.
Template Settings, Tags and Scenes	Available	Disabled

5 Template Scripting

Viz Pilot Edge supports dynamic and advanced customization of the fill-in form through template scripting. Scripts are authored in TypeScript and are compiled on the fly to JavaScript for execution in the browser environment. Both the TypeScript source and the compiled JavaScript are stored in the Viz Pilot backend.

The scripting API provides three main entry points:

- Global functions
- The `vizrt` object, gives access to template fields, events, and template-specific data structures.
- The `app` object, provides application-level functionality.

With these components, template authors can control the appearance and behavior of templates, automate workflow steps, and populate fields with data from external sources.

The following sections cover these topics in more detail:

5.1 Script Technology and Security

The scripting language in the template editor is TypeScript. All JavaScript is valid Typescript, so you can choose which language to use inside the editor. However, syntax highlighting and error checking expect TypeScript. The script inside the editor is automatically compiled into JavaScript and stored as both Typescript and JavaScript within the template. When a template script executes in the browser, it runs through the browser's Web API.

When executing scripts through the browser, security is subject to the browser's default security policies. This includes:

- **CORS (Cross-Origin Resource Sharing):** Requests to external servers may be blocked if the correct CORS headers are not set.
- **Same-Origin Policy:** Scripts can only interact with resources from the same domain unless explicitly permitted.
- **Sandboxing:** If hosted within an iframe, additional restrictions may apply depending on the host's configuration.

To avoid unexpected issues, ensure that your server is properly configured to allow cross-origin access if needed.

5.1.1 Fetching Data from External Servers

If your script fetches data from external sources (for example, sports statistics or news feeds) and these servers **do not support CORS**, you must:

1. **Use a proxy server** to relay the request.
2. **Make the request through the proxy**, which must be configured to handle the external API call and return the data with appropriate CORS headers.

5.2 Jump to Preview Point

To refresh the preview or jump to a specified preview point in the current graphics, use the *app.jumpToPreviewPoint* method with a named preview point as a parameter. Sending an empty string, jumps to the default preview point in the scene.

```
vizrt.onClick = (name: string) => {
    if (name === "jump2") {
        console.log("jumping to preview point 2")
        app.jumpToPreviewPoint("pilot2")
    }
}
```

5.3 Temporary Storage

In some scenarios, temporary storage beyond standard script variables, is useful. This storage functions similarly to a field in the model and can be referenced as a source in a **Dynamic Drop-Down**.

5.3.1 Using vizrt.$data

A special object, `vizrt.$data`, is available in scripting. This object acts as a key-value map where you can freely define your own keys (the values must be strings).

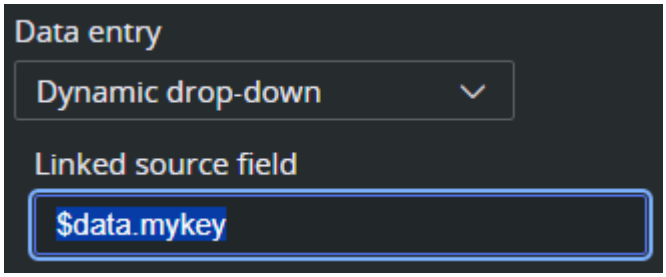
```
vizrt.$data.mykey = "myvalue"
```

In this example, the key `mykey` is assigned the value `"myvalue"`.

5.3.2 Dynamic Drop-Down Integration

You can use a key in the `$data` object as a source for dynamic drop-down, for instance `$data.mykey`, replacing the use of a traditional model field. The source data is not stored in the payload.

```
vizrt.$data.mykey = "[{"label": "Tottenham Hotspur", "value": "tottenham"}, {"label":
"Liverpool", "value": "liverpool"}, {"label": "Manchester City", "value":
"mancity"}]"
```



See also Dynamic Drop-Down.

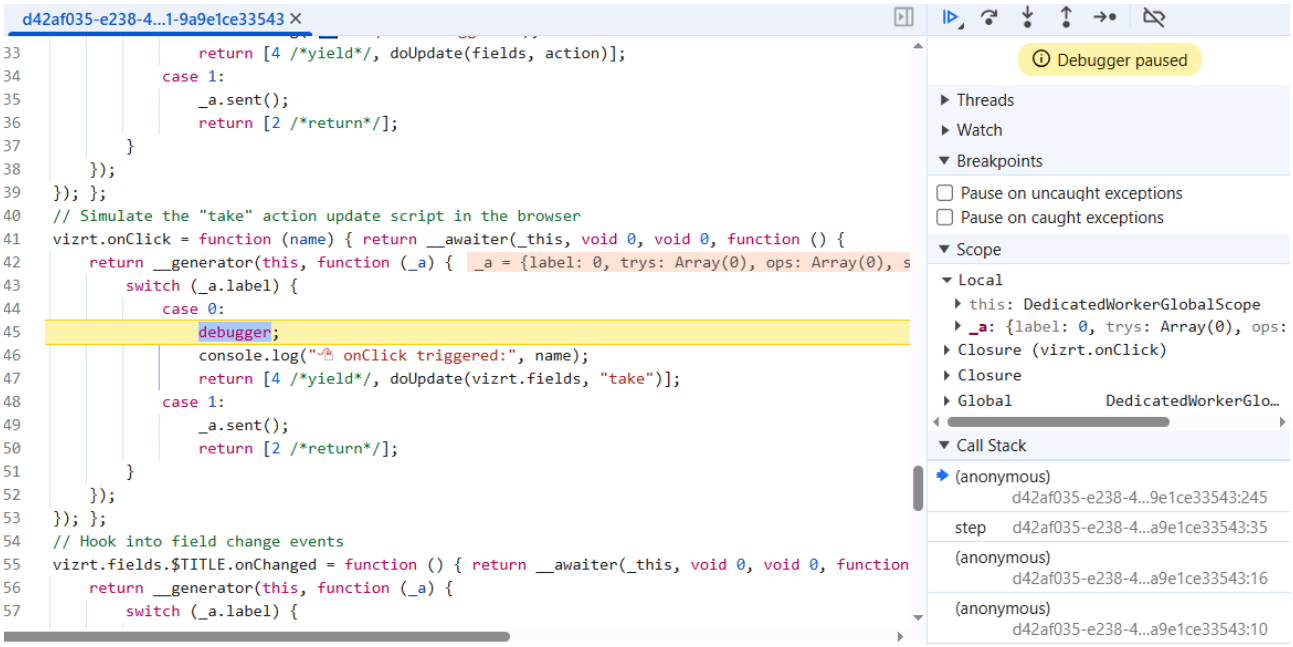
5.4 Debugging

Debugging scripts running in Template Builder can be challenging, as they execute within the embedded browser environment. However, you can force execution to pause at a specific line by inserting the JavaScript `debugger` statement, but the browser's Developer Tools must be open for the statement to take effect. When Developer Tools are open and the runtime hits a `debugger` statement, execution halts and you can inspect the call stack, scope variables and source files.

Example

```
// Simulate the "take" action update script in the browser
vizrt.onClick = async (name: string) => {
  debugger
  console.log("🖱️ onClick triggered:", name);
  await doUpdate(vizrt.fields, "take");
};
```

When this code runs, the browser pauses execution at the `debugger` line, enabling you to inspect the script using the browser's developer tools.



5.5 Field Access

5.5.1 Field Access

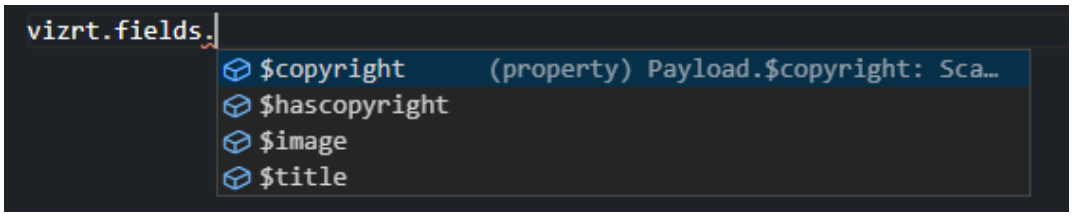
When using the scripting tool in the template, individual fields must be accessed through the global name space `vizrt.fields` (for example, `vizrt.fields.$singleline.value`).

> **Note:** Writing `$singleline.value` instead of `vizrt.fields.$singleline.value` does not work and results in a compile error.

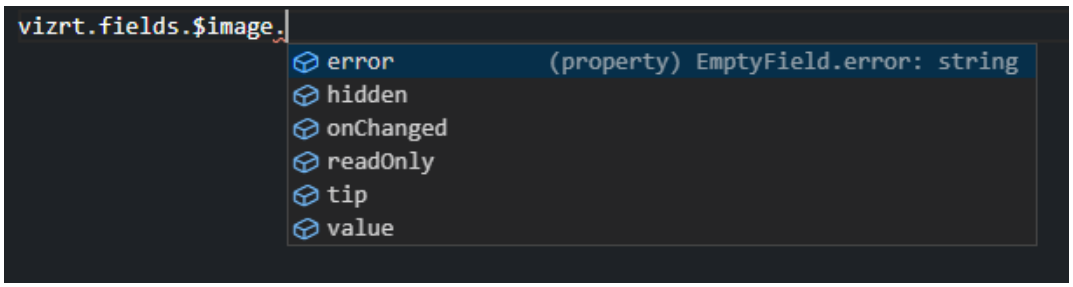
The script executes when a graphic element is opened or created with the scripted template in Viz Pilot Edge.

In Template Builder, the script is also re-loaded and restarted when there are changes made to it.

By typing `vizrt.fields`, the editor's autocomplete shows you the available fields to choose from.



You can read and write field values, react to value changes from outside the script, and access the properties *error*, *hidden*, *readOnly* and *tip* of the `vizrt` fields.



- **onChanged:** A property on fields that you can set as a function. If set, this function is called whenever the value of the field changes and receives the new value as an argument. If not set, it is *null*.

> **Note:** Changes made to field values by the template script *do not* trigger the *onChanged* function.

- **readOnly:** Read and write *boolean* access, to whether the field should be editable in the form or not. If *false* , the field and its input elements are editable in the UI. If *true* , they are read-only and greyed out in the UI, but are accessible, saved and loaded as part of the payload.
- **hidden:** Read and write *boolean* access, to whether the field should be editable in the form or not. If *false* , the field and its input elements are present and visible in the UI. If *true* , they are hidden from the UI but are accessible, saved and loaded as part of the payload.
- **error:** Read and write *string* access, to an error to display for this field. It overrides other error messages associated with the field when non-empty (e.g. errors due to input length or regex constraints specified in the field definition).
- **tip:** Read and write *string* access to provide a tip for this field. It overrides the tip specified in the field definition when non-empty.

> ⚠ **Note:** Dashes cannot be used in TypeScript with the dot syntax. Use `vizrt.fields["$01-week"]` syntax to access such fields.

Typescript lets you access fields using known names (for example, `vizrt.fields["$field_2"]` *)*, but does not allow expressing the field names using variables (for example, `vizrt.fields["$field_" + num]` *)*, as it is unable to determine whether the field name is valid and what type the field is.

This must be used, for instance, when addressing the layers in a Transition Logic combo template, such as: `vizrt.fields["$-vizlayer-TL_FS"]` .

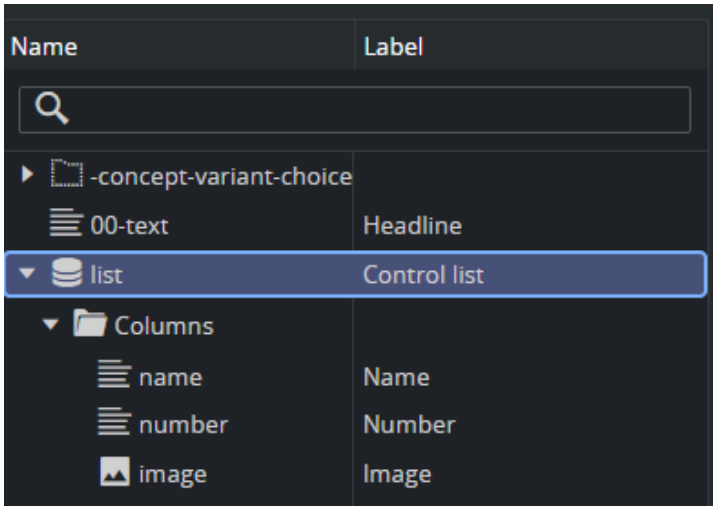
To overcome this, the type checking associated with *vizrt.fields* can be locally ignored by using the *any* type. You may do so inline using for example *(vizrt.fields* ***as any****)["$field_" + num]*, or storing it as a new variable, as shown in the example below.

```
function resetValues() {
    for (let playerNumber = 1; playerNumber <= 2; playerNumber++) {
        const untypedFields = vizrt.fields as any

        untypedFields["$player" + playerNumber + "_firstname"].value = ""
        untypedFields["$player" + playerNumber + "_lastname"].value = ""
        untypedFields["$player" + playerNumber + "_age"].value = 0
    }
}
```

> ⚠ **Note:** When using an object cast to *any*, the script editor cannot help you catch errors such as setting a boolean value to a string field, or trying to access a field that does not exist.

List Fields

In TemplateBuilder, a **list field** represents a repeating set of rows, similar to a table. Even though Viz Artist uses the term **ControlList**, the underlying VDF field type is simply a **list**, and each row contains a fixed set of child fields (effectively table columns).



A list field is accessed as follows:

```
const list = vizrt.fields.$list.value;
```

`list` now contains an **array of rows**, and each row exposes its columns as simple child fields:

```
list[rowIndex].$name.value = "Example";
list[rowIndex].$number.value = "123";
list[rowIndex].$image.value = createImageAsset("http://example/image.jpg");
```

Field Naming

It is recommended to use clean names without hyphens for column names. Child fields are then accessed directly using:

- `$name`
- `$number`
- `$image`
- etc.

This makes the list behave like a typed table where each row has well-defined columns. If column names contain hyphens (for example, `$11-number`), you cannot use the dot syntax in TypeScript. Instead, access the field using bracket notation: `list[i]["$11-number"].value = "123";`.

See Quick Start Examples for a full list example.

Read-only Fields

Map fields are currently supported only for read-access by the scripting API.

Individual properties for the **Image** and **Video** fields (height, width, etc.) are read-access only. Changes to those fields must change the entire value.

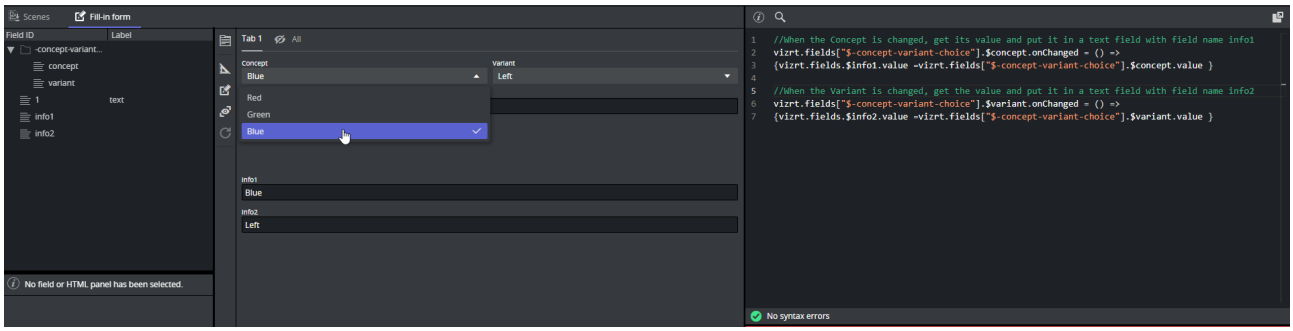
The following example shows how to retrieve the image height:

```
vizrt.fields.$ImageInfo.value = "No image info";

vizrt.fields.$image.onChanged =() => {
    vizrt.fields.$ImageInfo.value = 'Image changed';

    var v = vizrt.fields.$image.value;
    if(v != undefined && v.height != undefined)
        vizrt.fields.$ImageInfo.value = v.height.toString();
    else
        vizrt.fields.$ImageInfo.value = "No image info";
}
```

Note: Because images and videos can be undefined, they must be checked before they are used.

5.5.2 Accessing Concepts & Variants

To access information from concept and variants fields, because they contain dashes, you must use square brackets, as shown in the example below.




```
//When the Concept is changed, get its value and put it in a text field with field
name info1
vizrt.fields["$-concept-variant-choice"].$concept.onChanged = () =>
{vizrt.fields.$info1.value = vizrt.fields["$-concept-variant-choice"].
$concept.value }

//When the Variant is changed, get the value and put it in a text field with field
name info2
vizrt.fields["$-concept-variant-choice"].$variant.onChanged = () =>
{vizrt.fields.$info2.value = vizrt.fields["$-concept-variant-choice"].
$variant.value }
```

5.5.3 Fetching Data From External Sources

Whether on template load, or as a reaction to a field change, you can initiate HTTP, HTTPS or REST calls to fetch values from third-party or external services.

This can be done via the browser's built-in *fetch* API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

> **Info:** See the Quick Start Examples section for a short example of a REST call triggered by an *onChanged* event.

5.5.4 Image Metadata

Every image has some amount of metadata attached to it, and with this field you are able to access this metadata by using the script editor.

You can upload images and the corresponding metadata to asset source servers. By accessing an image's metadata, you can auto-fill the name field within a template or display or hide an image that might be copyrighted.

> **Note:** Image scripting metadata currently works for images retrieved from the following asset source servers: Vizone, Vos, GH and OMS.

How to Use the Image Metadata

If you are using a newly created template, choose any image, and then query that image's metadata in the script editor by querying the image's metadata map. With an existing template that already contains an image created before version 2.4 of Template Builder and Viz Pilot Edge, click on the current image and re-select it from the asset selector, or select another image and then re-select the original one. Once this is done, you can proceed by querying the image's metadata map.

This example checks if *imageName* has a metadata key named *test* and then tries to get the value of that metadata key. You can use the script syntax shown below:

```
let hasMetadataKey: boolean = vizrt.fields.$imageName.metadata.has("test") //true if
the metadata contains the key test
let metadataValue: string = vizrt.fields.$imageName.metadata.get("test") //it is set
to the value it has in the metadata, otherwise it is undefined
```

This code example reacts when a new image is selected. When the *image2* field gets assigned a new image, it tries to get the description from the metadata associated with the new image, and set it into the text field *img2_txt*. If the description does not exist, a message displays explaining it was not found.

```
vizrt.fields.$image2.onChanged = () => {
    if (vizrt.fields.$image2.metadata != undefined) {
        let keyName = "description"
        let a = vizrt.fields.$image2.metadata.get(keyName) // get the metadata value
that has the given key
        if (a != undefined) {
            vizrt.fields.$img2_txt.value = a // if the value is not undefined, then
set it into a string field within the template
        } else {
            vizrt.fields.$img2_txt.value = "The key '" + keyName + "' was not found
inside the metadata map"
            // Alternatively, set it to nothing: vizrt.fields.$img2_txt.value = ""
```

```
        }
    }
}
```

To access the entire metadata list:

```
vizrt.fields.$03.onChanged = v => {
     if (vizrt.fields.$03 != undefined && vizrt.fields.$03.metadata != undefined) {
         vizrt.fields.$metadatalist.value = ""
         vizrt.fields.$03.metadata.forEach(printCustom)
     }

     function printCustom(value: any, index: string){
         if (vizrt.fields.$03 != undefined && vizrt.fields.$03.metadata != undefined)
{
             vizrt.fields.$metadatalist.value += index + ": " + vizrt.fields.
$03.metadata.get(index)+"\n"
         }
     }
 }
```

Image Metadata XML

An image's metadata is stored within asset source servers in XML format, and the XML metadata structure should follow a simple field-value (key-value) structure. This guarantees that all metadata is correctly mapped and accessible through the script editor.
However, since many image metadata XMLs are disorderly, a parser has been created to handle most XML structures, although this has some consequences that should be made aware of. All these example scripts are based on an image field named "image1".

