

CUE  
**Tech Guide**  
3.16.12-4

# Table of Contents

- [1 Installing CUE](#)..... 5
- [2 Configuring CUE](#)..... 7
  - [2.1 Mandatory Tasks](#)..... 7
    - [2.1.1 Basic CUE Configuration](#).....7
    - [2.1.2 Nginx Configuration](#)..... 8
    - [2.1.3 Web Service CORS Configuration](#)..... 9
  - [2.2 Optional Tasks](#).....10
    - [2.2.1 Third-Party Authentication](#)..... 10
    - [2.2.2 Create new Dialog](#).....12
    - [2.2.3 Heading Levels](#).....13
    - [2.2.4 Automatic div Removal](#).....13
    - [2.2.5 Content Type Selection for Binaries](#)..... 14
    - [2.2.6 Defining Custom Icons](#)..... 15
    - [2.2.7 Metadata Panel Sections](#)..... 15
    - [2.2.8 CUE Composer Integration for Storylines](#)..... 18
    - [2.2.9 Smart Quotes](#)..... 21
    - [2.2.10 Spelling Checker](#)..... 22
    - [2.2.11 Semantic Shortcut Key Combination](#).....23
    - [2.2.12 Content Creation Shortcuts](#)..... 24
    - [2.2.13 The Publish Shortcut](#).....25
    - [2.2.14 Content Card Date Type](#)..... 26
    - [2.2.15 Storyline Symbol Insertion](#).....26
    - [2.2.16 Section Page Metadata Panel Width](#)..... 28
    - [2.2.17 Search Filters](#)..... 28
    - [2.2.18 Dashboards \(Content Store only\)](#).....30
    - [2.2.19 Asset Picker Custom Search Filters \(Content Store only\)](#).....31
    - [2.2.20 Autosave Interval](#).....32
    - [2.2.21 Quick View](#).....32
    - [2.2.22 HTML Source Editing](#)..... 34
    - [2.2.23 Cleaning up Pasted Content](#)..... 34
    - [2.2.24 CUE Print Access for Freelancers](#)..... 34
    - [2.2.25 Teaser Anchors in Section Page Previews](#).....35
    - [2.2.26 Metadata Panel Section List Length](#)..... 36
    - [2.2.27 Default Tag Relevance](#).....36

2.2.28 Sections Side Panel Preview.....	37
2.2.29 Storyline Metrics (Content Store only).....	37
2.2.30 Preview Control Dialog.....	41
2.2.31 Inline Link Target Window Default.....	42
2.2.32 Date Picker Default Time.....	42
2.2.33 CUE Print Handling in Create New Dialog.....	43
2.2.34 Access Token Refreshment Timing.....	43
2.2.35 Environment Visualization.....	43
2.2.36 Disabling Search-As-You-Type.....	44
2.2.37 Enabling User Tracking.....	44
2.2.38 Automated Curation With Sophi.....	45
3 Installing and Configuring Plug-ins.....	49
3.1 cue-content-duplication-enrichment-service.....	50
3.1.1 Installing cue-content-duplication-enrichment-service.....	50
3.1.2 Configuring cue-content-duplication-enrichment-service.....	50
3.2 cue-spellcheck.....	52
3.2.1 Installing cue-spellcheck.....	52
3.2.2 Configuring cue-spellcheck.....	52
3.2.3 Starting cue-spellcheck.....	53
3.2.4 Specifying Publication Language.....	54
4 Extending CUE.....	55
4.1 Web Components.....	55
4.1.1 Creating a Web Component.....	56
4.1.2 The CUE Web Component API.....	58
4.2 Enrichment Services.....	129
4.2.1 Configuring Enrichment Services in CUE.....	130
4.2.2 Creating an Enrichment Service.....	137
4.2.3 Multi-select Enrichment Services.....	140
4.2.4 Some Examples.....	140
4.2.5 Learning More About Enrichment Services.....	144
4.3 Drop Resolvers.....	144
4.3.1 Configuring Drop Resolvers in CUE.....	144
4.3.2 Drop Resolver Parameters.....	146
4.3.3 Drop Resolver Return Values.....	146
4.4 URL-based Content Creation.....	147
4.4.1 Content Creation URL Structure.....	147
4.4.2 Example Script.....	150

<a href="#">4.5 URL-based Content Editing</a>	151
<a href="#">4.5.1 Content Editing URL Structure</a>	151
<a href="#">4.6 Logout Triggers</a>	152
<a href="#">4.7 CUE Safe Mode</a>	152
<a href="#">4.8 Custom Capabilities (Content Store only)</a>	153
<a href="#">5 DC-X Integration</a>	155
<a href="#">5.1 DC-X Drop Resolver Installation</a>	155
<a href="#">5.2 DC-X Extension Configuration</a>	155
<a href="#">5.2.1 Endpoint Configuration</a>	156
<a href="#">5.2.2 Side Panel Configuration</a>	156
<a href="#">5.2.3 Drop Resolver Configuration</a>	157
<a href="#">5.2.4 Content Type Configuration</a>	158
<a href="#">5.2.5 Zipline Configuration</a>	158
<a href="#">5.3 Login Credentials</a>	165
<a href="#">5.4 Using The Main DC-X Integration</a>	165
<a href="#">5.5 Using The DC-X Wire Integration</a>	167

# 1 Installing CUE

CUE requires the use of an **SSE Proxy** to manage the delivery of Server-sent Events from the CUE Content Store to CUE clients. This means that an CUE SSE Proxy must have been installed and configured to manage SSE for the Content Store, and the Content Store must have been configured to direct SSE connection requests to the SSE Proxy. For general information on how to install and configure an SSE Proxy, see the [SSE Proxy documentation](#). For specific guidance on how to configure the Content Store and the SSE Proxy to work together with CUE, see [Configure an SSE Proxy Connection for CUE](#).

CUE is available as a standard Debian installation package, making installation on Ubuntu or other Debian-based Linux systems very straightforward. CUE is a standalone web application. Although it needs to be connected to an CUE Content Store and/or a CCI Newsgate back end, it does not need to be co-located with either of them. It can be installed on the same server as a Content Store instance, but it does not need to be. An application server such as Tomcat is not required to serve CUE. Since it is a pure HTML/Javascript application, a web server such as nginx or Apache is sufficient.

## A note about version code names

CUE and all related applications (CUE Content Store, CUE Print, Content Store plug-ins and so on) are released on a synchronized schedule where all product versions in a given release are known to work well together. Only these approved version combinations are supported. Each set of compatible product versions is identified by a code name, and during installation you can use this code name instead of the individual product's version number, thereby simplifying the installation process.

In the case of CUE and other Linux applications installed on Ubuntu using **apt**, the code name is actually the name of a repository containing compatible versions of all products. This means that in order to ensure version compatibility, all you need to do is add the name of the required repository to your `/etc/apt/sources.list.d/escenic.list` file. Once you have done this you do not need to specify any version numbers when installing individual packages - **apt** will just install the latest maintenance release from that repository.

Note that code names cannot be used in this way on Red Hat installations, where the application packages to be installed must still be identified by their version numbers. This is also the case for CUE Print.

The code name for CUE 3.16.12-4 is **platinum**. To find the correct CUE Print version to install for the **platinum** release, check the [CUE Print release notes](#).

## Installation procedure

The instructions given here are based on the use of an nginx web server, running on Ubuntu.

To install CUE:

1. Log in via SSH from a terminal window.
2. Switch user to root:

```
$ sudo su
```

3. If necessary, download and set the Escenic **apt** repository key:

```
| # curl --silent https://user:password@apt.escenic.com/repo.key | apt-key add -
```

where *user* and *password* are your Stibo DX download credentials (the same ones you use to access the Stibo DX Maven repository). If you do not have any download credentials, please contact [Stibo DX support](#).

4. Add the current version repository name to your list of sources.

```
| # echo "deb https://user:password@apt.escenic.com platinum main non-free" >> /etc/  
| apt/sources.list.d/escenic.list
```

5. You need to install version 1.7.5 or higher of nginx. The version available in the Ubuntu 14.04 repositories is too old, so in order to ensure that you install a new enough version, you need to add a repository containing a more recent version:

```
| # add-apt-repository ppa:nginx/stable
```

6. Update your package lists:

```
| # apt-get update
```

7. Download and install CUE:

```
| # apt-get install cue-web
```

8. Download and install nginx

```
| # apt-get install nginx
```

## 2 Configuring CUE

In order to complete the installation of CUE, you must:

- Carry out a basic configuration of CUE itself and the nginx web server that serves the CUE application.
- Configure nginx to support cross-origin communication between CUE and the CUE Content Store's web service

These mandatory configuration tasks are both described in [section 2.1](#).

There are in addition a number of more or less optional configuration tasks that you may need to carry out, depending on your specific requirements. These tasks are described in [section 2.2](#).

### 2.1 Mandatory Tasks

The configuration tasks described in this section are **required** in order to get CUE up and running.

#### 2.1.1 Basic CUE Configuration

CUE configuration involves configuring CUE itself, and also configuring the nginx web server that serves the CUE application.

The actual CUE configuration consists of editing YAML format configuration files, identified by the file type extension `.yaml`. The delivered system includes a number of such configuration files containing CUE's default configuration settings. These files are located in the `/etc/escenic/cue-web` folder.

The `/etc/escenic/cue-web` folder also contains a file called `config.yaml.template`, containing the property settings that you always need to set when installing CUE. To use this file you rename it to `config.yaml` and then edit the contents.

To configure CUE:

1. If necessary, switch user to `root`.  

```
| $ sudo su
```
2. Copy `/etc/escenic/cue-web/config.yaml.template` to `config.yaml`:  

```
| # cp /etc/escenic/cue-web/config.yaml.template /etc/escenic/cue-web/config.yaml
```
3. Open the new `/etc/escenic/cue-web/config.yaml` for editing. For example  

```
| # nano /etc/escenic/cue-web/config.yaml
```
4. Uncomment and set the required endpoint parameters (which you will find at the top of the file):  

```
| endpoints:  
|   escenic: "http://escenic-host:81/webservice/index.xml"
```

```
newsgate: "http://newsgate-host/newsgate-cf/"
```

where:

- *escenic-host* is the IP address or host name of the Content Store CUE is to provide access to
- *newsgate-host* is the IP address or host name of the CCI Newsgate system CUE is to provide access to. If no CCI Newsgate system is present, then do not uncomment the **newsgate:** line.

5. If your CUE configuration makes use of an Escenic-CCI Newsgate bridge, then you will need to add a third line under **endpoints:**

```
endpoints:
  escenic: "http://escenic-host:81/webservice/index.xml"
  newsgate: "http://newsgate-host/newsgate-cf/"
  bridge: "http://bridge-host:7001/ngece-bridge/"
```

where *bridge-host* is the IP address or host name of an Escenic-CCI Newsgate bridge. (A bridge is a service capable of converting Escenic content to Newsgate format, and is required to support Newsgate write-to-fit functionality in CUE.)

6. Save the file.

7. Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the Content Store web service URL you specified in step 3.

You now need to configure the nginx web server to serve the CUE application, as described in [section 2.1.2](#).

## 2.1.2 Nginx Configuration

To configure nginx:

1. If necessary, switch user to **root**.

```
$ sudo su
```

2. Open **/etc/nginx/sites-available/default** for editing, and replace the entire contents of the file with the following:

```
server {
    listen 81 default;
    include /etc/nginx/default-site/*.conf;
}
```

3. Create a new folder to contain your site definitions:

```
# mkdir /etc/nginx/default-site/
```

4. Add three files to the new **/etc/nginx/default-site/** folder, called **cue-web.conf** and **webservice.conf**:

```
# touch /etc/nginx/default-site/cue-web.conf
# touch /etc/nginx/default-site/webservice.conf
# touch /etc/nginx/conf.d/request-entity-size-limit.conf
```

5. Open **/etc/nginx/default-site/cue-web.conf** for editing and add the following contents:

```
location /cue-web/ {
    alias /var/www/html/cue-web/;
    expires modified +310s;
```



```
| }
```

Depending on the version of nginx that you have installed, the alias specified in `cue-web.conf` may need to be set to `/var/www/cue-web/` instead of `/var/www/html/cue-web/`.

6. Open `/etc/nginx/default-site/webservice.conf` for editing and add the contents described in [section 2.1.3](#).
7. Open `/etc/nginx/conf.d/request-entity-size-limit.conf` for editing and add the following contents:

```
| # Disable default 1Mb limit of PUT and POST requests.
| client_max_body_size 0;
```

(If you do not add this setting, then nginx will not allow larger files such as images and videos to be uploaded to CUE.)

You will now need to set up cross-origin communication between CUE and the Content Store web service as described in [section 2.1.3](#).

### 2.1.3 Web Service CORS Configuration

Your `cue-web` application is now running on the nginx default port, 81. In order to be able to run correctly it needs to be able to send requests to the CUE Content Store's web service. This web service may possibly be running on a different host in a different domain. Even if it is running on the same host as nginx, it will most likely be listening on port 8080 (Tomcat's default port). This means that by default any requests from the `cue-web` application to the Content Store web service will be rejected as cross-origin scripting exploits.

You can, however, enable cross-origin communication between the `cue-web` application and the Content Store web service by setting up an nginx proxy for the web service that redirects requests to the actual web service and also adds the [CORS](#) headers needed to ensure that the requests will not be rejected.

Here is an example of a suitable `/etc/nginx/default-site/webservice.conf`:

```
| location ~ "/(escenic|studio|webservice|webservice-extensions)/(.*)" {
|     if ($http_origin ~* (https?://[^\/*]\.dev\.my-cue-domain\.com(?:[0-9]+)?)$) {
|         set $cors "true";
|     }
|     if ($request_method = 'OPTIONS') {
|         set $cors "${cors}options";
|     }
|     if ($request_method = 'GET') {
|         set $cors "${cors}get";
|     }
|     if ($request_method = 'HEAD') {
|         set $cors "${cors}get";
|     }
|     if ($request_method = 'POST') {
|         set $cors "${cors}post";
|     }
|     if ($request_method = 'PUT') {
|         set $cors "${cors}post";
|     }
|     if ($request_method = 'DELETE') {
|         set $cors "${cors}post";
```

```

    }
    if ($cors = "trueget") {
        add_header "Access-Control-Allow-Origin" "$http_origin" always;
        add_header "Access-Control-Allow-Credentials" "true" always;
        add_header "Access-Control-Expose-Headers" "Link,X-ECE-Active-
Connections,Location,ETag,Allow" always;
    }
    if ($cors = "truepost") {
        add_header "Access-Control-Allow-Origin" "$http_origin" always;
        add_header "Access-Control-Allow-Credentials" "true" always;
        add_header "Access-Control-Expose-Headers" "Link,X-ECE-Active-
Connections,Location,ETag" always;
    }
    if ($cors = "trueoptions") {
        add_header 'Access-Control-Allow-Origin' "$http_origin";
        add_header 'Access-Control-Allow-Credentials' 'true';
        add_header 'Access-Control-Max-Age' 1728000;
        add_header 'Access-Control-Allow-Methods' 'GET, POST, HEAD, OPTIONS, PUT,
DELETE';
        add_header 'Access-Control-Allow-Headers' 'Authorization,Content-
Type,Accept,Origin,User-Agent,DNT,Cache-Control,X-Mx-ReqToken,Keep-Alive,X-Requested-
With,If-Modified-Since,If-Match,If-None-Match,X-Escenic-Locks,X-Escenic-media-
filename,X-Escenic-home-section-uri,X-Escenic-Container-Destinations';
        add_header 'Content-Length' 0;
        add_header 'Content-Type' 'text/plain charset=UTF-8';
        return 204;
    }
    proxy_set_header Host $http_host;
    proxy_pass http://127.0.0.1:8080;
}

```

In the origin filter at the top of the file:

```

if ($http_origin ~* (https?://[^\/*]\.dev\.my-cue-domain\.com(?:[0-9]+)?)$) {
    set $cors "true";
}

```

you must replace *my-cue-domain\.com* with the actual domain name of your CUE installation.

## 2.2 Optional Tasks

The configuration tasks described in this section are optional. Whether or not they are necessary depends on your system requirements.

Most of the configuration tasks involve editing YAML configuration files in the CUE configuration folder (*/etc/escenic/cue-web*). In some cases, however, it is also necessary to make server-side configuration changes. This usually involves editing XML files called Content Store **resources**.

### 2.2.1 Third-Party Authentication

Both CUE Content Store and CCI Newsgate can be configured to allow third-party authentication of users. This lets you log in to CUE using your Google or Facebook ID, for example, rather than by entering a CUE-specific user name and password.

In order to be able to make use of third-party authentication in CUE:

- The Content Store/CCI Newsgate back-end system(s) must have been configured to allow third-party authentication. For details of how to enable third-party authentication in CUE, see [Third-Party Authentication](#).
- CUE itself must be configured to display the UI for the third-party authentication methods that have been enabled.

CUE supports two third-party authenticators – Google and Facebook.

### 2.2.1.1 Google Authentication

If the relevant back-end system(s) have been set up to support Google Authentication, then you can configure CUE support by adding a YAML configuration file to the CUE configuration folder (`/etc/escenic/cue-web`).

When you are configuring third-party authentication for the Content Store as described in [Configure OAuth Authentication](#), you need to add a CUE redirect URI to the **Authorized redirect URI** in step 16. The URI must be your CUE URI followed by `/oauth_callback.html`: for example `http://your-cue-host/cue-web/oauth_callback.html`.

Your configuration file must contain the following settings:

```
oauth:
  name: "Google"
  label: "Log in with Google account"
  authURI: "https://accounts.google.com/o/oauth2/auth"
  scope: "email"
  clientId: "google-client-id"
```

where `google-client-id` is the client ID you created in the steps described above.

When setting up Google authentication for the Content Store, you create two client IDs – one for desktop clients and one for web clients. Make sure that you use the web client ID for configuring CUE.

When you have saved this file, enter (as the `root` user):

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made. The CUE login page will now include a **Log in with Google account** option.

### 2.2.1.2 Facebook Authentication

If the relevant back-end system(s) have been set up to support Facebook Authentication, then you can configure CUE support by adding a YAML configuration file to the CUE configuration folder (`/etc/escenic/cue-web`). The file must contain the following settings:

```
oauth:
  name: "Facebook"
  label: "Log in with Facebook account"
  authURI: "https://graph.facebook.com/oauth/authorize"
  scope: "email"
  clientId: "facebook-client-id"
```

where *facebook-client-id* is the the **web client ID** you created when configuring access to the back-end system(s) (see [Configure OAuth Authentication](#)).

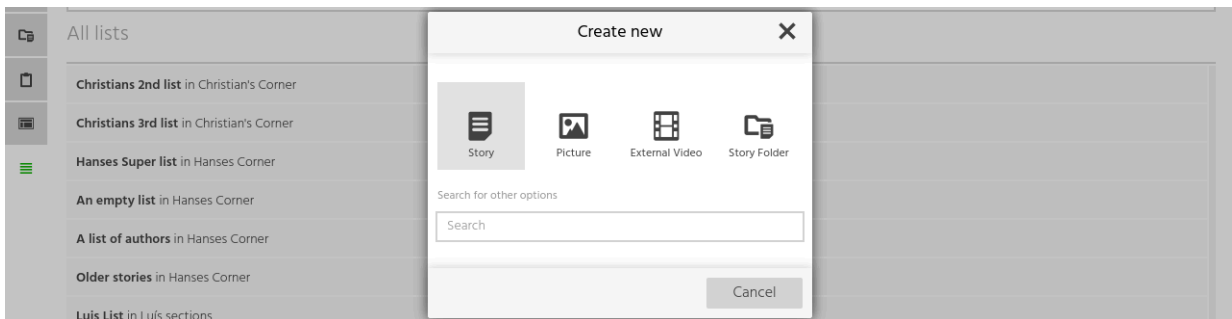
When setting up Facebook authentication for the Content Store, you create two client IDs – one for desktop clients and one for web clients. Make sure that you use the web client ID for configuring CUE.

When you have saved this file, enter (as the **root** user):

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made. The CUE login page will now include a **Log in with Facebook account** option.

## 2.2.2 Create new Dialog



The **Create new** dialog (shown above) is configurable: you can specify which content types are to be displayed as favorites in the top half of the dialog. There is space for a maximum of four favorites: all other options must be selected using the search field in the bottom half of the dialog.

To specify your required favourites:

1. If necessary, switch user to **root**.

```
$ sudo su
```

2. Open `/etc/escenic/cue-web/30-new-content-defaults.yml` for editing. For example

```
# nano /etc/escenic/cue-web/30-new-content-defaults.yml
```

3. Find the `newContentDefaults` parameter:

```
newContentDefaults:
- type: "story"
  icon: "story"
- type: "picture"
  icon: "picture"
- type: "video"
  icon: "video"
- type: "storyfolder"
  icon: "storyfolder"
- type: "gallery"
  icon: "picture"
```

4. Modify the list of content type/icon pairs to meet your requirements. If you use CUE to edit several publications that have different content types, then you may want to have more than four content types in the list even though a maximum of four can be displayed in the dialog. If

a publication has no **video** content type, for example, then the **Create new** dialog will display **story**, **picture**, **storyfolder** and **gallery** from the above list.

Note that if you create custom content type icons as described in [section 2.2.6](#), then any icon settings made in the **content-type** resource will override icon settings made here.

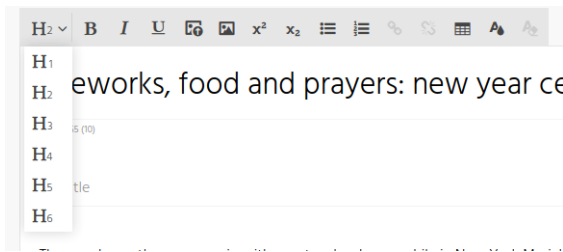
5. Save the file.
6. Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

### 2.2.3 Heading Levels

The rich text editor's formatting tool bar has a Heading button that you can use to insert HTML heading elements **h1**, **h2**, **h3** and so on. By default the button offers **h2** as the default selection, with headings **h1** and **h3** - **h6** as options in a drop-down menu:



You can, however, change this default configuration as follows:

1. If necessary, switch user to **root**.

```
$ sudo su
```

2. Open **etc/escenic/cue-web-3.16/plugins/internal/EscenicHeading/EscenicHeading.yml** for editing. For example:

```
# nano etc/escenic/cue-web-3.16/plugins/internal/EscenicHeading/EscenicHeading.yml
```

3. To change the default heading level, edit the **defaultHeadingLevel** property:

```
defaultHeadingLevel: 2
```

4. To change the contents of the drop-down menu, edit the **headingLevels** property:

```
headingLevels: "1, 2, 3, 4, 5, 6"
```

5. Save the file.
6. Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

### 2.2.4 Automatic div Removal

CUE can be configured to automatically remove HTML **div** elements from text pasted into rich text fields. This functionality is useful for some customers, but not for others and is therefore disabled by default. To enable it:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open **/etc/escenic/cue-web/config.yml** for editing. For example

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following setting:

```
|   removeDivsAutomatically: true
```

4. Save the file.

5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

You can disable the functionality by setting **removeDivsAutomatically** to **false**.

### 2.2.5 Content Type Selection for Binaries

When a binary file is dropped in CUE, a content item is automatically created to contain it. In order to be able to do this, CUE searches for a content type that is configured to handle the binary file's MIME type. If some MIME types can be handled by more than one content type, then by default CUE uses the first one it finds. You can, however configure CUE to allow the user to choose the content-type.

To configure this kind of content type selection:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open **/etc/escenic/cue-web/config.yml** for editing. For example

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following settings:

```
|   contentTypeSelection:  
|     enabled: true
```

4. Save the file.

5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

If JPEG file types can be handled by three different content types, **picture**, **graphic** and **special**, then users who drop a JPEG file into CUE will now be prompted to select which of the three content types CUE should use.

If you don't want all available content types to be offered as options, you can exclude some by including an **ignoreContentTypes** property in the configuration file:

```
|   contentTypeSelection:  
|     enabled: true  
|     ignoreContentTypes: ["special"]
```

**ignoreContentTypes** accepts an array of content type names, so you can exclude multiple content types from the user prompt if you wish. If you exclude all content types except one, then no prompt is displayed in CUE since the user no longer has a choice.

## 2.2.6 Defining Custom Icons

Icons are widely used to represent different types of objects in CUE:

- Content items
- Publications
- Story elements in storylines
- Workflow states
- Dashboards
- Macros

The icon used to represent a content item varies according to its type: stories are represented by document icons, pictures by image icons, and so on. Similarly, paragraph, image and table story elements are all represented by different icons, as are different publications, workflow states, dashboards and macros. A number of standard icons are supplied with CUE for use with default publications, content types, story element types, states and so on. It is of course possible to re-use some of these icons for your own content types, story element types and so on, but it is not recommended – you should define your own custom icons.

All of the above object types are defined in Content Store resource files of one kind or another as part of the publication definition process described in the Content Store [Publication Design Guide](#).

In general, defining an icon for one of the above object types involves creating the icon itself (an image file, for example) and then adding a `ui:icon` element that references the image to the object type's definition in the appropriate Content Store resource file. Suppose, for example, that you want to add an icon to your "Long Story" content type. In this case you would need to insert a `ui:icon` element as a child of the `<content-type name="longstory">` element in your publication's `content-type` resource. Similarly, for a "Special Para" story element type, you would need to insert a `ui:icon` element as a child of the `<story-element-type name="specialpara">` element in your `specialpara.xml` story element type resource.

The details of what kind of image you should create and how you should use the `ui:icon` element vary between the above object types. In some cases only .PNG images may be used as icons, whereas in other cases more freedom is allowed (.SVG files, inline SVG code, Unicode characters and so on). For detailed instructions, see the Content Store [Publication Design Guide](#) and the reference description of the [ui:icon element](#).

## 2.2.7 Metadata Panel Sections

You can control which sections appear in the metadata panel on the right side of the CUE window (and the order in which they appear) by adding `metadata-panel` elements to your content type definitions in the Content Store `content-store` resource. For general information about the `content-store` resource and how to define CUE content types, see [The content-type Resource](#).

Using the `metadata-panel` element, you can define what sections are to be displayed in the metadata panel for each content type in a publication, and the order in which they are to appear. The `metadata-panel` element belongs to a special CUE-specific namespace: `http://`

**xmlns.cuepublishing.com/configuration**. Before you add any **metadata-panel** elements to your **content-type** resource, therefore, you should declare this namespace in the file's root element, and define a prefix for it (**cue** is recommended). For example:

```
<content-types xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  xmlns:doc="http://xmlns.vizrt.com/2010/documentation"
  xmlns:media="http://xmlns.escenic.com/2013/media"
  xmlns:video="http://xmlns.escenic.com/2010/video"
  xmlns:livecenter="http://xmlns.escenic.com/2015/live-center"
  xmlns:cci="http://cci/extension/integration"
  xmlns:cue="http://xmlns.cuepublishing.com/configuration"
  version="4">
  ...
</content-types>
```

Once you have done this, you can control the metadata panel sections displayed for items of a particular content type by adding a **cue:metadata-panel** element as a child of its defining **content-type** element. For example:

```
<content-type name="story">
  ...
  <cue:metadata-panel>
    my.relation-headshot
    cue.general-info
    my.extra-info
    cue.section
    ...
  </cue:metadata-panel>
  ...
</content-type>
```

The content of the **metadata-panel** element must be a white space-separated list of metadata panel section names. Only the sections you specify in the list will be displayed for content items of this type, and they will be displayed in the order specified. Content types for which you do not specify a **cue:metadata-panel** element will get the default metadata panel sections, displayed in the default order.

The built-in sections must be specified using their CUE **tag names**, all of which start with the prefix "**cue.**". Most of the built in sections are omitted from the above example to keep it short. For a complete list of all the built-in metadata sections and their tag names, see [section 2.2.7.1](#). The tag names of any metadata sections belonging to custom web components are defined in the web component configurations, as described in [section 4.1.2.7.1](#), for example.

### 2.2.7.1 Metadata Section Tag Names

CUE metadata panel sections are identified by **tag names**. All the built-in sections have tag names that start with the characters "**cue.**". To avoid possible future name clashes, you should choose a different prefix when naming any custom metadata sections you create.

#### CUE online sections

UI Label	Tag name
Schedule	<b>cue.schedule</b>



UI Label	Tag name
Authors	<code>cue.authors</code>
General info	<code>cue.general-info</code>
Related	<code>cue.related</code>
Section	<code>cue.section</code>
Tags	<code>cue.tags</code>
Usages	<code>cue.usage</code>
Measurements	<code>cue.online-measurements</code>
Metrics	<code>cue.online-metrics</code>
Semantic analysis (if CUE Semantic is installed)	<code>cue.semantic</code>

### CUE Print sections

UI Label	Tag name
Properties	<code>cue.story-folder</code>
General info	<code>cue.text-general-info</code>
Package Properties	<code>cue.text-package-properties</code>
Package Properties (for storylines)	<code>cue.print-storyline-package-properties</code>
Proofreading	<code>cue.text-proof-state</code>
Related	<code>cue.text-related</code>
Related (for storylines)	<code>cue.print-storyline-related</code>
Text Properties	<code>cue.text-text-properties</code>
Assignment	<code>cue.assignment</code>
Assignment metadata	<code>cue.assignment-usage</code>
Measurements	<code>cue.print-measurements</code>

### Shared sections

UI Label	Tag name
Versions	<code>cue.versions</code>

## 2.2.8 CUE Composer Integration for Storylines

It is possible to open a print story in CUE Composer directly from CUE. No configuration is required to make this integration available for rich text-based stories but for storyline stories, you need to explicitly enable it by including a `cue:integration-target` element in the storyline's content type definition. This element must contain the value `cue-print`.

The `cue:` namespace prefix must be declared (usually in the content-type resource's root element as follows):

```
<content-types xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  xmlns:doc="http://xmlns.vizrt.com/2010/documentation"
  xmlns:media="http://xmlns.escenic.com/2013/media"
  xmlns:video="http://xmlns.escenic.com/2010/video"
  xmlns:livecenter="http://xmlns.escenic.com/2015/live-center"
  xmlns:cci="http://cci/extension/integration"
  xmlns:cue="http://xmlns.cuepublishing.com/configuration"
  version="4">
  ...
</content-types>
```

The `cue:integration-target` element must then be included in the content type definitions of all the storyline content types that you want to be able to open in CUE Composer. For example:

```
<content-type name="storyline">
  ...
  <cue:integration-target>cue-print</cue:integration-target>
  ...
</content-type>
```

If your installation includes multiple CUE Print instances (test, staging, production for example), they must all be configured with different system names. Otherwise this feature may open stories in the wrong instance of CUE Composer.

For storylines that are configured in this way, the following additional CUE Print-related features are available:

- CUE Print-driven locking of storylines and story elements
- CUE Print measurement data, including "write to fit" line counts

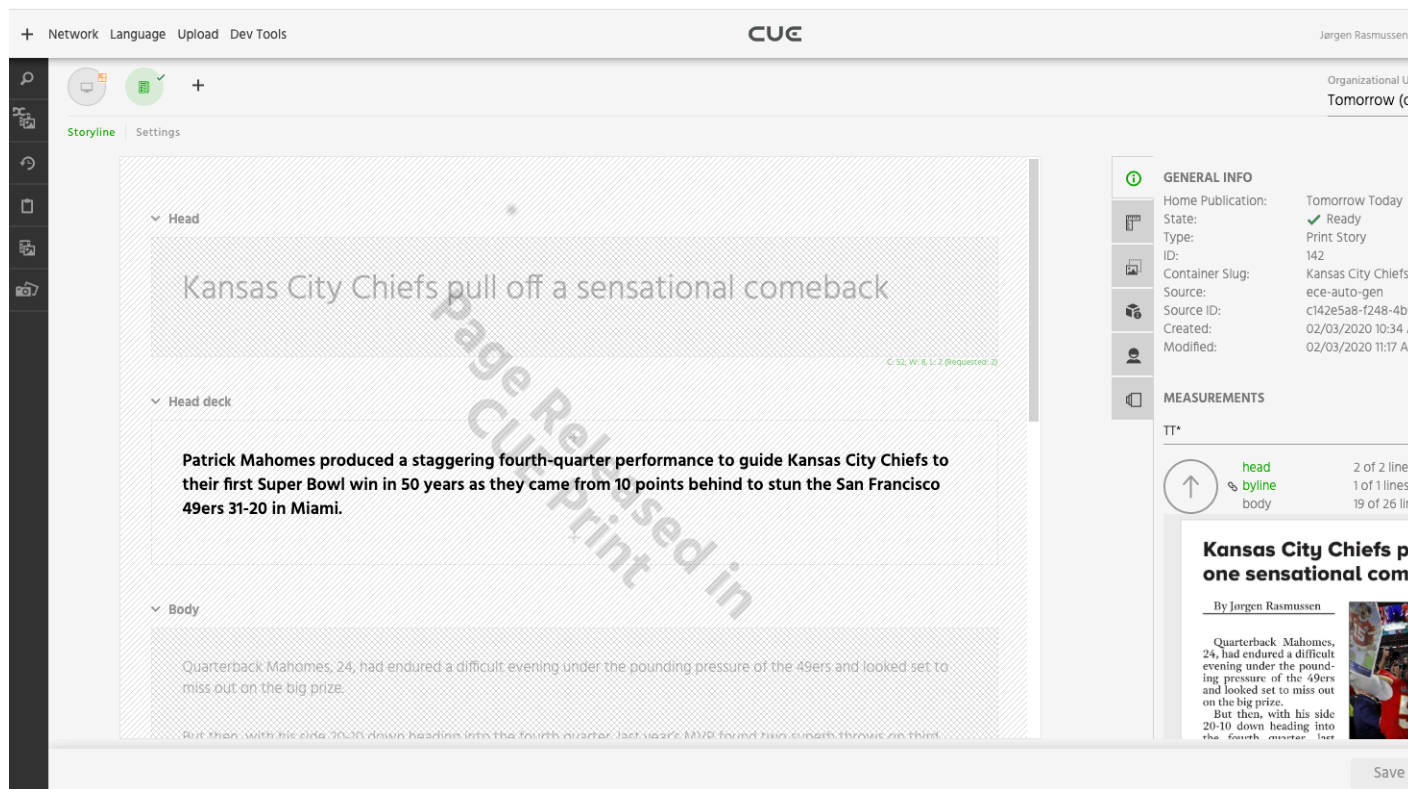
These features are described in the following sections. Both features require the addition of `cue:cue-print` elements to the story element types used in your storylines. A `cue:cue-print` element establishes a mapping between the story element type to which it belongs and the story element type's target CUE Print element tag:

```
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  xmlns:cue="http://xmlns.cuepublishing.com/configuration"
  name="headline">
  ...
  <cue:cue-print elementTag="Headline">
  ...
</story-element-type>
```

The above example indicates that the **headline** story element type is represented by the **Headline** element tag in CUE Print. Note that the **cue:** namespace prefix needs to be declared in the root element of any story element definition to which you add a **cue:cue-print** element (as highlighted in the example above).

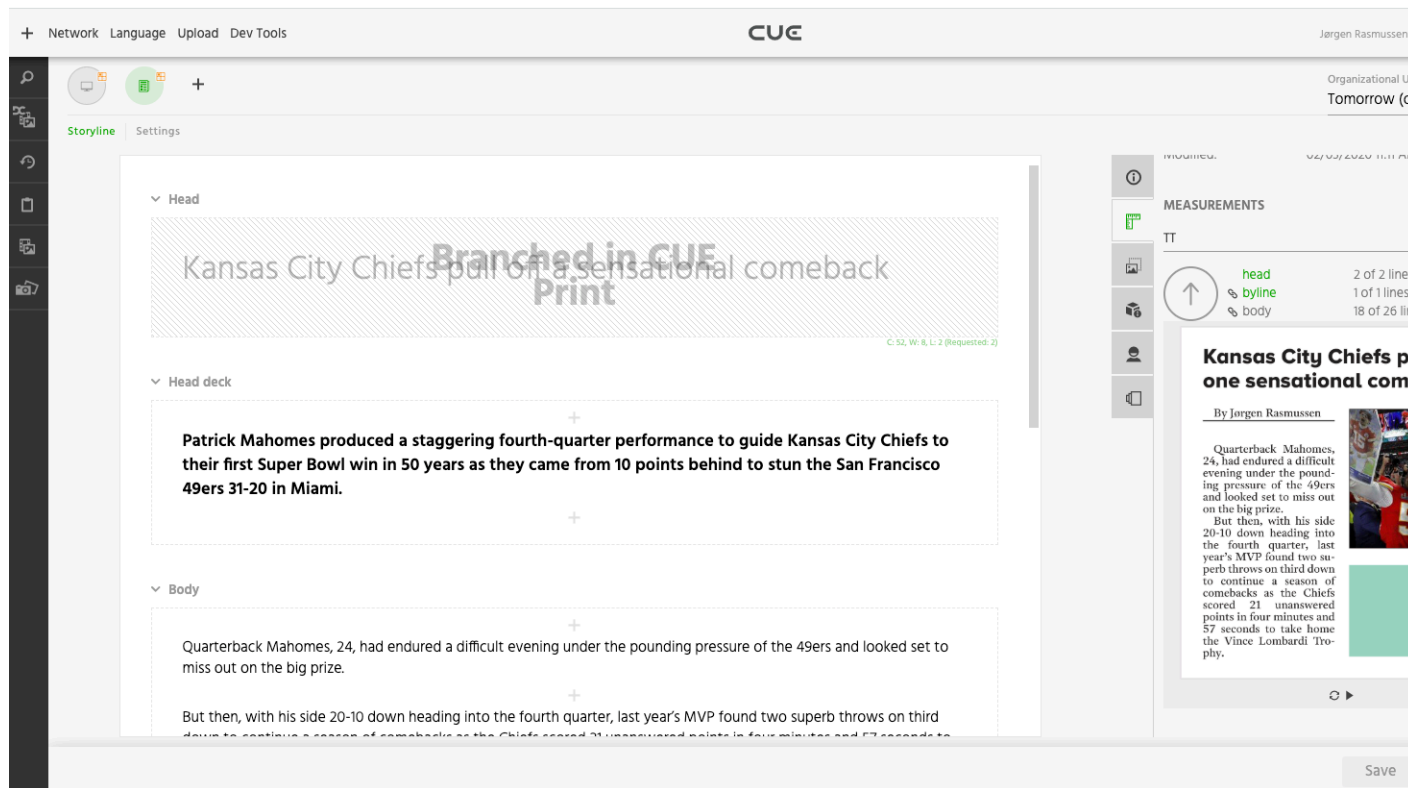
### 2.2.8.1 CUE Print-driven Locking

Once the CUE Print package that inherits a print storyline is released, its content is locked and no changes can be made to it. This change of status is made visible by locking the print storyline in CUE as well. It is no longer possible to edit the storyline, and it is stamped with a message indicating its locked status:



Should the package be "unreleased" in CUE Print, then it is also unlocked in CUE, and its stamp is removed.

In addition, the story elements that make up a print storyline can be individually locked and stamped if the elements they are mapped to are branched in CUE Print:



Should the changes made to an element in CUE Print be reverted, then the related story element is unlocked in CUE and its stamp is removed.

Note that story elements can only be locked in this way if they have been correctly configured with a `cue:cue-print` element.

### 2.2.8.2 Displaying CUE Print Measurement Data

You can configure the story elements in a CUE Print-integrated storyline to display "as you type" measurement data consisting of character, word and "write to fit" line counts:



The line count changes color according to its write-to-fit status:

- Black if the current line count is less than the requested line count
- Green if the current line count matches the requested line count
- Red if the current line count exceeds the requested line count

To enable this functionality for a story element, you must add a `ui:count` element story element's type definition (in addition to the `cue:cueprint` element that specifies its target CUE Print element tag):

```
<?xml version="1.0" encoding="UTF-8"?>
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  xmlns:cue="http://xmlns.cuepublishing.com/configuration"
  name="headline">
  <ui:label>Headline</ui:label>
  <ui:icon>headline</ui:icon>
  <ui:priority>900</ui:priority>
  <ui:count show="true"/>
  <cue:cue-print elementTag="Headline">
  <field name="headline" type="basic" mime-type="text/plain">
    <ui:title-field/>
  </field>
  <ui:style>
    .story-element-headline [contenteditable='true'] {
      font-size: 2.5em;
    }
  </ui:style>
</story-element-type>
```

Note that:

- The `ui:count` element's `for` attribute (used when enabling word/character counts for online storylines, see [section 2.2.29.1](#)) is not used in this context.
- The `ui:count` element can only be used in this way with story elements. Although the `ui:count` element can be added to individual fields of a story element when enabling word/character counts for online storylines (see [section 2.2.29.1](#)), this is not the case when enabling CUE Print-based measurements.

## 2.2.9 Smart Quotes

CUE has a "smart quotes" function that can automatically convert default "straight" single or double quotes to "curly" quotes of various kinds. Different languages (and different publishers) have different quotation mark conventions, so this function is configurable, allowing you to set up CUE to use the quotation marks you require.

Smart quoting is disabled by default. To enable it:

1. If necessary, switch user to `root`.
 

```
| $ sudo su
```
2. Open `/etc/escenic/cue-web/config.yml` for editing. For example:
 

```
| # nano /etc/escenic/cue-web/config.yml
```
3. Add a `useSmartQuotes` property, and set it to `true`:
 

```
| useSmartQuotes: true
```

This enables the smart quotes function.

4. Add a `smartQuotes` property with four child properties called `openDoubleCurly`, `closeDoubleCurly`, `openSingleCurly` and `closeSingleCurly`. Use these properties to

specify the quotation marks you want to use. Straight double quotation marks are replaced by the characters you specify with **openDoubleCurly** and **closeDoubleCurly**, and straight single quotation marks are replaced by the characters you specify with **openSingleCurly** and **closeSingleCurly**. The following settings, for example:

```
useSmartQuotes: true
smartQuotes:
  openDoubleCurly: „"
  closeDoubleCurly: ""
  openSingleCurly: `
  closeSingleCurly: `
```

will replace "quotation" with „quotation” and 'quotation' with ‘quotation’.

5. If you want to limit the smart quotes functionality to rich text fields only, add the following:

```
disableSmartQuotesInNonRichTextFields: true
```

The smart quotes functionality will then not work in plain text fields.

6. Save the file.

7. Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

## 2.2.10 Spelling Checker

CUE can be configured to provide spelling and grammar checking in a number of different ways:

### Use the **cue-spellcheck** service

**cue-spellcheck** is a microservice for CUE, that provides an interface to external spelling/grammar checking services. Currently **cue-spellcheck** only works with storyline content, but support for all content items fields is planned for the future. You are recommended to use this spelling checker if possible.

### Use the CUE Print spelling checker

If your installation includes CUE Print, then you can configure CUE to use the CUE Print spelling checker. The CUE Print spelling checker only works with classic CUE content (not with storylines).

### Depend on the browser

Chrome includes a built-in spelling checker, and there are also spelling checker plug-ins for Chrome. If you do not add any explicit spelling checker configuration, then whatever spelling/grammar checker is configured in the browser will be used.

If you are going to use the **cue-spellcheck** service, then before you configure CUE, you will need to install and configure **cue-spellcheck**, as described in [section 3.2](#).

To enable the **cue-spellcheck** service or the CUE Print spelling checker:

1. If necessary, switch user to **root**.

```
$ sudo su
```

2. Open **/etc/escenic/cue-web/config.yml** for editing. For example:

```
# nano /etc/escenic/cue-web/config.yml
```

- For the CUE Print spelling checker, just add the following settings:

```
cueSpellCheck:
  enabled: true
  defaultState: on
```

For **cue-spellcheck**, you need to include one additional setting:

```
cueSpellCheck:
  enabled: true
  defaultState: on
  serviceURL: 'https://cue-spellcheck-url/spellcheck'
```

where *cue-spellcheck-url* is the URL of the **cue-spellcheck** micro-service.

The above settings both enable the selected spelling checker and switch it on by default for all users. If you don't want it to be on by default for all users, then set **defaultState** to **off** instead.

- Save the file.
- Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

Whether you set the spelling checker on or off by default, CUE users can subsequently switch it on or off for themselves. The option can be found under **Personal preferences** on the **Settings** page. Their selected setting is saved on the device, so users who use multiple devices will need to make the setting separately on each device.

If you do not add a spelling checker configuration as described above, then whatever spelling/grammar checker is configured in the browser will be used.

### 2.2.11 Semantic Shortcut Key Combination

CUE has a **semantic shortcut** feature that provides keyboard-only access to CUE features. By default, a semantic shortcut is introduced by pressing the **Shift** key twice in quick succession: this displays a small dialog listing additional keys the user can press to complete a shortcut and execute an action. You can, however, replace the **Shift Shift** introductory key sequence with some other key sequence or key combination if required. You can also change the maximum interval between the keypresses in an introductory key sequence.

To change the default semantic shortcut settings:

- If necessary, switch user to **root**.

```
$ sudo su
```

- Open **/etc/escenic/cue-web/config.yml** for editing. For example:

```
# nano /etc/escenic/cue-web/config.yml
```

- To replace the default **Shift Shift** sequence add the following setting:

```
keyboardShortcuts:
  semanticToggle: "new-sequence"
```

where *new-sequence* is a key sequence specification such as **mod mod** (which specifies a key **sequence**) or **mod+alt+a** (which specifies a key **combination**).

- If you are using a key sequence (either the default shift shift or a sequence you have defined yourself), you can also control how quickly the user has to type the sequence in order for it to be recognized. By default, the interval between the two keypresses must not exceed 500 milliseconds. To increase the interval to 600 milliseconds, for example, you would need to enter a **resetSequenceTimeout** property as a child of the same **keyboardShortcuts** property:

```
keyboardShortcuts:
  resetSequenceTimeout: 600
```

- Save the file.
- Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the change you made.

Note that:

- The key identifier **mod** represents the **ctrl** key on Windows or the **command** key on Mac. You should always use **mod** rather than **ctrl** or **command** to ensure that semantic shortcuts will work on both platforms.
- You can in theory use any sequence or combination of keys to introduce semantic shortcuts, but in order to avoid problems you are recommended to stick to either a modifier sequence such as shift shift or a combination of modifiers and characters such as **mod+alt+a**. You should also take care to avoid combinations that are already in use either by CUE itself or by the browser.

For more detailed information about supported key combinations and how to specify them, see the documentation of the Javascript library used to provide this functionality: [Mousetrap](#).

## 2.2.12 Content Creation Shortcuts

You can make custom shortcuts for creating new content items of specific types by adding a **quickCreateShortcuts** entry to one of the configuration files in `/etc/escenic/cue-web/`. The **quickCreateShortcuts** must contain an array of definitions, each one defining the shortcut(s) for creating a different type of content item. For example:

```
quickCreateShortcuts:
- name: "Regular Story"
  keystroke: "r"
  quickCombo: ["ctrl+alt+r"]
  contentType: "regular-news-story"

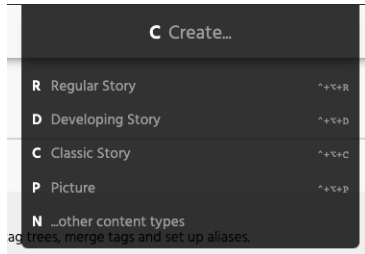
- name: "Developing Story"
  keystroke: "d"
  quickCombo: ['ctrl+alt+d']
  contentType: "developing-story"

- name: "Classic Story"
  keystroke: "c"
  quickCombo: ['ctrl+alt+c']
  contentType: "story"

- name: "Picture"
  keystroke: "p"
  quickCombo: ['ctrl+alt+p']
  contentType: "picture"
```



The primary purpose of this configuration is to create custom **semantic** shortcuts for content creation. Once you have created at least one such shortcut, then the existing **Shift Shift C (Create)** semantic shortcut will, instead of immediately displaying the **Create new** dialog, display the semantic shortcuts you have created, plus an **N ...other content types** option that displays the **Create new** dialog:



Selecting any of the custom shortcuts creates the new content item immediately.

You can, however, (as in the example shown above) create additional key combination shortcuts.

Each shortcut definition consists of the following properties:

**name (required)**

The label to use in the semantic shortcut dialog.

**keystroke (required)**

The semantic shortcut keystroke to follow **Shift Shift C**. You can use any key except **N**. Omit this property if you only want to create a standard key combination shortcut.

**quickCombo (optional)**

An alternative key combination shortcut. Make sure you avoid the combinations already used by CUE and the browser. Omit this property if you only want to create a semantic shortcut.

**contentType (required)**

The name of the content type to create.

### 2.2.13 The Publish Shortcut

By default, the **Publish** buttons in CUE are assigned the shortcut **ctrl+shift+s / command+shift+s**. You can, however, replace this default with another shortcut or remove it entirely by adding a **publishButtonShortcut** entry to one of the configuration files in **/etc/escenic/cue-web/**. Removing the shortcut from these buttons reduces the risk of users unintentionally publishing content while editing.

To remove the default shortcut from the **Publish** buttons, set the value of **publishButtonShortcut** to an empty array:

```
| publishButtonShortcut: []
```

To replace the default shortcut with your own shortcut, fill the array with the key combinations you want to use:

```
| publishButtonShortcut: ['mod+alt+p']
```

**mod** is shorthand for both the **ctrl** key on Windows and the **command** key on Macs, so the above is equivalent to:

```
| publishButtonShortcut: ['ctrl+alt+p','command+alt+p']
```

Be careful not to choose a key combination that is already in use – either by CUE, the browser or the operating system.

## 2.2.14 Content Card Date Type

The content cards displayed in lists such as search results lists include a date field that can be used as a sorting key. By default, the date shown in this field is the content item's creation date (or in fact its creation time). You can optionally replace the creation date/time with the last-modified date/time if you consider this to be a more useful sort key.

To replace creation date/time with last-modified date/time:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open **/etc/escenic/cue-web/config.yml** for editing. For example:

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following setting:

```
| useModificationTimeOnContentCard = true
```

4. Save the file.

5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the change you made.

## 2.2.15 Storyline Symbol Insertion

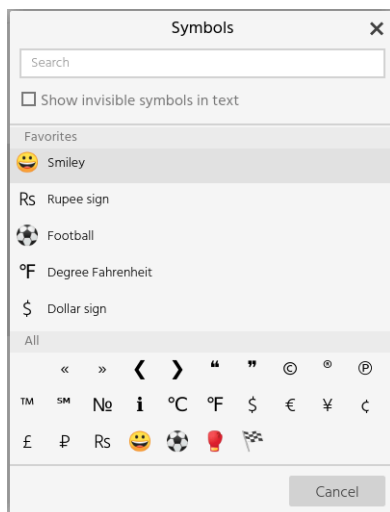
CUE's storyline editor includes a tool for easily inserting symbols and special characters in the storyline text: non-Latin characters, mathematical symbols, currency symbols, emojis and invisible characters such as soft hyphens and non-breaking spaces. The purpose of the symbol insertion tool is not to enable the insertion of **any** character (computer operating systems in any case provides methods for doing that) but to make the insertion of commonly-used special characters easy. Before you can use the symbol insertion tool, therefore, you need to configure it with a selection of the characters and symbols likely to be actually required at your installation.

Once it is configured, the symbol insertion tool provides the following capabilities:

- Keyboard shortcuts for inserting characters and symbols
- A **Symbols** dialog from which characters and symbols can be selected
- A function for revealing the location of invisible characters such as soft hyphens

The symbol insertion functionality is only available in text-only story elements such as **headline**, **lead-text**, **paragraph**, **pull-quote** and so on. It will not work in story elements that contain non-text fields: none of the symbol insertion features will work in in the **caption** field of an **image** story element, for example.

The **Symbols** dialog is displayed by means of a semantic shortcut (**Shift Shift Y**) and looks like this:



All the special characters configured for use at your site are displayed at the bottom of the dialog under **All**. In addition, each user's 5 most frequently used characters are displayed at the top of the dialog under **Favorites**. Any displayed character can be inserted by clicking on it, and hovering the mouse over a character will display its name (useful for invisible characters such as soft hyphens). There is also a **Search** field at the top of the dialog that you can use to search for symbols by name.

Checking the **Show invisible symbols in text** option highlights any invisible characters in the storyline in blue, rendering them visible. If you check or uncheck this option, then an **Apply** button is displayed in the dialog so that you can close the dialog and apply the change. In this way you can switch highlighting of invisible characters on and off.

Depending on how the symbol insertion tool is configured, it may also be possible to enter some characters just by pressing a keyboard shortcut, without displaying the **Symbols** dialog (see the description of the **shortcut** property below).