Empty field-value pairs get added accordingly:

```
<field name="car"/>
<field name="color">
    <value/>
</field>

// Accessing these can be done using:

let a = vizrt.fields.$image1.metadata.get("car")
let b = vizrt.fields.$image1.metadata.get("color")

// Both a and b will be ""
```

Nested structures get stored based on their hierarchy, with "/" being the parent-child separator:

```
<field name="access-rights">
    <field name="user-rights">
        <value>true</value>
```

```
    </field>
</field>

// To access the user-rights value:

let a = vizrt.fields.$image1.metadata.get("access-rights/user-rights")

// a will then be set to true.
```

Any fields with duplicate names get assigned a unique name with an incrementing suffix:

```
<field name="file-link-id">
    <value>id123</value>
</field>
<field name="file-link-id">
    <value>id456</value>
</field>
<field name="file-link-id">
    <value>id789</value>
</field>
```

Access duplicate names like this using the incrementing suffix:

```
let a = vizrt.fields.$image1.metadata.get("file-link-id")
let b = vizrt.fields.$image1.metadata.get("file-link-id(2)")
let c = vizrt.fields.$image1.metadata.get("file-link-id(3)")

a will be "id123"
b will be "id456"
c will be "id789"
```

5.6 Script Units

Script Units are reusable TypeScript modules that centralize logic and share it across multiple templates in Template Builder. Instead of duplicating utility functions or business rules in each template, you place them in a Script Unit and import them wherever needed.

Script Units are stored in Viz Pilot's backend. This means they function as shared, centralized code:

- When you **create or update** a Script Unit, the change is immediately available to all templates that import it.
- Any template using that unit automatically picks up the new behavior as soon as the unit is saved, no redeployment of individual templates is required.

A Script Unit is imported and used in a template as follows:

```
import { toTitleCase, toShoutingCaps } from '@unit/formatting';

const guestName = toTitleCase("good title casing");     // "Good Title Casing"
const location  = toShoutingCaps("Chicago");            // "CHICAGO"
```

And the unit itself might look like:

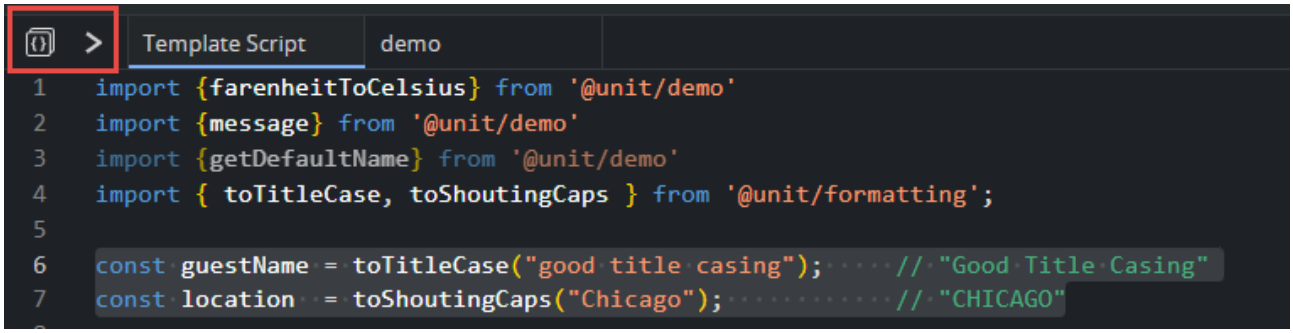
```
// @unit/formatting
export function toTitleCase(name: string): string {
    return name
        .toLowerCase()
        .replace(/\b\w/g, c => c.toUpperCase());
}

export function toShoutingCaps(text: string): string {
    return text.trim().toUpperCase();
}
```

Script Units provide a clean way to maintain consistency, reduce duplication, and ensure that updates to shared logic propagate instantly across your entire template ecosystem.

5.6.1 Create, Modify and Delete Script Units

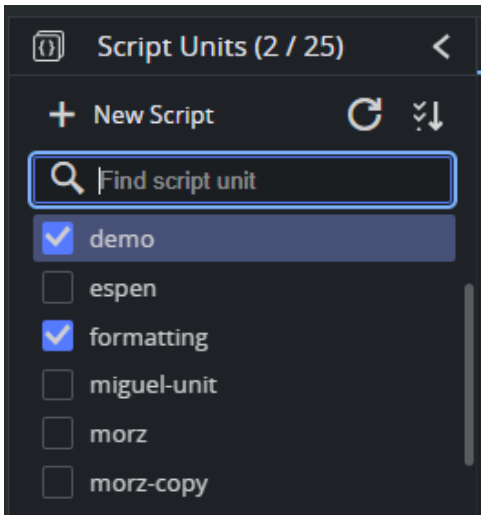
The Viz Pilot backend maintains a record of which templates depend on which Script Units. Therefore, it is important that the Script Unit list in Template Builder accurately reflects all units referenced in your template scripts.

To manage Script Units, click the **Script Units** icon to open the list:



From the Script Units panel, you can:

- **Create** a new Script Unit.
- **Search** and **sort** existing units.
- **Attach** a Script Unit to the current template by selecting the checkbox next to its name.



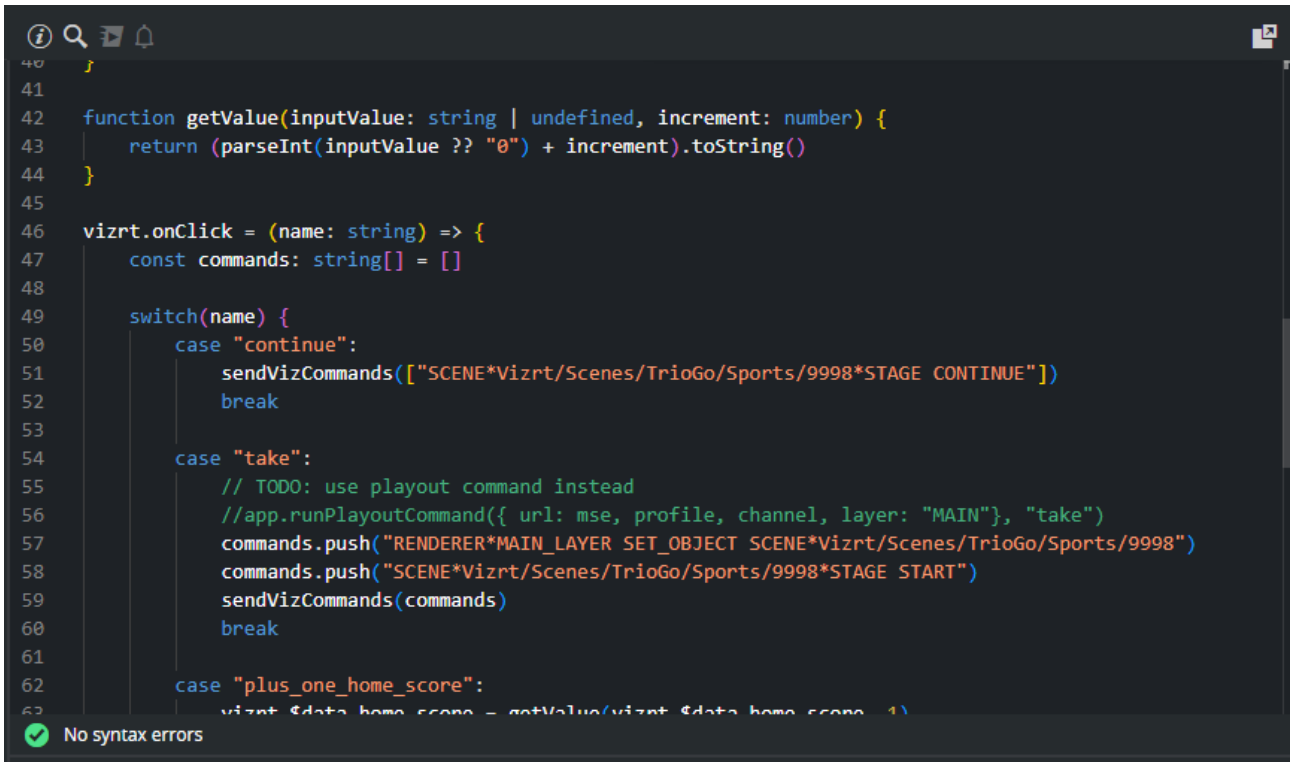
These operations ensure that your template always has access to the Script Units it requires, and that the backend can track dependencies correctly.

5.7 Script Editor

The internal script editor in Template Builder is built on top of the Monaco Editor, the same open-source editor that powers Visual Studio Code. It provides a powerful and efficient development experience with many built-in features that help you write and manage scripts effectively.

This page outlines the general editor features available in the script editor.

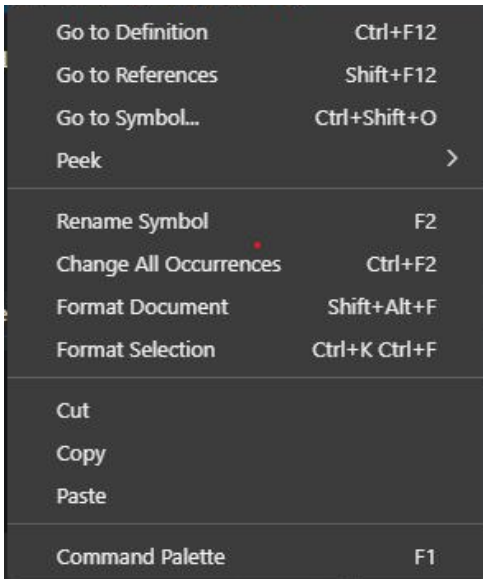
- Core Editor Features
- Context Menu
- Keyboard Shortcuts



5.7.1 Core Editor Features

- **Syntax Highlighting** for TypeScript and JavaScript.
- **IntelliSense** for code suggestions, autocompletion, and tooltips.
- **Real-Time Error Checking** based on syntax and type inference.
- **Bracket Matching** highlights matching brackets and braces, helping you keep track of code blocks and function definitions.
- **Code Folding** collapses or expands code blocks using the small arrows in the margin, making it easier to navigate large scripts.
- **Multi-Cursor Editing** by holding `Alt` and clicking to place multiple cursors.

5.7.2 Context Menu



Right-click anywhere in the editor to open the context menu. This menu provides access to many common actions and code navigation tools:

- **Go to Definition:** Jumps to where a function or variable is defined.
- **Go to References:** Finds all usages of the selected symbol.
- **Go to Symbol...:** Opens a list of all functions and variables in the file, allowing quick navigation.
- **Peek:** A submenu that lets you preview definitions and references inline without leaving your current position.
- **Rename Symbol:** Renames a symbol across the script, updating all references automatically.
- **Change All Occurrences:** Selects and edits all occurrences of the current word or variable at once.
- **Format Document:** Re-formats the entire script based on consistent style rules.
- **Format Selection:** Formats only the selected lines.
- **Cut / Copy / Paste:** Standard editing options to manipulate code.
- **Command Palette:** Opens a searchable list of all available commands in the editor.

5.7.3 Keyboard Shortcuts

Here are some of the most useful Monaco Editor shortcuts that work inside Template Builder:

Shortcut	Action
`Ctrl + Shift + O`	Open a searchable list of functions, variables, and other symbols in the document. By typing `:` the symbols are grouped by category.
`Ctrl + Space`	Trigger suggestion/autocomplete

Shortcut	Action
`F12`	Go to Definition
`Alt + F12`	Peek Definition
`Shift + F12`	Find All References
`F2`	Rename Symbol
`Ctrl + F`	Search (and replace)
`F1`	Open Command Palette
`Alt + Up / Alt + Down`	Move line up/down
`Ctrl + Shift + K`	Delete line
`Ctrl + Z / Ctrl + Y`	Undo / Redo
`Shift + Alt + F`	Format the document
`Ctrl + K, F`	Format the selected text

5.8 Quick Start Examples

In this section you can find short examples of how to use the scripting functionality.

5.8.1 Fields

Setting Default Values on Data Element Creation

```
vizrt.onCreate = () => {
  vizrt.fields.$headline.value = "Enter headline"
  vizrt.fields.$showLogo.value = true
  vizrt.fields.$headline.decoration.maxlength = 60
}
```

Conditionally Hide Fields based on a Boolean

```
vizrt.fields.$showSubtitle.onChanged = show => {
    vizrt.fields.$subTitle.hidden = !show
}
```

Accessing Tables

The following example shows how to populate a list or table of Formula 1 drivers when the user clicks a button named `"fetch"`.

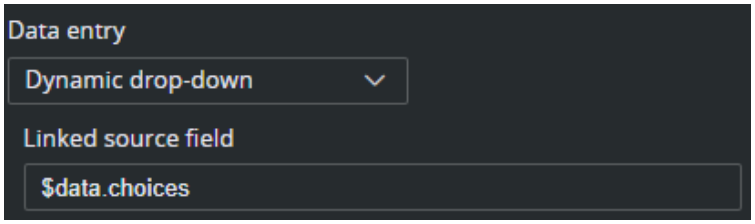
```
// Drivers list, usually fetched from an external resource
const drivers = [
    { name: "Lewis Hamilton", wins: "105" },
    { name: "Michael Schumacher", wins: "91" },
    { name: "Max Verstappen", wins: "68" },
    { name: "Sebastian Vettel", wins: "53" },
    { name: "Alain Prost", wins: "51" },
    { name: "Ayrton Senna", wins: "41" },
    { name: "Fernando Alonso", wins: "32" }
];

// Helper: extract last name, lowercase, and add .jpg
function getImageFileName(name: string): string {
    const parts = name.trim().split(" ");
    const lastName = parts[parts.length - 1].toLowerCase();
    return `${lastName}.jpg`;
}

vizrt.onClick = (name: string) => {
    if (name === "fetch") {

        // Reference to the list/table field in the payload
        const list = vizrt.fields.$list.value;

        // Base path for image assets
        const baseUrl = "http://myserver.com/imageshare/";

        // Iterate and populate the rows
        drivers.forEach((driver, i) => {
            if (!list[i]) return; // Safety if fewer rows exist

            const imageFile = getImageFileName(driver.name);

            list[i].$name.value = driver.name;
            list[i].$number.value = driver.wins;
            list[i].$image.value = createImageAsset(baseUrl + imageFile);
        });
    }
};
```

Dynamic Drop-down and ControlOMO

This example shows how to fill a dynamic drop-down (a drop-down that can change run-time) with options, controlling a ControlOMO field in the template.

```
const select = [
  { label: "Image and text", value: 0 },
  { label: "Text", value: 1 },
  { label: "Bullets", value: 2 },
  { label: "Image and bullets", value: 9.0 }
];

vizrt.$data["choices"] = JSON.stringify(select)
```

In the template, **Data Entry** for the ControlOMO field must be set to **Dynamic drop-down** and the Linked source field should be `$data.choices`.



React to Field Changes and Update other Fields

This example shows how to listen to user input (`onChanged`) and programmatically modify other fields, including dynamically controlling visibility via the `hidden` property.

```
// React when the user edits the headline field
vizrt.fields.$headline.onChanged = value => {
  // Automatically update another field
  vizrt.fields.$slug.value = value?.toUpperCase() ?? ""

  // Show/hide a field based on input
  vizrt.fields.$subtitle.hidden = !value
}
```

> **Note:** Field names that contain dashes (`-`) cannot be accessed using dot notation in TypeScript. Instead, use bracket notation: `vizrt.fields["$01-week"]`.
> When defining new fields, it is recommended to use camelCase or underscores (for example, `01thisIsMyField` or `01_week`). This allows the fields to be accessed using standard dot notation, which is more concise and readable.

Validate Before Save

```
vizrt.onBeforeSave = () => {
  const headline = vizrt.fields.$headline.value
  if (!headline || headline.trim().length === 0) {
```

```
    vizrt.fields.$headline.error = "Headline cannot be empty"
    app.notify("error", "Please fill in the headline before saving.")
    return false
  }
  vizrt.fields.$headline.error = ""
  return true
}
```

5.8.2 Control Playout through MSE

```
const mse = app.createMseConnection("http://mse-server:8580")

vizrt.onClick = name => {
  const target = { profile: "STUDIO_A", channel: "GFX", layer: "MAIN" as const }
  switch (name) {
    case "take":
      mse.take(target)
      break
    case "update":
      mse.update(target)
      break
    case "out":
      mse.out(target)
      break
  }
}
```

5.8.3 Working with MOS Metadata

These patterns show how scripts can both **consume metadata from the newsroom system** and **inject structured data back into the MOS item**, enabling tighter integration between templates and editorial workflows.

Create MOS Metadata

This shows how to initialize MOS metadata, when a new Data Element is created, including automation-related properties, such as `continueCount`.

```
vizrt.onCreate = () => {
  // Standard MOS fields
  vizrt.$pilot.mos.slug = "BREAKING-NEWS"
  vizrt.$pilot.mos.description = "Breaking news lower third"

  // Control automation behavior
  vizrt.$pilot.mos.continueCount = 1

  // Optional: initialize Mosart timing
  if (vizrt.$pilot.mos.mosart) {
```

```
    vizrt.$pilot.mos.mosart.inMode = "auto"
    vizrt.$pilot.mos.mosart.outMode = "auto"
  }
}
```

Read Custom MOS External Metadata (MEM) from NRCS

This example reads a custom MEM block provided by the NRCS, parses the XML payload and uses it to populate template fields.

```
vizrt.onLoad = () => {
  const schema = "http://example.com/schemas/suggested-headlines"

  const block = vizrt.$pilot.mos.mem.get(schema)
  if (!block) return

  try {
// Example MEM block:
// <mosExternalMetadata>
//  <mosSchema>http://example.com/schemas/suggested-headlines</mosSchema>
//  <mosScope>PLAYLIST</mosScope>
//  <mosPayload>
//    <headlines>
//      <headline>Main headline</headline>
//      <headline>Alternative headline</headline>
//    </headlines>
//  </mosPayload>
// </mosExternalMetadata>

    const parser = new DOMParser()
    const xml = parser.parseFromString(block.payload, "application/xml")

    const headlines = Array.from(xml.getElementsByTagName("headline"))
      .map(node => node.textContent)
      .filter(Boolean)

    // Use the first suggested headline
    if (headlines.length > 0) {
      vizrt.fields.$headline.value = headlines[0]!
    }
  } catch (e) {
    console.error("Failed to parse MEM block", e)
  }
}
```

Create a Custom MOS External Metadata (MEM) Block

This example creates (or updates) a custom MEM block that can be consumed by the NRCS, or downstream systems. The payload can contain any agreed XML structure, such as playout instructions or editorial metadata.

```
vizrt.onAfterSave = () => {
  const schema = "http://example.com/schemas/graphics-instructions"

  const payload = `
    <instructions>
      <take mode="auto" />
      <duration seconds="10" />
      <transition type="fade" />
    </instructions>
  `.trim()

  vizrt.$pilot.mos.mem.set({
    schema,
    payload,
    scope: "PLAYLIST",
  })
}
```

The resulting MOS XML is:

```
<mosExternalMetadata>
  <mosSchema>http://example.com/schemas/graphics-instructions</mosSchema>
  <mosScope>PLAYLIST</mosScope>
  <mosPayload>
    <instructions>
      <take mode="auto"/>
      <duration seconds="10"/>
      <transition type="fade"/>
    </instructions>
  </mosPayload>
</mosExternalMetadata>
```

5.8.4 Handle Errors in Asynchronous Operations

```
fetch("https://api.example.com/config")
  .then(res => res.json())
  .then(config => {
    vizrt.fields.$headline.value = config.defaultHeadline
  })
  .catch(e =>
    reportErrors(() => {
      throw e
    }, "load remote config"),
  )
```

5.8.5 Script Unit with Click Handler

The following example demonstrates how to use a script unit to add individual click handlers for buttons in HTML fragments. The exported method `onClick` replaces the usual pattern with something simpler and modular.

```
vizrt.onClick = (name) => {
  switch (name) {
    case "BtnA": ...
    case "BtnB": ...
  }
}
```

This method allows you to register individual click handlers by name, without writing a large `switch` block. Each call to `onClick(name, callback)` stores a callback in a map and installs a global `vizrt.onClick` dispatcher that forwards the event to the correct callback. It also returns a **dispose** function so you can unregister the handler.

The unit and the usage of this module may look like this in the template:

```
const onClickMap: Record<string, () => void> = {}

/**
 * A better vizrt.onClick to avoid switch statements...
 * It returns a dispose function when it's no longer needed
 * @param name the vizrt-click-name to register
 * @param callback The callback to call when clicked
 * @returns a dispose function to stop updating
 */
export const onClick = (name: string, callback: () => void) => {
  onClickMap[name] = callback
  vizrt.onClick = (name) => onClickMap[name]?.()

  //console.log("from onClick")
  return () => delete onClickMap[name]
}
```

First add a new HTML fragment to the template and create a button with a click name:

```
<button vizrt-click-name="my-button">MY BUTTON</button>
```

Then, in scripting, the handler for this button is expressed as follows:

```
import { onClick } from "@unit/global-onclick"

const dispose = onClick("my-button", () => {
    app.notify("info", "The button was clicked!");
```

```
});
```

You can later call the *dispose()* method if you need to disconnect the click handler.

5.8.6 Fetch Data from REST Service

In this example, the script automatically fills the `title` field by fetching data from a REST endpoint. It demonstrates how to use the standard `fetch` API and update fields based on an external response.

```
vizrt.onLoad = () => {
  fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => {
      // Assume the response contains: { "title": "Some headline" }
      vizrt.fields.$title.value = data.title ?? ""
    })
    .catch(error => {
      console.error("Failed to fetch title:", error)
      app.notify("error", "Could not load title from service")
    })
}
```

5.8.7 Accessing Views (Tabs)

Templates can expose multiple *views* (tabs) to organize content or configurations. From scripting, it is possible to:

- Iterate over available views.
- Check which view is active.
- Activate a view programmatically.
- React when the user changes the selected view.

> ⚠ **Note:** The special **All tab** is a UI concept only. It is not included in the views API and cannot be activated from scripting.

```
// Activate a view
vizrt.views[1].activate();

// Check which view is active
const activeView = vizrt.views.find(view => view.isActive);

if (activeView) {
  console.log("Active view:", activeView.title);
}

// Listen for activation
vizrt.onViewActivated((selected, old) => {
  if (selected === undefined) {
```

```
    console.log("All tab selected");
  } else {
    console.log("Active view:", vizrt.views[selected].title);
  }
});

// Change the decorative title of a view
vizrt.views[0].title = "Overview - Q1";
```

Working with ViewItems

Each view exposes a collection of items representing fields and UI elements (fragments/iframes). These can be inspected, repositioned and conditionally hidden at runtime to create more dynamic layouts.

A `ViewItem` is either:

- a **field item** (`type: "field-item"`) identified by its `path` , or
- a **UI item** (`type: "ui-item"`) identified by its `name` .

All items share a common API for layout via `position` and `setPosition()` .

Example: reposition and hide items

```
const items = vizrt.views[0].items;

// Move a field item (identified by field path)
const headline = items.find(
  item => item.type === "field-item" && item.path === "headline"
);

headline?.setPosition({
  x: 0,
  y: 0,
  width: 12,
  height: 2
});

// Hide the headline if needed
if (headline) {
  headline.hidden = true;
}

// Move a UI fragment (identified by name)
const fragment = items.find(
  item => item.type === "ui-item" && item.name === "fragment"
);

fragment?.setPosition({
  x: 6,
  y: 0,
  width: 5,
```

```
  height: 14
});
```

Notes

- `setPosition()` accepts a partial rectangle, this way, you are able to only update what you need:

  ```
  headline?.setPosition({ y: 2 });
  ```

- Layout changes are applied in a single update per call.
- Use `type` to distinguish between field items and UI items.
- Field items use `path`, UI items use `name` for identification.
- `hidden` can be used to programmatically show/hide items. If a field has a hidden-expression, that logic overrides the `hidden` property.

This allows scripts to dynamically rearrange and control visibility of the UI, based on user input or context.

5.9 API Reference

The Template Builder scripting API provides a TypeScript-based interface for controlling and customizing the behavior of graphics templates within Viz Pilot Edge. Scripts are authored in the Monaco editor inside Template Builder, and execute at runtime in Viz Pilot Edge when users interact with templates.

Scripts have access to the template's payload fields, MOS metadata, life cycle events, view management, MSE playout control, etc, all through a set of globally available objects and functions documented below.

5.9.1 Global Objects

vizrt

The primary global object for interacting with the scripting API. Provides access to template fields, environment variables, Pilot metadata, MOS data, views, life cycle callbacks and spell check.

```
declare const vizrt: {
  readonly $env: Partial<Readonly<Record<string, string>>>
  readonly $pilot: PilotData
  readonly $data: Partial<Record<string, string>>
  readonly fields: Payload
  readonly views: readonly View[]
  onViewActivated: undefined | ViewActivatedCallback
  onCreate: undefined | (() => void)
  onLoad: undefined | (() => void)
  onClick: undefined | ((name: string) => void)
  onUpdate: undefined | ((fields: Payload, action: string) => void | Promise<void>)
  onAfterSave: undefined | (() => void)
  onBeforeSave: undefined | (() => boolean | Promise<boolean>)
  readonly spellCheck: (languageCode?: string, fields?: readonly string[]) =>
Promise<boolean>
  /** @deprecated Use `app.jumpToPreviewPoint` instead. */
  readonly jumpToPreviewPoint: (name: string) => void
```

```
}
```

Property	Type	Description
`$env`	`Partial<Readonly<Record<string, string>>>`	Read-only access to environment variables defined for the template.
`$pilot`	`PilotData`	Read-only access to Pilot metadata including MOS data, Viz Mosart configuration, and element info.
`$data`	`Partial<Record<string, string>>`	Temporary key-value data store, accessible from dynamic drop-down editors.
`fields`	`Payload`	The template's payload fields. The `Payload` type is generated per model and provides strongly-typed access to every field defined in the template.
`views`	`readonly View[]`	Array of views (tabs) defined in the template model. Excludes the "All" tab which is a UI-only concept.

app

Global object providing access to application-level commands such as UI notifications, MSE connections and preview navigation.

```
declare const app: {
  readonly notify: (severity: "warning" | "info" | "error" | "success", message:
string) => void
  readonly createMseConnection: (url: string) => MseConnection
  readonly jumpToFieldPreview: (path: string) => void
  readonly jumpToPreviewPoint: (name: string) => void
}
```

Method	Description
`notify(severity, message)`	Displays a notification toast in the UI with the given severity level and message text.
`createMseConnection(url)`	Creates a connection to a Media Sequencer Engine (MSE) at the specified REST URL. Returns an `MseConnection` object for playout control.

Method	Description
`jumpToFieldPreview(path)`	Navigates the graphics preview to the field identified by the given path.
`jumpToPreviewPoint(name)`	Navigates the graphics preview to a named preview point.

5.9.2 Callbacks

Life cycle callbacks allow scripts to respond to events in the template life cycle. Assign a function to the corresponding property on the `vizrt` object to register a handler.

vizrt.onCreate

Called when a template is opened in Viz Pilot Edge, which creates a new Data Element. Typically used to initialize default values for fields.

- **Not triggered** when creating a Data Element from an existing one (for example, duplicate).
- **Not triggered** from Template Builder.

```
vizrt.onCreate = () => {
  vizrt.fields.$headline.value = "Default Headline"
}
```

vizrt.onLoad

Called when an existing data element is opened in Viz Pilot Edge. This event also fires in browse mode when a data element is selected. Use this event to prepare the MOS XML that is sent to the newsroom, which is relevant when data elements are sent directly from browse mode.

- **Not triggered** from Template Builder.

```
vizrt.onLoad = () => {
  if (!vizrt.fields.$headline.value) {
    vizrt.fields.$headline.value = "Fallback"
  }
}
```

vizrt.onUpdate

This event is invoked by the Script Runner when the template is configured to use the Update Service. Note that this handler executes in a Node.js server process and, therefore, may expose a different JavaScript API. See Update Service for more details.

- **Not triggered** from Template Builder or Viz Pilot Edge.

```
vizrt.onUpdate = async (fields: Payload, action: string) => {
  fields.headline.value = "Updated by automation"
}
```

vizrt.onBeforeSave

Called before a save operation. Return `false` (or a `Promise<false>`) to cancel the save, return `true` to proceed.

```
vizrt.onBeforeSave = () => {
  if (!vizrt.fields.$headline.value) {
    app.notify("error", "Headline is required")
    return false
  }
  return true
}
```

vizrt.onAfterSave

Called after a save operation of the currently open Data Element has completed successfully.

```
vizrt.onAfterSave = () => {
  app.notify("success", "Element saved")
}
```

vizrt.onClick

Called when a click event is triggered from an HTML fragment UI. The `name` parameter corresponds to the `vizrt-click-name` attribute of the clicked element.

```
vizrt.onClick = (name: string) => {
  if (name === "reset-button") {
    vizrt.fields.$headline.value = ""
  }
```

```
}
```

vizrt.onViewActivated

Called when the active view (tab) changes in the UI. Receives the index of the newly selected view and the previously selected view. An index of `undefined` means the "All" tab is selected.

```
vizrt.onViewActivated = (selected, old) => {
  if (selected !== undefined) {
    console.info(`Switched to view: ${vizrt.views[selected].title}`)
  }
}
```

5.9.3 Payload & Fields

Payload (Generated Interface)

The `Payload` type is **generated at design time** based on the template's VDF model. Each field defined in the model becomes a strongly-typed property on the `Payload` interface. The actual shape depends on the template.

> ⚠ **Note:** All field names in the generated `Payload` interface are prefixed with `$` (for example, `$headline`, `$body`). This naming convention is applied automatically during type generation and distinguishes payload fields from built-in properties.

Access fields through `vizrt.fields`:

```
// Read a field value
const title = vizrt.fields.$headline.value

// Write a field value
vizrt.fields.$headline.value = "Breaking News"

// Access a nested field
vizrt.fields.$settings.$color.$value = "#FF0000"

// Listen for external changes to a field
vizrt.fields.$headline.onChanged = newValue => {
  console.info("Headline changed to:", newValue)
}
```

5.9.4 Field Types

EmptyField

The base interface for all VDF fields. Represents a field with no value and no list, only behavioral properties.

> **Info:** If `hidden`, `readOnly`, or `tip` is `undefined` at the script level, the value is **inherited** from the field definition in the model, and further from the parent field.

```
interface EmptyField {
  /** Controls the visibility of the field. Inherited from parent and field
definition. */
  hidden: boolean
  /** Controls whether the field is read-only. Inherited from parent and field
definition. */
  readOnly: boolean
  /** The tooltip text for the field. Inherited from the field definition. */
  tip: string
  /** An error message to display on the field. Not inherited. */
  error: string
  /** The decoration object controlling display properties. */
  decoration: BaseDecoration
}
```

ScalarField<T>

Extends `EmptyField` with a typed value (this is the most commonly used field type). `T` corresponds to the field's media type (for example, `string`, `number`, `boolean`, `RichText`, `ImageAsset`, etc.).

```
interface ScalarField<T> extends EmptyField {
  /** Read/write access to the field value. */
  value: T
  /** Handler invoked when the value is changed from outside the script (e.g., by the
user in the UI). */
  onChanged?: (value: T) => void
  /** Read-only access to the field's metadata key-value pairs defined in the model.
*/
  readonly metadata: Readonly<Map<string, string>> | undefined
}
```

TextScalarField<T>

Extends `ScalarField<T>` with text-specific decoration support. Used for single-line text, multi-line text and rich text fields.

```
interface TextScalarField<T = string> extends ScalarField<T> {
  /** Text-specific decoration with properties like maxlength. */
  decoration: TextDecoration
}
```

VdfList<Columns>

Represents a list field containing rows of typed column values. The list is read-only (you cannot add or remove rows from script), but individual cell values can be modified.

```
type VdfList<Columns extends object> = ReadonlyArray<{
  [Col in keyof Columns]: VdfListItem<Columns[Col]>
}>
```

Iterating a list example:

```
for (const row of vizrt.fields.$items) {
  console.info(row.label.value)
}
```

VdfListItem<T>

A mapped type that extracts the scriptable properties from a field type when used inside a list. For fields extending `EmptyField`, only properties starting with `$` and the `value` property are preserved.

```
type VdfListItem<T> = T extends EmptyField ? { [K in Extract<keyof T, `$${string}` |
"value">]: VdfListItem<T[K]> } : T
```

5.9.5 Decoration Types

Decorations allow scripts to control the display properties of fields dynamically.

BaseDecoration

Base decoration available on all field types.

```
interface BaseDecoration {
  /** Display label for the field in the UI. */
  label?: string
}
```

Example:

```
vizrt.fields.$headline.decoration.label = "Main Title"
```

TextDecoration

Extended decoration for text fields, adding text-specific constraints.

```
interface TextDecoration extends BaseDecoration {
  /** Maximum number of characters allowed in the text field. */
  maxlength?: number
}
```

Example:

```
vizrt.fields.headline.decoration.maxlength = 50
```

5.9.6 Value Types

RichText

Represents formatted text content. `RichText` objects are **immutable** and must be created using the `createRichText` factory function.

```
interface RichText {
  /** The text content without any formatting (plain text representation). */
  readonly plainText: string
}
```

MediaAsset

Base interface for all media asset values. Contains common metadata shared by image and video assets.

```
interface MediaAsset {
  /** The type of the media: `"image"` or `"video"`. */
  readonly mediaType: MediaType
```

```
  /** The display title of the asset. */
  readonly title: string | undefined
  /** Timestamp of the last update to the asset. */
  readonly updated: Readonly<Date> | undefined
  /** URL to a thumbnail representation of the asset. */
  readonly thumbnailUrl?: string | undefined
  /** The path to the media resource. */
  readonly path: string
}
```

ImageAsset

Stores information about an image asset. Extends `MediaAsset` with dimensional metadata.

```
interface ImageAsset extends MediaAsset {
  /** The width of the image in pixels, if available. */
  readonly width: number | undefined
  /** The height of the image in pixels, if available. */
  readonly height: number | undefined
}
```

VideoAsset

Stores information about a video asset. Extends `MediaAsset` with dimensional and duration metadata.

```
interface VideoAsset extends MediaAsset {
  /** The width of video frames in pixels, if available. */
  readonly width: number | undefined
  /** The height of video frames in pixels, if available. */
  readonly height: number | undefined
  /** The duration of the video in seconds, if available. */
  readonly duration: number | undefined
}
```

Duplet

Represents a two-dimensional value (media type `"application/vnd.vizrt.duplet"`). The object is **immutable**.

```
class Duplet {
  /** The x-value of the duplet. */
  x: number
  /** The y-value of the duplet. */
  y: number
  constructor(x: number, y: number)
```

```
}
```

Triplet

Represents a three-dimensional value (media type `"application/vnd.vizrt.triplet"`). The object is **immutable**.

```
class Triplet {
  /** The x-value of the triplet. */
  x: number
  /** The y-value of the triplet. */
  y: number
  /** The z-value of the triplet. */
  z: number
  constructor(x: number, y: number, z: number)
}
```

VizMap

Represents a Viz Engine map value (media type `"application/vnd.vizrt.curious.map"`). Contains data in a `|||`-delimited envelope format (for example, `Protocol=1|||XML=...|||`). The object is **immutable**.

```
class VizMap {
  /** The raw `|||`-delimited map string. */
  readonly mapString: string
  constructor(mapString: string)
}
```

MediaType

Union type for the supported media asset categories.

```
type MediaType = "image" | "video"
```

5.9.7 View Management

Views represent tabs in the payload editor UI. They are defined in the template model and allow organizing fields into logical groups.