To enable the symbol insertion functionality, you need to add a **specialChars** property like this to one of your CUE configuration files:

```
specialChars:
- name: 'Soft Hyphen'
  icon: '-'
  invisible: true
- name: 'Left double-angle quote'
  icon: '«'
  shortcut: ["ctrl+alt+," , "command+alt+," ]
- name: 'Right double-angle quote'
  icon: '»'
  shortcut: ["ctrl+alt+." , "command+alt+." ]
- name: 'Copyright'
  icon: '©'
```

The **specialChars** property must contain an array of entries defining the characters you want users to be able to insert. Each entry consists of the following properties:

**name**

The character/symbol name as you want it to appear in the **Symbols** dialog. **(Required)**

**icon**

The actual character or symbol to be inserted. **(Required)**

Note that in the example shown above, the **icon** appears to be an empty string, but is in fact a soft hyphen character (U+00AD).

**shortcut**

An array of one or more shortcut definitions that can be used to insert the character. Make sure to avoid shortcuts that are already used by the operating system or the browser. **(Optional)**

**invisible**

Set this to true for invisible characters that you want to be able to highlight using the **Show invisible symbols in text** function. **(Optional)**

## 2.2.16 Section Page Metadata Panel Width

You can increase the width of the metadata panel in the section editor has been increased. Some users may prefer this layout, since much of the section page editor's functionality is located in the metadata panel.

To enable an extra-wide section page metadata panel:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open **/etc/escenic/cue-web/config.yml** for editing. For example:

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following setting:

```
| wideSectionMetadataPanel = true
```

4. Save the file.

5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the change you made.

## 2.2.17 Search Filters

The CUE search panel offers a set of search filters that allow users to narrow down the results of a search by limiting the results to all documents of a specified type or all documents created after a certain date, and so on. It is difficult to design a set of filters that meets all customers' requirements, so the CUE search filters are configurable. By editing a Content Store configuration file, you can:

- Determine which filters appear in the search panel's filters list
- Determine the order in which the filters appear
- Add your own custom filters

Exactly how you modify the search filters offered in the CUE search panel depends on whether your CUE installation has a CUE Content Store back end, or an Escenic Content Engine back end:

- If you have a CUE Content Store back end, then it is a Content Store configuration task. No configuration work is required in CUE itself. For instructions on how to modify CUE's search filters, see [Custom Search Filter Definitions](#).
- If you have an Escenic Content Engine back end, then see [section 2.2.17.1](#) below.

### 2.2.17.1 Modifying the Search Filter Panel (Escenic Back End)

Custom filters are simpler than the predefined filters: they are simple tests that the CUE user can only turn on or off. You could, for example, create a "Premium content" filter that selects only content items with a Boolean `premium` field that is set to `true`.

To modify the search filter panel:

1. If necessary, switch user to `root`.

```
| $ sudo su
```

2. Open `/etc/escenic/cue-web/40-search-filter.yml` for editing. For example:

```
| # nano /etc/escenic/cue-web/40-search-filter.yml
```

3. Modify the default search filter layout to meet your requirements:

```
| searchFilter:
|   - id: "document-types"
|     name: "Document Types" #translate
|   - id: "document-states"
|     name: "Document States" #translate
|   - id: "creation-date"
|     name: "Creation date" #translate
|   - id: "authors"
|     name: "Authors" #translate
|   - id: "sections"
|     name: "Sections" #translate
|   - id: "tags"
|     name: "Tags" #translate
```

You can, for example, change the order of the predefined filters and remove any you don't need by commenting them out:

```
| searchFilter:
|   - id: "document-types"
|     name: "Document Types" #translate
|   - id: "document-states"
|     name: "Document States" #translate
|   - id: "creation-date"
|     name: "Creation date" #translate
|   - id: "sections"
|     name: "Sections" #translate
|   - id: "authors"
|     name: "Authors" #translate
| # - id: "tags"
| #   name: "Tags" #translate
```

You can also add custom filters of your own. You can insert a custom filter anywhere you like, for example:

```
| searchFilter:
|   - id: "document-types"
|     name: "Document Types" #translate
|   - id: "document-states"
```

```

    name: "Document States" #translate
  - id: "creation-date"
    name: "Creation date" #translate
  - id: "premium-content"
    name: "Premium Content" #translate
    query: "premium_b:true"
  - id: "sections"
    name: "Sections" #translate
  - id: "authors"
    name: "Authors" #translate
# - id: "tags"
#   name: "Tags" #translate

```

4. Save the file.

5. Enter:

```
# dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

Note the following:

- A custom filter's **query** property of must contain a valid Solr query clause. This means that in order to write such a clause you need to know both [Solr query syntax](#) and your Solr schema (in order to know what fields are indexed and how to identify the fields correctly).
- The predefined search filters have fixed IDs. Make sure that your custom filter IDs do not clash with them.

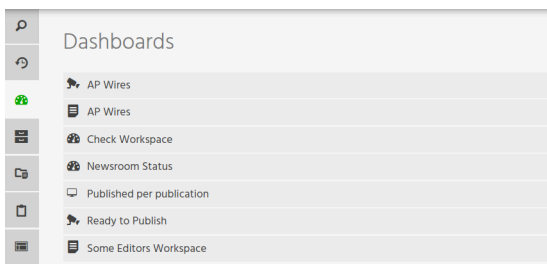
## 2.2.18 Dashboards (Content Store only)

The functionality described in this topic depends on use of the CUE Content Store. If you are using CUE with an Escenic Content Engine back end, then it is not available.

A **dashboard** is a panel that contains one or more constantly updated lists of content items maintained by CUE. Dashboards provide an easy way for editors and others to maintain control over the editorial workflow. A dashboard might, for example, contain two lists: one showing all draft content items in the Sports section and the other showing all approved content items in the same section. Another dashboard might contain a single list showing all image content items that are in the approved state. You can define any number of dashboards.

The content of a dashboard is updated every 30 seconds by default.

All the dashboards available to you can be displayed by selecting  from the left hand navigation menu:

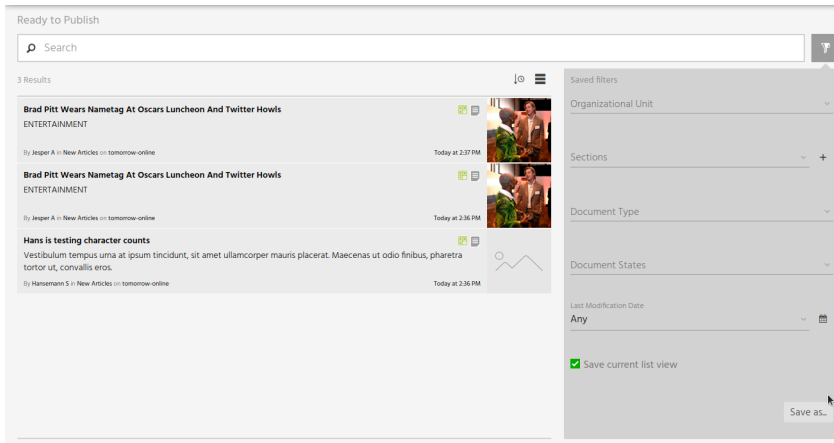


Double-click on one of the listed dashboards to display it in a new tab.

## CUE Tech Guide

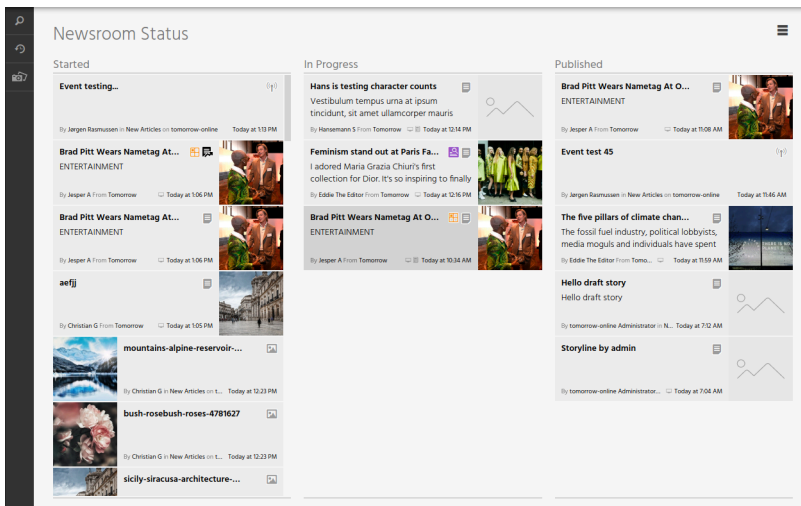
The content item lists displayed in a dashboard are the results of predefined searches. The appearance and functionality of a dashboard depends upon whether it contains just one search result list, or several.

A single-search dashboard looks more or less identical to the CUE search panel:



Like an ordinary search panel it has a search field at the top and a filter drop-down on the right that you can use to narrow down the contents of the list. And as with the standard search panel, you can also save searches. A saved search created in a dashboard belongs to the dashboard: you will not find it in the standard search panel or in any other dashboard.

A multiple search dashboard has a simpler layout, and the searches it provides cannot be modified:



There is no search field or filter panel: just lists of search results.

Creating dashboards is a Content Store configuration task. No configuration work is required in CUE itself. For instructions on how to create dashboards, see [Dashboard Definitions](#).

### 2.2.19 Asset Picker Custom Search Filters (Content Store only)

The asset picker dialogs displayed by CUE for selecting content items in various contexts (for example selecting an image, video or relation to include in a story) have a search filter button. Clicking this

button displays a search filter form that lets the user narrow down the list of selectable content items in various ways.

CUE is delivered with a standard search filter that is used in all search panels, dashboards and asset pickers by default. It is, however, possible to modify this standard search filter and to define your own search filters. If CUE has a Content Store back end then you can replace asset picker search filters with these custom search filters. This feature is not, however, available with Escenic Content Engine back ends.

You can configure an asset picker to use a custom search filter in two different ways:

- For a standard content item relation, you do it by inserting a `ui:search-filter-name` element as a child of the appropriate `relation-type` element in a content type definition (see [Customizing Relation Asset Picker Filters](#)).
- For story elements that represent relations such as "image", "video", "gallery" and "relation" story elements, you do it by inserting a `ui:search-filter-name` element as a child of the `link` type `field` element in the relevant `story-element-type` definition (see [Image Element Type Filters](#)).

In both cases the `ui:search-filter-name` element must contain the name of the search filter that you want to use:

```
| <ui:search-filter-name>my-custom-search-filter</ui:search-filter-name>
```

### 2.2.20 Autosave Interval

By default, changes made in a content editor are autosaved every three seconds. If you wish to change the autosave interval, you can do so by adding a configuration setting as follows:

1. If necessary, switch user to `root`.

```
| $ sudo su
```

2. Open `/etc/escenic/cue-web/config.yml` for editing. For example

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following setting:

```
| autoSaveInterval: interval
```

where *interval* is the required update interval in milliseconds (the default setting is 3000).

4. Save the file.
5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

### 2.2.21 Quick View

CUE has a **quick view** feature that lets you quickly display a content item without having to actually open it in an editor. To display a quick view you select the content item you are interested in (for example, in a search results list) and press the space bar. This displays a pop-up window containing a simplified view of the content item. The quick view is superficially similar to a content editor but usually contains only a subset of the content item's fields, and it is not editable. You can navigate up



and down in the content item list with the quick view open, and the contents of the view will change to show the currently selected item. To close the quick view, press the space bar a second time.

The quick view feature is available in:

- Search results lists
- Dashboard lists
- The **Recent** list
- The metadata panel **Related** and **Usage** sections
- The section page editor
- Inbox and list editors

In order to be useful, the quick view function must be configured by adding `ui:preview` elements to the fields in your content type and story element type definitions. Unless you do this, the main part of the quick view window will always be empty (although the metadata panel on the right of the window will contain some basic metadata).

To make a field show up in quick views, you need to add a `ui:preview` element as a child of its `field` type definition. For a storyline-based content item, this will be a `field` element in a story element type definition (see [Story Element Types](#)). For a rich text-based content item, it will be a field element in a content type definition (see [The content-type Resource](#)).

The following example shows a field definition containing a `ui:preview` element:

```
<field name="caption" type="basic" mime-type="text/plain">
  <ui:label>Caption</ui:label>
  <ui:preview/>
</field>
```

The `ui:preview` element currently only works for plain text, rich text and image binary fields. If you add it to other kinds of fields, it will be ignored.

### 2.2.21.1 Quick View "Thumbnail" Threshold

In order to ensure that the quick view feature is actually quick, large images are not displayed at full resolution. Any image above a specified size is replaced by a lower resolution copy. These quick view copies are referred to as "thumbnails", although they are considerably larger than normal thumbnail images.

By default, an image is replaced by a thumbnail version if it is larger than 1 megabyte. You can, however change this default by adding a `quickViewThumbnailThreshold` property setting to one of your CUE configuration files:

```
quickViewThumbnailThreshold: 1500000
```

The threshold must be specified in bytes.

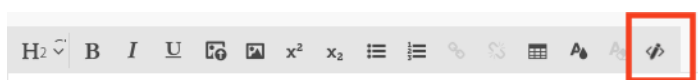
The quick view thumbnail feature depends on the Content Store metadata extraction feature (see [Metadata Extraction](#)) in order to be able to determine the size of the original image. If this Content Store feature is not enabled then the original image will always be used, irrespective of the value set with `quickViewThumbnailThreshold`.

## 2.2.22 HTML Source Editing

By default, CUE's rich text editor (used for editing XHTML-based rich text fields, not storyline content) does not include a source editing feature as in most cases it is not required. In some cases, however, rich text fields may end up containing unwanted or incorrect HTML markup. In such cases, access to a source editor may be needed to correct the problem.

You can therefore optionally enable a source editing option in the rich text editor. The editor cannot be globally enabled, but you can enable it for specific content types and / or specific rich text fields by adding `ui:allow-source-editor` elements to a publication's content-type resource. For detailed instructions on how to do use this element, see [allow-source-editor](#).

For any rich text field where HTML source editing is enabled, the following button is added to the editor toolbar:



The HTML source editing option is intended to be used for the correction/removal of invalid or unwanted HTML, not for the insertion of custom HTML content.

## 2.2.23 Cleaning up Pasted Content

Content copied from external sources such as web pages can contain a lot of unwanted and potentially dangerous markup. CUE therefore filters all content pasted into the rich text editor, removing everything except a small subset of HTML formats that are considered to be both useful and harmless. CUE's has two whitelists of allowed formats: a very restrictive one for print stories:

```
b i u sub sup p br ul ol li table thead tbody tfoot tr th td
```

and a slightly less restrictive one for online stories that includes headings, images and links:

```
h1 h2 h3 h4 h5 h6 b i u sub sup p br a[href] a[target] a[rel] ul ol li img[src]
img[alt] img[width] img[height] table thead tbody tfoot tr th td
```

The print whitelist is fixed, but you can override the online whitelist for a rich text fields by adding a `ui:whitelisted-elements-onpaste` element to the field definition in your publication's `content-type` resource. The `ui:whitelisted-elements-onpaste` element must be added as a child of the `field` element. If you want to change the whitelist of all the rich text fields in your publication then you must add a `ui:whitelisted-elements-onpaste` element to all the rich text `field` elements in your content-type resource.

Here is an example whitelist definition that is more restrictive than the default online whitelist:

```
<ui:whitelisted-elements-onpaste>
  h1 h2 h3 b i p br a[href] a[target] a[rel] a[class=myclass]
</ui:whitelisted-elements-onpaste>
```

For further information about the `ui:whitelisted-elements-onpaste` element, see [here](#).

## 2.2.24 CUE Print Access for Freelancers

To give freelancers full access to CUE print, including content creation rights:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open `/etc/escenic/cue-web/config.yml` for editing. For example:

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following setting:

```
| contractorExtendedAccess = true
```

4. Save the file.

5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the change you made.

Note that this freelancer access is limited to CUE print, it does not include online access.

## 2.2.25 Teaser Anchors in Section Page Previews

CUE can generate teaser-specific section page previews – previews that automatically scroll to a selected teaser. This can be very useful when working on publications with very long section pages. Instead of having to search for the teaser you are interested in after displaying the preview, you just select the teaser in CUE before you generate the preview: the preview then automatically scrolls to the correct location in the preview.

CUE generates a teaser-specific preview by including a fragment identifier in the URL passed to the preview tab:

```
| http://mywebsite.com/#ece-12345?poolId=104&token=-1361898978
```

This URL will cause the displayed preview to scroll down to the teaser marked with the anchor `<a id="ece-12345"></a>`. CUE only includes a fragment ID in the preview URL if a teaser is selected in the section page editor when the preview button is pressed. The fragment ID has the form:

*prefix content-item-id*

where *prefix* is by default **ece-** and *content-item-id* is the internal ID of the selected teaser.

You can change the prefix that is used as follows:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open `/etc/escenic/cue-web/config.yml` for editing. For example:

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add the following setting:

```
| previewAnchorPrefix = "your-preferred-prefix"
```

4. Save the file.

5. Enter:

```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the change you made.

## Required front-end configuration

In order for this feature to work, you must configure your front end application (i.e the CUE Front Waiter) to add matching HTML anchors to section page teasers. The Waiter application in the Tomorrow Online demo supplied with CUE Front 1.9 or higher includes a Twig template for this purpose called **anchor.twig**:

```
| <a id="ece-{{ articleId }}"></a>
```

If you want to know more, download the Tomorrow Online demo and look in the Twig templates to see how it is used.

### 2.2.26 Metadata Panel Section List Length

At installations where content items may appear in very many sections, displaying the complete list of sections in the metadata panel can become a performance problem. You can, however, reduce the impact of this problem by limiting the length of the metadata panel section list.

This option is disabled by default. To enable it, add the following setting to one of your CUE configuration files:

```
| filteredSectionPanel: true
```

When this option is enabled, a content item's section list will only contain the section in which the content item was created and the publication home section. You can expand the list to show all sections by selecting the **ADVANCED** link at the top of the metadata panel.

### 2.2.27 Default Tag Relevance

You can define the default relevance assigned to newly-created tags by adding the following setting to one of your CUE configuration files:

```
| defaultTagRelevance: relevance
```

where *relevance* is one of the following values:

- 0.2  
Corresponds to one bar in the UI.
- 0.4  
Corresponds to two bars in the UI.
- 0.6  
Corresponds to three bars in the UI.
- 0.8  
Corresponds to four bars in the UI.
- 1.0  
Corresponds to five bars in the UI.

If the **defaultTagRelevance** property is not present or is set to any other value, then tags are assigned a default relevance of 1.0 (five bars).

Tags added by CUE Semantic are assigned a relevance by the back end tagging service, and are therefore not affected by the **defaultTagRelevance** setting.

## 2.2.28 Sections Side Panel Preview

By default, the **Sections** side panel used to display the section tree also displays a preview of the currently selected section in CUE's main panel. Generating this preview can negatively affect performance in some cases, and it is therefore possible to disable it. To do so, add the following setting to one of your CUE configuration files:

```
hideSectionPreview: true
```

## 2.2.29 Storyline Metrics (Content Store only)

CUE can display character and word counts for the story elements in a storyline and for individual fields within story elements. If the CUE installation includes a CUE Print back end, then it is also possible to display "write to fit" line counts. In addition to the counts displayed at the bottom of each field or story element, a **Metrics** metadata panel can also be used to display summary counts for the whole storyline.

It is possible to set length constraints for storylines and story element fields, specified as maximum and/or minimum word and/or character counts. If constraints are specified, then they are reflected in the metrics displayed for a story. The specified limits are displayed together with the current word / character counts, and in addition the word / character counts change color to highlight content that is breaking constraints (yellow if the current count is below the specified minimum and red if it is above the specified maximum). These constraints are purely advisory. CUE will not prevent the user from saving or publishing a story that is breaking a length constraint.

The storyline metrics functionality is extremely flexible, and can be configured to display exactly the figures you need.

The storyline metrics described in this section is primarily aimed at online content. At installations with a CUE Print back-end, storylines can alternatively be configured to display measurements supplied by the CUE Print back end. These measurements include print-specific "write-to-fit" line counts. For further information about this, see [section 2.2.8.2](#).

### 2.2.29.1 Configuring Character and Word Counts

CUE can display a character / word count below story elements or fields in story elements:

The word 'plastic' is ringing in society's ears. It seems we hear non-stop about single-use plastics in our everyday lives, and we're taking action: people are eschewing disposable coffee cups, refusing plastic straws and calling out supermarkets for wrapping produce in the stuff. But when you look at the plastic-wrapped tomatoes on a supermarket shelf, what you may fail to see is the plastic that was used to produce the food in the first place. What if the plastic problem goes back much further?

(30) / (84)

The counts are constantly updated as the user types, so they are always correct. The counts are only displayed where they are configured to appear.

To add a count to a story element you need to add a **ui:count** element to the story element's type definition. The **ui:count** element must be inserted as the child of the **story-element-type** element:

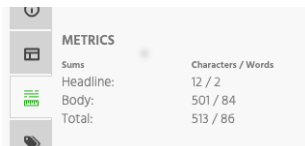
```
<?xml version="1.0" encoding="UTF-8"?>
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="headline">
  <ui:label>Headline</ui:label>
```

```

<ui:icon>headline</ui:icon>
<ui:priority>900</ui:priority>
<ui:count show="true" for="total headlines"/>
<field name="headline" type="basic" mime-type="text/plain">
  <ui:title-field/>
</field>
<ui:style>
  .story-element-headline [contenteditable='true'] {
    font-size: 2.5em;
  }
</ui:style>
</story-element-type>

```

The **show="true"** attribute causes counts to be displayed below story elements of this type. If you set **show=false**, then the characters and words are counted, but not displayed below the story elements. The **for="total headlines"** attribute causes the counts to be added to storyline summaries that can be displayed on a **Metrics** metadata panel:



METRICS	
Sums	Characters / Words
Headline:	12 / 2
Body:	501 / 84
Total:	513 / 86

Specifically, **for="total headlines"** says that the counts are to be added to summaries called **total** and **headlines**. For information about how storyline count summaries are defined, see [section 2.2.29.2](#).

When you add a count to a story element type in this way, all of the text in the story element is counted: all of its text fields, and any child story elements it contains. For some story element types, you may not want to do this. For an image story element type, for example, you might want to count the content of the **caption** field, but not the content of the **copyright** field. In this case, instead of adding a **ui:count** element to the whole **story-element-type** definition, you can just insert it to the field definitions you are interested in:

```

<?xml version="1.0" encoding="UTF-8"?>
<story-element-type xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  name="image">
  ...
  <field name="caption" type="basic" mime-type="text/plain">
    <ui:label>Caption</ui:label>
    <ui:count show="true" for="total body"/>
  </field>
  ...
</story-element-type>

```

### 2.2.29.2 Configuring a Metrics Panel

A metrics section is only included in a content item's metadata panel if it is configured to do so in the content item's type definition:

```

<?xml version="1.0" encoding="UTF-8"?>
<content-type name="story">
  ...
  <cue:metadata-panel>
    cue.general-info

```

```

    cue.section
    cue.metrics
    ...
  </cue:metadata-panel>
  ...
</content-type>

```

For general information about defining metadata panel sections, see [section 2.2.7](#).

To define the summary counts displayed in the **Metrics** panel, you need to add a **storylineMetrics** entry like this to one of your CUE configuration files:

```

storylineMetrics:
  metricPanel:
    - identifier: "headlines"
      label: "Headline"
    - identifier: "body"
      label: "Body"
    - identifier: "total"
      label: "Total"

```

You can define as many summary counts as you like in this way, and they will all be displayed in the **Metrics** panel using the specified **labels**. The **identifiers** are the summary names that must be referenced in the **ui:count** element's **for** attribute. In general, the order of the summary counts on the **Metrics** panel is determined by the order in which story elements appear in the storyline. A summary count with the identifier **total**, however, is always displayed last.

### 2.2.29.3 Configuring Length Constraints

If you configure length constraints, then they are included in metrics counts as shown below:

79 / 12	No constraints
79 (80) / 12	Within max constraint
86 (80) / 14	Outside max constraint
79 (20-) / 12	within min constraint
13 (20-80) / 3	Outside min constraint
79 (20-80) / 12	Within min and max constraint
86 (20-80) / 14	Outside max constraint

The above examples include only character constraints, but word constraints are shown in exactly the same way.

You can configure separate length constraints for storylines and story element fields. The way you configure constraints for storylines is different to the way you do it for story element fields.

#### 2.2.29.3.1 Story Element Field Length Constraints

To set length constraints for story elements or fields, all you need to do is add **ui:minchars**, **ui:maxchars**, **ui:minwords** and **ui:maxwords** elements as children of the **story-element-type** or field's **ui:count** element. For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="headline">
  <ui:label>Headline</ui:label>
  <ui:icon>headline</ui:icon>

```

```

<ui:priority>900</ui:priority>
<field name="headline" type="basic" mime-type="text/plain">
  <ui:title-field/>
  <ui:count show="true" for="total headlines">
    <ui:minchars>5</ui:minchars>
    <ui:maxchars>50</ui:maxchars>
    <ui:maxwords>8</ui:maxwords>
  </ui:count>
</field>
<ui:style>
  .story-element-headline [contenteditable='true'] {
    font-size: 2.5em;
  }
</ui:style>
</story-element-type>

```

You only need to add the constraints you are actually interested in – the `ui:minwords` constraint is omitted from the above example.

### 2.2.29.3.2 Storyline Length Constraints

Storyline length constraints are defined in storyline templates. A storyline template can contain several sets of constraints for different story sizes (short, medium and long, for example). The actual constraints used can then be selected by the CUE user when a story based on the template is actually created. The constraints are defined in a `ui:content-length-restrictions` element that must be inserted in the storyline template as a child of the `elements` element. For example:

```

<elements>
  ...
  <ui:content-length-restrictions>
    <ui:content-length-constraint name="small">
      <ui:label>Small</ui:label>
      <ui:minchars>50</ui:minchars>
      <ui:maxchars>200</ui:maxchars>
      <ui:maxwords>40</ui:maxwords>
    </ui:content-length-constraint>
    <ui:content-length-constraint name="medium" default="yes">
      <ui:label>Medium</ui:label>
      <ui:minchars>150</ui:minchars>
      <ui:maxchars>800</ui:maxchars>
    </ui:content-length-constraint>
    <ui:content-length-constraint name="large">
      <ui:label>Large</ui:label>
      <ui:minchars>500</ui:minchars>
      <ui:maxchars>5000</ui:maxchars>
      <ui:minwords>250</ui:minwords>
      <ui:maxwords>1000</ui:maxwords>
    </ui:content-length-constraint>
  </ui:content-length-restrictions>
  ...
</elements>

```

The above example defines three different sets of constraints, **small**, **medium** and **large**, with **medium** defined as the default. This means that when a content item is created based on this storyline template, the **medium** constraints will be preselected. If none of the `content-length-constraint` elements include a `default="yes"` attribute, then when a content item is created based on this storyline template, no constraints will be set.



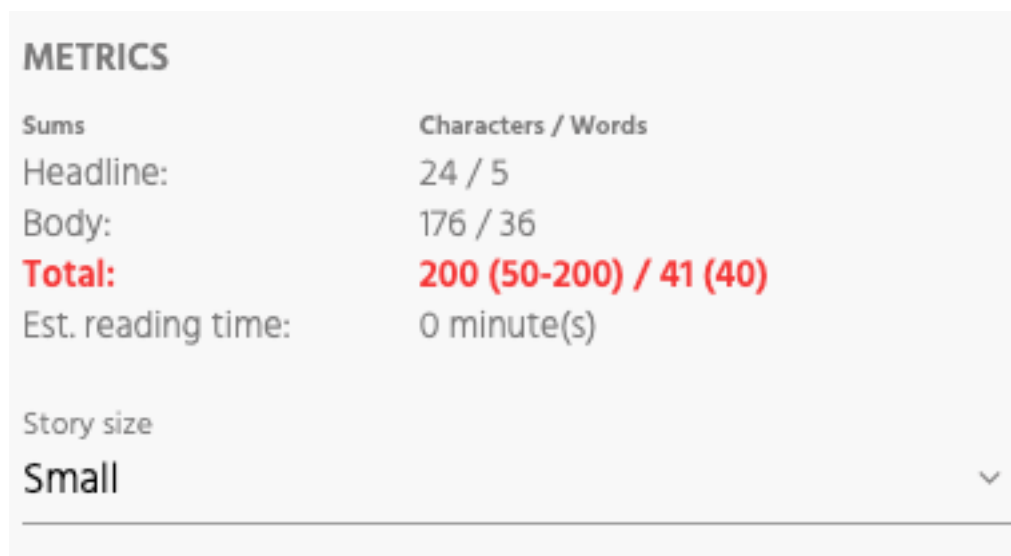
Content types that reference storyline templates containing length constraints will usually need to include a story length field, so that CUE users are able to select the constraint set they want to use. A story length field is a **field** element that contains a **ui:story-size** and a **ui:hidden** element. For example:

```
<field name="story-size" type="basic" mime-type="text/plain">
  <ui:hidden />
  <ui:story-size />
</field>
```

The **ui:story-size** element identifies the field as a story size field, triggering CUE to display it as a drop-down field at the bottom of the **Metrics** panel, where it can be used to select the required length constraint set for the storyline. You must include a **ui:hidden** element to ensure that the field is not displayed as an ordinary content item field. Note the following:

- It does not matter where you put the story size field definition in the content type definition, as long as it is a valid location for a **field** element.
- The field's **type** and **mime-type** attributes are ignored: the field is always displayed as a drop-down in CUE.
- The name of the field is not used by CUE, and nor is any **ui:label** element, should you include one in the field definition.

The following screenshot shows a **Metrics** panel containing a story size field:



If a storyline template contains only one set of constraints and its **default** attribute is set to **yes** then a story length field is not required.

For a full description of the **ui:content-length-restrictions** element and its children, see [here](#).

### 2.2.30 Preview Control Dialog

CUE previews can be modified by means of a floating control dialog displayed over the preview. You can configure this dialog by adding the following settings to one of your CUE configuration files:

```
previewControlSetting:  
  previewAll: false  
  visible: false
```

**previewAll** is set to **true** by default. If you set it to **false** then the dialog's **Also preview unpublished content** checkbox will be unchecked by default and previews will not include unpublished related content.

**visible** is set to **true** by default. If you set it to **false** then the preview control dialog will not be displayed, meaning the end user cannot control what is shown in previews.

Please note that if the preview controller is enabled then the preview is shown in an iframe. This may pose a problem for some websites due to how authorization is handled in iframes. If your website cannot be displayed in an iframe then the preview controller should be disabled.

### 2.2.31 Inline Link Target Window Default

The **Link target** field in the rich text editor's **Inline Link Properties** dialog lets CUE users select whether an inline link they insert should be opened in the current window or a new window. By default, the default selection it offers to users is **This window**. You can, however, change the default selection to **New window** by adding the following setting to one of your CUE configuration files:

```
xhtmlDefaultInlineLinkTarget: "_blank"
```

**xhtmlDefaultInlineLinkTarget** has two possible values:

**\_self (default)**

The default selection offered by CUE is **This window**.

**\_blank**

The default selection offered by CUE is **New window**.

### 2.2.32 Date Picker Default Time

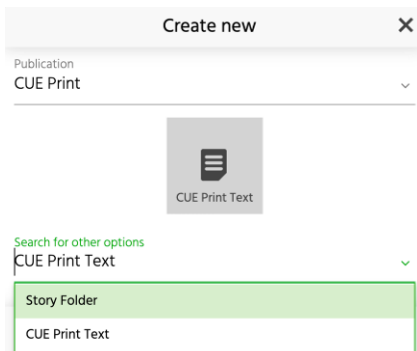
By default, CUE date pickers show the current date and time when first opened. You can if you wish, disable setting of the current time by adding the following settings to one of your CUE configuration files:

```
currentTimeControlSetting:  
  currentTimeInDatePicker: false
```

Date pickers will then still show the current date when first opened, but the time will always be set to 00:00 (12:00 AM).

### 2.2.33 CUE Print Handling in Create New Dialog

At installations with a CUE Print back end, **CUE Print** appears by default as an option in the **Create new** dialog's **Publication** drop-down. Selecting **CUE Print** offers the user the option of creating either a **CUE Print text** or a **Story Folder**:



You can if you wish modify this default behavior by adding a `disableCuePrintOptionsInNewContentDialog` property to one of your configuration files and setting it to `true`. The result of doing this is that no **CUE Print** option will appear in the **Publication** drop-down. **CUE Print text** and **Story Folder** will then be listed as content type options for all publications, in the **Search for other options** list.

### 2.2.34 Access Token Refreshment Timing

At CUE installations where login is managed by CUE User Manager or Google/Facebook, the client is required to refresh its access tokens periodically. By default, CUE refreshes its access token 30 seconds before it is due to expire. You can, however modify this default by adding a `refreshTokenBeforeInSeconds` property to one of your configuration files. You can, for example increase the setting to 60 seconds as follows:

```
refreshTokenBeforeInSeconds: 60
```

### 2.2.35 Environment Visualization

Most CUE installations have multiple environments: a production environment, a test environment and a staging environment. Many CUE users will only ever work in the production environment, but others (developers, testers, maintenance staff and so on) may frequently switch between environments. For such users, it is very important to keep track of which system they are working in: much greater care is needed when working in a production environment than when working in a test environment.

It is therefore possible to set a couple of `environment` properties when configuring CUE to clearly visualize which environment this instance is running in. Adding the following to one of your configuration files, for example:

```
environment:
  name: "Test"
  color: "yellow"
```

will cause CUE's menu bar to be displayed with a yellow background and will also display the name "Test" in the menu bar. This makes it a lot easier for users to distinguish the CUE instances from one another.

You can use any valid CSS color specification when setting the `color` property.

### 2.2.36 Disabling Search-As-You-Type

By default, the CUE search function starts searching as soon as you start typing. Every character you type starts a new, more tightly specified search. In most cases, this is the most efficient way to execute searches: you will often see the result you are looking for before you have completed typing the search term in your head. However, for some organizations with very large databases, this method of searching may actually prove too demanding, and cause performance issues. It is therefore possible to disable it.

To disable the search-as-you-type feature, add the following property setting to one of your configuration files:

```
searchAsYouType: false
```

In order for the search-as-you-type feature to work, CUE adds wildcards to the search terms you enter. When you disable it, these wild cards are no longer added. Search will therefore return different (narrower) result sets with search-as-you-type disabled than it does when the feature is enabled.

### 2.2.37 Enabling User Tracking

CUE's user tracking feature must never be activated unless you have first entered a written agreement with Stibo DX.

CUE's user tracking feature is a product development aid intended to help Stibo DX developers improve the CUE user experience. When it is enabled, CUE user activity is monitored, recorded and sent to Stibo DX for analysis, where it may be stored for up to six months. The data captured includes button clicks, link clicks, page views, JavaScript errors, browser types and geographic regions. No personally identifiable information is collected: IP addresses are 2-byte masked blocks and cookie user IDs contain no personally identifiable information. Nevertheless, the collection of such data is strictly controlled by law in many countries. Therefore, this feature should only be enabled if:

- The customer's CUE users have consented to the collection of the data
- The customer has granted Stibo DX permission to collect and use the data, in writing

To enable user tracking, add the following property settings to one of your configuration files:

```
analyticsConfig:  
  enabled: true  
  customerId: 'jira-project-key'
```

If the `analyticsConfig` property is not present in your configuration or if `analyticsConfig/enabled` is set to any value other than `true`, then the user tracking feature will remain disabled.

`analyticsConfig/customerId` must be set to your organization's project ID in the Stibo DX Jira system (that is, the project key you use when reporting CUE bugs to Stibo DX). In other words if your CUE bug report URLs end with `/MYPROJECT-nnn`, then your `customerId` is `MYPROJECT`.

## 2.2.38 Automated Curation With Sophi

Sophi is a third-party web service (<https://www.sophi.io/>) that uses AI to offer automated content curation based on a sophisticated analysis of both website content and usage data. It can be used with CUE to provide automated deskings of content on section pages and realtime feedback regarding the performance of published content.

In order to enable this feature, you must first set up a Sophi account, and obtain the credentials you need to access Sophi services. Once you have done so, you can configure your CUE installation to support automated curation. You can apply automated curation to all your content, or just parts of it.

Stibo DX accepts no responsibility for the security of content or usage data exported to Sophi, or to any other third party service.

How does the automated curation work? First of all, Sophi must be provided with the information it needs. This is done in two ways. Special Javascript code provided by Sophi is included in all of a publication's content pages. This monitoring code ensures that lots of information about user interaction with the publication content is sent to Sophi. Secondly, the Content Store must send actual content to Sophi every time a content item is published or republished on the site. Sophi uses these inputs (article content plus article usage) to assign a performance score (called a **Sophi score**) to each content item on the site.

Sophi uses the information it gathers to create curation recommendations, which are passed to CUE by filling special Sophi-curated lists. In each section of your publication that you want to be automatically curated you must create one or more lists, and make them available to Sophi. These **curated lists** will be under Sophi's control, and filled with content items by Sophi. CUE users will not be able to open or edit them, only desk them on the section page.

The use of lists as an interface to Sophi means that you can mix automated and manual curation throughout a publication. You can choose to only make use of automated curation in some sections, and within each section page you can manually curate some areas and desk curated lists in others. Sophi continuously monitors the publication content and its performance and updates the content of the curated lists accordingly.

Content item Sophi scores are used to generate icons called **halos** for display in CUE content cards. A content item's halo provides a graphical indication of how well it is performing in its current location on the section page, using a combination of color and size. Green indicates satisfactory or good performance and red indicates unsatisfactory/poor performance. The size of the halo's inner circle represents the content item's current Sophi score.

Setting up automated curation is quite complicated, and requires configuring several components of a CUE installation, not just CUE itself. These are the setup tasks you need to carry out:

1. Create a [Sophi](#) account and obtain from Sophi all the information you need (endpoint URLs, login credentials, tracking code and instructions for how to insert the tracking code).
2. Insert the tracking code in your publication templates in accordance with Sophi's instructions. Precisely where and how you do this will depend on what front-end technology you use for your publications.
3. Decide where you want to make use of automated curation (which parts of which section pages). You can fill an entire section page with automatically curated content, or limit the use of curated content to specific areas on a section page.

4. Create the required lists in CUE. You will need to create at least one list in each section where you want to make use of automated curation, but you might possibly need to create more than one list in some sections. Note that a Sophi-curated list cannot be used to hold anything other than Sophi recommendations. Once you declare that the list is a curated list (see [section 2.2.38.2](#)), you can no longer edit its contents in CUE.
5. Get the web service URIs of each of the lists you have created. You can do this by using the web service to search for each list you have created. Enter a URI like this in your browser URI field:

```
https://content-store-host/webservice/publication/publication-name/escenic/lists?
searchTerms=list-name
```

This returns an XML document. You will find the list's URI in one of the `/feed/entry/link` elements of that document (the one where the `rel` attribute is set to `"edit"`).

You will need these URIs when configuring CUE.

6. Configure a Content Store proxy service that CUE can use to access Sophi. For further information about this, see [section 2.2.38.1](#).
7. Configure the CUE Zipline Sophi plugin. The Sophi plugin performs two functions:
  - It monitors the Content Store, and every time a content item is published or republished, it sends the content item to Sophi for analysis.
  - It polls Sophi at regular intervals for changes to curated lists, and applies the changes supplied by Sophi.

For instructions on how to configure the Sophi plugin, see [The Sophi Plugin](#).

8. Configure CUE itself - see [section 2.2.38.2](#) for details.
9. In CUE, desk the curated lists in the required locations.

### 2.2.38.1 Sophi Proxy Service Configuration

In order for CUE to be able to display halos representing content item performance, it must be able to retrieve content items' Sophi scores from Sophi, for which authentication is necessary. Instead of CUE itself being configured with the necessary credentials, all requests to Sophi are directed via a suitably configured Content Store proxy service. The proxy service forwards all requests to Sophi, and forwards all responses back to CUE.

For general information about how to create a Content Store proxy service, see [Content Store Proxy Services](#). For information on how to set up a proxy service to provide the OAuth authentication required by Sophi, see [OAuth Client Credentials Flow](#).

A correctly configured proxy service for accessing the Sophi halo service should consist of three configuration layer `.properties` files that look something like this:

```
/com/escenic/webservice/proxy/ProxyResourceConfig.properties
| serviceMapping.sophi-halos=https://halo-api.sophi.io/

/com/escenic/webservice/proxy/
ProxyResourceAuthorizationHeaderProvider.properties
| provider.sophi-halos=./WebServiceAuthorizationHeaderProvider

/com/escenic/webservice/proxy/
WebServiceAuthorizationHeaderProvider.properties
| $class=com.escenic.webservice.proxy.auth.OAuth2ClientCredentialsAuthorizationHeaderProvider
```

```
tokenUrl=https://sophi-prod.auth0.com/oauth/token
clientId=client-id
clientSecret=client-secret
customField.audience=https://api.sophi.io
customField.grant_type=client_credentials
```

where *client-id* and *client-secret* are the credentials supplied by Sophi

### 2.2.38.2 Sophi CUE Configuration

Configuring CUE to make use of Sophi automated curation involves the following steps:

1. Decide where you want to make use of automated curation (which parts of which section pages). You can fill an entire section page with automatically curated content, or limit the use of curated content to specific areas on a section page.
2. Create the required lists in CUE. You will need to create at least one list in each section where you want to make use of automated curation, but you might possibly need to create more than one list in some sections. Note that a Sophi-curated list cannot be used for anything else. Once you declare that the list is a curated list (see step 4 below), you can no longer edit its contents in CUE.
3. Get the web service URIs of each of the lists you have created. You can do this by using the web service to search for each list you have created. Enter a URI like this in your browser URI field:

```
https://content-store-host/webservice/publication/publication-name/escenic/lists?
searchTerms=list-name
```

This returns an XML document. You will find the list's URI in one of the `/feed/entry/link` elements of that document (the one where the `rel` attribute is set to `"edit"`).

4. Add the following to one of your CUE configuration files:

```
sophiIntegration:
  sophiControlledLists:
    - 'https://content-store-host/webservice/escenic/list/list-id'
    - 'https://content-store-host/webservice/escenic/list/list-id'
    ...etc
  performanceUpdateIntervalMinutes: 5
  performanceScoreType: '30m'
```

where:

**sophiControlledLists**

Contains the URIs of the curated lists you have created (see [section 2.2.38](#) for a description of how to find these URIs).

**performanceUpdateIntervalMinutes**

Specifies how frequently you want the halos displayed on content cards to be updated.

**performanceScoreType**

Must be set to one of the following values: **10m**, **30m**, **60m** or **180m**. It specifies the length of time (in minutes) over which content item performance is evaluated when generating the Sophi scores from which halos are generated. The default value is **30m**, which means that when a content item's halo is updated, it is based on the content item's performance during the preceding 30 minutes.

5. Save the file.
6. Enter:

```
# dpkg-reconfigure cue-web-3.16
```

Once you have done this, the lists you have added to the **sophiControlledLists** entry in the configuration file will be under Sophi's control. You will no longer be able to open them for editing in CUE



## 3 Installing and Configuring Plug-ins

CUE's capabilities can be extended by installing **plug-ins**. CUE plug-ins fall into three categories:

- Base plug-ins supplied by Stibo DX that provide self-contained functional extensions. These plug-ins have no dependencies other than CUE itself and freely available system components such as the nodeJS engine or Java, unless specifically stated. All the information you need to install and configure base plug-ins is here. The following base plug-ins are currently available:

### **cue-content-duplication-enrichment-service**

This plug-in adds content duplication functions to the home page **Search** and **Latest Opened** panels in CUE and to the **Search** side panel. After installing the plug-in, the context menu displayed by right-clicking or long-pressing a content item in these panels will contain two new options, **Duplicate** and **Duplicate as**. These options allow you to quickly make copies of content items.

### **cue-spellcheck**

This plug-in adds a spelling/grammar checker to CUE. **cue-spellcheck** is a micro-service that can potentially connect CUE to a number of different language service providers. Currently, it will only connect to a [LanguageTool](#) service. LanguageTool is an open source, multilingual spelling/grammar checker. There are both free and paid public LanguageTool services available, and it is also possible to run your own private LanguageTool service.

Base plug-in packages follow CUE version numbering: you should only install base plug-ins that have the same version number as CUE.

- CUE plug-ins supplied by Stibo DX. These CUE plug-ins are dependent on Content Store plug-ins as follows:

### **cue-plugin-live**

Depends on CUE Live.

### **cue-plugin-menu-editor**

Depends on the CUE Menu Editor plug-in.

### **cue-plugin-video**

Depends on the CUE Video plug-in.

These plug-ins are automatically installed together with CUE. Any configuration that might be required is described in the appropriate Content Store plug-in guide.

- Third-party plug-ins that are not made by Stibo DX. These plug-ins may or may not have dependencies other than CUE itself. The information you need to install and configure these plug-ins must be provided by the plug-in supplier.

Base plug-ins are installed in the same way as CUE itself, using **apt-get install**, and can either be installed together with CUE, or at any time later. To install the **cue-content-duplication-enrichment-service** plug-in together with CUE, for example, you would do as follows:

```
# apt-get update
# apt-get install cue-web-3.16 cue-content-duplication-enrichment-service-3.16
```

To install it on its own after the installation of CUE, you would only need to enter:

```
# apt-get update
```

```
# apt-get install cue-content-duplication-enrichment-service-3.16
```

For additional instructions regarding the installation of the **cue-content-duplication-enrichment-service** plug-in, see [section 3.1](#).

## 3.1 cue-content-duplication-enrichment-service

This plug-in depends on nodeJS, version 14.16.0 or higher. The **node** command must be available in **\$PATH**. To check whether this is the case, enter:

```
$ which node
```

If this command does not return the path of the **node** executable, then you need to either install it or add its location to **\$PATH**. If **node** is available, make sure you check its version, since the version installed by default on Ubuntu systems is too old:

```
$ node -v
```

If the version number is less than 14.16.0, then you need to replace it with a newer version. For advice on how to do this on Ubuntu, see (for example) [this page](#).

### 3.1.1 Installing cue-content-duplication-enrichment-service

You can install the **cue-content-duplication-enrichment-service** plug-in either at the same time as you install CUE itself, or at any time later. The version number of **cue-content-duplication-enrichment-service** must match the version number of CUE. To install **cue-content-duplication-enrichment-service** on its own after the installation of CUE, log in as **root** and enter:

```
# apt-get update
# apt-get install cue-content-duplication-enrichment-service-3.16
```

This installs the enrichment service and starts it immediately.

### 3.1.2 Configuring cue-content-duplication-enrichment-service

To configure the **cue-content-duplication-enrichment-service**:

1. Log in as **root** if necessary.
2. Open **/etc/escenic/content-duplication-service-3.16/content-duplication-service.yaml** in an editor and add the following content:

```
server:
  port: port-number
  ziplineEndpoint: http://ziplinehost/cue-print-zipline/escenic/convert/default
  endpoint: http://content-store-host/webservice/index.xml
```

```
maxPayloadLimit: "max-payload-size"
```

where:

- *port-number* is your preferred port number (the default is **8082**)
- *ziplinehost* is the domain name of your CUE Zipline service host (see note below)
- *content-store-host* is the domain name of your Content Store host (see note below)
- *max-payload-size* is the maximum amount of data that will be handled by the duplication function. The default maximum size is **1mb**. If this is insufficient you can set it to a larger value. You can specify the payload size using a variety of units: **b**, **kb**, **mb**, etc. - for all options, see the documentation of the [bytes](#) node.js library.

3. You can also optionally add configurations like this for handling unmatched relations:

```
unmatchedRelationsMapping:
  - contentType: storyline
    relationGroup: relations
```

Without such a section, when you duplicate a content item as a different content type, only relations that have a matching relation type are copied to the new content item. Specifying **unmatchedRelationsMapping** allows you prevent these unmatched relations being lost. For each target content type, you can specify a relation group to which unmatched relations can be copied.

4. Save the file.
5. Restart the service as follows:

```
# /etc/init.d/content-duplication-service restart
```

**Note:** The **ziplineEndpoint** property is only required if you need the duplication service to support the conversion of classic rich text-based stories to storyline containers, since this functionality is dependent on CUE Zipline.

You also need to configure CUE to access the duplication service. To do this:

1. Create a file called `/etc/escenic/cue-web/content-duplication-service.yml`, open it in an editor and add the following content:

```
enrichmentServices:
  - name: "Duplicate Service"
    href: "http://myhost:port-number/contentDuplicationService"
    title: "Duplicate Service"
    triggers:
      - name: "on-duplicate"
        properties: {}

authorizedEndpoints:
  - "http://myhost:port-number/"

extendedContextMenuItems:
  - name: "duplicate-service"
    title: "Duplicate"
    trigger: "on-duplicate"
  - name: "duplicate-as-service"
    title: "Duplicate as ..."
```

```
trigger: "on-duplicate"
```

where:

- *myhost* is your CUE host's domain name
  - *port-number* is the same port number you specified in the duplication service configuration file
2. Save the configuration file.
  3. Apply your configuration changes by entering:

```
# dpkg-reconfigure cue-web-3.16
```

You should now be able to duplicate content items using the **Duplicate** and **Duplicate as** context menu options in CUE.

## 3.2 cue-spellcheck

**cue-spellcheck** is a micro-service that connects CUE to an external language service provider. Currently, it will only connect to a [LanguageTool](#) service. LanguageTool is an open source, multilingual spelling/grammar checker. There are both free and paid public LanguageTool services available, and it is also possible to run your own private LanguageTool service.

### 3.2.1 Installing cue-spellcheck

You can install the **cue-spellcheck** plug-in either at the same time as you install CUE itself, or at any time later. The version number of **cue-spellcheck** must match the version number of CUE. To install **cue-spellcheck** on its own after the installation of CUE, log in as **root** and enter:

```
# apt-get update
# apt-get install cue-spellcheck-3.16
```

### 3.2.2 Configuring cue-spellcheck

To configure **cue-spellcheck**:

1. Log in as **root** if necessary.
2. Open `/etc/escenic/cue-spellcheck/cue-spellcheck.yaml` in an editor. It will contain something like this:

```
providers:
  - type: language_tool
    options:
      url: "https://api.languagetool.org"
      username: "user@company.com"
      apiKey: "xxxxxxxxxxxxx"
localeProviderMap:
  en-US: language_tool
  en-GB: language_tool
  de-DE: language_tool
server:
  applicationConnectors:
    - type: http
      port: 9690
```

```

adminConnectors:
  - type: http
    port: 9691
logging:
  level: INFO
loggers:
  com.escenic.cue.spellcheck: ERROR

```

### 3. Edit the highlighted fields to match your requirements:

```
url: "https://api.languagetool.org"
```

Set this property to point to the LanguageTool service you want to use:

- <https://api.languagetool.org> is a free public service. It is rate-limited, and therefore only really useful for test purposes.
- <https://api.languagetoolplus.com/> is a paid public service (sign up at <https://languagetool.org/proofreading-api>). If you use this service, you must configure the **username** and **apiKey** as described below.
- Alternatively you can enter the URL of your own private instance.

```
username: "user@company.com"
```

Set this property to the username you use to login to LanguageTool. This property is require only if you use <https://api.languagetoolplus.com/> as the **url**.

```
apiKey: "xxxxxxxxxxxxxx"
```

Set this property to the API key of LanguageTool. This property is require only if you use <https://api.languagetoolplus.com/> as the **url**.