View

Represents a single view/tab. Scripts can read the active state, activate a view programmatically, change the display title and inspect contained items.

```
interface View {
  /** Whether this view is currently the active/selected tab. */
  readonly isActive: boolean
  /** Activates this view, switching the UI to display this tab. */
  readonly activate: () => void
  /**
   * A decorative title that can be set by scripts for display purposes.
   * Does not affect the underlying model. Resets to the model default on session
reload.
   */
  title: string
  /** The items contained in this view. */
  readonly items: readonly ViewItem[]
}
```

Switching views example:

```
// Activate the second view
vizrt.views[1].activate()

// Rename a view tab
vizrt.views[0].title = "General Settings"
```

ViewItem

A union type representing any item within a view. Either a field item or a UI item (fragment/iframe).

```
type ViewItem = FieldViewItem | UiViewItem
```

FieldViewItem

A field item placed within a view, identified by its field path.

```
interface FieldViewItem extends BaseViewItem {
  readonly type: "field-item"
  /** The field path identifying this item within the payload. */
  readonly path: string
}
```

UiViewItem

A UI item (HTML fragment or iframe) placed within a view, identified by its name.

```
interface UiViewItem extends BaseViewItem {
```

```
  readonly type: "ui-item"
  /** The name identifying this UI item. */
  readonly name: string
}
```

BaseViewItem

Base interface for all view items, providing position management within the view's grid layout.

```
interface BaseViewItem {
  /** The current grid position and dimensions of this item. */
  readonly position: Rect
  /**
   * Updates the position of this item. Accepts a partial `Rect` — provided
   * properties are merged with the existing position.
   */
  readonly setPosition: (position: Partial<Rect>) => void
  /** The discriminator for the item type. */
  readonly type: "field-item" | "ui-item"
  /** The field path (only set for field items). */
  readonly path?: string
  /** The UI item name (only set for UI items). */
  readonly name?: string
  /** Whether the view item is hidden or not in the view
   *
   *  **Note**: Field's hidden expression evaluation overrides this value
   **/
  hidden?: boolean
}
```

Rect

Defines a rectangle within the view's 2D grid layout system. All values are in grid units.

```
type Rect = {
  /** The x-coordinate (column) of this rectangle. */
  readonly x: number
  /** The y-coordinate (row) of this rectangle. */
  readonly y: number
  /** The width of this rectangle in grid columns. */
  readonly width: number
  /** The height of this rectangle in grid rows. */
  readonly height: number
}
```

Repositioning a field and a UI component in a view example:

```
const items = vizrt.views[0].items
const headlineItem = items.find(item => item.type === "field-item" && item.path ===
"headline")
headlineItem?.setPosition({ x: 0, y: 0, width: 12, height: 2 })

const myFragment = items.find(item => item.type === "ui-item" && item.name ===
"fragment")
myFragment?.setPosition({ x: 6, y: 0, width: 5, height: 14 })
```

5.9.8 Pilot Data

PilotData

Provides read-only access to Pilot-specific metadata associated with the current template, including MOS data, Viz Mosart configuration, and Data Element information.

```
type PilotData = {
  /** Read-only Mosart configuration defaults. */
  readonly mosartConfig?: MosartConfig
  /** Mosart timing panel state, editable by scripts. */
  readonly mosartTimingPanel?: MosartTimingPanel
  /** MOS item data — description, slug, abstract, timing, and MEM blocks. */
  readonly mos: MOS
  /** Information about the current Data Element, if available. */
  readonly element?: PilotElementInfo
}
```

PilotElementInfo

Contains identification information for the current Data Element.

```
type PilotElementInfo = {
  /** The unique identifier of the Data Element. */
  readonly elementId?: string
}
```

5.9.9 MOS & MEM

MOS

Provides read/write access to MOS (Media Object Server) item fields (the standard metadata associated with newsroom items). Also provides access to MEM blocks and Viz Mosart timing values.

```
type MOS = {
  /** Mosart timing values (playout channel, timing, in/out modes). */
  mosart?: MosartTimingValues
  /** Continue count for Mosart playout automation. */
  continueCount?: number
  /** The MOS item description. */
  description?: string
  /** The MOS item abstract. */
  abstract?: string
  /** The MOS item slug. */
  slug?: string
  /** Access to MOS External Metadata (MEM) blocks. */
  readonly mem: MEMAccess
}
```

Setting MOS metadata example:

```
vizrt.$pilot.mos.description = "Election results lower third"
vizrt.$pilot.mos.slug = "ELECTION-2026"
```

MEMAccess

Provides CRUD operations for MOS External Metadata (MEM) blocks. MEM blocks are XML fragments stored within MOS items, each identified by a unique schema URL.

```
type MEMAccess = {
  /** Retrieves a MEM block by its schema URL, or `undefined` if not found. */
  readonly get: (schema: string) => MEMBlock | undefined
  /** Creates or updates a MEM block. */
  readonly set: (block: MEMBlock) => void
  /** Removes a MEM block by its schema URL. */
  readonly delete: (schema: string) => void
}
```

Managing MEM blocks example:

```
// Set a custom MEM block
vizrt.$pilot.mos.mem.set({
  schema: "http://example.com/schemas/custom-metadata",
  payload: "<metadata><key>value</key></metadata>",
  scope: "OBJECT",
})

// Read a MEM block
const block = vizrt.$pilot.mos.mem.get("http://example.com/schemas/custom-metadata")
if (block) {
  console.info("MEM payload:", block.payload)
}
```

```
// Remove a MEM block
vizrt.$pilot.mos.mem.delete("http://example.com/schemas/custom-metadata")
```

MEMBlock

Represents a single MOS External Metadata block.

```
type MEMBlock = {
  /** The schema URL uniquely identifying this MEM block type. */
  readonly schema: string
  /** The XML payload content of the MEM block. */
  readonly payload: string
  /** The scope of the MEM block, determining its visibility and lifecycle. */
  readonly scope: "PLAYLIST" | "STORY" | "OBJECT"
}
```

Scope	Description
`"PLAYLIST"`	The MEM block is scoped to the entire playlist.
`"STORY"`	The MEM block is scoped to the current story.
`"OBJECT"`	The MEM block is scoped to the individual MOS object.

MosartTimingValues

Defines the Viz Mosart playout timing configuration for a template. These values control how the graphics element is triggered and removed during broadcast automation.

```
type MosartTimingValues = {
  /** The playout channel name. */
  channel: string
  /** The start time for playout. */
  start: string
  /** The duration of playout. */
  duration: string
  /** The in-transition mode. */
  inMode: InMode
  /** The out-transition mode. */
  outMode: OutMode
  /** Whether this element is used as a locator graphic. */
  isLocator: boolean
}
```

MosartConfig

Read-only Viz Mosart configuration defaults for the template. These values are set by the system and cannot be modified by scripts.

```
type MosartConfig = {
  /** Whether the Mosart timing panel should be displayed. */
  readonly showTimingPanel: boolean
  /** The default playout destination channels. */
  readonly destinations: readonly string[]
  /** Whether the "is Locator" checkbox should be shown by default in the timing
panel. */
  readonly showIsLocator: boolean
}
```

MosartTimingPanel

The editable state of the Viz Mosart timing panel for the current template. Scripts can control which options are available and the panel's default expanded state.

```
type MosartTimingPanel = {
  /** The playout destinations available for this template. */
  destinations: readonly string[]
  /** Whether to display the "is Locator" checkbox. */
  showIsLocator: boolean
  /** Whether to display the "Continue Count" option. */
  showContinueCount: boolean
  /** Whether the timing panel is expanded by default. Defaults to `true`. */
  expanded: boolean
}
```

InMode

The in-transition mode for Viz Mosart playout automation.

```
type InMode = "auto" | "manual"
```

Value	Description
`"auto"`	The graphic is taken on-air automatically by the automation system.
`"manual"`	The graphic requires manual triggering by an operator.

OutMode

The out-transition mode for Vi Mosart playout automation.

```
type OutMode = "auto" | "story-end" | "background-end" | "open-end"
```

Value	Description
`"auto"`	The graphic is taken off-air automatically after its duration.
`"story-end"`	The graphic remains on-air until the current story ends.
`"background-end"`	The graphic remains on-air until the background element ends.
`"open-end"`	The graphic remains on-air indefinitely until manually removed.

5.9.10 MSE Connection & Playout

MseConnection

Provides methods for controlling graphics playout through a Media Sequencer Engine (MSE). Create an instance using `app.createMseConnection(url)`.

```
type MseConnection = {
  sendVizCommand: (target: PlayoutTarget<undefined>, ...commands: readonly string[])
=> void
  take: (target: PlayoutTarget<VizLayer>) => void
  update: (target: PlayoutTarget<VizLayer>) => void
  continue: (target: PlayoutTarget<VizLayer | string>) => void
  out: (target: PlayoutTarget<VizLayer | string>) => void
  checkConnection: () => Promise<boolean>
}
```

Method	Description
`sendVizCommand(target, ...commands)`	Sends one or more raw Viz Engine commands through the MSE. The `layer` property on the target is not used.

Method	Description
`take(target)`	Executes a *take* command, bringing the graphic on-air on the specified target and layer.
`update(target)`	Executes an *update* command, refreshing the on-air graphic with current field values.
`continue(target)`	Executes a *continue* command, advancing the graphic to its next animation state.
`out(target)`	Executes an *out* command, removing the graphic from air on the specified target.
`checkConnection()`	Returns a `Promise<boolean>` indicating whether the MSE connection is valid.

Playout workflow example:

```
const mse = app.createMseConnection("http://mse-host:8580")

const target = { profile: "STUDIO_A", channel: "GFX", layer: "MAIN" as const }

// Check connection first
const connected = await mse.checkConnection()
if (connected) {
  mse.take(target)
}
```

PlayoutTarget<Layer>

Identifies the MSE profile, channel and optional Viz Engine layer for a playout operation.

```
type PlayoutTarget<Layer> = {
  /** The MSE profile name. */
  readonly profile: string
  /** The MSE channel name within the profile. */
  readonly channel: string
  /** The Viz Engine layer to target. Defaults to `"MAIN"` if omitted. */
  readonly layer?: Layer
}
```

VizLayer

The available Viz Engine rendering layers.

```
type VizLayer = "MAIN" | "FRONT" | "BACK"
```

Layer	Description
`"MAIN"`	The primary rendering layer for the main graphics scene.
`"FRONT"`	The front layer, rendered on top of the main layer.
`"BACK"`	The back layer, rendered behind the main layer.

5.9.11 Factory Functions

createRichText

Creates an immutable `RichText` object from a plain text string.

```
declare function createRichText(plainText: string, escape?: boolean): RichText
```

Parameter	Type	Default	Description
`plainText`	`string`	—	The plain text content.
`escape`	`boolean`	`true`	When `true`, escapes HTML characters (`<`, `>`) so they display literally. Set to `false` if the text is already properly formatted.

Example:

```
vizrt.fields.$body.value = createRichText("Breaking news: <important>")
// The "<important>" text is escaped and displayed literally
```

Formatted Text fields can be assigned by creating an object of type RichText in the method *createRichText*:

```
vizrt.fields.$name.onChanged = value => {
    vizrt.fields["$01-richtext"].value = createRichText(
    `<fo:wrapper bold="false" italic="false" underline="false">Hello </fo:wrapper>` +
    `<fo:wrapper bold="true" italic="true" underline="true">${value}!</fo:wrapper>`,
false)
```

```
}
```

Info: The second argument to the method is set to false. This ensures the formatted text is sent to the Viz Engine unescaped. If set to true, the text is escaped, but the formatting is lost.

false: When the actual RichText XML should be sent to the Viz Engine.
true: When the text does not contain RichText, but contains characters that require escaping, such as `<` and `>`.

createImageAsset

Creates an ImageAsset object that can be assigned to Image fields.

```
declare function createImageAsset(image: string, title?: string): ImageAsset
```

Parameter	Type	Description
`image`	`string`	External URL, Graphic Hub image path or UUID.
`title`	`string`	Optional display title. Not required for Graphic Hub images.

Example:

```
vizrt.fields.$insta.$image2.value =
createImageAsset("IMAGE*&lt;520E0100-56BD-5C4D-884D-5598ACFD75A4&gt;")
vizrt.fields.$insta.$image2.value = createImageAsset("IMAGE*01_GLOBALS/IMAGE/
logo_wide")
vizrt.fields.$insta.$image2.value = createImageAsset("https://images.pexels.com/
photos/169647/pexels-photo-169647.jpeg", "HEY")
```

createVideoAsset

Creates a `VideoAsset` from a local file path on the playout engine.

```
declare function createVideoAsset(video: string, title?: string): VideoAsset
```

Parameter	Type	Description
`video`	`string`	The file path to the video on the playout engine. Can be a full path or relative to the clip root on the Viz Engine.
`title`	`string`	Optional display title for the video.

Example:

```
vizrt.fields.$00Video.value = createVideoAsset("C:\\clips\\CVJPHFMXDHBGZFMF.mxf",
"Title")
vizrt.fields.$00Video.value = createVideoAsset("opener.mxf", "Title")
```

createImageAssetFromXml

Creates an `ImageAsset` from a raw XML string containing an `atom:entry` representation of the image.

```
declare function createImageAssetFromXml(imageXml: string): ImageAsset
```

createVideoAssetFromXml

Creates a `VideoAsset` from a raw XML string containing an `atom:entry` representation of the video.

```
declare function createVideoAssetFromXml(videoXml: string): VideoAsset
```

5.9.12 Utility Functions

reportErrors

Catches and reports errors from callback or asynchronous functions to the UI. Use this to wrap `catch` handlers in promise chains so that exceptions are surfaced to the user rather than silently swallowed.

```
declare function reportErrors(f: () => void, stage: string): void
```

Parameter	Type	Description
`f`	`() => void`	A function to execute. Any exception thrown within it is reported to the UI.
`stage`	`string`	A descriptive label for the operation, shown in the error notification.

Example:

```
fetch("https://api.example.com/data")
  .then(response => response.json())
```

```
  .then(data => {
    vizrt.fields.$headline.value = data.title
  })
  .catch(e =>
    reportErrors(() => {
      throw e
    }, "fetch data"),
  )
```

5.9.13 Spell Check

vizrt.spellCheck

Invokes the spell check dialog on text fields. The user can review and correct misspelled words interactively.

```
readonly spellCheck: (languageCode?: string, fields?: readonly string[]) =>
Promise<boolean>
```

Parameter	Type	Default	Description
`languageCode`	`string`	`"en_US"`	The dictionary language code (e.g., `"en_US"`, `"nb_NO"`). If omitted, uses the `vizrt-spellcheck-language` environment variable, or defaults to `"en_US"`.
`fields`	`readonly string[]`	all text fields	An array of field paths to check. If omitted, all applicable text fields are checked.

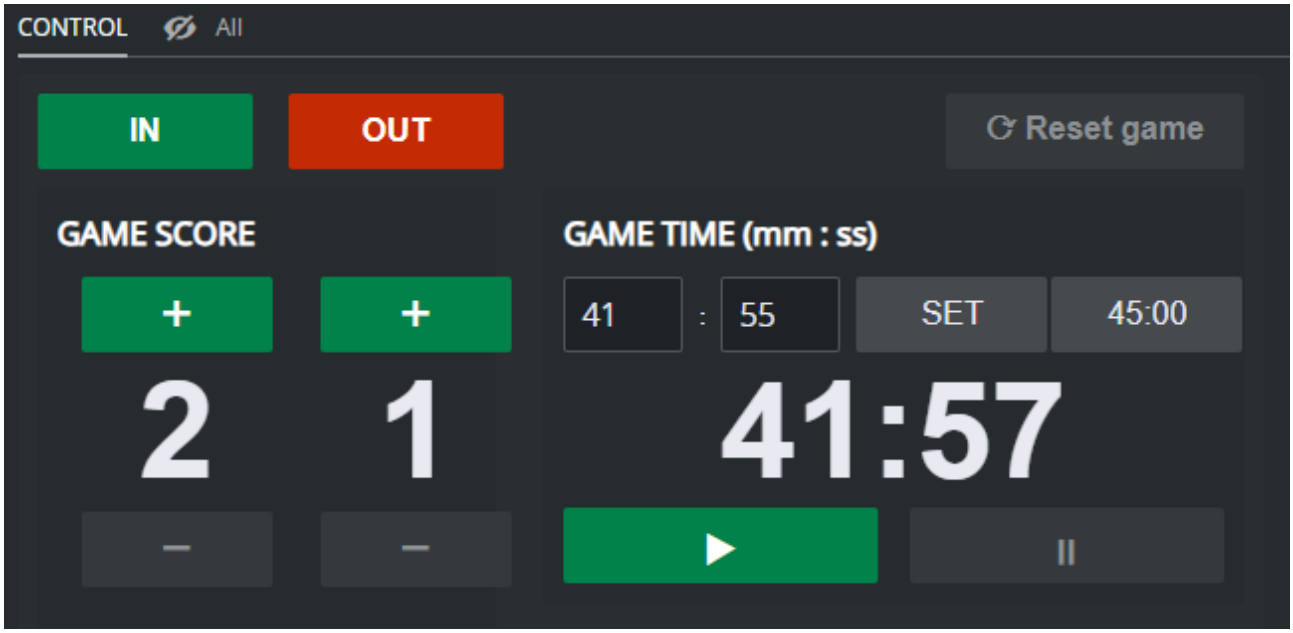
Returns `Promise<boolean>`, resolves to `true` if the spell check completed, or `false` if the user cancelled.

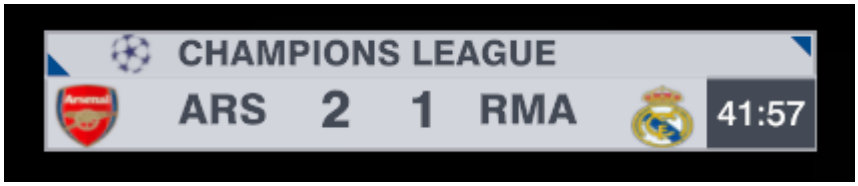
Example:

```
// Check all text fields in Norwegian
const completed = await vizrt.spellCheck("nb_NO")

// Check specific fields only
await vizrt.spellCheck("en_US", ["headline", "body"])
```

6 Action Panels

Action Panels are regular Viz Pilot Edge templates or data elements, but are designed to remain open in the client and function as control surfaces. In Viz Pilot Edge, add the URL parameter **&action** to open the template in action mode. This page walks through a practical approach to building an action panel template to control a typical **clock-and-score panel**.





This page contains the following topics:

Like all templates in Template Builder, Action Panels require an imported scene. However, if the panel's actions are not directly tied to the scene content, the scene can be a dummy scene containing only a ControlObject plugin.

When an Action Panel is created and a scene is imported, Template Builder automatically generates input fields for the controllable objects defined in the scene. These fields can then be used to drive scene playback from the template. For example, in a **clock and score panel**, you might expose the home and away team names as text fields. When this template is opened in Viz Pilot Edge, users can enter team names and save the template as a **data element** for a specific match. This way, action panels can be prefilled and reused with specific content.

It is important to emphasize that **Action Panels are regular templates and data elements** (all standard mechanisms to build templates in Template Builder apply).

> ⚠ **Note:** To host action panel templates or data elements in your own client, add the *&action* URL parameter to the Viz Pilot Edge URL.

See the section Working with Templates for details on template creation and editing.