```

localeProviderMap:
  en-US: language_tool
  en-GB: language_tool
  de-DE: language_tool

```

Add or remove lines to specify the languages for which you require assistance, using the appropriate [IETF BCP47 language tag](https://www.ietf.org/rfc/rfc4747.txt) and the keyword **language\_tool**. See <https://dev.languagetool.org/languages> for a list of supported languages.

No other settings should be modified.

### 4. Save the file.

## 3.2.3 Starting cue-spellcheck

To start **cue-spellcheck**, enter:

```
/etc/init.d/cue-spellcheck start
```

Once it is started, **cue-spellcheck** listens for requests from CUE on the port specified in the configuration file (**server/applicationConnectors/port**). In a production environment, **cue-spellcheck** will normally be accessed via an **nginx** server. In order to complete the setup, you need to configure CUE with the **cue-spellcheck**'s API endpoint (see [section 2.2.10](#)).

### 3.2.4 Specifying Publication Language

If you require spelling/grammar checking in any language(s) other than US English, then you need to specify the language(s). You do this by setting the [com.esenic.cue.spellcheck.language](#) feature on each publication. Features are publication properties set in Content Store **feature** resource files. For general information about feature resources, see [The feature Resource](#). For information about how to update publication resources, see [Update Resources](#). If this property is not set, then a publication's default language is US English.

## 4 Extending CUE

CUE is more than a simple editor for Content Store - it's an extensible platform. It includes three extension mechanisms that you can use to add your own functionality and to integrate external services into your editorial workflows. The extension mechanisms are:

### Web components

CUE web components are HTML/CSS/Javascript components that you can use to add custom functionality to CUE. See [section 4.1](#) for further information.

### Enrichment services

Enrichment services are a very powerful and flexible mechanism for extending CUE's functionality. An enrichment service is an HTTP service that communicates with CUE via a very simple protocol. You can implement your own enrichment services to provide additional functionality and integrate CUE with other systems in various ways. See [section 4.2](#) for further information.

### Drop resolvers

Drop resolvers are HTTP services, rather like enrichment services. Drop resolvers, however, are specifically designed to handle the processing and import of foreign objects dropped into CUE. See [section 4.3](#) for further information.

### URL-based content creation

CUE lets you create a draft content item by simply passing a URL to a browser. A script running in some other application such as Trello, Google Sheets or Slack can simply construct a CUE URL containing the details of a new content item and pass the URL to a browser. CUE will then start in the browser and create the requested content item, ready for the user to continue editing (if required), save and publish. See [section 4.4](#) for further information.

### Logout triggers

A logout trigger is a simple HTTP **GET** request that is sent to a specified URL when the user logs out from CUE. It provides a mechanism for integrators to automatically perform other actions (such as logging out of a VPN) on logout from CUE. For further information see [section 4.6](#).

When a problem arises in CUE, it is sometimes difficult to determine whether the problem is in CUE itself or in an extension you have added. CUE therefore includes a **safe mode** feature that lets you easily disable extensions as a diagnostic aid. For details, see [section 4.7](#).

## 4.1 Web Components

[Web components](#) is the name given to a set of features being added to the W3C HTML and DOM specifications that support the creation of reusable components in web documents and web applications.

CUE makes use of this technology to enable the following types of extensions:

### Editor side panel

An editor side panel is displayed as a pop-out panel on the left side of a CUE editor window (similar to an editor **Search** panel). A custom editor side panel works in the same way as the standard side panels: a new button is added to the column on the left side of the display, and selecting this button opens and closes the panel.

### Editor metadata section

An editor metadata section is displayed in the pop-out attributes panel on the right side of a CUE editor window (similar to the **General info** and **Authors** sections). A metadata section works in the same way as the standard attributes sections: a new button is added to the column on the right side of the display, and selecting this button opens and closes the panel, focused on the appropriate section.

### Custom field editor

A custom field editor extension changes the appearance and behavior of a content item field. You can, for example, configure CUE to display an integer field in a content item as a graphical slider instead of displaying a simple text field. You can also use it to display much more complex components containing many different controls and elements: a color picker component that offers the user several different ways to choose a color, for example.

### Custom storyline element field editor (Content Store only)

A custom storyline element field editor extension changes the appearance and behavior of a storyline element field. It works in much the same way as a custom field editor, and enables the same kinds of possibilities. The map and table included in the CUE Content Store's starter pack are implemented using custom storyline element field editors.

### Home page panel

A home page panel occupies the main work area of the CUE home tab. A custom home page panel works in the same way as the standard **Search** and **Sections** panels: a new button is added to the column on the left side of the display, and selecting this button displays the panel in the main work area.

### Home page metadata section

A home page metadata section is displayed in the pop-out attributes panel on the right side of a CUE editor window (similar to the **General info** and **Pages** sections displayed with the **Sections** home page panel). A metadata section works in the same way as the standard attributes sections: a new button is added to the column on the right side of the display, and selecting this button opens and closes the panel, focused on the appropriate section.

All you need to do to add a web component to CUE is:

- Create a JavaScript file containing the definition of your web component.
- Put the web component definition in a web location that is accessible to CUE.
- Add information about the web component to a YAML configuration file and save the file in the CUE configuration folder (`/etc/escenic/cue-web`). You can either create a separate configuration file for each of your web components, or create a single configuration file for all of them.

This process is described in more detail in the following sections.

## 4.1.1 Creating a Web Component

A web component is an ECMAScript (ES) module. It contains:

- A class extending `HTMLElement`. The class can use the `shadowRoot` to define **local** CSS styles. These styles are **only** applied to HTML elements inside that `shadowRoot` – they will not affect any elements in documents where the web component is displayed.
- A statement to register the class as a custom element. The custom element name **must** contain a `-`.

Here is a skeleton web component that you can use as a basis for your own web components:



```
/**
 * Creating the web component
 */
class MyComponent extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host { width: 100%; display: block; } /* Styles the web component tag */
      </style>

      <!-- Add your web component HTML here -->
    `;
  }

  connectedCallback() {
    console.log('The CUE interface of the web component:', this.cueInterface);
    // The web component is now attached.
    // Add your logic here.
  }

  disconnectedCallback() {
    // The web component is now detached.
    // Add your clean-up logic here.
  }
}
customElements.define('my-component', MyComponent);

/**
 * Creating the icon (if required)
 */
class MyComponentIcon extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `<!-- Add your web component icon HTML here -->`;
  }

  connectedCallback() {
    console.log('The CUE interface of the icon:', this.cueInterface);
    // The icon is now attached.
    // Add your logic here.
  }

  disconnectedCallback() {
    // The web component is now detached.
    // Add your clean-up logic here.
  }
}

customElements.define('my-component-icon', MyComponentIcon);
```

Field editor web components have no use for an icon, so in this case the icon class can be omitted.

### Drag and drop from web components

You can drag objects from all web components except rich text field extensions to drop zones in CUE. Anywhere in the CUE interface that you can drop an uploaded file, you can also drop an object that has been dragged from a web component, as long as the object is correctly constructed. A correctly constructed draggable object is a JSON object with a single property, **files**. This property is an array of objects, each object being composed of three properties:

**name**

The file name of this object

**mimeType**

The mime type of this object

**dataURL OR url**

For external objects, the third property is called **dataURL**, and holds the content of the object, encoded as a [data URL](#). The dropped object may, however, in some cases be an existing CUE content item, in which case the third property is called **url** and holds the URL of the content item.

The entire JSON object must be supplied as the drag event's **dragData** property and be assigned the mime type **application/x-web-component-data**.

#### 4.1.2 The CUE Web Component API

A web component in general contains a class that extends **HTMLElement**. For building CUE web components, CUE provides its own base class that extends **HTMLElement**, **cue.core.webcomponents.CUEElement**, plus a number of subclasses for use in specific types of CUE extension. To make a CUE web component you need to extend one of these subclasses.

**cue.core.webcomponents.CUEElement** itself is defined as follows:

```
export abstract class CUEElement extends HTMLElement
  implements webcomponent.CUEElement {
  public user: webcomponent.User;
  public endpoints: StringMap<uri.URI>;
  public credentials: StringMap<string>;
  public dialog: Dialog;
  public notification: webcomponent.Notification;
  public abstract getTitle(): Promise<Nullable<string>>;
  public abstract getLink(): Promise<webcomponent.Nullable<webcomponent.Link>>;
}
```

This class provides four properties and two functions:

**user**

The current user

**endpoints**

The URL(s) of CUE's back end(s)

**credentials**

The credentials needed to access CUE's back end(s), available as:

```
credentials.esenic
credentials.newsgate
credentials.dc-x
```

**dialog**

An interface that exposes methods for creating dialogs of various kinds. For details, see [section 4.1.2.15](#).

**notification**

An interface that exposes methods for showing and hiding notifications from web components. For details, see [section 4.1.2.14](#).

**getTitle ()**

Returns the title of the current editor.

**getLink ()**

Returns the URL of the content item being edited.

The subclasses fall into three main groups:

**Home page panel / Editor side panel extensions**

There are two classes you can use for adding home page/editor side panels:

**cue.core.webcomponents.SidePanel**

Extend this class to display a home page panel that occupies the main work area of the CUE home tab or a pop-out panel on the left side of a CUE editor window. A custom home page panel works in the same way as the standard **Search** and **Sections** panels: a new button is added to the column on the left side of the display, and selecting this button displays the panel in the main work area. On an editor page, it is displayed as a pop-out panel on the left side, similar to an editor **Search** panel.

**cue.core.webcomponents.ListEditor**

Extend this class to display a pop-out panel on the left side of a CUE list editor window. The **ListEditor** class can only be used in this specific context.

**Metadata panel extensions**

There are several classes you can use for adding custom sections to the pop-out metadata panel displayed on the right hand side of various pages. They all work in the same way: a new button is added to the column on the right side of the display, and selecting this button opens and closes the panel, focused on the appropriate section. The following classes are available:

**cue.core.webcomponents.MetadataPanelCUEElement**

Extend this class to add a custom metadata panel section to the following home page panels:

- The **Search** panel
- The **Recent** panel
- The **Dashboard** panel
- The **Archive** panel

**cue.core.webcomponents.AssignmentEditorMetadataPanel**

Extend this class to add a custom metadata panel section to CUE assignment editors.

**cue.core.webcomponents.SectionsMetadataPanel**

Extend this class to add a custom metadata panel section to the **Sections** home page panel.

**cue.core.webcomponents.SectionPageMetadataPanel**

Extend this class to add a custom metadata panel section to the CUE section page editor.

**cue.core.webcomponents.TextEditorMetadataPanel**

Extend this class to add a custom metadata panel section to CUE content editors (rich text).

**cue.core.webcomponents.StorylineEditorMetadataPanel**

Extend this class to add a custom metadata panel section to CUE content editors (storyline).

**cue.core.webcomponents.StoryFolderEditorMetadataPanel**

Extend this class to add a custom metadata panel section to CUE story folder editors.

**cue.core.webcomponents.ContentSummaryEditor**

Extend this class to add functionality to content summaries.

**Editor extensions**

There is currently only one such extension:

**cue.core.webcomponents.CustomEditorPanel**

Extend this class to add a custom editor panel to the storyline editor:

**Custom field editors**

There are two classes for creating custom field editors:

**cue.core.webcomponents.CustomFieldEditor**

Extend this class to change the appearance and behavior of a content item field (make an integer field in a content item be displayed as a graphical slider, for example).

**cue.core.webcomponents.CustomStoryElementEditor**

Extend this class to change the appearance and behavior of a story element field (make an integer field in a story element be displayed as a graphical slider, for example).

These classes and their use are described in more detail in the following sections.

The CUE base classes replace an earlier method of implementing CUE web components based on passing a **cueInterface** object from CUE to the web component. This mechanism is still available but is deprecated – the **cueInterface** object will be withdrawn in a future release. You should therefore use the new API described here when implementing new web components. If you need information about the old **cueInterface**-based API, please refer to the CUE 3.4 documentation.

**4.1.2.1 SidePanel**

**cue.core.webcomponents.SidePanel** can be used to display either a home page panel that occupies the main work area of the CUE home tab or a pop-out panel on the left side of a CUE editor window (similar to an editor **Search** panel).

It is defined as follows:

```
export abstract class SidePanel extends CUEElement
  implements webcomponent.Panel {
  public name: string; // Name of panel
  public homeScreen: boolean; // true: home page panel, false: editor side panel
  public active: boolean; // Active state of the panel

  // Function to be called whenever active state changes
  public abstract addActiveWatcher(fn: (active: boolean) => void): () => void;

  // Function to be called whenever active editor changes
  public abstract addActiveEditorWatcher(
```

```

    callback: (editor: Nullable<webcomponent.CUEElement>) => void
  ): () => void;
}

```

#### 4.1.2.1.1 SidePanel Example Configuration

```

sidePanels:
  - id: "twitter-home-panel"
    name: "Twitter Timelines"
    directive: "cue-custom-panel-loader"
    isAngular: true
    webComponent:
      modulePath: "webcomponents/twitter/twitter-home-panel.js"
      icon: "twitter-home-panel-icon"
    mimeTypes: []
    homeScreen: true
    metadata: []
    active: false
    order: 705

```

#### 4.1.2.1.2 SidePanel Example Implementation

```

window.twttr = (function(d, s, id) {
  var js, fjs = d.getElementsByTagName(s)[0],
      t = window.twttr || {};
  if (d.getElementById(id)) return t;
  js = d.createElement(s);
  js.id = id;
  js.src = "https://platform.twitter.com/widgets.js";
  fjs.parentNode.insertBefore(js, fjs);

  t._e = [];
  t.ready = function(f) {
    t._e.push(f);
  };

  return t;
})(document, "script", "twitter-wjs");
/**
 * Twitter Timeline
 */
class TwitterTimeline extends cue.core.webcomponents.SidePanel {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host { margin: 0 20px 0 20px; padding: 0; width: 100%; display: block; }
      </style>
      <h1>Twitter Timelines</h1>
      <div id="timeline"></div>
    `;
  }

  connectedCallback() {
    twttr.ready() => {
      twttr.widgets.load();
      twttr.widgets.createTimeline(
        {

```

```

        sourceType: 'profile',
        screenName: 'escenic'
    },
    this.shadowRoot.querySelector('#timeline'),
    {
        height: 1000
    }
    );
});
}
}

customElements.define('twitter-home-panel', TwitterTimeline);

/**
 * Twitter icon
 */
class TwitterIcon extends cue.core.webcomponents.SidePanel {
    constructor() {
        super();

        this.attachShadow({ mode: 'open' });
        this.shadowRoot.innerHTML = `
            <style>
                :host { margin: 0 0px 0 0px; width: 26px; display: inline; float: left;
margin-right: 18px; }
                img { width: 20px; position: relative; top: 4px; left: 10px; }
            </style>
            <img class="icon">
        `;

        this.activeIconPath = 'twitter-home-panel-icon-active.png';
        this.inactiveIconPath = 'twitter-home-panel-icon.png';
    }

    connectedCallback() {
        this.activeStateChanged(this.active);
        this.addActiveWatcher(active => {
            this.activeStateChanged(active);
        });
    }

    activeStateChanged(active) {
        let img = this.shadowRoot.querySelector('img.icon');
        if (active) {
            img.src = this.getAbsolutePath(this.activeIconPath);
        }
        else {
            img.src = this.getAbsolutePath(this.inactiveIconPath);
        }
    }

    getAbsolutePath(path) {
        const baseURI = import.meta.url;
        return baseURI.substring(0, baseURI.lastIndexOf('/') + 1) + path;
    }
}

customElements.define('twitter-home-panel-icon', TwitterIcon);

```

#### 4.1.2.2 ListEditor

`cue.core.webcomponents.ListEditor` can be used to display a pop-out panel on the left side of a CUE list editor window (similar to a **Search** panel).

It is defined as follows:

```
export abstract class SidePanel extends CUEElement
  implements webcomponent.Panel {
  public name: string; // Name of panel
  public homeScreen: boolean; // true: home page panel, false: editor side panel
  public active: boolean; // Active state of the panel

  // Function to be called whenever active state changes
  public abstract addActiveWatcher(fn: (active: boolean) => void): () => void;

  // Function to be called whenever active editor changes
  public abstract addActiveEditorWatcher(
    callback: (editor: Nullable<webcomponent.CUEElement>) => void
  ): () => void;
}

abstract class ListEditor extends cue.core.webcomponents.CUEElement {

  // Returns active list
  getList(): Nullable<List>;

  // Function to be called whenever active list changes
  addListWatcher(watcher: () => void): () => void;
}

interface List {
  link: Nullable<Link>;
  title: Nullable<string>;
  items: ListItem[];
  changelogURI: Nullable<uri.URI>;
  listPoolURI: Nullable<uri.URI>;
  section: Nullable<string>;
  publication: Nullable<Link>;
}

interface ListItem {
  about: uri.URI;
  handle: Nullable<uri.URI>;
  pinned: boolean;
  priority: number;
}
```

##### 4.1.2.2.1 ListEditor Example Configuration

```
sidePanels:
  - id: "list-info"
    name: "List Info"
    directive: "cue-custom-panel-loader"
    mimeTypes: [ "x-ece/list" ]
    homeScreen: false
    requires: ["escenic"]
    webComponent:
```

```

    modulePath: "webcomponents/list-info/list-info.js"
    icon: "list-info-icon"
    order: 709

```

#### 4.1.2.2.2 ListEditor Example Implementation

```

class ListInfo extends cue.core.webcomponents.ListEditor {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0;
          width: 100%
        }
        h1 {
          color: #9c9c9c;
          font-size: 24px;
          font-weight: 300;
        }
      </style>

      <h1>List Info</h1>
      <div id="list-info-wrapper"></div>
    `;

    this.blobUrl = undefined;
  }

  connectedCallback() {
    this.addListWatcher(() => this.showListInfo());
    this.showListInfo();
  }

  showListInfo() {
    const list = this.getList();
    const wrapper = this.shadowRoot.querySelector('#list-info-wrapper');
    wrapper.innerHTML = list
      ? `List Title: ${list.title}, Items: ${list.items.length}.`
      : '';
  }
}

customElements.define('list-info', ListInfo);

class ListInfoIcon extends cue.core.webcomponents.ListEditor {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          display: block;
        }
        .icon:before {

```



```

        font: 16px 'cf';
        font-style: normal;
        font-weight: normal;
        color: #444444;
        content: '\\e846';
        -webkit-font-smoothing: antialiased;
        -moz-osx-font-smoothing: grayscale;
    }
    .icon.active:before {
        color: #09ab00;
    }
</style>
<span class="icon"></span>
`;
}

connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher(active => {
        this.activeStateChanged(active);
    });
}

activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
        $(icon).addClass('active');
    } else {
        $(icon).removeClass('active');
    }
}
}
}
customElements.define('list-info-icon', ListInfoIcon);

```

#### 4.1.2.3 HomePageMetadataPanel

`cue.core.webcomponents.HomePageMetadataPanel` can be used to add a custom metadata panel section to the **Search**, **Recent**, **Dashboard** or **Archive** home page panel:

It is defined as follows:

```

export abstract class HomePageMetadataPanel extends CUEElement
    implements webcomponent.CollectionPanel<webcomponent.Content> {
    public name: string; // Name of the metadata panel
    public active: boolean; // Active state of the metadata panel

    // Function to be called whenever active state changes
    public abstract addActiveWatcher(fn: (active: boolean) => void): () => void;

    // Function to be called whenever selection in home page panel changes
    // Entries that are not accessible by the current user are passed as `null` in the
    array
    public abstract addFocusWatcher(
        fn: (entries: (webcomponent.Content | null)[]) => void
    ): () => void;

    // Returns currently selected contents
    public abstract getSelections(): Promise<webcomponent.Content[]>;

```

```
// Following functions return info about current content item
getArticleId: (content: webcomponent.Content) => Nullable<string>;
getArticleUri: (content: webcomponent.Content) => Nullable<string>;
getContentType: (content: webcomponent.Content) => Nullable<string>;
getState: (content: webcomponent.Content) => Nullable<ContentState>;
getPublishedDate: (content: webcomponent.Content) => Nullable<Date>;
}
```

This **PreviewPanel** class makes the **this.addFocusWatcher()** method available to the web component, so that it can provide a callback function that will be called whenever a new selection is made in the panel.

#### 4.1.2.3.1 HomePageMetadataPanel Configuration

The following properties must be defined to configure a home page metadata section based on **HomePageMetadataPanel**:

- **homePanels**

An array of directive names of the home screen panel on which the metadata should be made available. The following directive names may be specified:

<b>cue-search-sidepanel</b>	CUE home screen <b>Search</b> panel
<b>cue-latest-opened-sidepanel</b>	CUE home screen <b>Recent</b> panel
<b>cue-dashboard-sidepanel</b>	CUE home screen <b>Dashboard</b> panel
<b>cue-lists-sidepanel</b>	CUE home screen <b>Lists</b> panel

Remember also that the **homePanels** property name must be preceded by a hyphen (-).

**directive**

The tag name of the web component. The name you specify **must** contain a hyphen.

**name**

The display name of the component. The name is only actually displayed when the mouse is held over the metadata section button.

**webComponent**

Information about the web component:

**modulePath**

The URL of the web component

**icon**

The tag name of the web component's icon. The name you specify **must** contain a hyphen.

**order**

Determines the position of this section in the attributes panel (and the position of the button). The sections are arranged in numerical order from lowest to highest.

All the properties must be entered as a list item belonging to a **homeScreenMetadata** property. They must be indented correctly and the **homePanels** property must be preceded by a hyphen (-) to indicate the start of a new list item. The following example shows the required format:

```

homeScreenMetadata:
  - homePanels: ["cue-search-sidepanel", "cue-latest-opened-sidepanel"]
    directive: "content-preview"
    name: "Preview"
    webComponent:
      modulePath: "http://www.example.com/webcomponents/preview/preview.js"
      icon: "content-preview-icon"
    order: 804

```

#### 4.1.2.3.2 HomePageMetadataPanel Example

```

class PreviewPanel extends cue.core.webcomponents.HomePageMetadataPanel {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
padding: 0;
width: 100%
}
h1 {
color: #9c9c9c;
font-size: 24px;
font-weight: 300;
}
img, video {
max-width: 100%;
}
</style>

<h1>Preview</h1>
<div id="preview-wrapper"></div>
`;

    this.blobUrl = undefined;
  }

  connectedCallback() {
    this.addFocusWatcher(contents => {
      // Because a content can be `null` if the user doesn't have access to it
      if(contents[0]){
        this.focusedResultChanged(contents[0]);
      }
    });
  }

  focusedResultChanged(content) {
    if (this.blobUrl) {
      window.URL.revokeObjectURL(this.blobUrl);
    }
    let hide = true;
    if (content && content.mimeType) {
      hide = !_.includes(['x-ece/picture', 'x-ece/video'], content.mimeType);
    }
    this.info.hidden = hide;
    if (!hide) {
      this.showPreview(content.links['edit-media'].uri.toString(), content.mimeType);
    }
  }
}

```

```

    }
  }

  showPreview(binaryLink, mimeType) {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', binaryLink, true);
    xhr.setRequestHeader('Authorization', this.credentials.escenic);
    xhr.responseType = 'blob';
    xhr.onload = () => {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          this.blobUrl = window.URL.createObjectURL(xhr.response);
          this.updatePreview(mimeType);
        }
        else {
          console.error(xhr.statusText);
        }
      }
    };
    xhr.onerror = () => {
      console.error(xhr.statusText);
    };
    xhr.send(null);
  }

  updatePreview(mimeType) {
    const wrapper = this.shadowRoot.querySelector('#preview-wrapper');
    if (mimeType === 'x-ece/video') {
      wrapper.innerHTML = '<video controls preload="metadata" src="' + this.blobUrl +
''>';
    }
    else {
      wrapper.innerHTML = '';
    }
  }
}

customElements.define('content-preview', PreviewPanel);

/**
 * Creating the icon (if required)
 */
class PreviewIcon extends cue.core.webcomponents.HomePageMetadataPanel {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
display: block;
}
.icon:before {
font: 16px 'cf';
font-style: normal;
font-weight: normal;
color: #444444;
content: '\\e879';
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;

```

```

    }
    .icon.active:before {
    color: #09ab00;
    }
</style>
<span class="icon"></span>
`;
    }
    connectedCallback() {
        this.activeStateChanged(this.active);
        this.addActiveWatcher(active => {
            this.activeStateChanged(active);
        });
    }
    activeStateChanged(active) {
        const icon = this.shadowRoot.querySelector('.icon');
        if (active) {
            $(icon).addClass('active');
        }
        else {
            $(icon).removeClass('active');
        }
    }
}
customElements.define('content-preview-icon', PreviewIcon);

```

#### 4.1.2.4 AssignmentEditorMetadataPanel

`cue.core.webcomponents.AssignmentEditorMetadataPanel` can be used to add a custom metadata panel section to CUE assignment editors.

It is defined as follows:

```

export abstract class AssignmentEditorMetadataPanel extends CUEElement
    implements webcomponent.AssignmentEditor {

    // Returns the assignment object opened in the editor
    public abstract getAssignment(): webcomponent.Nullable<
        webcomponent.Assignment
    >;
    // Returns the story folder to which the assignment opened in the editor belongs
    public abstract getStory(): webcomponent.Nullable<webcomponent.PrintStory>;
}

```

##### 4.1.2.4.1 AssignmentEditorMetadataPanel Configuration

```

editors:
  metadata:
    - name: "Assignment Info"
      directive: "assignment-info"
      mimeTypes: [ "x-cci/assignment; type=picture" ]
      webComponent:
        modulePath: "webcomponents/{assignment-editor-metadata-panel-web-component}"
        icon: "assignment-info-icon"
      order: 735

```

## 4.1.2.4.2 AssignmentEditorMetadataPanel Example

```

class AssignmentInfo extends cue.core.webcomponents
  .AssignmentEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0;
          width: 100%
        }
        h1 {
          color: #9c9c9c;
          font-size: 24px;
          font-weight: 300;
        }
      </style>

      <h1>Assignment Info</h1>
      <div id="assignment-info-wrapper"></div>
      <div id="story-info-wrapper"></div>
    `;
  }

  connectedCallback() {
    this.showAssignmentInfo();
  }

  showAssignmentInfo() {
    const assignment = this.getAssignment();
    const printStory = this.getStory();
    const assignmentWrapper = this.shadowRoot.querySelector(
      '#assignment-info-wrapper'
    );
    const storyWrapper = this.shadowRoot.querySelector('#story-info-wrapper');
    assignmentWrapper.innerHTML = assignment
      ? `Assignment Name: ${assignment.name}, Items: ${
          assignment.items.filter(item => !item.empty).length
        }.`
      : '';
    storyWrapper.innerHTML = printStory
      ? `Story Name: ${printStory.name}.`
      : '';
  }
}

customElements.define('assignment-info', AssignmentInfo);

class AssignmentInfoIcon extends cue.core.webcomponents
  .AssignmentEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {

```