6.1 Key Concepts

Template Builder provides several mechanisms that are especially useful when building an action panel template:

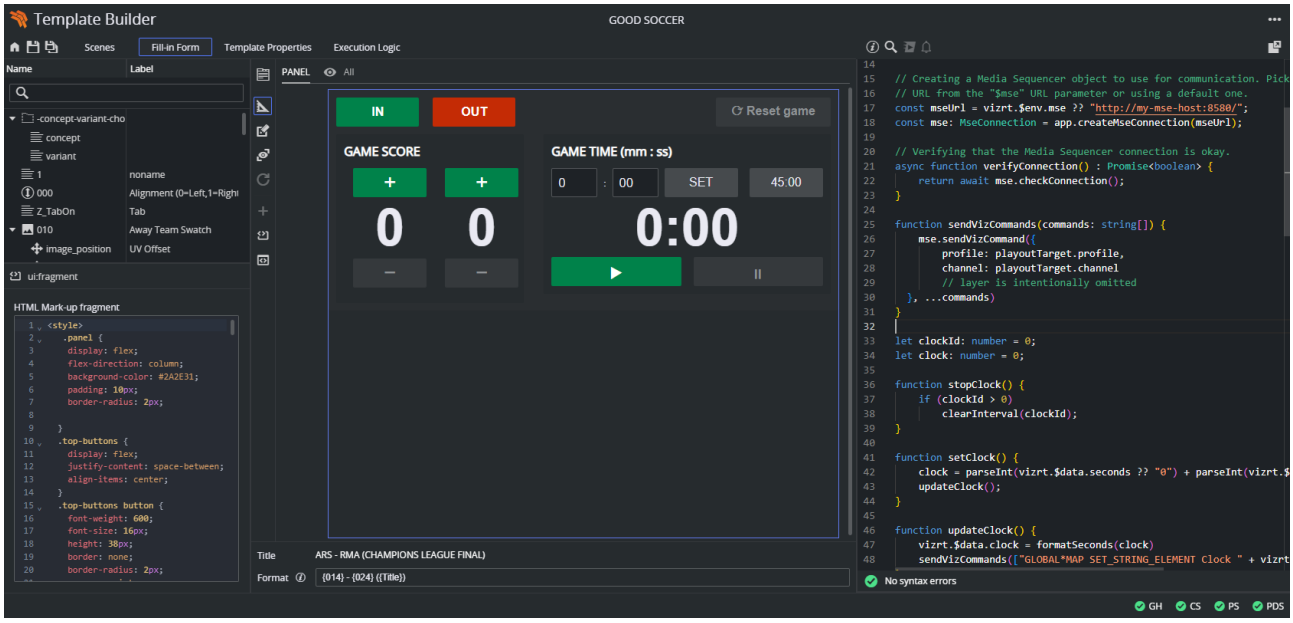
- **HTML fragments** inside the template allow fully customized panel layouts:
  - Style the content using standard CSS.
  - Use `<button>` elements with the `vizrt-click-name` attribute to define click-event handlers in internal scripting.
  - Use HTML form controls such as `<input>` and `<select>` with the `vizrt-input-name` attribute to enable two-way binding between control values in the HTML fragment, and corresponding fields in the template or the `vizrt.$data` map used in internal scripting.
- **Internal script commands** let you perform common actions programmatically:
  - **Media Sequencer operations:** `app.createMseConnection(...)` creates a Media Sequencer object and lets you execute Media Sequencer commands such as `"take"`, `"continue"`, `"update"`, or `"out"` for the current template or data element.
  - **Viz commands:** `mse.sendVizCommand(...)` sends Viz Engine commands (through Media Sequencer), such as setting shared memory variables, starting directors, or loading and updating scenes.
  - **User notifications:** `app.notify(...)` displays a warning, error, or info message to the user.

6.2 UI Design with HTML Fragments

The user interface of an action panel template can be built just like any ordinary template. However, a particularly powerful technique is using HTML fragments to customize the layout. In fact, the entire UI can be constructed within a single HTML fragment.

To get started:

- Add a new **tab** in your template.
- Add an **HTML fragment** to this tab and resize it to the desired size.
- Use **CSS** for layout and visual styling, either inline or inside a `<style>` tag within the fragment.



6.2.1 Enable User Interaction

There are two key mechanisms that enable interaction in the Action Panel UI:

1. Buttons

You can add `<button>` elements inside the HTML fragment to trigger scripted actions. Use the special attribute `vizrt-click-name` to identify which action should be triggered when a button is clicked.

```
<button vizrt-click-name="take">IN</button>
```

In your internal script, the `vizrt.onClick(name)` handler receives the value of `vizrt-click-name` and can react accordingly:

```
vizrt.onClick = (name: string) => {
  switch(name) {
    case "take":
      mse.take(playoutTarget)
      break
    // more cases...
  }
}
```

Use case: You want the panel operator to manually trigger a "take" playout with a single button click.

2. HTML Form Controls with Bindings

To capture input from the user, use standard HTML form controls like `<input>` and `<select>`. These elements can be **two-way bound** to either:

- A **template field** (linked to a control object in the scene).
- A field in the `vizrt.$data` map (used in internal scripting).

Use the `vizrt-input-name` attribute to bind the control:

Example 1 - Bind to Field

```
<input vizrt-input-name="title" type="text" value="MATCH DAY">
```

This binds the value of the input box to the field “title” in the template.

Example 2 - Bind to Scripting

```
<input vizrt-input-name="$data.home_score" type="number" value="0">
```

This example shows an input box where the user can type a numeric value, a two-way value bound to the value of the given key in the *$data* map in internal scripting: `vizrt.$data.home_score`. This variable can also be manipulated in the script, for instance increasing the score with 1, the new value is shown in the corresponding HTML form control.

Use case: You want the operator to input or adjust values like team names, scores, or time values, which the script can then access or modify.

6.3 Internal Scripting

Once the UI is in place, internal scripting provides the logic to execute actions in response to user input.

6.3.1 Trigger a Media Sequencer (MSE) Command

To create an object to perform Media Sequencer operations, use:

```
const mse: MseConnection = app.createMseConnection(mseUrl);
```

Creating the object with an URL as parameter does not automatically connect to Media Sequencer. The object uses the Media Sequencer REST API, and does not have callback events.

The first operation to execute after creating this object, is to check whether the connection is valid. To do this, you may create a function:

```
// Verifying that the Media Sequencer connection is okay.
async function verifyConnection() : Promise<boolean> {
    return await mse.checkConnection();
}
```

And call it as such:

```
if (!(await verifyConnection())) {
  app.notify("error", "Cannot connect to the Media Sequencer.");
}
```

If this operation returns true, you are ready to trigger playout commands on the current template or data element.

All commands have a `PlayoutTarget` object as a parameter. This object has the following members:

- `profile` : The MSE profile to use.
- `channel` : The channel to play out on.
- `layer` (optional): Viz layer (one of, `"MAIN"` , `"BACK"` , `"FRONT"`).

```
// Creating a default playout target.
const defaultPlayoutTarget: PlayoutTarget<VizLayer> = {
    profile: "default",
    channel: "A",
    layer: "MAIN"
};
```

Use Environment Variables from URL Parameters

In the example above, the playout target and Media Sequencer URL is hard coded in scripting. These values can also be provided from environment variables specified in the Viz Pilot Edge URL parameters, for instance:

`http://pds-host:8177/app/pilotedge/pilotedge.html?template=28935&$mse=http://custom-mse-host:8580/&$profile=myprofile1&$channel=B&action`

> ⚠ **Note:** These `$` -prefixed parameters are only parsed by Viz Pilot Edge and Template Builder, and are not treated as regular query parameters in HTTP requests.

URL parameters starting with a dollar sign are treated as an environment variable. In scripting, the environment variables can be accessed through the `vizrt.$env` dictionary:

```
// Creating a Media Sequencer object to use for communication. Picking up the
// URL from the "$mse" URL parameter or using a default one.
const mseUrl = vizrt.$env.mse ?? "http://default-mse-host:8580/";
const mse: MseConnection = app.createMseConnection(mseUrl);
```

```
// Creating the playout target to use, based on URL parameters or the default target
above.
const playoutTarget: PlayoutTarget<VizLayer> = {
    profile: vizrt.$env.profile ?? defaultPlayoutTarget.profile,
    channel: vizrt.$env.channel ?? defaultPlayoutTarget.channel,
};
```

6.3.2 Send Viz Engine Commands

To send commands directly to Viz Engine (for example, set shared memory, update scenes), use `mse.sendVizCommand (playoutTarget, commands)`. Commands is an array of strings containing the Viz Engine commands to send. The `PlayoutTarget` parameter is described above.

> **Note:** When using `mse.sendVizCommand(...)`, you **must not** include the `layer` property in the `PlayoutTarget`. If you do, **Template Builder raises a compile-time error**, because the layer must be specified explicitly in the command strings, not in the target.

> **Use case:** You want to update a scene's data or trigger a director without playing out a new template.

The `commands` array contains Viz command strings.

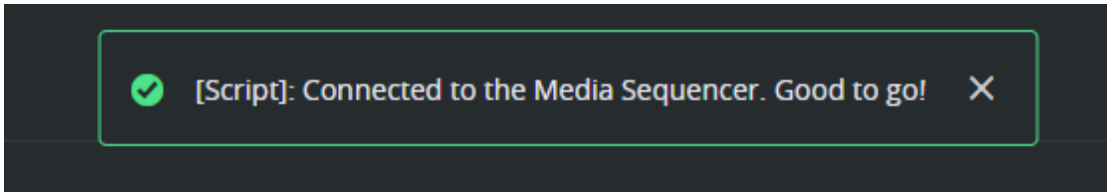
```
function sendVizCommands(commands: string[]) {
    mse.sendVizCommand({
        profile: playoutTarget.profile,
        channel: playoutTarget.channel
        // layer is intentionally omitted
  }, ...commands)
}

function updateAwayScore() {
    sendVizCommands(["GLOBAL*MAP SET_STRING_ELEMENT Soccer_AwayTeamScore " + vizrt.
$data.away_score])
}
```

6.3.3 Display a Message to the User

Use `app.notify(...)` to show non-blocking notifications (toast messages) in the UI:

```
app.notify("success", "Connected to the Media Sequencer. Good to go!");
```

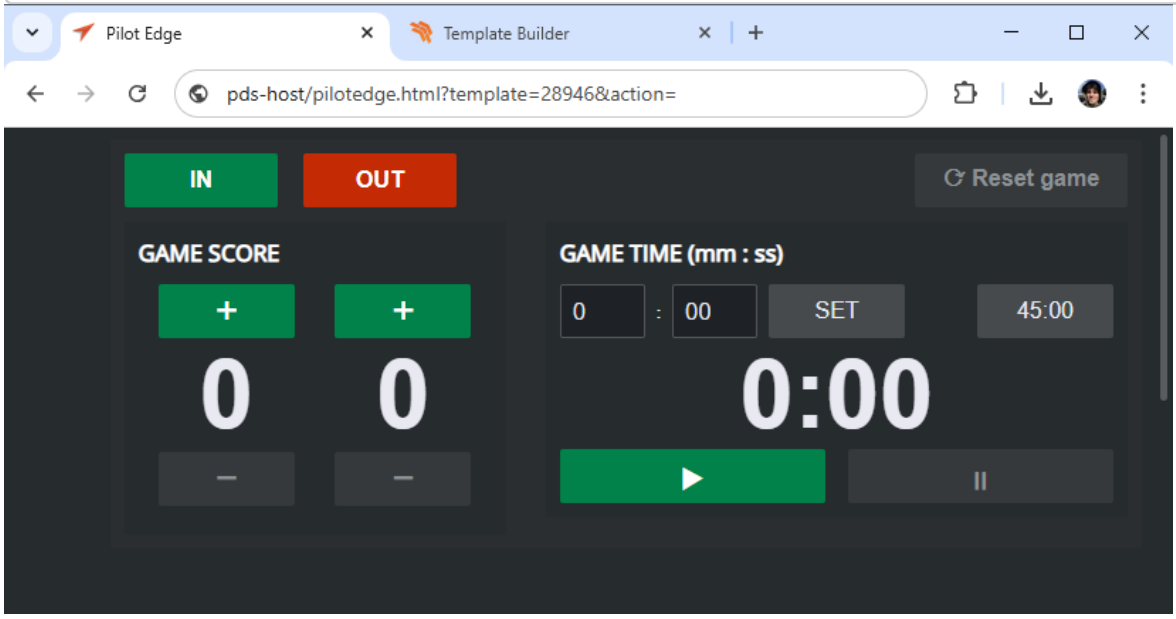
- `severity` : `“success", "info"`, `"warning"`, or `"error"`.
- `message` : The text to display.



6.4 Host the Action Panel

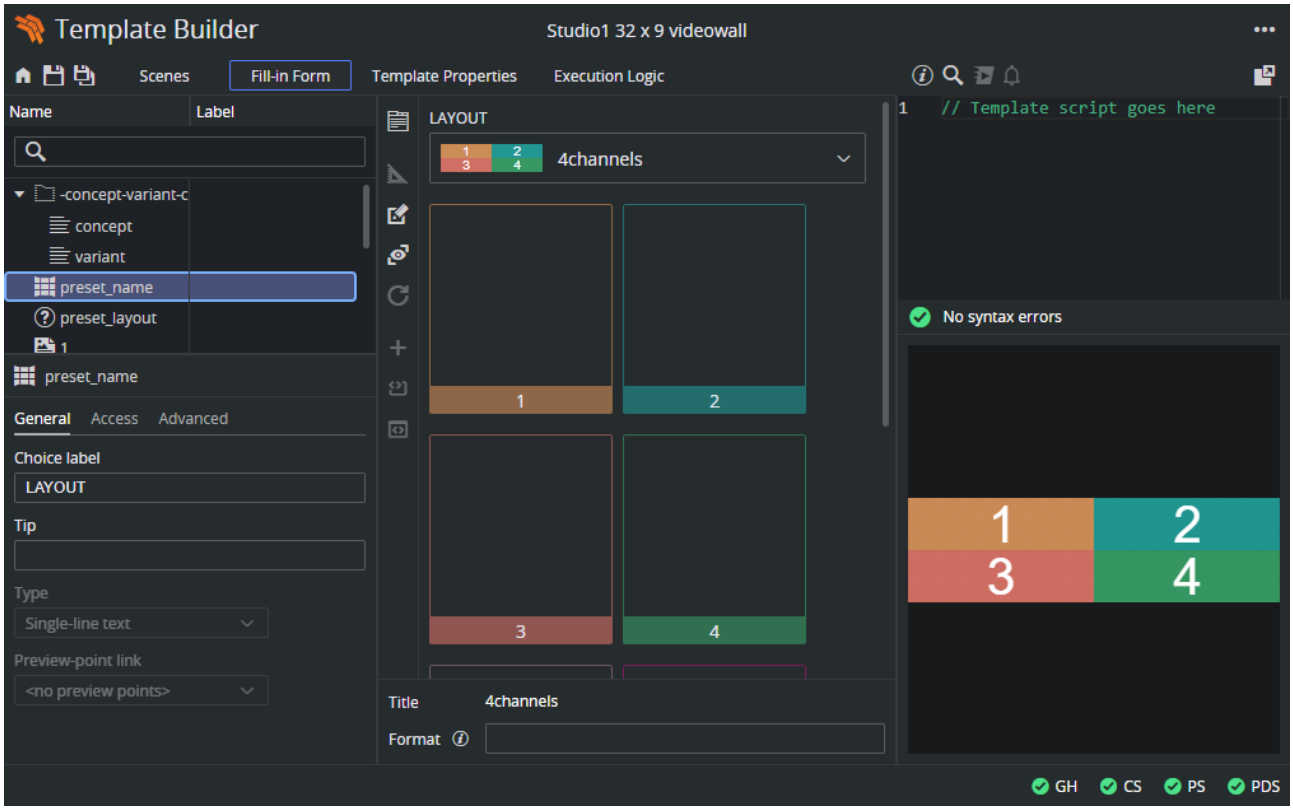
The Action Panel template can be tested from Template Builder and Viz Pilot Edge. When hosted in Viz Pilot Edge or custom web clients that embed Viz Pilot Edge, the template can be shown in "action mode" using the `&action` parameter, displaying only the Action Panel interface. This can be done by adding the URL parameter `&action` to the Viz Pilot Edge URL.

```
http://pds-host:8177/app/pilotedge/pilotedge.html?template=28580&action
```



7 Multiplay Presets

Template Builder supports importing Viz Multiplay preset scenes, that typically use the Viz Artist Preset plugin to define layout presets with superchannels. Each preset contains a layout with up to 16 superchannels. When imported into Template Builder, a preset template is created. The fill-in form displays a dropdown menu to select among the available presets (layouts), along with an editor for each superchannel in the selected preset.



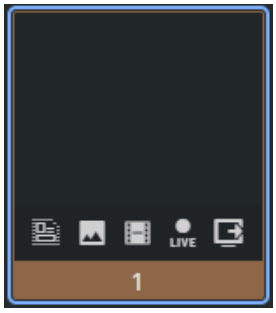
7.1 Creating a Preset Template

A Preset Template is based on a Viz Artist scene that includes the Preset plugin. This plugin can be manually added in Viz Artist, where presets can also be created, edited and removed. However, a more convenient workflow is to use **Viz Multiplay**, which includes a Video Wall Designer with intuitive drag-and-drop functionality to design layouts visually.

When a preset scene is saved in Viz Multiplay, it generates a Viz Artist scene containing all preset data (such as the dimensions and positions of superchannels, and a name for each preset). This scene is then published to **Graphic Hub**, from where Template Builder can import it into any existing concept, using the standard Pilot workflow. The result is a regular Viz Pilot Edge template.

7.2 Adding Default Content to Channels

Each superchannel can be filled with default content such as graphics, images, videos, live channels, or an "out" action to clear the channel. Leaving a superchannel empty is interpreted as **"ignore"**, meaning the current content in the channel remains unchanged.



Each superchannel field in the template includes the following options:

- **Graphics:** Select graphics to display in the channel. This can be either an existing data element or a template that needs to be filled out.
- **Images:** Browse and select an image to display. The image can be cropped with the image editor.
- **Videos:** Browse and select a video clip to be played in the channel.
- **Live:** Select a live input source. The actual content depends on how input channels are configured on the Viz Engine playout machine.
- **Out:** Clears the channel by taking the current content out.

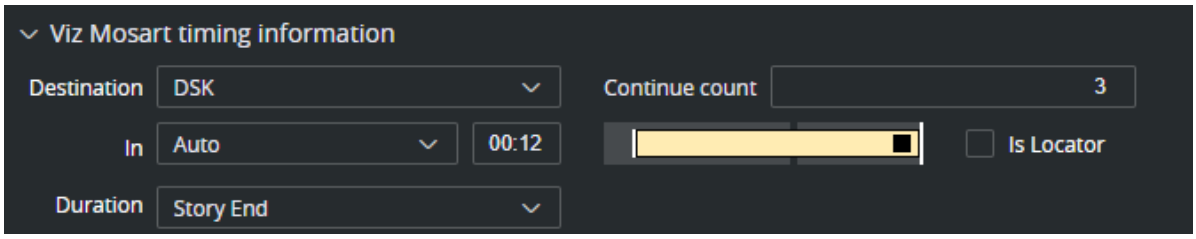
7.3 Saving a Preset Template

When a preset template is saved, the currently selected layout preset and the default content for each superchannel are stored in the template. This defines the default state when the template is opened in Viz Pilot Edge.

Preset templates are standard Viz Pilot Edge templates and can belong to any concept. To control their visibility in the client, tags can be applied to the templates and used in combination with the **Pilot Collections** feature, which allows templates to be shown or hidden based on tags.

8 Viz Mosart Timing Information

The Viz Mosart Timing Panel is now integrated directly into Viz Pilot Edge, eliminating the need to embed the timing UI within individual templates.



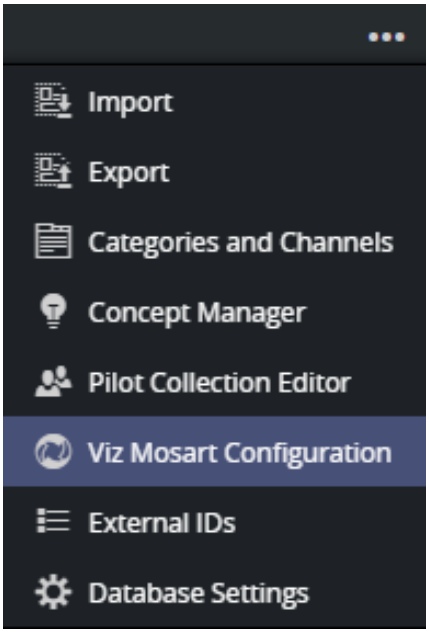
Historically, Viz Mosart timing information was stored directly within data elements, meaning each element had a fixed timing configuration. With the introduction of the Viz Mosart Timing Information mechanism, timing is now transmitted only via the generated MOS XML. This change delegates the lifetime and ownership of the timing data to the newsroom system, enabling one data element to be reused with varying in/out points in different rundowns.

This section contains the following topics:

8.1 Configuration

The Viz Pilot backend must be aware of available Viz Mosart destinations. These must be configured manually via Template Builder:

1. Open **Viz Mosart Configuration** from the Template Builder tools menu.

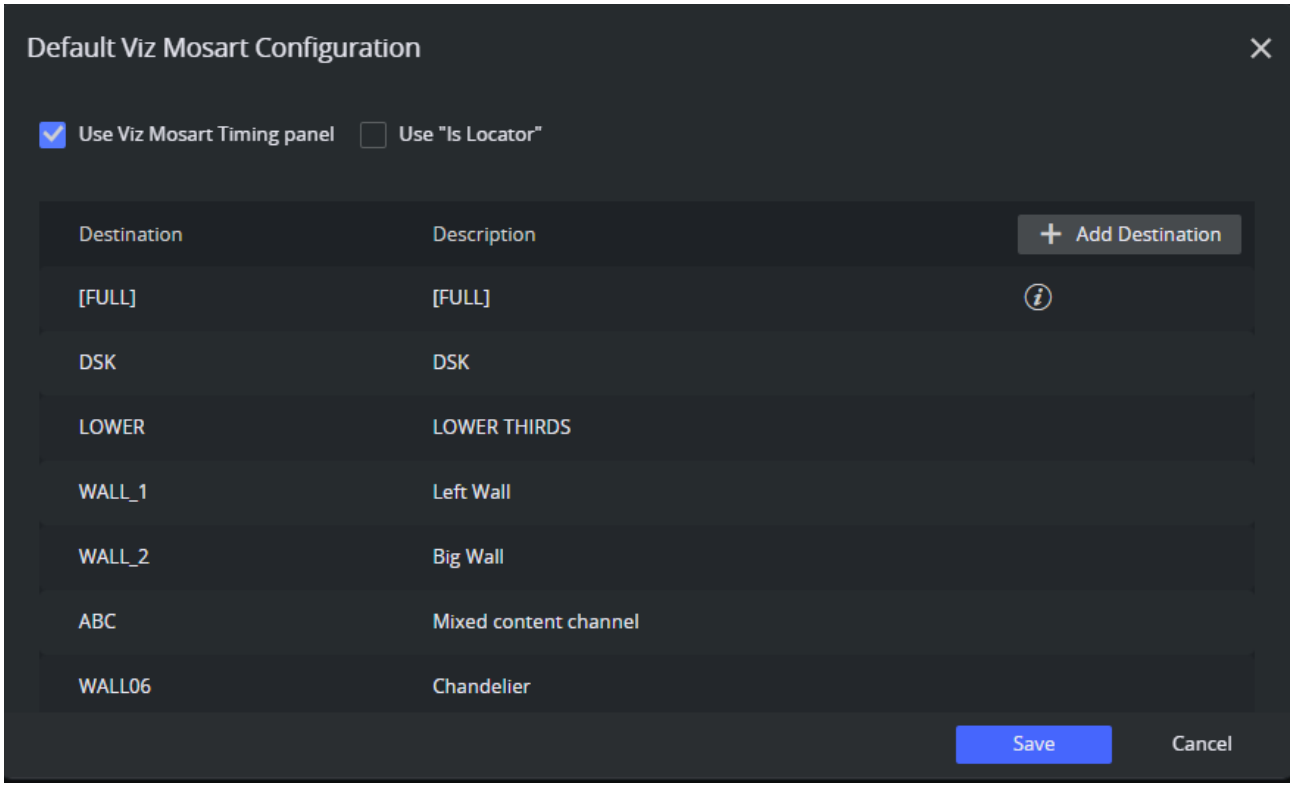


2. Add Viz Mosart destinations to match your Viz Mosart system. A destination consists of a Channel and an optional Description. The description is used in the UI only.

Note: One default destination called **[FULL]** is represented internally as an empty string (""). This indicates the graphic is a primary full-screen event, not a secondary event, and therefore cannot have timing information in a Viz Mosart system.

3. Enable the timing panel using:
   - **Use Viz Mosart Timing Panel**: Displays the panel at the bottom of regular Viz Pilot Edge templates.
   - **Use "Is Locator"**: Includes a checkbox in the timing panel to set the "Locator" flag.

Info: In Viz Mosart, a **Locator** is a graphic linked to a video switcher crosspoint. Taking the crosspoint (as a switcher or keyed crosspoint) also takes the graphic.



When the configuration is saved, the Viz Mosart Timing Panel becomes visible in all applicable Viz Pilot Edge templates.

> ⚠ **Note:** The [FULL] entry is mandatory. It results in an empty <channel> field in the MOS XML and disallows timing, as full-screen graphics are treated as primary events.

8.2 MOS XML

When Viz Mosart Timing Panel is enabled, timing data is not stored in the data element but included in a `mosExternalMetadata` block within the generated MOS XML:

```
<mosExternalMetadata>
    <mosScope>PLAYLIST</mosScope>
    <mosSchema><http://www.vizrt.com/mosObj/mosart</mosSchema>>
    <mosPayload>
        <channel>DSK1</channel>
        <start>00:10</start>
        <duration>00:05</duration>
        <inMode>auto</inMode>
        <outMode>auto</outMode>
        <isLocator>false</isLocator>
    </mosPayload>
</mosExternalMetadata>
```

Notes:

- Timing information is **not persisted** in the data element itself.
- The newsroom system retains control over timing.
- Older mechanisms (for example, storing timing in the `mosart` field or description) remain supported.

8.3 Customization through Scripting

You can use scripting to tailor the behavior of the timing panel, per template. The following objects are accessible:

- **$pilot.mosartConfig** (read-only): Structure containing the Viz Mosart configuration in the database settings.
  - **.destinations[]** (list of strings): List of configured Viz Mosart destinations.
  - **.showTimingPanel** (boolean | undefined): Whether to show the panel.
  - **.showIsLocator** (boolean | undefined): Whether the “is Locator” checkbox is shown or not in the Viz Mosart timing panel.
- **$pilot.mosartTimingPanel**
  - **.destinations []** (list of strings): Overrides available destinations per template.
  - **.showIsLocator** (boolean): Overrides the global "Is Locator" setting.
  - .**showContinueCount** (boolean): Show/hide "Continue Count" input.
  - **.expanded** (boolean): Expand/collapse the panel by default (default: `true`).
- **$pilot.mos**
  - **.mosart**
    - **.channel** (string): Destination channel.
    - **.start** (string): Start time.
    - **duration** (string): Duration.
    - **inMode** (InMode): Can be set to "auto" or "manual".
    - **outMode** (OutMode): Graphics outMode can be set to "auto", "story-end", "background-end" or "open-end".
    - **isLocator** (boolean): Indicates whether this graphics should be treated as a locator.
  - **.continueCount** (number | undefined): Optional override for the current data element.

8.4 Examples

8.4.1 Limit Destinations per Template

Only allows a specific set of destinations for a given template:

```
// When a new data element is created, we set the default destinations of the Mosart
// timing panel and the default value to be selected. We need to check whether  these
// objects are defined. They may be undefined if configured to not be used, or no
// Mosart configuration is added to the Pilot system.
vizrt.onCreate = () => {
  if (vizrt.$pilot.mosartTimingPanel && vizrt.$pilot.mos.mosart) {
    vizrt.$pilot.mosartTimingPanel.destinations = ["LOWER", "DSK"];
```

```
    vizrt.$pilot.mos.mosart.channel = "LOWER";
  }
}
```

8.4.2 Filter Destinations

Excludes destinations that begin with "WALL":

```
if (vizrt.$pilot.mosartTimingPanel && vizrt.$pilot.mos.mosart && vizrt.
$pilot.mosartConfig) {
  vizrt.$pilot.mosartTimingPanel.destinations = vizrt.
$pilot.mosartConfig.destinations.filter(
    (destination: string) => !destination.startsWith("WALL"));
  }
}
```

8.4.3 Enabling only Full Screen

Only use the `[FULL]` destination (represented by an empty string):

```
// Empty channel is treated as the full screen destination in Viz Mosart.
vizrt.onCreate = () => {
  if (vizrt.$pilot.mosartTimingPanel && vizrt.$pilot.mos.mosart) {
    vizrt.$pilot.mosartTimingPanel.destinations = [""];
    vizrt.$pilot.mos.mosart.channel = "";
  }
}
```

8.4.4 Conditionally Show "Is Locator"

Enable the "Is Locator" checkbox, only in specific templates:

```
vizrt.onCreate = () => {
  if (vizrt.$pilot.mosartTimingPanel) {
    vizrt.$pilot.mosartTimingPanel.showIsLocator = true;
  }
};
```

8.4.5 Set Default Timing Values

Prepopulate the timing fields:

```
vizrt.onCreate = () => {
```

```
  // Setting initial timing values
  if (vizrt.$pilot.mos.mosart) {
    vizrt.$pilot.mos.mosart.start = "00:12";
    vizrt.$pilot.mos.mosart.inMode = "auto";
    vizrt.$pilot.mos.mosart.outMode = "story-end";
    vizrt.$pilot.mos.mosart.channel = "DSK";
  }

  // Overriding the continue count of the scene
  if (vizrt.$pilot.mosartTimingPanel) {
    vizrt.$pilot.mosartTimingPanel.showContinueCount = true;
    vizrt.$pilot.mos.continueCount = 3;
  }
};
```

9 Asset Hosting

9.1 Introduction

Asset Hosting allows Viz Pilot Edge to embed a custom **asset editor UI** inside an `<iframe>`. This enables third-party or custom web applications to act as *asset guests* that create, modify, or visualize assets (images and videos) used by templates.

Communication between the **host** (Viz Pilot Edge) and the **guest** (your asset UI) is done using the browser standard `window.postMessage` API. The host controls lifecycle and context, while the guest is responsible for presenting UI and emitting updated asset data.

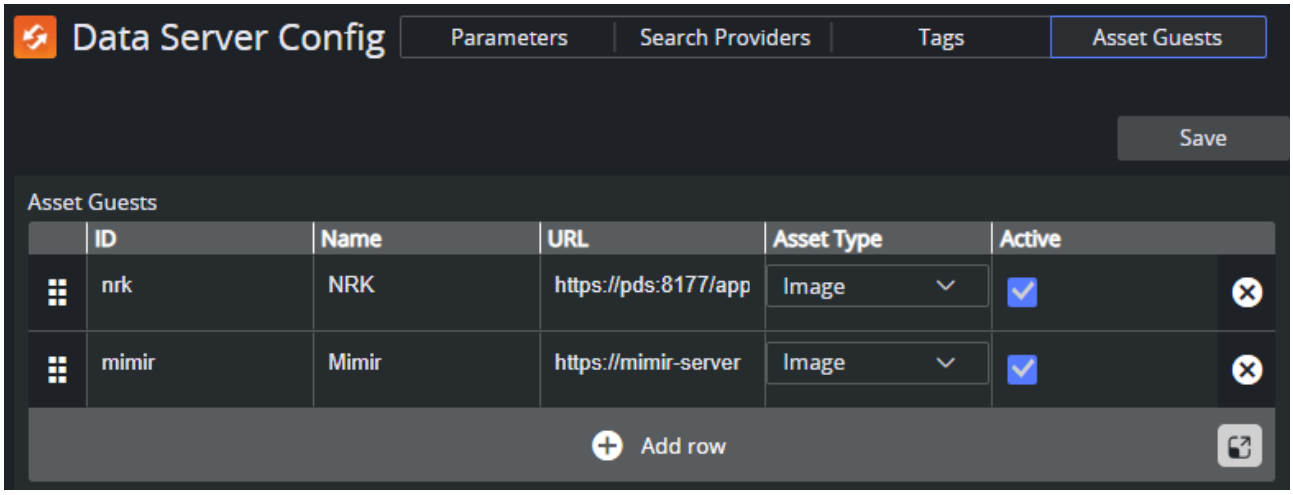
The basic architecture is as follows:

- The **host** embeds the guest using an `<iframe>`.
- The **guest** reads configuration from query parameters.
- Both sides exchange structured messages using `window.postMessage`.

9.2 Configuration

This section describes how Asset Guests are configured to appear in Template Builder and Viz Pilot Edge.

Asset Guests are registered in **Data Server Config**, where each guest editor is defined and made available to the system.



- Open **Data Server Config**.
- Navigate to the **Asset Guests** section.
- Add one row for each Asset Guest you want to register.
  - **ID:** A unique identifier used by Viz Pilot Edge to associate assets with a specific guest editor. The value can be any unique string, but must remain stable once the assets are created using this guest.
  - **Name:** The display name of the Asset Guest. This name appears in the search and selection UI when users choose an editor.
  - **URL:** The full URL of the Asset Guest UI. Viz Pilot Edge loads this URL inside an `<iframe>` when the guest editor is opened.
  - **Asset Type:** The type of assets this asset guest provides, which can be an image or video.
  - **Active:** Controls whether the Asset Guest is available in Viz Pilot Edge. When disabled, the guest is not shown in the UI.
- Click **Save**.

9.3 Asset Host API

The Asset Host API is a small, message-based protocol over `window.postMessage`. Messages are JavaScript objects with a mandatory `type` property and optional additional fields.

The API is **asymmetric**:

- The **host** drives state by sending the current asset to the guest.
- The **guest** notifies the host when the asset has changed.

All messages sent **from the guest to the host** must:

- Include the `guestid`.
- Use the host origin as `targetOrigin`.

9.3.1 Query Parameters

When loading the asset guest iframe, the host (Viz Pilot Edge) appends query parameters that the guest must read.

- `asset_host_origin` : Specifies the origin of the host application. The guest **must** use this value as the `targetOrigin` when calling `postMessage` .
- `guestid` : Uniquely identifies this guest instance. This value must be echoed back in all messages sent to the host.

9.3.2 Guest > Host Messages

- `asset_guest_loaded` : Sent by the guest when it has finished loading and is ready to receive messages.

Example:

```
window.parent.postMessage(
  { type: "asset_guest_loaded", guestid },
  hostOrigin
)
```

The purpose of this asset is to signal readiness, and allows the host to safely send `set_asset` .

- `asset_changed` : Sent by the guest when the user selects an asset to be displayed in the host.

Example:

```
window.parent.postMessage(
  {
    type: "asset_changed",
    guestid,
    xml: "<entry xmlns=...>...</entry>"
  },
  hostOrigin
)
```

⚠ **Note:** The `xml` must be a valid Atom `<entry>` representing the asset.

Example XML for Image Assets

```
<entry xmlns="http://www.w3.org/2005/Atom" xmlns:media="http://search.yahoo.com/
mrss/">
  <content type="image/jpeg" src="https://example.com/image.jpg"/>
  <title>Headline of the Day</title>
  <id>17737204</id>
  <updated>2026-01-21T13:40:23Z</updated>
  <media:content url="https://example.com/image.jpg" type="image/jpeg" width="1200"
height="675"/>
  <media:thumbnail url="https://example.com/thumb.jpg"/>
</entry>
```

9.3.3 Host > Guest Messages

`set_asset` : Sent by the host to apply an existing asset entry to the guest UI. The guest typically selects this asset (if possible) in its UI.

Example:

```
window.addEventListener("message", ev => {
  if (ev.data.type === "set_asset") {
    const assetXml = ev.data.xml
    // Update UI from asset XML
  }
})
```

The purpose of this asset is to initialize the guest UI, and update the guest when the selection changes externally.

9.3.4 Minimal Lifecycle Example

1. The host loads the guest iframe with query parameters.
2. The guest initializes and sends `asset_guest_loaded` .
3. The host sends `set_asset` .
4. The user edits the asset in the guest UI.
5. The guest sends `asset_changed` .