```

        margin: 0;
        display: block;
    }
    .icon:before {
        font: 16px 'cf';
        font-style: normal;
        font-weight: normal;
        color: #444444;
        content: '\\e846';
        -webkit-font-smoothing: antialiased;
        -moz-osx-font-smoothing: grayscale;
    }
    .icon.active:before {
        color: #09ab00;
    }
</style>
<span class="icon"></span>
`;
}

connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher(active => {
        this.activeStateChanged(active);
    });
}

activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
        $(icon).addClass('active');
    } else {
        $(icon).removeClass('active');
    }
}
}
customElements.define('assignment-info-icon', AssignmentInfoIcon);

```

#### 4.1.2.5 SectionsMetadataPanel

`cue.core.webcomponents.SectionsMetadataPanel` can be used to add a custom metadata panel section to the **Sections** home page panel.

It is defined as follows:

```

export abstract class SectionsMetadataPanel extends CUEElement
    implements webcomponent.SectionsPanel {
    public active: boolean; // Active state of the metadata panel

    // Function to be called whenever selection in Sections home page panel changes
    public abstract addFocusWatcher(
        fn: (section: webcomponent.Section) => void
    ): () => void;

    // Returns the sections that are currently selected in the section tree
    public abstract getSelections(): webcomponent.Section[];

    // Function to be called whenever active state changes
    public abstract addActiveWatcher(fn: (active: boolean) => void): () => void;
}

```

```

// Following functions return info about current section
getSectionUri: () => Nullable<string>;
getSectionId: () => Nullable<string>;
getSectionName: () => Nullable<string>;
}

```

#### 4.1.2.5.1 SectionsMetadataPanel Configuration

The following properties must be defined to configure a home page metadata section based on **SectionsMetadataPanel**:

##### - **homePanels**

An array of directive names of the home screen panel on which the metadata should be made available. The array may only contain one directive name in this case: **cue-sections-sidepanel**, specifying the home screen **Sections** panel.

Remember also that the **homePanels** property name must be preceded by a hyphen (-).

##### **directive**

The tag name of the web component. The name you specify **must** contain a hyphen.

##### **name**

The display name of the component. The name is only actually displayed when the mouse is held over the metadata section button.

##### **webComponent**

Information about the web component:

##### **modulePath**

The URL of the web component

##### **icon**

The tag name of the web component's icon. The name you specify **must** contain a hyphen.

##### **order**

Determines the position of this section in the attributes panel (and the position of the button). The sections are arranged in numerical order from lowest to highest.

All the properties must be entered as a list item belonging to a **homeScreenMetadata** property. They must be indented correctly and the **homePanels** property must be preceded by a hyphen (-) to indicate the start of a new list item. The following example shows the required format:

```

homeScreenMetadata:
- homePanels: ["cue-sections-sidepanel"]
  directive: "section-info"
  cssClass: "section-info"
  title: "General info" #translate
  name: "General Info"
  webComponent:
    modulePath: "webcomponents/section-info/section-info.js"
    icon: "section-info-icon"
  order: 702

```

#### 4.1.2.5.2 SectionsMetadataPanel Example

```

class SectionInfo extends cue.core.webcomponents.SectionsMetadataPanel {
  constructor() {

```



```
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
padding: 0;
width: 100%;
}
::selection {
background: rgba(9, 171, 0, 0.5);
color: white;
}
h1 {
display: inline-block;
line-height: 48px;
font-size: 14px;
font-weight: 600;
text-transform: uppercase;
color: #797878;
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
height: 38px;
margin-bottom: 0;
}
.property {
display: flex;
flex-direction: row;
flex-wrap: wrap;
margin-bottom: 20px;
}
.property .row {
display: flex;
flex-direction: row;
width: 100%;
font-weight: 300;
font-size: 14px;
}
.property .row .left {
flex-grow: 1;
width: 20%;
color: #9c9c9c;
}
.property .row .right {
flex-grow: 1;
width: 80%;
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
}
.selectable {
user-select: auto;
}
a {
color: #457dce;
text-decoration: none;
}
</style>
```

```

<h1>General info</h1>
<div class="property">
<div id="id" class="row" title="ID">
<div class="left">ID:</div>
<div class="right selectable"></div>
</div>
<div id="name" class="row" title="Name">
<div class="left">Name:</div>
<div class="right"></div>
</div>
<div id="uri" class="row" title="URI">
<div class="left">URI:</div>
<div class="right"></div>
</div>
<div class="property">
<div id="created" class="row" title="Created">
<div class="left">Created:</div>
<div class="right"></div>
</div>
<div id="modified" class="row" title="Modified">
<div class="left">Modified:</div>
<div class="right"></div>
</div>
<div id="published" class="row" title="Published">
<div class="left">Published:</div>
<div class="right"></div>
</div>
</div>
`;
}

connectedCallback() {
  this.addFocusWatcher(section => {
    this.focusedSectionChanged(section);
  });
}

focusedSectionChanged(section) {
  const addCredentials = xhr => {
    xhr.setRequestHeader('Authorization', this.credentials.esenic);
  };

  $.ajax({
    url: section.uri.toString(),
    type: 'GET',
    beforeSend: addCredentials
  })
  .done(document => {
    this.updateView(document);
  })
  .fail(error => {
    console.error(error);
  });
};

updateView(document) {
  const resolver = function (namespace) {
    switch (namespace) {

```

```

    case 'atom':
      return 'http://www.w3.org/2005/Atom';
    case 'app':
      return 'http://www.w3.org/2007/app';
    case 'dcterms':
      return 'http://purl.org/dc/terms/';
  }
};

const formatDate = date => {
  return moment(date, moment.ISO_8601, true).format('lll');
};

const created = document.evaluate('./atom:entry/dcterms:created', document,
resolver)
  .iterateNext().firstChild.nodeValue;
const modified = document.evaluate('./atom:entry/app:edited', document, resolver)
  .iterateNext().firstChild.nodeValue;
const publishedNode = document.evaluate('./atom:entry/atom:published', document,
resolver)
  .iterateNext();
const published = publishedNode ? formatDate(publishedNode.firstChild.nodeValue) :
'';

this.shadowRoot.querySelector('#id .right').innerHTML =
this.cueInterface.homePanel.getSectionId();
this.shadowRoot.querySelector('#name .right').innerHTML =
this.cueInterface.homePanel.getSectionName();
this.shadowRoot.querySelector('#uri .right').innerHTML = "<a
target='_blank' href='" + this.cueInterface.homePanel.getSectionUri() + "'> +
this.cueInterface.homePanel.getSectionUri() + "</a>";
this.shadowRoot.querySelector('#created .right').innerHTML = formatDate(created);
this.shadowRoot.querySelector('#modified .right').innerHTML =
formatDate(modified);
this.shadowRoot.querySelector('#published .right').innerHTML = published;
};
}
customElements.define('section-info', SectionInfo);

class SectionInfoIcon extends cue.core.webcomponents.SectionsMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
display: block;
}
.icon:before {
font: 16px 'cf';
font-style: normal;
font-weight: normal;
color: #444444;
content: '\\e8ad';
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: greyscale;
}
.icon.active:before {

```

```

color: #09ab00;
}
</style>
<span class="icon"></span>
`;
}

connectedCallback() {
  this.activeStateChanged(this.active);
  this.addActiveWatcher(active => {
    this.activeStateChanged(active);
  });
}

activeStateChanged(active) {
  const icon = this.shadowRoot.querySelector('.icon');
  if (active) {
    $(icon).addClass('active');
  }
  else {
    $(icon).removeClass('active');
  }
};
}
customElements.define('section-info-icon', SectionInfoIcon);

```

#### 4.1.2.6 SectionPageMetadataPanel

`cue.core.webcomponents.SectionPageMetadataPanel` can be used to add a custom metadata panel section to the CUE section page editor.

It is defined as follows:

```

export abstract class SectionPageMetadataPanel extends CUEElement
  implements webcomponent.SectionPageEditor {
  public name?: string; // Name of the metadata panel
  public active: boolean; // Active state of the metadata panel
  public mimeType: string; // MIME type of content being edited (always x-ece/section-
    page in this case)

  // Function to be called whenever selection in section page editor changes
  // Entries that are not accessible by the current user are passed as `null` in the
  array
  public abstract addFocusWatcher(
    fn: (entries: (webcomponent.Content | null)[]) => void
  ): () => void;

  // Function to be called whenever something is changed in the section page's owning
  section
  addSectionWatcher(
    watcher: () => void
  ): () => void;

  // Returns content items currently selected in the section page editor
  public abstract getSelections(): Promise<webcomponent.Content[]>;

  // Sets the value of the specified teaser field in the currently selected content
  item on the section page
  public abstract setFieldValue(fieldName: string, value: any): void;

```

```

// Returns the section page's owning section
getSection: () => Section;

// Returns the section page
getSectionPage: () => Promise<SectionPage>;
}

```

#### 4.1.2.6.1 SectionPageMetadataPanel Configuration

```

editors:
  metadata:
    - name: "Preview"
      directive: "section-page-content-preview"
      mimeTypes: ["x-ece/section-page"]
      webComponent:
        modulePath: "webcomponents/preview/preview.js"
        icon: "section-page-content-preview-icon"
      order: 730

```

#### 4.1.2.6.2 SectionPageMetadataPanel Example

```

class PreviewPanel extends cue.core.webcomponents.SectionPageMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0;
          width: 100%
        }
        h1 {
          color: #9c9c9c;
          font-size: 24px;
          font-weight: 300;
        }
        img, video {
          max-width: 100%;
        }
      </style>

      <h1>Preview</h1>
      <div id="preview-wrapper"></div>
    `;

    this.blobUrl = undefined;
  }

  connectedCallback() {
    this.addFocusWatcher(contents => {
      // Because a content can be `null` if the user doesn't have access to it
      if(contents[0]){
        this.focusedResultChanged(contents[0]);
      }
    });
  }
}

```

```

focusedResultChanged(content) {
  if (this.blobUrl) {
    window.URL.revokeObjectURL(this.blobUrl);
  }
  let hide = true;
  if (content && content.mimeType) {
    hide = !_.includes(['x-ece/picture', 'x-ece/video'], content.mimeType);
  }
  this.info.hidden = hide;
  if (!hide) {
    this.showPreview(content.links['edit-media'].uri.toString(), content.mimeType);
  }
}

showPreview(binaryLink, mimeType) {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', binaryLink, true);
  xhr.setRequestHeader('Authorization', this.credentials.escenic);
  xhr.responseType = 'blob';
  xhr.onload = () => {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        this.blobUrl = window.URL.createObjectURL(xhr.response);
        this.updatePreview(mimeType);
      }
      else {
        console.error(xhr.statusText);
      }
    }
  };

  xhr.onerror = () => {
    console.error(xhr.statusText);
  };

  xhr.send(null);
}

updatePreview(mimeType) {
  const wrapper = this.shadowRoot.querySelector('#preview-wrapper');
  if (mimeType === 'x-ece/video') {
    wrapper.innerHTML = '<video controls preload="metadata" src="' + this.blobUrl +
'>';
  }
  else {
    wrapper.innerHTML = '</span>
    `;
  }

  connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher(active => {
      this.activeStateChanged(active);
    });
  }

  activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
      $(icon).addClass('active');
    }
    else {
      $(icon).removeClass('active');
    }
  }
}

customElements.define('section-page-content-preview-icon', PreviewIcon);

```

#### 4.1.2.7 TextEditorMetadataPanel

`cue.core.webcomponents.TextEditorMetadataPanel` can be used to add a custom metadata panel section to CUE content editors (rich text). Note that `TextEditorMetadataPanel` inherits from `StorylineEditorMetadataPanel`, allowing you to use it to create web components that will work for both storylines and classic content items. The `isStoryline()` method can be used to determine what kind of content item is currently loaded, allowing you modify the component's behavior as required.

It is defined as follows:

```

export abstract class TextEditorMetadataPanel extends StorylineEditorMetadataPanel
  implements webcomponent.TextEditor, webcomponent.Panel {
  public name: string; // Name of the metadata panel section
  public mimeType: string; // MIME type of content being edited
  public active: boolean; // Active state of the metadata panel section

```

```

public selection: webcomponent.EditorSelection; // Current text selection in editor

// Returns true if the content item in the editor is a storyline
// in which case you can use the functions inherited from
StorylineEditorMetadataPanel
public abstract isStoryline(): boolean;

// Sends the specified trigger to the specified enrichment service
public abstract triggerService(
  triggerName: string,
  serviceName: string
): void;

// Sets the value of the specified field in the content editor.
// Will throw error if CUE fails to set the value
public abstract setFieldValue(fieldName: string, value: any): void;

// Function to be called whenever some content is changed in the editor
public abstract addContentWatcher(
  watcher: (content: webcomponent.Content) => void
): () => void;

// Gets the content being edited
public abstract getContent(): Promise<webcomponent.Content>;

// Returns the xml representation of the content being edited
public abstract getContentXML(): Promise<string | undefined>;

// Returns current container object
getContainer: () => webcomponent.Container;

// Returns an up-to-date preview URL for the current content item
public abstract getPreviewURL(): Promise<string | undefined>;

// Sets current container slug
setContainerSlug: (slug: string) => void;

// Returns whether container slug is editable
isContainerSlugEditable: () => boolean;

// Returns references to all the content items in which this
// content item appears as a relation or inline relation
getContentUsages: () => Promise<webcomponent.Content[]>;

// Returns the Content Store id of the content item being edited
getArticleId(): Nullable<string>;

// Returns the preview URI of the content item being edited
getArticleUri(): Nullable<string>;

// Returns the content type of the content item being edited
getContentType(): Nullable<string>;

// Returns the state of the content item being edited
getState(): Nullable<webcomponent.ContentState>;

// Returns the published date of the content item being edited
getPublishedDate(): Nullable<Date>;

// Returns the tags assigned to the content item being edited

```



```
    getTags(): Tag[];  
}
```

The **TextEditorMetadataPanel** class provides read/write access to the current text selection in the editor via its **selection** property:

```
interface EditorSelection {  
    getCurrentSelection: () => Selection | undefined;  
    replaceSelection: (newContent: string, selection: Selection) => void;  
    forEachBlockInSelection: (  
        range: Range,  
        forEachBlock: (element: Element) => void  
    ) => void;  
    replaceElement: (  
        element: Element,  
        elementName: string,  
        className: string,  
        text?: string  
    ) => void;  
    replaceBlockElement: (  
        element: Element,  
        elementName: string,  
        className: string  
    ) => void;  
    addSelectionWatcher: (  
        watcher: (selection: Selection | undefined) => void  
    ) => void;  
}
```

### 4.1.2.7.1 TextEditorMetadataPanel Configuration

The following properties must be defined to configure an editor metadata section based on **TextEditorMetadataPanel**:

- **name**

The name of the web component, preceded by a hyphen (-). By convention it is usually the same as the web component's **tagName**, but does not have to be.

**tagName**

The tag name of the web component. The name you specify here must contain a hyphen and must be the name specified with **customElements.define()** in the web component definition.

**modulePath**

The URL of the web component

**attributes**

Information about the web component:

**title**

The display name of the component. The name is only actually displayed when the mouse is held over the metadata section button.

**icon**

The tag name of the web component's icon. The name you specify **must** contain a hyphen.

All the properties must be entered as a list item belonging to a **customComponents** property. They must be indented correctly and the **name** property must be preceded by a hyphen (-) to indicate the start of a new list item. The following example shows the required format:

```
customComponents
  - name: "content-history"
    tagName: "content-history"
    modulePath: "webcomponents/history/history.js"
    attributes:
      title: "Content History"
      icon: "content-history-icon"
```

In order for a metadata section defined in this way to actually appear in CUE, you also need to add a configuration to one or more **content-type** resources in the Content Store. For further information about this, see [section 2.2.7](#).

#### 4.1.2.7.2 TextEditorMetadataPanel Example

```
class TextModification extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host { width: 100%; display: block; } /* Styles the web component tag */
        h1 {
          color: #9c9c9c;
          font-size: 24px;
          font-weight: 300;
        }
        button {
          display: block;
          font-size: 16px;
          font-family: "Hind", Helvetica Neue, Helvetica, Arial, Sans-serif;
          line-height: 32px;
          height: 32px;
          text-align: center;
          border: none;
          border-radius: 3px;
          background-color: #d3d3d3;
          color: #444444;
          cursor: pointer;
          padding: 0 10px;
          margin-bottom: 10px;
        }
        button:disabled {
          background-color: #efefef;
          color: #999999;
          pointer-events: none;
        }
        button:hover {
          background-color: #efefef;
        }
      </style>

      <h1>Text Modification</h1>
      <button class="character-tag">Insert Character Tag</button>
      <button class="macro-tag">Insert Macro Tag</button>
```

```

    <button class="em-dash">Insert Em dash</button>
    <button class="queen">Insert #</button>
  `;
}

connectedCallback() {
  if (this.selection) {
    this.addButtonEventListeners();
    this.currentSelection = this.selection.getCurrentSelection();
    this.selection.addSelectionWatcher(newSelection => {
      this.currentSelection = newSelection;
      this.setButtonStates(!newSelection);
    });
  }
  this.setButtonStates(!this.currentSelection);
}

addButtonEventListeners() {
  this.addCharacterTagEventListener();
  this.addMacroTagEventListener();
  this.addEmDashEventListener();
  this.addQueenEventListener();
}

addCharacterTagEventListener() {
  const button = this.shadowRoot.querySelector('.character-tag');
  $(button).on('click', () => {
    if (this.currentSelection.rangeCount) {
      this.selection.forEachBlockInSelection(this.currentSelection.getRangeAt(0),
element => {
        this.selection.replaceElement(element, 'span', 'quote_attr');
      });
    }
  });
}

addMacroTagEventListener() {
  const button = this.shadowRoot.querySelector('.macro-tag');
  $(button).on('click', () => {
    this.selection.replaceSelection('<span class="cci-codes">&lt;extra_leading&gt;</span>', this.currentSelection);
  });
}

addEmDashEventListener() {
  const button = this.shadowRoot.querySelector('.em-dash');
  $(button).on('click', () => {
    this.selection.replaceSelection('-', this.currentSelection);
  });
}

addQueenEventListener() {
  const button = this.shadowRoot.querySelector('.queen');
  $(button).on('click', () => {
    this.selection.replaceSelection('#', this.currentSelection);
  });
}

setButtonStates(enabled) {

```

```

    const buttonSelectors = ['.character-tag', '.macro-tag', '.em-dash', '.queen'];

    _._forEach(buttonSelectors, selector => {
      const button = this.shadowRoot.querySelector(selector);
      $(button).prop('disabled', !enabled);
    });
  }
}
customElements.define('text-modification', TextModification);

class TextModificationIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host { margin: 0; padding: 2px; display: block; } /* Styles the web
component icon tag */
        .icon:before {
          font: 16px 'cf';
          font-style: normal;
          font-weight: normal;
          color: #444444;
          content: '\\e8a6';
          -webkit-font-smoothing: antialiased;
          -moz-osx-font-smoothing: grayscale;
        }
        .icon.active:before {
          color: #09ab00;
        }
      </style>

      <span class="icon"></span>
    `;
  }

  connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher(active => {
      this.activeStateChanged(active);
    });
  }

  activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
      $(icon).addClass('active');
    }
    else {
      $(icon).removeClass('active');
    }
  }
}
customElements.define('text-modification-icon', TextModificationIcon);

```

#### 4.1.2.7.3 TextEditorMetadataPanel / Enrichment Service Example

This example shows how a `TextEditorMetadataPanel` web component can be used to invoke an enrichment service. The `TextEditorMetadataPanel` configuration looks like this:

```
editors:
  metadata:
    - name: "Enrichment service"
      directive: "enrichment-service"
      mimeTypes: ["x-ece/story"]
      webComponent:
        modulePath: "webcomponents/enrichment-service.js"
        icon: "enrichment-service-icon"
      order: 731
```

In this case, an enrichment service configuration is also required:

```
enrichmentServices:
  - name: "Text plain service"
    href: "http://localhost:8082/textPlainService"
    title: "Text plain service"
    triggers:
      - name: "on-click"
        properties: {}
  - name: "VDF payload service"
    href: "http://localhost:8082/vdfPayloadService/payload"
    title: "VDF payload service"
    triggers:
      - name: "on-click"
        properties: {}
```

The web component implementation looks like this:

```
class EnrichmentService extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host {
  margin: 0;
  padding: 0;
  width: 100%;
}
::selection {
  background: rgba(9, 171, 0, 0.5);
  color: white;
}
h1 {
  color: #9c9c9c;
  font-size: 24px;
  font-weight: 300;
}
.property {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  margin-bottom: 20px;
}
`
```

```

    .property .row {
      display: flex;
      flex-direction: row;
      width: 100%;
      font-weight: 300;
      font-size: 14px;
    }
    .property .row .left {
      flex-grow: 1;
      width: 30%;
      color: #9c9c9c;
    }
    .property .row .right {
      flex-grow: 1;
      width: 70%;
      white-space: nowrap;
      overflow: hidden;
      text-overflow: ellipsis;
    }
  </style>

  <h1>Enrichment service</h1>
  <div class="property">
    <div>
      <button class="text-plain">Invoke Service (text/plain)</button>
    </div>
    <hr/>
    <div>
      <button class="vdf-payload">Invoke Service (vdf)</button>
    </div>
  </div>
  `;
  };

  connectedCallback() {
    const textPlain = this.shadowRoot.querySelector('.text-plain');
    $(textPlain).on('click', () => {
      this.triggerService('on-click', 'Text plain service');
    });
    const vdfPayload = this.shadowRoot.querySelector('.vdf-payload');
    $(vdfPayload).on('click', () => {
      this.triggerService('on-click', 'VDF payload service');
    });
  }
}

customElements.define('enrichment-service', EnrichmentService);

class EnrichmentServiceIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
  <style>
    :host {
      margin: 0;
      display: block;
    }
    .icon:before {

```

```

        font: 16px 'cf';
        font-style: normal;
        font-weight: normal;
        color: #444444;
        content: '\\e0f3';
        -webkit-font-smoothing: antialiased;
        -moz-osx-font-smoothing: greyscale;
    }
    .icon.active:before {
        color: #09ab00;
    }
</style>
<span class="icon"></span>
`;
}

connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher((active) => {
        this.activeStateChanged(active);
    });
}

activeStateChanged(active) {
    let icon = this.shadowRoot.querySelector('.icon');
    if (active) {
        $(icon).addClass('active');
    }
    else {
        $(icon).removeClass('active');
    }
}
}

customElements.define('enrichment-service-icon', EnrichmentServiceIcon);

```

For detailed information about enrichment services, see [section 4.2](#).

#### 4.1.2.7.4 TextEditorMetadataPanel / History Example

This example shows how to add a "history" metadata section using **TextEditorMetadataPanel**. The configuration looks like this:

```

customComponents
- name: "content-history"
  tagName: "content-history"
  modulePath: "webcomponents/history/history.js"
  attributes:
    title: "Content History"
    icon: "content-history-icon"

```

The web component implementation looks like this:

```

class HistoryElement extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `

```

```

<style>
  :host {
    margin: 0;
    padding: 0;
    width: 100%
  }
  h1 {
    color: #9c9c9c;
    font-size: 24px;
    font-weight: 300;
  }
  .entry {
    width: 100%;
    display: flex;
    flex-direction: row;
  }
  .state {
    width: 25%
  }
  .date {
    width: 42%;
  }
  .author {
    width: 33%;
  }
</style>

<h1>History</h1>
<div class="entries"></div>
`;
}

connectedCallback() {
  this.fetchHistory();
  this.addContentWatcher(() => {
    this.fetchHistory();
  });
}

fetchHistory() {
  this.getContent().then(content => {
    const historyLinks = content.links['http://www.vizrt.com/types/relation/log'];
    if (historyLinks && historyLinks.length > 0) {
      const historyLink = historyLinks[0];
      $.ajax({
        url: historyLink.uri.toString(),
        type: 'GET',
        accept: historyLink.mimeType.format(),
        beforeSend: xhr => {
          xhr.setRequestHeader('Authorization', this.credentials.escenic);
        },
        success: result => {
          this.displayHistory(result);
        },
        error: (request, error) => {
          console.error(error);
        }
      });
    }
  });
}
});

```



```

    }

    displayHistory(document) {
      const entriesDiv = this.shadowRoot.querySelector('.entries');
      const parsedEntries = this.parseDocument(document);
      entriesDiv.innerHTML = '';
      parsedEntries.forEach(entry => {
        entriesDiv.innerHTML += '<div class="entry">' +
          '<span class="state">' + entry.state + '</span>' +
          '<span class="date">' + entry.updated.format('lll') + '</span>' +
          '<span class="author">' + entry.author + '</span>' +
          '</div>';
      });
    }

    parseDocument(document) {
      const resolver = function (namespace) {
        switch (namespace) {
          case 'app':
            return 'http://www.w3.org/2007/app';
          case 'atom':
            return 'http://www.w3.org/2005/Atom';
          case 'vaext':
            return 'http://www.vizrt.com/atom-ext';
        }
      };

      const entries = document.evaluate('//atom:entry', document, resolver);
      let entry = entries.iterateNext();
      let parsedEntries = [];
      while (entry) {
        const updated = document.evaluate('./atom:updated', entry,
        resolver).iterateNext().firstChild.nodeValue;
        const state = document.evaluate('./app:control/vaext:state', entry,
        resolver).iterateNext().attributes.getNamedItem('name').value;
        const authorNode = document.evaluate('./atom:author/atom:name', entry,
        resolver).iterateNext().firstChild;
        const author = authorNode ? authorNode.nodeValue : '';
        parsedEntries.push({
          updated: moment(updated, moment.ISO_8601, true),
          state: state,
          author: author
        });
        entry = entries.iterateNext();
      }
      return parsedEntries;
    }
  }

  customElements.define('content-history', HistoryElement);

  /**
   * Creating the icon (if required)
   */
  class HistoryIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
    constructor() {
      super();

      this.attachShadow({ mode: 'open' });
      this.shadowRoot.innerHTML = `
        <style>

```

```

        :host {
            margin: 0;
            display: block;
        }
        .icon:before {
            font: 16px 'cf';
            font-style: normal;
            font-weight: normal;
            color: #444444;
            content: '\\e8b9';
            -webkit-font-smoothing: antialiased;
            -moz-osx-font-smoothing: grayscale;
        }
        .icon.active:before {
            color: #09ab00;
        }
    </style>
    <span class="icon"></span>
`;
}

connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher(active => {
        this.activeStateChanged(active);
    });
}

activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
        $(icon).addClass('active');
    }
    else {
        $(icon).removeClass('active');
    }
}
}
}
customElements.define('content-history-icon', HistoryIcon);

```

#### 4.1.2.7.5 TextEditorMetadataPanel / Content XML Example

This example shows how to add a "Content XML" metadata section using **TextEditorMetadataPanel**. The configuration looks like this:

```

customComponents
- name: "content-xml"
  tagName: "content-xml"
  modulePath: "webcomponents/content-xml/content-xml.js"
  attributes:
    title: "Content XML"
    icon: "content-xml-icon"

```

The web component implementation looks like this:

```

const script = document.createElement('script');
script.src = '//cdn.jsdelivr.net/gh/highlightjs/cdn-release@9.18.0/build/highlight.min.js';

```

```

document.head.appendChild(script);

class ContentXml extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
<style>
@import url('https://cdn.jsdelivr.net/gh/highlightjs/cdn-release@9.18.0/build/styles/default.min.css');
:host {
margin: 0;
padding: 0;
width: 100%;
}
::selection {
background: rgba(9, 171, 0, 0.5);
color: white;
}
h1 {
color: #9c9c9c;
font-size: 24px;
font-weight: 300;
}
</style>

<h1>Content XML</h1>
<pre><code class="xml"></code></pre>
<button class="content-xml">Content XML</button>
`;
  }

  async connectedCallback() {
    this.addButtonEventListener();
  }

  addButtonEventListener() {
    $(this.shadowRoot.querySelector('.content-xml')).on('click', () => this.prettyXML());
  }

  async prettyXML() {
    const xmlElement = this.shadowRoot.querySelector('.xml');
    const sourceXML = await this.getContentXML();
    const html = this.htmlEscape(this.prettifyXml(sourceXML));
    $(xmlElement).html(html);
    hljs.highlightBlock(xmlElement);
  }

  prettifyXml(sourceXml) {
    const xmlDoc = new DOMParser().parseFromString(sourceXml, 'application/xml');
    const xsltDoc = new DOMParser().parseFromString([
    '<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">',
    '  <xsl:strip-space elements="*" />',
    '  <xsl:template match="para[content-style][not(text())]">',
    '    <xsl:value-of select="normalize-space(.)" />',
    '  </xsl:template>',
    '  <xsl:template match="node()|@*">',
    '    <xsl:copy><xsl:apply-templates select="node()|@*" /></xsl:copy>',
    ]
  
```

```

' </xsl:template>',
' <xsl:output indent="yes"/>',
'</xsl:stylesheet>',
].join('\n'), 'application/xml');

const xsltProcessor = new XSLTProcessor();
xsltProcessor.importStylesheet(xsltDoc);
const resultDoc = xsltProcessor.transformToDocument(xmlDoc);
return new XMLSerializer().serializeToString(resultDoc);
}

htmlEscape(s) {
return s
.replace(/&/g, '&amp;');
.replace(/</g, '&lt;');
.replace(/>/g, '&gt;');
}
}

customElements.define('content-xml', ContentXml);

class ContentXmlIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
constructor() {
super();

this.attachShadow({mode: 'open'});
this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
display: block;
}
.icon:before {
font: 16px 'cf';
font-style: normal;
font-weight: normal;
color: #444444;
content: '<x>';
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: greyscale;
}
.icon.active:before {
color: #09ab00;
}
</style>
<span class="icon"></span>
`;
}

connectedCallback() {
this.activeStateChanged(this.active);
this.addActiveWatcher(function (active) {
this.activeStateChanged(active);
}).bind(this);
}

activeStateChanged(active) {
var icon = this.shadowRoot.querySelector('.icon');
if (active) {
$(icon).addClass('active');
}
}
}

```

```

    } else {
      $(icon).removeClass('active');
    }
  }
}

customElements.define('content-xml-icon', ContentXmlIcon);

```

#### 4.1.2.7.6 TextEditorMetadataPanel / Container slug Example

This example shows how to add a "Container Slug" metadata section using **TextEditorMetadataPanel** to modify a container slug. The configuration looks like this:

```

customComponents
- name: "slug-modification"
  tagName: "slug-modification"
  modulePath: "webcomponents/slug-modification/slug-modification.js"
  attributes:
    title: "Slug Modification"
    icon: "slug-modification-icon"

```

The web component implementation looks like this:

```

class SlugModification extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host { width: 100%; display: block; }
h1 {
color: #9c9c9c;
font-size: 24px;
font-weight: 300;
}
.slug-text {
padding: 10px 5px;
}
.slug-text:hover {
border: 1px solid grey;
}
.slug-input {
width: 100%;
}
</style>

<h1>Slug Modification</h1>
<div class="slug-text" title="Click to Edit"></div>
<input type="text" class="slug-input" hidden>
`;
  }

  connectedCallback() {
    this.loadSlug();
    this.addSlugEventListener();
    this.addContentWatcher(() => {

```

```

this.loadSlug()
});
}

loadSlug() {
const container = this.getContainer();
if(container.slug) {
const slugText = this.shadowRoot.querySelector('.slug-text');
$(slugText).html(container.slug);
}
}

addSlugEventListener() {
const slugInput = this.shadowRoot.querySelector('.slug-input');
const slugText = this.shadowRoot.querySelector('.slug-text');

$(slugText).on('click', () => {
if(this.isContainerSlugEditable()) {
$(slugInput).css('display', 'block');
const container = this.getContainer();
$(slugInput).val(container.slug);
$(slugText).css('display', 'none');
}
});

$(slugInput).on('keyup', e => {
if (e.keyCode === 13) {
this.setContainerSlug($(slugInput).val());
$(slugInput).css('display', 'none');
$(slugText).html($(slugInput).val());
$(slugText).css('display', 'block');
}
});
}
}

customElements.define('slug-modification', SlugModification);

class SlugModificationIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
constructor() {
super();

this.attachShadow({ mode: 'open' });
this.shadowRoot.innerHTML = `
<style>
:host { margin: 0; padding: 2px; display: block; } /* Styles the web component icon
tag */
.icon:before {
font: 16px 'cf';
font-style: normal;
font-weight: normal;
color: #444444;
content: 'Sl';
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;
}
.icon.active:before {
color: #09ab00;
}
</style>

```

```

<span class="icon"></span>
`;
}

connectedCallback() {
  this.activeStateChanged(this.active);
  this.addActiveWatcher(active => {
    this.activeStateChanged(active);
  });
}

activeStateChanged(active) {
  const icon = this.shadowRoot.querySelector('.icon');
  if (active) {
    $(icon).addClass('active');
  }
  else {
    $(icon).removeClass('active');
  }
}

customElements.define('slug-modification-icon', SlugModificationIcon);

```

#### 4.1.2.7.7 TextEditorMetadataPanel / Field editor Example

This example shows how to add a "Field editor" metadata section using **TextEditorMetadataPanel** to modify a field. The configuration looks like this:

```

customComponents
- name: "field-editor"
  tagName: "field-editor"
  modulePath: "webcomponents/field-editor/field-editor.js"
  attributes:
    title: "Field editor"
    icon: "field-editor-icon"

```

The web component implementation looks like this:

```

class FieldEditor extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host { width: 100%; display: block; }
h1 {
color: #9c9c9c;
font-size: 24px;
font-weight: 300;
}
.slug-text {
padding: 10px 5px;
}
.slug-text:hover {
border: 1px solid grey;

```

```

    }
    .slug-input {
      width: 100%;
    }
  </style>

  <h1>Field Editor</h1>
  <input type="text" class="title-field" placeholder="Title">
  `;
}

async connectedCallback() {
  this.addContentWatcher(content => {
    this.setViewValue(content.values['title']);
  });
  const content = await this.getContent();
  this.setViewValue(content.values['title']);

  this.addFieldChangeListener();
}

addFieldChangeListener() {
  const field = this.shadowRoot.querySelector('.title-field');
  $(field).on('change', () => {
    this.setValue('title', $(field).val());
  });
}

setViewValue(value) {
  const field = this.shadowRoot.querySelector('.title-field');
  $(field).val(value);
}

setValue(key, value) {
  try {
    this.setFieldValue(key, value);
  } catch (e) {
    console.error(`Failed to set ${key} value!`, e);
  }
}

customElements.define('field-editor', FieldEditor);

class FieldEditorIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
    <style>
    :host { margin: 0; padding: 2px; display: block; } /* Styles the web component icon
      tag */
    .icon:before {
      font: 16px 'cf';
      font-style: normal;
      font-weight: normal;
      color: #444444;
      content: 'E';
      -webkit-font-smoothing: antialiased;

```



```

-moz-osx-font-smoothing: grayscale;
}
.icon.active:before {
color: #09ab00;
}
</style>

<span class="icon"></span>
`;
}

connectedCallback() {
this.activeStateChanged(this.active);
this.addActiveWatcher(active => {
this.activeStateChanged(active);
});
}

activeStateChanged(active) {
const icon = this.shadowRoot.querySelector('.icon');
if (active) {
$(icon).addClass('active');
}
else {
$(icon).removeClass('active');
}
}
}
customElements.define('field-editor-icon', FieldEditorIcon);

```

### 4.1.2.7.8 TextEditorMetadataPanel / Usages Example

This example shows how to add a "Usages" metadata section using **TextEditorMetadataPanel** to return references to all the content items in which this content item appears as a relation or inline relation. The configuration looks like this:

```

customComponents
- name: "usages"
  tagName: "content-usages"
  modulePath: "webcomponents/usages.js"
  attributes:
    title: "Usages" #translate
    icon: "usages-icon"

```

The web component implementation looks like this:

```

class Usages extends cue.core.webcomponents.TextEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host { width: 100%; display: block; }
h1 {
color: #9c9c9c;

```

```

font-size: 24px;
font-weight: 300;
}
.slug-text {
padding: 10px 5px;
}
.slug-text:hover {
border: 1px solid grey;
}
.slug-input {
width: 100%;
}
</style>

<h1>Content Usages</h1>
<ul class="usages"></ul>
`;
}

async connectedCallback() {
const usages = await this.getContentUsages();
usages.forEach(usage => {
const ul = $(this.shadowRoot.querySelector('.usages'));
ul.append(`<li><a href="${usage.links['alternate']}" ?
usage.links['alternate'].uri.toString() : usage.links['self'].uri.toString()"><span>
${usage.values['title']}</a></span></li>`);
});
}
}
customElements.define('content-usages', Usages);

class UsagesIcon extends cue.core.webcomponents.TextEditorMetadataPanel {
constructor() {
super();

this.attachShadow({ mode: 'open' });
this.shadowRoot.innerHTML = `
<style>
:host { margin: 0; padding: 2px; display: block; } /* Styles the web component
icon tag */
.icon:before {
font: 16px 'cf';
font-style: normal;
font-weight: normal;
color: #444444;
content: 'U';
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;
}
.icon.active:before {
color: #09ab00;
}
</style>

<span class="icon"></span>
`;
}

connectedCallback() {
this.activeStateChanged(this.active);
}

```

```
    this.addActiveWatcher(active => {
      this.activeStateChanged(active);
    });
  }

  activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
      $(icon).addClass('active');
    }
    else {
      $(icon).removeClass('active');
    }
  }
}

customElements.define('content-usages-icon', UsagesIcon);
```

### 4.1.2.8 StorylineEditorMetadataPanel

`cue.core.webcomponents.StorylineEditorMetadataPanel` can be used to add a custom metadata panel section to CUE content editors (storyline).

It is defined as follows:

```
export abstract class StorylineEditorMetadataPanel extends CUEElement {

  // Returns the storyline being edited
  public storyline: webcomponent.Storyline;

  // Function to be called whenever the storyline changes
  public abstract addStorylineWatcher(
    watcher: (storyline: webcomponent.Storyline) => void
  ): () => void;

  // Function to be called whenever a new story element gets the focus in the editor
  public abstract addStoryElementFocusWatcher(
    watcher: (storyelement: webcomponent.StoryElement) => void
  ): () => void;

  // Function to be called whenever the text selection changes in the editor
  public abstract addTextSelectionWatcher(
    watcher: (selection: webcomponent.StorylineTextSelection) => void
  ): () => void;

  // Gets the story element currently in focus
  getFocusedStoryElement(): webcomponent.StoryElement | undefined;

  // Sets the value of the specified field in the specified story element in the
  editor
  public abstract setStoryElementFieldValue(
    storyElementId: string,
    fieldName: string,
    fieldValue: any
  ): void;

  // Sets the value of the story element in the editor
  public abstract updateStoryElementValue(
    storyElementId: string,
    fieldValue: any
```

```

    ): void;
  }

```

#### 4.1.2.8.1 StorylineEditorMetadataPanel Configuration

```

editors:
  metadata:
    - name: "storyline-stat"
      directive: "storyline-stat"
      cssClass: "storyline-stat"
      title: "Storyline Stat" #translate
      webComponent:
        modulePath: "webcomponents/storyline/storyline-stat.js"
        icon: "storyline-stat-icon"
      mimeTypes: ["x-ece/story", "x-ece/new-content; type=story"]
      order: 731

```

#### 4.1.2.8.2 StorylineEditorMetadataPanel Example

```

const titleShortcut = 'wct';
const dummyTitle = 'Cool Title from Storyline stat web component';
const bodyShortcut = 'wcbd';
const dummyBody = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor incididunt ut ' +
  'labore et dolore magna aliqua. Magna etiam tempor orci eu. Sed libero enim ' +
  'sed faucibus turpis in eu mi. Urna porttitor rhoncus dolor purus non enim ' +
  'praesent elementum. Magna fermentum iaculis eu non diam phasellus vestibulum
  lorem.'

class StorylineStatPanel extends cue.core.webcomponents.StorylineEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
padding: 0;
width: 100%;
}
::selection {
background: rgba(9, 171, 0, 0.5);
color: white;
}
h1 {
display: inline-block;
line-height: 48px;
font-size: 14px;
font-weight: 600;
text-transform: uppercase;
color: #797878;
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
height: 38px;
margin-bottom: 0;
}
h2 {

```

```
display: inline-block;
font-size: 13px;
font-weight: 600;
color: #797878;
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
margin-bottom: 0;
}
.property {
display: flex;
flex-direction: row;
flex-wrap: wrap;
margin-bottom: 20px;
}
.property .row, .elementId {
display: flex;
flex-direction: row;
width: 100%;
font-weight: 300;
font-size: 14px;
}
.property .row .left {
flex-grow: 1;
width: 80%;
color: #9c9c9c;
}
.property .row .right {
flex-grow: 1;
width: 20%;
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
}
.property .elementId .left {
flex-grow: 1;
width: 10%;
color: #9c9c9c;
}
.property .elementId .right {
flex-grow: 1;
width: 90%;
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
}
.property .row.total {
background: #efefef;
font-weight: bold;
}
</style>

<h1>Storyline Stats</h1>
<div class="property">
<div id="headline" class="row">
<div class="left">Number of headlines:</div>
<div class="right"></div>
</div>
<div id="paragraph" class="row">
<div class="left">Number of paragraph:</div>
```

```

<div class="right"></div>
</div>
<div id="lead_text" class="row">
<div class="left">Number of Lead Text:</div>
<div class="right"></div>
</div>
<div id="image" class="row">
<div class="left">Number of images:</div>
<div class="right"></div>
</div>
<div id="video" class="row">
<div class="left">Number of videos:</div>
<div class="right"></div>
</div>
<div id="embed" class="row">
<div class="left">Number of embeds:</div>
<div class="right"></div>
</div>
<div id="others" class="row">
<div class="left">Others Elements:</div>
<div class="right"></div>
</div>
<div id="total" class="row total">
<div class="left">Total Count:</div>
<div class="right"></div>
</div>
</div>
<h2>Focus Element Stats:</h2>
<div class="property">
<div id="id" class="elementId">
<div class="left">Id:</div>
<div class="right"></div>
</div>
<div id="general" class="row">
<div class="left">Number of general fields:</div>
<div class="right"></div>
</div>
<div id="settings" class="row">
<div class="left">Number of settings fields:</div>
<div class="right"></div>
</div>
<div id="chars" class="row">
<div class="left">Number of Chars:</div>
<div class="right"></div>
</div>
</div>

<h2>Selection Stats:</h2>
<div class="property">
<div id="selWords" class="row">
<div class="left">Number of words:</div>
<div class="right"></div>
</div>
<div id="selChars" class="row">
<div class="left">Number of Chars:</div>
<div class="right"></div>
</div>
<div id="selVowels" class="row">
<div class="left">Number of vowels:</div>
<div class="right"></div>

```

```

</div>
</div>
`;

}

connectedCallback() {
  this.updateView();
  this.addStorylineWatcher(storyline => {
    this.storyline = storyline;
    this.updateView();
    this.fillStoryElementTitleAndBody();
  });
  this.addStoryElementFocusWatcher(storyelement =>
this.updateViewOnStoryElementFocus(storyelement));
  this.addTextSelectionWatcher(selection =>
this.updateViewOnSelectionChange(selection));
}

updateView() {
  const totalCount = this.storyline.elements.length;
  const storyElements = this.getStoryElements();
  const getElementsByType = type => storyElements
.filter(element => element.model.name === type);

  const headlineCount = getElementsByType('headline').length;
  const paragraphCount = getElementsByType('paragraph').length;
  const leadTextCount = getElementsByType('lead_text').length;
  const imageCount = getElementsByType('image').length;
  const videoCount = getElementsByType('video').length;
  const embedCount = getElementsByType('embed').length;
  const others = totalCount -
    (headlineCount
+ paragraphCount
+ leadTextCount
+ imageCount
+ videoCount
+ embedCount);

  this.shadowRoot.querySelector('#headline .right').innerHTML = headlineCount;
  this.shadowRoot.querySelector('#paragraph .right').innerHTML = paragraphCount;
  this.shadowRoot.querySelector('#lead_text .right').innerHTML = leadTextCount;
  this.shadowRoot.querySelector('#image .right').innerHTML = imageCount;
  this.shadowRoot.querySelector('#video .right').innerHTML = videoCount;
  this.shadowRoot.querySelector('#embed .right').innerHTML = embedCount;
  this.shadowRoot.querySelector('#others .right').innerHTML = others;
  this.shadowRoot.querySelector('#total .right').innerHTML = totalCount;
}

updateViewOnStoryElementFocus(storyElement) {
  const generalFieldCount = storyElement.model.fields.filter(field =>
field.isVisible && !field.isSettings).length;
  const settingsFieldCount = storyElement.model.fields.filter(field =>
field.isSettings).length;
  const stringFieldValues = storyElement.model.fields
.filter(field => field.type === 'string')
  .map(field => storyElement.values[field.name])
  .filter(value => !!value);
  const charsCount = stringFieldValues.reduce((result, value) => result +
value.length, 0);

```

```

    this.shadowRoot.querySelector('#id .right').innerHTML = storyElement.id;
    this.shadowRoot.querySelector('#general .right').innerHTML = generalFieldCount;
    this.shadowRoot.querySelector('#settings .right').innerHTML = settingsFieldCount;
    this.shadowRoot.querySelector('#chars .right').innerHTML = charsCount;
  }

  updateViewOnSelectionChange(selection) {
    const selectedText = selection.selectedText;
    const charCount = selectedText.length;
    const vowelCount = (selectedText.match(/[aeiou]/gi) || '').length;
    const wordCount = selectedText.trim().split(/\s+/).length

    this.shadowRoot.querySelector('#selChars .right').innerHTML = charCount;
    this.shadowRoot.querySelector('#selVowels .right').innerHTML = vowelCount;
    this.shadowRoot.querySelector('#selWords .right').innerHTML = wordCount;
  }

  fillStoryElementTitleAndBody() {
    const storyElements = this.getStoryElements();
    const textElements = storyElements
    .filter(element => element.model.name === 'lead_text'
    || element.model.name === 'headline'
    );
    textElements.forEach(storyElement => {
      const fieldName = storyElement.model.fields[0].name;
      const fieldValue = storyElement.values[fieldName];
      if(fieldValue && (fieldValue.search(titleShortcut) > -1 ||
fieldValue.search(bodyShortcut) > -1 )) {
        const newValue = fieldValue.replace(titleShortcut, dummyTitle)
          .replace(bodyShortcut, dummyBody);
        this.updateStoryElementValue(storyElement.id, newValue);
      }
    });
  }

  getStoryElements() {
    return this.storyline.elements
      .map(elementId => this.storyline.storyElements.get(elementId));
  }
}

customElements.define('storyline-stat', StorylineStatPanel);

class StorylineStatIcon extends cue.core.webcomponents.MetadataPanelCUEElement {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host { margin: 0; padding: 2px; display: block; } /* Styles the web component icon
tag */
.icon:before {
font: 16px 'cf';
font-style: normal;
font-weight: normal;
color: #444444;
content: 'St';

```



```

-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;
}
.icon.active:before {
color: #09ab00;
}
</style>

<span class="icon"></span>
`;
}

connectedCallback() {
  this.activeStateChanged(this.active);
  this.addActiveWatcher(active => {
    this.activeStateChanged(active);
  });
}

activeStateChanged(active) {
  const icon = this.shadowRoot.querySelector('.icon');
  if (active) {
    $(icon).addClass('active');
  }
  else {
    $(icon).removeClass('active');
  }
}
}
customElements.define('storyline-stat-icon', StorylineStatIcon);

```

#### 4.1.2.9 StoryFolderEditorMetadataPanel

`cue.core.webcomponents.StorylineEditorMetadataPanel` can be used to add a custom metadata panel section to CUE story folder editors.

It is defined as follows:

```

export abstract class StoryFolderEditorMetadataPanel extends CUEElement {

  // Returns the storyfolder being edited
  public abstract getStoryFolder: () => webcomponent.Nullable<
    webcomponent.StoryFolder
  >;

  // Function to be called whenever the story folder content changes in the editor
  public abstract addStoryFolderWatcher(watcher: () => void): () => void;
}

```

##### 4.1.2.9.1 StoryFolderEditorMetadataPanel Configuration

```

editors:
  metadata:
    - name: "story-folder-info"
      directive: "story-folder-info"
      cssClass: "story-folder-info"
      title: "Story Folder Info" #translate
      webComponent:
        modulePath: "webcomponents/story-folder-info/story-folder-info.js"

```

```

    icon: "story-folder-info-icon"
    mimeTypes: [ "x-cci/storyfolder" ]
    order: 735

```

#### 4.1.2.9.2 StoryFolderEditorMetadataPanel Example

```

class StoryFolderInfo extends cue.core.webcomponents
  .StoryFolderEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0;
          width: 100%
        }
        h1 {
          color: #9c9c9c;
          font-size: 24px;
          font-weight: 300;
        }
      </style>

      <h1>Story Folder Info</h1>
      <div id="story-folder-info-wrapper"></div>
    `;
  }

  connectedCallback() {
    this.addStoryFolderWatcher(() => this.showStoryFolderInfo());
    this.showStoryFolderInfo();
  }

  showStoryFolderInfo() {
    const storyFolder = this.getStoryFolder();
    const wrapper = this.shadowRoot.querySelector('#story-folder-info-wrapper');
    wrapper.innerHTML = storyFolder
      ? `<div> Stories: ${storyFolder.stories.length}</div>
        <div> Assignments: ${storyFolder.assignments.length}</div>
        <div> Assets: ${storyFolder.contents.length}</div>
        <div> Packages: ${storyFolder.packages.length}</div>`
      : '';
  }
}

customElements.define('story-folder-info', StoryFolderInfo);

class StoryFolderInfoIcon extends cue.core.webcomponents
  .StoryFolderEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;

```

```

        display: block;
    }
    .icon:before {
        font: 16px 'cf';
        font-style: normal;
        font-weight: normal;
        color: #444444;
        content: '\\e846';
        -webkit-font-smoothing: antialiased;
        -moz-osx-font-smoothing: grayscale;
    }
    .icon.active:before {
        color: #09ab00;
    }
</style>
<span class="icon"></span>
`;
}

connectedCallback() {
    this.activeStateChanged(this.active);
    this.addActiveWatcher(active => {
        this.activeStateChanged(active);
    });
}

activeStateChanged(active) {
    const icon = this.shadowRoot.querySelector('.icon');
    if (active) {
        $(icon).addClass('active');
    } else {
        $(icon).removeClass('active');
    }
}
}
customElements.define('story-folder-info-icon', StoryFolderInfoIcon);

```

#### 4.1.2.10 CustomEditorPanel

**cue.core.webcomponents.CustomEditorPanel** can be used to add a custom editor panel to a storyline editor. Its purpose is to allow specific storyline types to be extended with custom functionality. You might, for example, extend a Facebook storyline type with a panel containing a preview of the post being created. The panel is displayed to the right of the storyline editor itself, between the editor and the metadata panel. Note that although **CustomEditorPanel** inherits from **TextEditorMetadataPanel**, it can only be used together with storyline editors, not with classic CUE editors.

It is defined as follows:

```

export abstract class CustomEditorPanel
    extends TextEditorMetadataPanel
    implements webcomponent.CustomEditor
{
}

```

## 4.1.2.10.1 CustomEditorPanel Configuration

The following properties must be defined to configure an editor panel based on **CustomEditorPanel**:

- **name**  
The name of the web component, preceded by a hyphen (-). By convention it is usually the same as the web component's **tagName**, but does not have to be.
- modulePath**  
The URL of the web component
- attributes**  
Any properties required by this web component

All the properties must be entered as a list item belonging to a **customComponents** property. They must be indented correctly and the **name** property must be preceded by a hyphen (-) to indicate the start of a new list item. The following example shows the required format:

```
customComponents
  - name: "custom-preview"
    modulePath: "webcomponents/preview/custom-preview.js"
    attributes:
      renderDelay: 2
```

The **renderDelay** attribute specified in the example above is a parameter required by the example **custom-preview** web component (see [section 4.1.2.10.2](#)).

In order for a custom editor panel defined in this way to actually appear in CUE, you also need to add a **ui:custom-editor** element inside one or more **storyline-template** resources in the Content Store. A custom editor panel is only displayed alongside a storyline editor if the edited storyline's template contains a **ui:custom-editor** element. The **ui:custom-editor** element specifies which custom editor panel is to be displayed, its minimum screen width requirement, and how much of the storyline editor's width it is allowed to occupy. The **ui:custom-editor** element must appear as a child of the storyline template's **elements** element:

```
<?xml version="1.0"?>
<storyline-template
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  name="strict">
  <elements>
  ...
  <ui:custom-editor
    webcomponent="custom-preview"
    width="45%"
    minresolution="960px"/>
  ...
  </elements>
</storyline-template>
```

For general information about storyline templates and how to edit them, see [Storyline Templates](#). For a detailed description of the **ui:custom-editor** element, see [custom-editor](#).

## 4.1.2.10.2 CustomEditorPanel Example

The following example provides a "live preview" window for storyline editors, that updates as users edit the storyline. The **renderDelay** attribute set in the configuration file specifies how long (in seconds) to wait after the user stops typing before rendering a new preview in the window.