9.4 Complete Example

A complete HTML/JavaScript example implementing an image asset editor using an external feed from NRK, can be used as a reference implementation:

```
<!DOCTYPE html>
<html>
<head>
<script>
  addEventListener('DOMContentLoaded', () => {
    const ATOM_NS = "http://www.w3.org/2005/Atom"

    function getQueryParameter(name) {
      const params = new URLSearchParams(window.location.search)
      return params.get(name)
    }

    /**
     * Get the host origin specified by the "asset_host_origin" query parameter.
     */
    function getHostOrigin() {
      return getQueryParameter("asset_host_origin")
```

```
    }

    /**
     * Get the guest identifier specified by the "guestid" query parameter.
     */
    function getGuestIdentifier() {
      return getQueryParameter("guestid")
    }

    /** @param {string} text */
    function xmlEscape(text) {
      return text
        .replace(/&/g, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
    }

    /**
     * @typedef {Object} FeedItem
     * @property {string} id
     * @property {string} title
     * @property {string} type
     * @property {string} url
     * @property {string} updated
     */

    /**
     * Create Atom XML for an asset.
     * @param {FeedItem} item
     * @param {number} width
     * @param {number} height
     * @returns {string}
     */
    function getXml(item, width, height) {
      const dims = width && height ? ` width="${width}" height="${height}"` : ""
      return `<entry xmlns="http://www.w3.org/2005/Atom" xmlns:media="http://
search.yahoo.com/mrss/">
        <title>${xmlEscape(item.title)}</title>
        <id>${xmlEscape(item.id)}</id>
        <updated>${xmlEscape(item.updated)}</updated>
        <content type="${xmlEscape(item.type)}" src="${xmlEscape(item.url)}" />
        <media:content url="${xmlEscape(item.url)}" type="${xmlEscape(item.type)}"$
{dims} />
        <media:thumbnail url="${xmlEscape(item.url)}" />
      </entry>`
    }

    /**
     * Parses an Atom entry XML string into a FeedItem.
     * @param {string} assetXml
     * @returns {FeedItem | undefined}
     */
    function getItemFromXml(assetXml) {
```

```
      const entryEl = new DOMParser().parseFromString(assetXml, "application/
xml").documentElement

      const titleEls = entryEl.getElementsByTagNameNS(ATOM_NS, "title")
      const idEls = entryEl.getElementsByTagNameNS(ATOM_NS, "id")
      const updatedEls = entryEl.getElementsByTagNameNS(ATOM_NS, "updated")
      const contentEls = entryEl.getElementsByTagNameNS(ATOM_NS, "content")

      if (
        titleEls.length !== 1 ||
        idEls.length !== 1 ||
        updatedEls.length !== 1 ||
        contentEls.length !== 1
      ) return undefined

      const contentEl = contentEls.item(0)

      const type = contentEl.getAttribute("type")
      const url = contentEl.getAttribute("src")
      const title = titleEls.item(0).textContent
      const id = idEls.item(0).textContent
      const updated = updatedEls.item(0).textContent

      if (!type || !url || !title || !id || !updated) return undefined

      return Object.freeze({ id, title, url, type, updated })
    }

    const titleDiv = document.getElementById("title")
    const hostValuesDiv = document.getElementById("hostValues")

    /** @type {HTMLImageElement} */
    const imgEl = document.getElementById("image")

    /** @type {HTMLSelectElement} */
    const itemSelect = document.getElementById("itemsSel")

    const guestid = getGuestIdentifier()
    const hostOrigin = getHostOrigin()

    /** @type {FeedItem[]} */
    const items = []

    let selItem = undefined
    let pendingChangeAsset = false

    window.addEventListener("message", ev => {
      if (hostOrigin && ev.origin !== hostOrigin) return

      const type = ev.data?.type ?? ""

      switch (type) {
        case "set_asset": {
```

```
      selItem = typeof ev.data.xml === "string"
        ? getItemFromXml(ev.data.xml)
        : undefined

      if (items.length) {
        itemSelect.value = selItem?.id ?? ""
      }

      titleDiv.textContent = selItem?.title ?? ""
      break
    }

    case "provide_host_values": {
      const { type, ...rest } = ev.data
      hostValuesDiv.textContent = JSON.stringify(rest)
      break
    }
  }

  console.log("message from host:", ev.data)
})

imgEl.addEventListener("load", () => {
  if (!pendingChangeAsset || !selItem) return

  const xml = getXml(selItem, imgEl.naturalWidth, imgEl.naturalHeight)

  if (hostOrigin) {
    window.parent.postMessage(
      { type: "asset_changed", guestid, xml },
      hostOrigin
    )
  }

  titleDiv.textContent =
    `${selItem.title} (${imgEl.naturalWidth}x${imgEl.naturalHeight})`

  pendingChangeAsset = false
})

itemSelect.addEventListener("change", () => {
  const newValue = itemSelect.value
  const oldValue = selItem?.id ?? ""

  if (newValue === oldValue) return

  const newItem = items.find(item => item.id === newValue)

  if (newItem) {
    selItem = newItem
    imgEl.src = selItem.url
    pendingChangeAsset = true
  }
```

```
    })

    async function initialize() {
      itemSelect.disabled = true

      let feedText = ""

      try {
        const response = await fetch("https://www.nrk.no/toppsaker.rss")
        if (!response.ok) throw new Error(`HTTP ${response.status}`)
        feedText = await response.text()
      } catch (err) {
        console.error("Failed to load feed", err)
        return
      }

      const feedDoc = new DOMParser().parseFromString(feedText, "application/xml")
      const itemEls = feedDoc.getElementsByTagName("item")

      for (let i = 0; i < itemEls.length; ++i) {
        const itemEl = itemEls.item(i)

        const titleEls = itemEl.getElementsByTagName("title")
        const idEls = itemEl.getElementsByTagName("guid")
        const contentEls = itemEl.getElementsByTagNameNS("http://search.yahoo.com/
mrss/", "content")
        const dateElm = itemEl.querySelector("pubDate")

        if (!titleEls.length || !idEls.length || !contentEls.length || !dateElm)
continue

        const title = titleEls.item(0).textContent
        const id = idEls.item(0).textContent
        const updated = dateElm.textContent

        const contentEl = contentEls.item(0)
        const url = contentEl.getAttribute("url")
        const type = contentEl.getAttribute("type")

        if (!title || !id || !url || type !== "image/jpeg" || !updated) continue

        items.push(Object.freeze({ title, id, url, type, updated }))
      }

      for (const item of items) {
        const opt = document.createElement("option")
        opt.value = item.id
        opt.textContent = item.title
        itemSelect.appendChild(opt)
      }

      itemSelect.value = selItem?.id ?? ""
      itemSelect.disabled = false
```

```
    }

    initialize()

    if (hostOrigin) {
      window.parent.postMessage(
        { type: "asset_guest_loaded", guestid },
        hostOrigin
      )
    }
  })
</script>

<style>
  .label {
    padding-top: 8px;
    font-size: x-small;
    display: block;
  }
  .content {
    min-height: 20px;
  }
</style>
</head>

<body style="color: white; background-color: darkslateblue;">
  <h4>I am a hosted asset editor guest!</h4>

  <div style="font-size: smaller;">
    I allow you to create an image asset from the NRK news feed.
  </div>

  <div style="padding-top: 8px; font-size: smaller">
    NRK news feed items:
  </div>

  <select id="itemsSel"></select>

  <label class="label">Host values:</label>
  <div id="hostValues" class="content"></div>

  <label class="label">Title:</label>
  <div id="title" class="content"></div>

  <label class="label">Image:</label>
  <img id="image" src="" height="64" />
</body>
</html>
```

9.5 Best Practices

- Always validate `event.origin` for incoming messages.
- Always use `asset_host_origin` as `targetOrigin`.
- Treat the XML asset entry as immutable input.
- Only send `asset_changed` when the asset is complete and valid.

10 Shared Memory Support

Viz Engine provides a *Shared Memory* (SHM) mechanism that allows external applications to push data into a running graphics scene in near-realtime. This is commonly used for scenarios like live financial tickers, sports scores, election results, and weather data. Anywhere a graphic must reflect rapidly changing values without re-triggering the entire element.

The **ApplySharedMemory** control plugin is the Viz Engine component that receives SHM data and applies it to scene containers. Each piece of data is addressed by a *key*, a hierarchical path such as `/stocks/random/MSFT`, that the plugin uses to route incoming values to the correct container in the scene tree.

10.1 How It Works End-to-End

The complete data flow for a shared memory ticker looks as follows:

1. **External data source** (for example, Vizrt Datacenter, a custom UDP/TCP application, or any SHM-capable sender) continuously pushes key-value updates into Viz Engine's shared memory.
2. **ApplySharedMemory plugin** on the scene reads those keys and maps them to scene containers.
3. **Template Builder** imports the scene and exposes the control field backed by the plugin. The field uses the special media type `application/vnd.vizrt.smmkey`, which stores the SHM key string.
4. **Viz Pilot Edge** (or Viz Ticker Client) lets the user select which data item to display by browsing a feed and choosing an entry. The *locator* element in the feed entry carries the SHM key.
5. At playout time, the selected key is written into the payload. Viz Engine's ApplySharedMemory plugin uses that key to look up the matching realtime data in shared memory and apply it to the graphic.

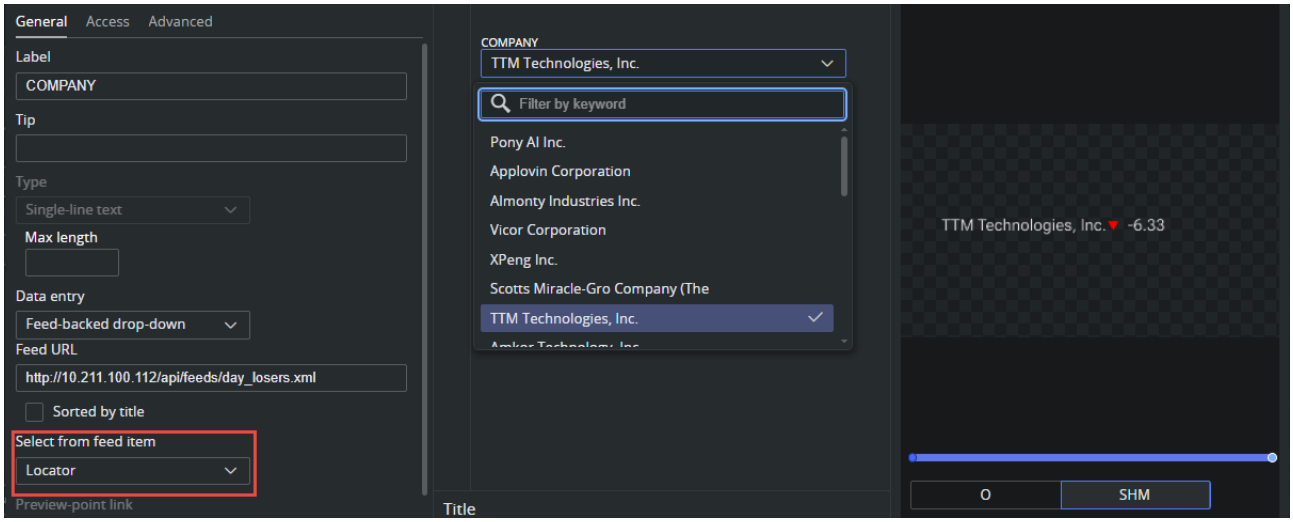
10.2 Importing a Scene with ApplySharedMemory

When importing a Viz Engine scene that uses the ApplySharedMemory control plugin, Template Builder automatically creates a field with the media type `application/vnd.vizrt.smmkey`. This field appears as a single-line text editor in Viz Pilot Edge, but its purpose is to hold the shared memory key path.

10.3 Linking the Field to a Feed

To let users select which data item to display (for example, which stock symbol), link the SHM key field to a feed that contains locator entries:

1. Select the **SHM key** field in the model editor.
2. Set the **Data entry** type to either:
   - **Feed Drop-down:** Presents feed entries as a drop-down list for quick selection.
   - **Feed Browser:** Presents a collection view for browsing and searching feed entries, while also allowing manual key entry.
3. Set the **Feed URL** to point to the Atom feed containing the available data items (for example, a list of stock symbols served by Pilot Data Server or a custom feed endpoint).
4. Set the **Select from feed item** to **Locator**. This tells the system to extract the value of the `<viz:locator>` element from the selected feed entry and store it as the field value.



When **Select from feed item** is set to **Locator**, the system matches the type attribute of the `<viz:locator>` element against the field's media type (`application/vnd.vizrt.smmkey`). If they match, the text content of the locator element, the SHM key path, is extracted and written to the field.

10.4 Feed Entry Format

Feed entries that provide SHM key data, must include a <locator> element from the Vizrt Atom Extension namespace (`http://www.vizrt.com/atom-ext`). The type attribute must be `application/vnd.vizrt.smmkey` to indicate that the value is a shared memory key.

10.4.1 Example Atom Entry

```
<entry xmlns="http://www.w3.org/2005/Atom">
    <id>MSFT</id>
    <published>2012-04-08T16:42:58+07:00</published>
    <updated>2012-09-25T10:34:29Z</updated>
    <title type="text">Microsoft</title>
    <locator type='application/vnd.vizrt.smmkey'
        xmlns='http://www.vizrt.com/atom-ext'>/stocks/random/MSFT
    </locator>
</entry>
```

Element / Attribute	Description
<id>	A unique identifier for the feed entry (for example, the stock ticker symbol).

Element / Attribute	Description
<title>	The human-readable label shown to the user in the feed browser or drop-down (for example, "Microsoft").
<locator>	The Vizrt Atom Extension element carrying the SHM key.
locator/@type	Must be `application/vnd.vizrt.smmkey` to match the field's media type.
locator/@xmlns	Must be `http://www.vizrt.com/atom-ext`.
Locator text content	The shared memory key path (for example, `/stocks/random/MSFT`). This is the value that gets stored in the payload field and used by the ApplySharedMemory plugin at playout time.

11 Troubleshoot

A list of known issues and their fixes are listed below.

11.1 Create New Button Not Displayed on UI

An outdated PDS version (<8.5) is installed. Install version 8.5 or above. Preview Server must also be updated to 4.4.1 or above.

> **Note:** Pilot Data Server version 8.6 is mandatory for Transition Logic support.

11.2 GH Scenes Tree Not Displayed when Pressing Create New

Make sure that `http://<PDS server>:8177/app/DataServerConfig/DataServerConfig.html → graphic_hub_url` is properly set.

11.3 An Error Message is Shown when attempting to Open a Scene

An outdated GH REST version (<3.4.2) is installed. Install version 3.4.2 or later.

11.4 Preview Server Error Message Shown when trying to Open a Scene

Check that the *http://<PDS server>:8177/app/DataServerConfig/DataServerConfig.html* → **preview_server_uri** property is set.

11.5 Scene Blocked due to Outdated or Empty Geom

If the Geom of a scene is outdated or empty when creating a transition logic template, Template Builder blocks the use of the scene.
To fix this, save or update the scene in Viz Artist > 4.2.

Important: The feature below must be enabled in the Viz Artist config file.

Enable automatic creation of merged geometries when saving a transition logic scene: AutoExportTransitionLogicGeometries = 1.

See the Viz Artist User Guide for more information on editing the Viz Artist config file.

11.6 Support

Support is available at the Vizrt Support Portal.

12 Additional Information

The appendix contains the following pages:

- Keyboard Shortcuts
- Overview of Media Types
- Transition Logic and Combo Templates
- Previewing Content
- Overview of Control Plugins

12.1 Keyboard Shortcuts

This page lists available keyboard shortcuts in Template Builder.

Shortcut	Description
CTRL + O	Open the **Open Template** dialog where you can select a template to open.
CTRL + S	Save a template.
CTRL + Z	Undo.
CTRL + Y	Redo.

Warning: The shortcut **CTRL + O** does not work properly in Firefox version 65.0.1 and later.

12.1.1 Graphics Preview Player Shortcuts

Use the following shortcuts for the **Graphics Preview** player:

Shortcut	Description
SPACE or **CTRL + SPACE**	Play/pause.
SHIFT + I	Go to the in-point.
SHIFT + O	Go to the out-point.
, (comma)	Move one frame back.
. (period)	Move one frame forward.

12.2 Overview of Media Types

The following media types are available for single value fields in Template Builder (click the links for W3C definitions):

Type		Media Type (XSD type)	Content of field/value element
Multi-line text		text/plain (string)	text
Single-line text		text/plain (normalizedString)	text
Formatted text		application/ vnd.vizrt.richtext+xml	XML (accepts plain text if unformatted)
Boolean		text/plain (boolean)	text (`true` or `false`)
Integer		text/plain (integer)	text (for example, `-42`)
Decimal		text/plain (decimal)	text using period as decimal point (for example, `123.456`)
Date and time		text/plain (dateTime)	text (for example, `2021-04-06T13:35:00Z`)
Date		text/plain (date)	text (for example, `2021-04-14`)
Two numbers (duplet)		application/ vnd.vizrt.duplet	text containing two decimal numbers separated by a space (for example, `0.6 0.8`)
Three numbers (triplet)		application/ vnd.vizrt.triplet	text containing three decimal numbers separated by spaces (for example, `3 4.5 5`)

Type		Media Type (XSD type)	Content of field/value element
Image		application/atom+xml; type=entry;media=image	The image path on GH (for example, `IMAGE*images/flags/denmark)`
Geometry		application/vnd.vizrt.viz.geom	The geometry path on GH (for example, `GEOM*objects/my-geom`)
Material		application/vnd.vizrt.viz.material	The material path on GH (for example, `MATERIAL*objects/my-material`)
Map		application/vnd.vizrt.curious.map	Proprietary format
Color		text/vnd.vizrt.color	text (for example, #140E7E or rgba(255, 0, 0, 1)).

12.3 Transition Logic and Combo Templates

This section covers transition logic and combo templates, and contains the following topics:

12.3.1 What is Transition Logic (TL)?

Transition Logic (TL) is a way of designing a graphics package that lets you maintain the look and feel of the graphics while letting journalists add graphics items to a rundown, without the need for technical knowledge. TL lets you independently control any number of graphics layers, providing a code-free and design-based method to build graphics that gracefully animate in and out, and transitions from one to another automatically.

Info: Transition Logic (TL) can be played out by most Vizrt control applications such as Viz Trio, Viz Pilot, Viz Multiplay and Viz Multichannel.

12.3.2 How does TL Work?

Master Scenes

This is accomplished by using a Master Scene (aka Background Scene) that coordinates the animation of independently controlled objects which make up the whole. The master scene commonly contains the background items of the graphics package. Such items can be looping backgrounds or the design items of the lower third, over the shoulders, and full-screen graphics. The *variable* or changing content, such as the text in a lower third, is stored separately in Object Scenes.

Object Scenes

When a lower third is played On Air, the object scene for the lower third is triggered. This tells the engine to load the master scene, place the object scene inside the master, and animate the timelines. TL handles all of this automatically.

Combo Templates

These are templates that contain multiple layers of TL scenes.

TL Terminology

- **Combo Templates:** A TL template that contains more than one layer of scenes.
- **Master Scenes:** A TL scene is not a single scene, but a set of Viz graphics scenes that consist of a master scene that may have multiple layers of graphics that can be On Air at the same time and independently controlled.
- **Object Scenes:** Each layer in the master scene may have multiple referring object scenes. However, only one object scene per layer can be active at any given time.
- **Layers:** Layers in the transition logic scene define how many scenes can be on air at the same time. TL layers are conceptual, not spatial.

Note: With Transition Logic scene design, *take in* and *take out* commands are still used as with standalone scene design. Where standalone scene design demands that only a single scene can be On Air at a time, however, Transition Logic allows for more than one scene to be On Air simultaneously. This means that using Transition Logic lets you have a graphic covering the lower third of the screen and another graphic covering the left and/or right side of the screen for over the shoulder graphics On Air at the same time.

12.3.3 Working with Transition Logic and Combo Templates

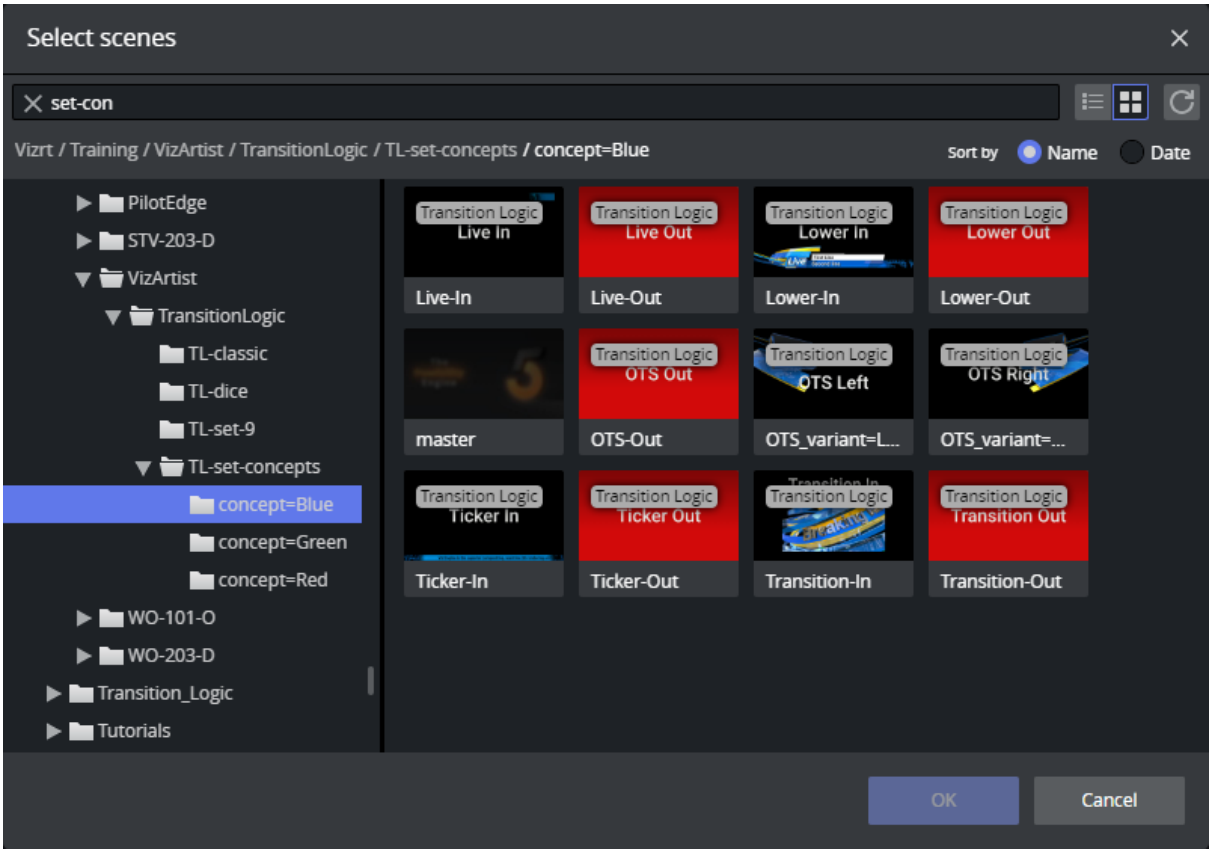
Follow the steps below to get started.

Note: Transition logic and combo templates require Viz Engine 4.3 or above.

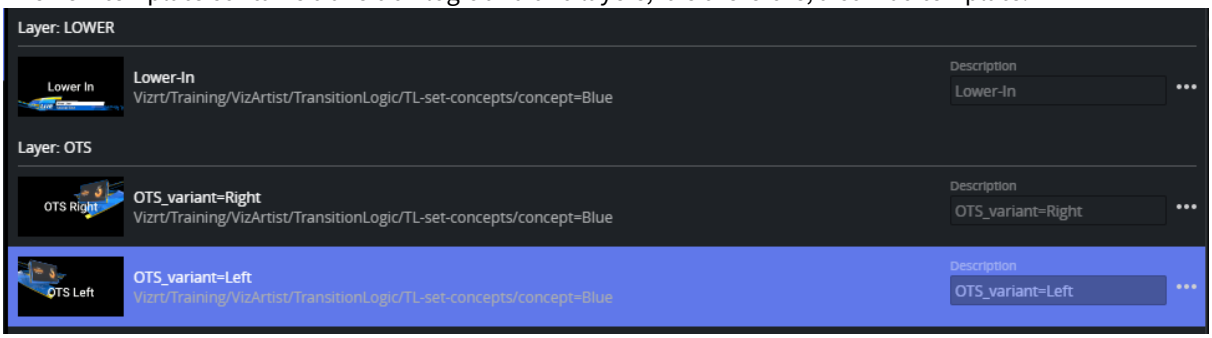
Creating a New Combo Template

Create a new template and add transition logic scenes. The following example use **Blue** and **Green** concepts.

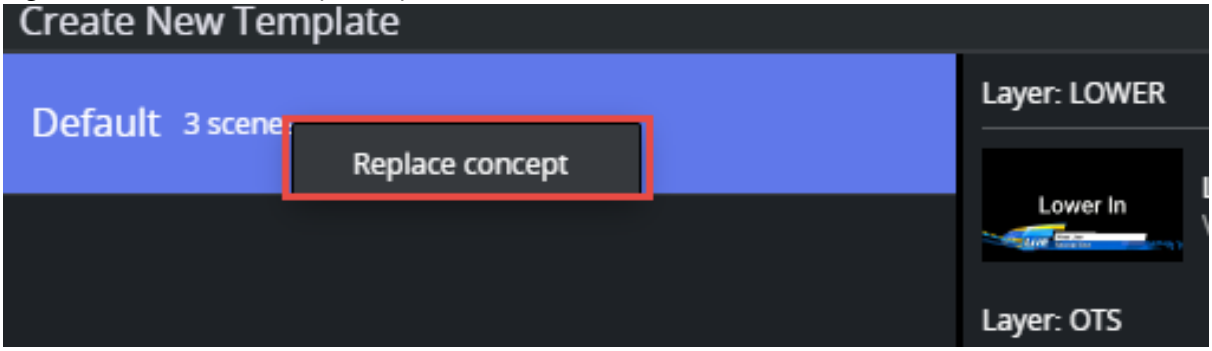
- Select scenes and click **OK**.



- The new template contains transition logic and two layers, it is therefore, a combo template:

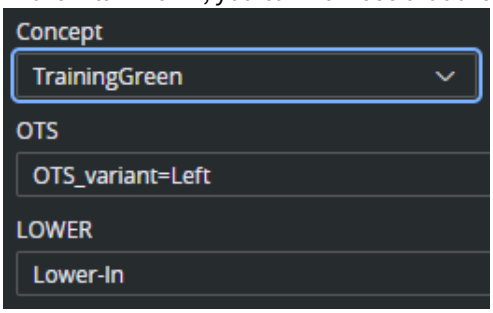


- Right click the Default concept to replace it with a new one:



- Click **+Add Concept** in the lower left corner of the screen. Enter a name for your new concept and click **OK**.

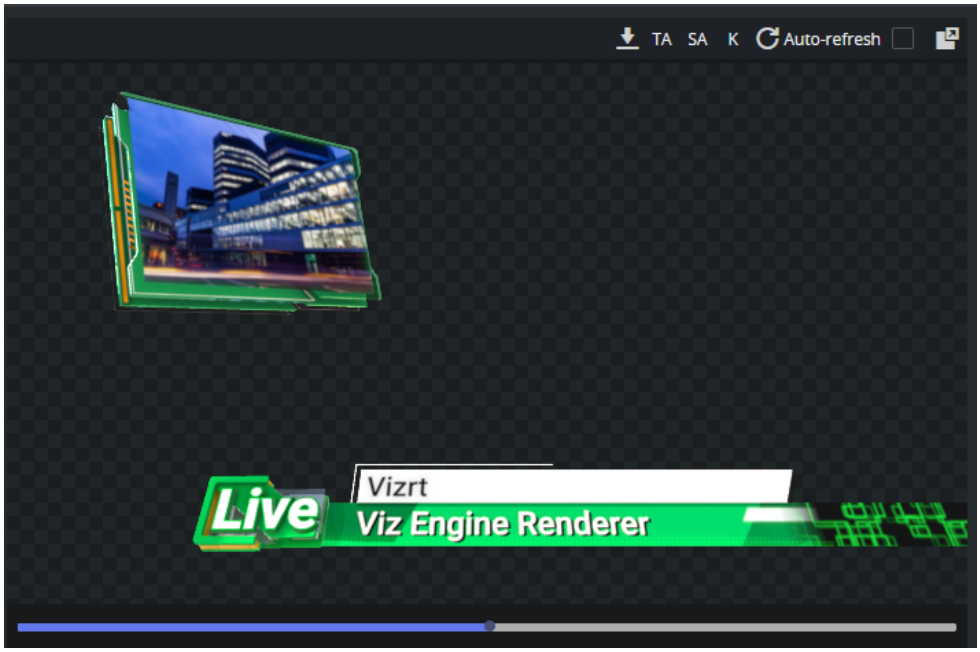
- Click the **+Add Scene** button.
- Select the *scenes with the same set of control objects* as those you selected for the first concept.
- In the **Fill-in form**, you can now see that the template contains two concepts and two layers.



12.4 Previewing Content

The **Graphics Preview** window is located to the right of the interface. It displays snapshots of the final output in an ongoing preview process, and provides an indication of how the graphics look when played out in high resolution on a Viz Engine.

Note: Template Builder sends requests to Preview Server which manages the Viz Engines that provide the snapshots.



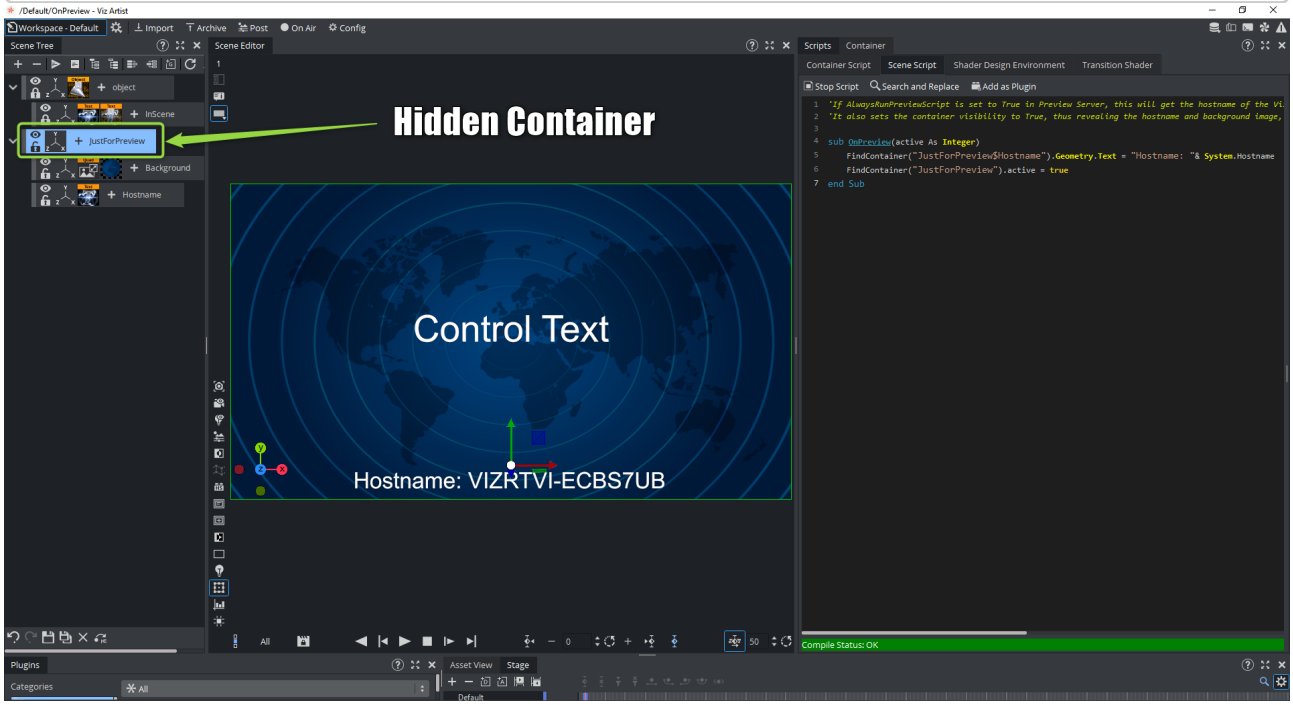
- **Preview points:** If the scene contains named preview points, such as stop points and/or tags in the Default director, these are displayed as a timeline on top of the preview. Small circles represent the preview points.
- **Download button:** Downloads the current preview snapshot as a PNG file in HD resolution.
- **TA:** Show/hide the Title Area.
- **SA:** Show/hide the Safe Area.
- **K:** Show the key signal for the graphics.
- **Refresh**: Visible when auto refresh is disabled. Sends a preview request to Preview Server with the current data. The request is unique, meaning that the preview server does not return a cached version of the snapshot. Preview Server also checks whether the scene itself is updated in Viz Artist and returns the latest saved version.
- **Auto-refresh:** Enabled by default. Send a preview request for any user generated change that may lead to the snapshot being changed.

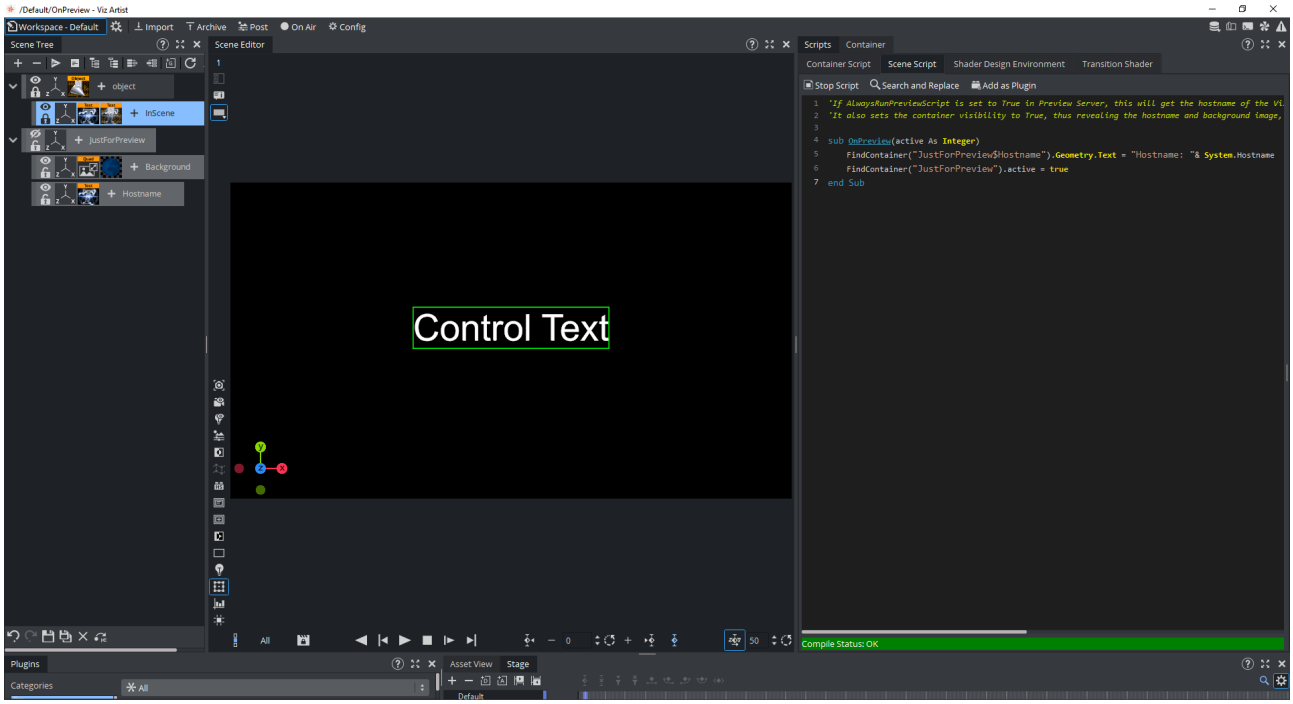
Info: Clicking on a preview point to request a preview sends a snapshot request with a named position to Preview Server. Clicking on the timeline sends a snapshot request with an absolute position to Preview Server. For more information, see the Preview Server REST API documentation.

12.4.1 Viz Scene - OnPreview()

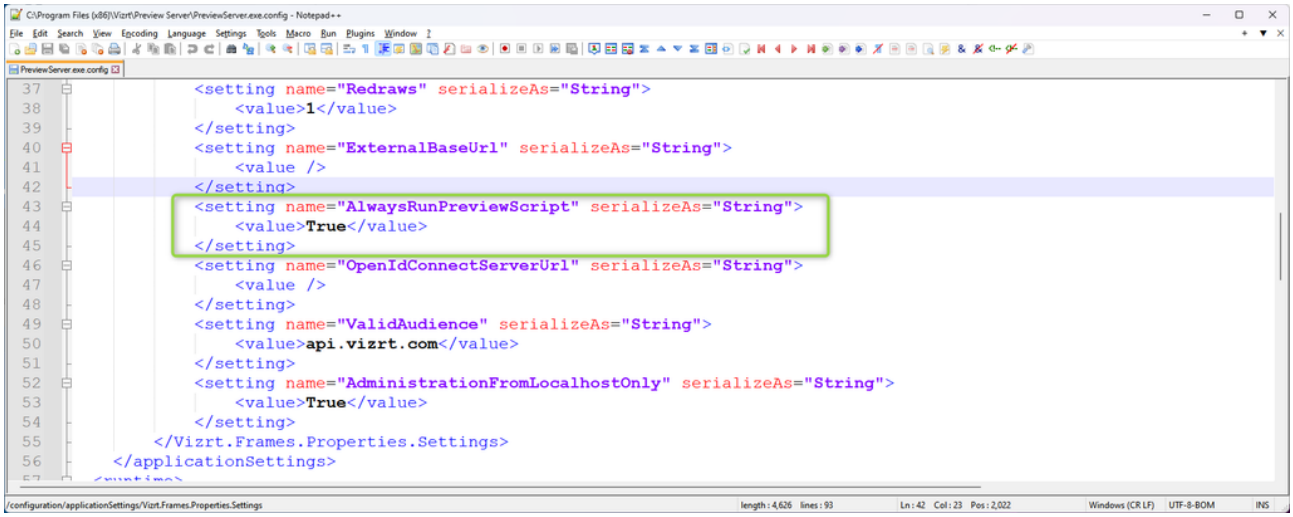
In the Viz Scene Script, you can use the *OnPreview()* function to customize the preview shown in Template Builder and Viz Pilot Edge.
Here is a sample code for Viz Scene Script:

```
//If AlwaysRunPreviewScript is set to True in Preview Server, this will get the
hostname of the Viz Engine rendering the preview and sets the returned value in the
text container from JustForPreview->Hostname.
//It also sets the container visibility to True, thus revealing the hostname and
background image, but only in Preview.

sub OnPreview(active As Integer)
    FindContainer("JustForPreview$Hostname").Geometry.Text = "Hostname: "&
System.Hostname
    FindContainer("JustForPreview").active = true
end Sub
```





To enable this functionality in Preview Server, navigate to the location of the *PreviewServer.exe.config* and set the value **True** for **AlwaysRunPreviewScript**:



You then have to restart the Preview Server Windows Service for this change to take effect. The preview now displays the hostname of the Viz Engine:

12.5 Overview of Control Plugins

12.5.1 Supported Viz Artist Control Plugins

Plugin Name	Comments
Control Chart	The Control Chart plug-in exposes control of chart data from the Visual Data Tools plugins.
Control Text	From Viz Engine 5.0.1 and above, Control Text exposes text from both the Classic and Viz Engine renderer pipeline.
Control Geom	The Control Geom plug-in exposes the control of geometry objects to the user.
Control Image	The Control Image plug-in creates an image control in the control clients.
Control List	The Control List plug-in allows you to create table controls. Normally there should be a Control Object for each row.
Control Material	The Control Material plug-in exposes the material control to the user. Remember to set up a search provider towards Graphic Hub, to use this.
Control Number	The Control Number plug-in (also known as Control Num) is used to be able to decide how a number input is to be formatted. It can be a value given by the control client user or by any external source. It should be used instead of Control Text when numbers are the input value.
Control Object	Control Object should always be added to a scene when you add other control plugins. It is in this plugin that you mark if a scene is part of a transition logic set or not. There should normally only be one control object in the scene. The exception is when you use a control list when there is a control object for every line.
Control OMO	The Control Object Moving (Omo) plug-in gives you the possibility to add a group of containers and reveal one at the time. This control plugin exposes an integer. It is typically nice to use a drop-down for these with proper field names.
Control Video	The Control Video plug-in exposes control over a video codec channel (ClipChannel). Does not support clips in soft clip players.
Control WoC	A replacement of the Control Maps plugin with more options and on-the-fly feedback from Viz Artist/Engine.

Info: If a Control Plugin is not listed in the table above, it is not supported by Graphic Hub and/or Template Builder.