```
class CustomPreviewPanel extends cue.core.webcomponents.CustomEditorPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        .preview-container {
          width: 100%;
          height: 100%;
        }
      </style>

      <div class="preview-container" id="preview"></div>
    `;
    this.frontBuffer = this.createBuffer();
    this.backBuffer = this.createBuffer();

    const previewDiv = this.shadowRoot.querySelector('#preview');
    previewDiv.append(this.frontBuffer, this.backBuffer);
  }

  async connectedCallback() {
    let typingDebounce; // Debounce timer for content updates
    let renderTimer; // Delay from typing debounce fires till we swap buffers
    let renderDelay = this.hasAttribute('renderDelay')
      ? this.getAttribute('renderDelay') * 1000
      : 1000;

    this.addContentWatcher(async () => {
      window.clearTimeout(typingDebounce);
      window.clearTimeout(renderTimer);
      typingDebounce = window.setTimeout(async () => {
        renderTimer = window.setTimeout(() => this.swapBuffers(), renderDelay);
        this.backBuffer.src = await this.getPreviewURL();
      }, 1000);
    });

    window.setTimeout(() => this.swapBuffers(), renderDelay);
    this.backBuffer.src = await this.getPreviewURL();
  }

  swapBuffers() {
    const tmp = this.backBuffer;
    this.backBuffer = this.frontBuffer;
    this.frontBuffer = tmp;
    this.backBuffer.style.display = 'none';
    this.frontBuffer.style.display = 'block';
  }

  createBuffer() {
    const buffer = document.createElement('iframe');
    buffer.style.width = '100%';
    buffer.style.height = '100%';
  }
}
```

```

        buffer.style.position = 'absolute';
        buffer.style.top = '0px';
        buffer.style.left = '0px';
        buffer.style.border = '1px solid #e6e6e6';
        buffer.style.display = 'none';
        return buffer;
    }
}
customElements.define('custom-preview', CustomPreviewPanel);

```

#### 4.1.2.11 ContentSummaryEditor

`cue.core.webcomponents.ContentSummaryEditor` can be used to extend the functionality of the content cards used to represent content items in the following contexts:

- The **Relations** metadata panel in a content editor, when a relation is expanded
- A section page metadata panel, when a teaser is selected

It is defined as follows:

```

export abstract class ContentSummaryEditor extends CUEElement {

    // Returns the content summary data as displayed
    public item: webcomponent.Content;

    // Returns the entire content summary
    public content: webcomponent.Content;

    // Returns the content editor tab URL for the content item represented by the
    // summary.
    public link: string;
}

```

##### 4.1.2.11.1 ContentSummaryEditor Configuration

```

customComponents:
  - name: "my-additional-editor"
    modulePath: "webcomponents/additional-editor/my-additional-editor.js"

```

##### 4.1.2.11.2 ContentSummaryEditor Example

```

class MyAdditionalEditor extends cue.core.webcomponents.ContentSummaryEditor {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0;
          width: 100%;
          display: block;
        }
        .wc-additional-editor-container {
          padding: 10px;
          height: 100px;
          overflow: hidden;
        }
      </style>
    `;
  }
}

```

```

.wc-additional-editor-group {
  width: 100%;
  height: auto;
  margin-bottom: 10px;
  font-family: "Hind", Helvetica Neue, Helvetica, Arial, sans-serif;
  color: #797878;
  font-size: 16px;
}
.wc-additional-editor-item-name {
  display: block;
  font-size: 12px;
}
.wc-additional-editor-item-content {
  display: block;
  font-size: 16px;
}
a.wc-additional-editor-link:link, a.wc-additional-editor-link:visited {
  text-decoration: none;
  color: #457dce;
}
a.wc-additional-editor-link:hover {
  text-decoration: underline;
}
</style>
<div class="wc-additional-editor-container">
  <div class="wc-additional-editor-group">
    <span class="wc-additional-editor-item-name">Title:</span>
    <span class="wc-additional-editor-item-content"
      id="wc-additional-editor-title">This is some title here</span>
  </div>
  <div class="wc-additional-editor-group">
    <span class="wc-additional-editor-item-name">Type:</span>
    <span class="wc-additional-editor-item-content" id="wc-additional-editor-
type">bipolar</span>
  </div>
  <div class="wc-additional-editor-group">
    <span class="wc-additional-editor-item-content">
      <a href="" target="_blank" class="wc-additional-editor-link"
id="wc-additional-editor-editor-link">Open content</a></span>
    </div>
  </div>
`;
}

connectedCallback() {
  const title = this.item.values.title;
  this.shadowRoot.querySelector(
    '#wc-additional-editor-title'
  ).innerHTML = title;
  const type = this.content.mimeType;
  this.shadowRoot.querySelector(
    '#wc-additional-editor-type'
  ).innerHTML = type;
  const editorLink = this.link;
  this.shadowRoot.querySelector(
    '#wc-additional-editor-editor-link'
  ).href = editorLink;
}
}

```

```
customElements.define('my-additional-editor', MyAdditionalEditor);
```

#### 4.1.2.12 CustomFieldEditor

`cue.core.webcomponents.CustomFieldEditor` can be used to create a custom field editor.

It is defined as follows:

```
export abstract class CustomFieldEditor extends CUEElement
  implements webcomponent.CustomFieldEditor {

  // Field MIME type as defined in the content type
  public mimeType: string;

  // Sets the value of the specified field (that is a different field from this one)
  public abstract setFieldValue(fieldName: string, value: any): void;

  // Function to be called whenever the value of this field changes
  public abstract addValueWatcher(watcher: (value: any) => void): () => void;

  // Sets the value of this field
  public abstract setValue(value: any): void;

  // The value of this field
  public abstract getValue(): any;

  // Function to be called whenever the read-only status of this field changes
  public abstract addReadOnlyWatcher(watcher: (value: boolean) => void): () => void;

  // The read-only status of this field
  public abstract isReadOnly(): boolean;

  // Returns the content being edited in the editor
  public abstract getContent(): Promise<webcomponent.Content>;

  // Returns the id of the content item being edited in the editor
  getArticleId(): Nullable<string>;

  // Returns the URI of the content item being edited in the editor
  getArticleUri(): Nullable<string>;

  // Returns the content type of the content item being edited in the editor
  getContentType(): Nullable<string>;

  // Returns the state of the content item being edited in the editor
  getState(): Nullable<webcomponent.ContentState>;

  // Returns the published date of the content item being edited in the editor
  getPublishedDate(): Nullable<moment.Date>;
}
```

##### 4.1.2.12.1 CustomFieldEditor Configuration

The following properties must be defined to configure a custom field editor based on `CustomFieldEditor`:



**- name**

The name of the web component. The name you specify **must** contain a hyphen. Remember also that the id property name must be preceded by a hyphen (-).

**tagName**

The name of the custom HTML element that is used to encapsulate the component: the name used in the `document.registerElement()` call in the component's `script` element.

**modulePath**

The URI of the component.

All the properties must be entered as a list item belonging to a `customComponents` property. They must be indented correctly and the `id` property must be preceded by a hyphen (-) to indicate the start of a new list item. The following example shows the required format:

```
customComponents:
  - name: "custom-slider"
    tagName: "my-slider"
    modulePath: "http://www.example.com/webcomponents/my-slider.js"
```

#### 4.1.2.12.2 Custom Field Editor Invocation

To use a custom field editor for a particular field, you need to add a `ui:editor` to the field's definition in the `content-type` resource. The `ui:editor` element has two attributes:

**type**

This must always be set to `web-component`.

**name**

This must be set to the name of the component as defined in the field editor configuration file.

To use the slider field editor defined in [section 4.1.2.12.1](#), for example, you would need to add the following `ui-editor` element to your field definition:

```
<field type="number" name="percentage">
  <ui:label>Percentage</ui:label>
  <ui:editor type="web-component" name="custom-slider"/>
</field>
```

#### 4.1.2.12.3 CustomFieldEditor Example

```
class NumberSlider extends cue.core.webcomponents.CustomFieldEditor {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
<style>
:host {
margin: 0;
padding: 0px;
width: 100%;
display: block;
}

.spacer {
padding: 10px;
height: 30px;
```

```

overflow: hidden;
}

#thefield {
width: 100%;
}
</style>
<div class="spacer">
<input id="thefield" type="range" max="100" min="0" value="0"><br/>
</div>
`;
}

connectedCallback() {
  this.updateViewValue();
  this.updateReadOnly();
  this.shadowRoot.querySelector('input').addEventListener('input', () =>
this.updateModelValue());
  this.addValueWatcher((value) => {
    this.updateViewValue();
  });
  this.addReadOnlyWatcher((value) => {
    this.updateReadOnly();
  });
}

updateViewValue() {
  this.shadowRoot.querySelector('input').value = this.getValue();
};

updateModelValue() {
  this.setValue(parseInt(this.shadowRoot.querySelector('input').value));
};

updateReadOnly() {
  this.shadowRoot.querySelector('input').readOnly = this.isReadOnly();
};
}
customElements.define('number-slider', NumberSlider);

```

#### 4.1.2.13 CustomStoryElementEditor

`cue.core.webcomponents.CustomStoryElementEditor` can be used to create a custom field editor for a field belonging to a story element.

It is defined as follows:

```

export abstract class CustomStoryElementEditor extends CustomFieldEditor

  /** from CustomFieldEditor interface */

  // Field MIME type as defined in the content type
  public mimeType: string;

  // Sets the value of a Content field outside the storyline
  public abstract setFieldValue(fieldName: string, value: any): void;

  // Function to be called whenever the value of this field changes
  public abstract addValueWatcher(watcher: (value: any) => void): () => void;

```

```
// Sets the value of this field
public abstract setValue(value: any): void;

// Returns the value of this field
public abstract getValue(): any;

// Function to be called whenever the read-only status of the story element changes
public abstract addReadOnlyWatcher(watcher: (value: boolean) => void): () => void;

// The read-only status of the story element
public abstract isReadOnly(): boolean;

// Returns the complete story content being edited in the editor
public abstract getContent(): Promise<webcomponent.Content>;

// Function to be called whenever the content changes
public abstract addContentWatcher(watcher: (content: webcomponent.Content) => void):
() => void;

// Returns the id of the content item being edited in the editor
getArticleId(): Nullable<string>;

// Returns the URI of the content item being edited in the editor
getArticleUri(): Nullable<string>;

// Returns the content type of the content item being edited in the editor
getContentType(): Nullable<string>;

// Returns the state of the content item being edited in the editor
getState(): Nullable<webcomponent.ContentState>;

// Returns the published date of the content item being edited in the editor
getPublishedDate(): Nullable<moment.Date>;

/** CustomStoryElementEditor interface */

// Returns the storyline being edited in the editor
public abstract getStoryline(): Promise<webcomponent.Storyline>;

// Returns story element. Undefined storyElementId means the story element where the
custom field resides
public abstract getStoryElement(storyElementId?: string):
Promise<webcomponent.StoryElement>;

// Returns value of field inside story element. Undefined storyElementId means the
story element where the custom field resides
public abstract getStoryElementFieldValue(fieldName: string, storyElementId?:
string): any;

// Sets value of field inside story element. Undefined storyElementId means the
story element where the custom field resides
public abstract setStoryElementFieldValue(fieldName: string, value: any,
storyElementId?: string): void;

// Function to be called whenever the storyline changes
public abstract addStorylineWatcher(watcher: (storyline: webcomponent.Storyline) =>
void): () => void;
```

```

// Function to be called whenever the story element changes. Undefined
storyElementId means the story element where the custom field resides
public abstract addStoryElementWatcher(watcher: (storyElement:
webcomponent.StoryElement) => void): () => void;

// Function to be called whenever the storylines text selection changes
public abstract addTextSelectionWatcher(watcher: (selection:
webcomponent.StorylineTextSelection) => void): () => void;

// Returns current text selection
public abstract getTextSelection(): webcomponent.StorylineTextSelection | undefined;
}

```

#### 4.1.2.13.1 CustomStoryElementEditor Configuration

The following properties must be defined to configure a custom field editor based on **CustomStoryElementEditor**:

- **name**

The name of the web component. The name you specify **must** contain a hyphen. Remember also that the id property name must be preceded by a hyphen (-).

**tagName**

The name of the custom HTML element that is used to encapsulate the component: the name used in the **document.registerElement()** call in the component's **script** element.

**modulePath**

The URI of the component.

All the properties must be entered as a list item belonging to a **customComponents** property. They must be indented correctly and the **id** property must be preceded by a hyphen (-) to indicate the start of a new list item. The following example shows the required format:

```

customComponents:
  - name: "custom-slider"
    tagName: "my-slider"
    modulePath: "http://www.example.com/webcomponents/my-slider.js"

```

#### 4.1.2.13.2 CustomStoryElementEditor Invocation

To use a custom field editor for a particular story element field, you need to add a **ui:editor** to the field's definition in the storyline definition. The **ui:editor** element has two attributes:

**type**

This must always be set to **web-component**.

**name**

This must be set to the name of the component as defined in the field editor configuration file.

To use the slider field editor defined in [section 4.1.2.13.1](#), for example, you would need to add the following **ui-editor** element to your field definition:

```

<field type="number" name="percentage">
  <ui:label>Percentage</ui:label>
  <ui:editor type="web-component" name="custom-slider"/>
</field>

```

## 4.1.2.13.3 CustomStoryElementEditor Example

```

class MySlider extends cue.core.webcomponents.CustomStoryElementEditor {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0px;
          width: 100%;
          display: block;
        }

        .spacer {
          padding: 10px;
          height: 30px;
          overflow: hidden;
        }

        #thefield {
          width: 100%;
        }
      </style>
      <div class="spacer">
        <input id="thefield" type="range" max="100" min="0" value="0"><br/>
      </div>
    `;
  }

  connectedCallback() {
    this.updateViewValue();
    this.updateReadOnly();
    this.shadowRoot.querySelector('input').addEventListener('input', () =>
    this.updateModelValue());
    this.addValueWatcher((value) => {
      this.updateViewValue();
    });
    this.addReadOnlyWatcher((value) => {
      this.updateReadOnly();
    });
  }

  updateViewValue() {
    this.shadowRoot.querySelector('input').value = this.getValue();
  };

  updateModelValue() {
    this.setValue(parseInt(this.shadowRoot.querySelector('input').value));
  };

  updateReadOnly() {
    this.shadowRoot.querySelector('input').readOnly = this.isReadOnly();
  };
}
customElements.define('custom-slider', MySlider);

```

#### 4.1.2.14 Sending Notifications from Web Components

You can send notifications from your web components. These notifications will appear in CUE's notification center in exactly the same way as CUE's own notifications. Notifications are managed by the **notification** object, which is exposed as a property of all CUE web component objects. It provides two methods - one for showing notifications, and one for hiding them.

```
interface cue.core.webcomponents.Notification {
  show(title: string, body: string): Promise<string>;
  hide(notificationId: string): Promise<boolean>;
}
```

##### 4.1.2.14.1 Web Component Notification Example

This example shows part of a web component in which a drop handler listens for drop events in a particular HTML element and sends notifications each time one occurs.

```
let notificationId;
connectedCallback() {
  this.shadowRoot.querySelector('div').addEventListener('dragover', event => {
    event.preventDefault();
    event.stopPropagation();
  });
  this.shadowRoot.querySelector('div').addEventListener('drop', event =>
    this.dropHandler(event));
}

dropHandler(event) {
  event.preventDefault();

  const uriList = event.dataTransfer.getData('text/uri-list');
  const uri = event.dataTransfer.getData('x-cue/uri');
  this.shadowRoot.querySelector('#uri-list .right').innerHTML = uriList;
  this.shadowRoot.querySelector('#uri .right').innerHTML = uri;
  const notification = this.notification;
  if (notification) {
    if (this.notificationId) {
      notification.hide(this.notificationId);
    }
    notification
      .show('Drop Data', `URI: ${uri}. URI List: ${uriList}.`)
      .then(notificationId => (this.notificationId = notificationId));
  }
}
```

#### 4.1.2.15 Adding Dialogs to Web Components

**cue.core.webcomponents.CUEElement** has a **dialog** property that provides a range of methods for adding dialogs to your web components:

```
interface Dialog {
  showOneButton(
    message: string,
    title: string,
    buttonLabel: string = 'OK'
  ): Promise<void>;
  showTwoButton(
    message: string,
```

```
        title: string,  
        okLabel: string = 'OK',  
        cancelLabel: string = 'Cancel',  
        setOkAsDefault: boolean = true  
    ): Promise<void>;  
    showError(message: string): Promise<void>;  
    showWarning(message: string): Promise<void>;  
    showVdf(  
        vdfPayload: string,  
        title: string,  
        okLabel: string = 'OK',  
        cancelLabel: string = 'Cancel',  
        setOkAsDefault: boolean = true  
    ): Promise<Content>;  
}
```

The **Dialog** methods can therefore be called from any of the CUE web components as follows:

```
this.dialog.function_name(_params)
```

For example:

```
this.dialog.showOneButton("Do you want to continue?", "Continue Dialog")
```

The methods are described in more detail in the following sections.

### 4.1.2.15.1 showOneButton

```
showOneButton(  
    message: string,  
    title: string,  
    buttonLabel?: string = 'OK'  
): Promise<void>;
```

Displays a simple one-button dialog. It has the following parameters:

**message**

The message to display in the dialog.

**title**

The dialog title.

**buttonLabel**

The label to display on the dialog's only button ("OK" by default).

### 4.1.2.15.2 showTwoButton

```
showTwoButton(  
    message: string,  
    title: string,  
    okLabel: string = 'OK',  
    cancelLabel: string = 'Cancel',  
    setOkAsDefault: boolean = true  
): Promise<void>;
```

Displays a simple two-button dialog. It has the following parameters:

**message**

The message to display in the dialog.

**title**

The dialog title.

**okLabel**

The label to display on the dialog's **OK** button ("OK" by default).

**cancelLabel**

The label to display on the dialog's **Cancel** button ("Cancel" by default).

**setOkAsDefault**

If **true** (the default) then the **OK** button in the dialog is set as the default choice.

#### 4.1.2.15.3 showError

```
showError(message: string): Promise<void>;
```

Displays a simple dialog containing an error message and a **Close** button. It has one parameter:

**message**

The error message to display in the dialog.

#### 4.1.2.15.4 showWarning

```
showWarning(message: string): Promise<void>;
```

Displays a simple dialog containing a warning message and a **Close** button. It has one parameter:

**message**

The warning message to display in the dialog.

#### 4.1.2.15.5 ShowVdf

```
showVdf(  
  vdfPayload: string,  
  title: string,  
  okLabel: string = 'OK',  
  cancelLabel: string = 'Cancel',  
  setOkAsDefault: boolean = true  
) : Promise<Content>;
```

Displays a dialog containing a **title**, custom contents defined in a VDF file and two buttons with the default labels **OK** and **Cancel**. The VDF file is submitted via the **vdfPayload** argument.

Displays a two-button dialog with custom content. It has the following parameters:

**vdfPayload**

A VDF document defining the content to display in the dialog. For further information, see [section 4.1.2.15.6](#).

**title**

The dialog title.

**okLabel**

The label to display on the dialog's **OK** button ("OK" by default).



**cancelLabel**

The label to display on the dialog's **Cancel** button ("**Cancel**" by default).

**setOkAsDefault**

If **true** (the default) then the **OK** button in the dialog is set as the default choice.

## 4.1.2.15.6 Defining Custom Dialogs

The **ShowVdf ()** method must be supplied with a VDF payload document defining the content of the dialog to be displayed. VDF is an XML format used to represent content items - for details, see the [Content Engine Integration Guide](#). Here, VDF is used to define a sequence of field values to be displayed in the dialog. For example:

```
<vdf:payload
  xmlns:vdf="http://www.vizrt.com/types"
  model="webcomponents/simple-dialog/vdfEditorModel.xml">
<vdf:field name="title">
  <vdf:value>This is a title</vdf:value>
</vdf:field>
<vdf:field name="body">
  <vdf:value>
    <div xmlns="http://www.w3.org/1999/xhtml">
      <p>This is some body text.</p>
    </div>
  </vdf:value>
</vdf:field>
</vdf:payload>
```

A VDF payload document only contains field values, it does not contain any metadata about the fields. You must therefore always create a VDF model document as well, which is referenced from the payload document. The example shown above references a model called **vdfEditorModel.xml**, which looks like this:

```
<vdf:model xmlns:vdf="http://www.vizrt.com/types"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  xmlns:atom="http://www.w3.org/2005/Atom">
<vdf:schema>
  <vdf:fielddef name="title" label="Title" mediatype="text/plain" xsdtype="string">
    <ui:label>Title</ui:label>
    <ui:description>The title of the article</ui:description>
  </vdf:fielddef>
  <vdf:fielddef name="body" label="Body text" mediatype="application/xhtml+xml"
  xsdtype="string">
    <ui:label>Body text</ui:label>
    <ui:description>The body text of the article.</ui:description>
    <ui:style>body { min-height: 200px; }</ui:style>
  </vdf:fielddef>
</vdf:schema>
</vdf:model>
```

Together, the VDF payload and model provide sufficient information for **ShowVdf ()** to construct the dialog.

You can use **ShowVdf ()** to create dialogs containing any of the field types supported by CUE.

#### 4.1.2.15.7 Dialog Example

The following example web component creates a metadata panel section containing a series of forms that you can use to display demo dialogs. It has no practical function, it is simply a demonstration of how the various dialog methods work, and what kind of dialogs they display. The web component is created using the `cue.core.webcomponents.StorylineEditorMetadataPanel` class and is configured as follows:

```
customComponents:
  - name: "simple-dialog"
    tagName: "simple-dialog"
    modulePath: "webcomponents/simple-dialog/simple-dialog.js"
    attributes:
      title: "Simple Dialog"
      icon: "simple-dialog-icon"
```

Here is the web component code:

```
class SimpleDialog extends cue.core.webcomponents.StorylineEditorMetadataPanel {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host { margin:0; padding: 0; width: 100%; display:block; }
        h1, h2, h3, h4 {
          color: #9c9c9c;
        }

        /* Panel specific inputs */
        .text-field {
          height: 32px;
          vertical-align: middle;
          font-family: 'Hind', Helvetica Neue, Helvetica, Arial, Sans-serif;
          font-weight: lighter;
          font-size: 18px;
          line-height: 1.3;
          color: #444444;
          cursor: pointer;
          border-radius: 3px;
          width: 100%;
        }

        .text-area {
          height: 128px;
          font-family: 'Hind', Helvetica Neue, Helvetica, Arial, Sans-serif;
          font-weight: lighter;
          font-size: 18px;
          color: #444444;
          cursor: pointer;
          border-radius: 3px;
          width: 100%;
        }

        button,
        input[type=submit],
        input[type=cancel],
        input[type=button] {
```

```

        height: 32px;
        border: none;
        background: #d3d3d3;
        vertical-align: middle;
        font-family: 'Hind', Helvetica Neue, Helvetica, Arial, Sans-serif;
        font-weight: lighter;
        font-size: 18px;
        line-height: 1.3;
        color: #444444;
        padding: 2px 10px 0 10px;
        cursor: pointer;
        border-radius: 3px;
        width: 100%;
    }
    button:hover {
        background: #e5e5e5;
    }
    .group {
        padding: 20px;
    }
    .spacer {
        padding:10px
    }
    p {
        margin: 0;
    }
</style>
<div>
    <h1>Simple Dialog API</h1>
    <h2>A Web Component</h2>
</div>
<hr>
<div class="group">
    <h3>Simple Customizable Dialogs</h3>
    <p></p>
    <div class="spacer"></div>
    <p>Title</p>
    <input class ="text-field" type="text" id="simple-title" value="Simple
Dialog">
    <p>Message</p>
    <textarea class="text-area" id="simple-message" rows="5" >description</
textarea>
    <p>Button Label</p>
    <input class ="text-field" type="text" id="simple-button-label"
value="Ok">
    <div class="spacer"></div>
    <button id="simple-button">Show One Button Dialog</button>
</div>
<hr>
<div class="group">
    <h3>Customizable Reactive Dialogs</h3>
    <p>The dialog API exposes customizable dialogs with positive and negative
options. The answer is a Promise that is resolved or rejected.</p>
    <div class="spacer"></div>
    <p>Title</p>
    <input class ="text-field" type="text" id="two-button-title" value="OK or
Cancel dialog">
    <p>Message</p>

```

```

        <textarea class="text-area" id="two-button-message" rows="5" >Using this
dialog one can prompt the user for a positive or negative answer. The answer can be
used for further action.</textarea>
        <p>Cancel label</p>
        <input class ="text-field" type="text" id="two-button-cancel-label"
value="Cancel">
        <p>Ok Label</p>
        <input class ="text-field" type="text" id="two-button-ok-label"
value="Ok">
        <label for="two-button-use-default-buttons"> Use default buttons</label>
        <input type="checkbox" id="two-button-use-default-buttons" checked>
        <div class="spacer"></div>
        <div class="spacer"></div>
        <button id="two-button-button">Show Two Button Dialog</button>
        <div class="spacer"></div>

        <p>output:</p>
        <input disabled class ="text-field" type="text" id="two-button-output"
value="">
    </div>
    <hr>
    <div class="group">

        <h3>Input validation using Dialogs</h3>
        <p>The Dialos API can be used to validate input from the user.</p>
        <div class="spacer"></div>
        <p>Which City was the European Capital of Culture 2017?</p>
        <input class ="text-field" type="text" id="answer" value="">
        <div class="spacer"></div>
        <button id="quiz-button">Answer question</button>

        <div class="spacer"></div>
        <p>Insert a number between 5 and 10</p>
        <input class ="text-field" type="number" id="numberField" value="">
        <div class="spacer"></div>
        <button type="submit" id="numValidation">Submit</button>
        <div class="spacer"></div>

    </div>
    <hr>
    <div class="group">
        <h3>Alert Dialogs</h3>
        <p>The dialog API exposes two levels of alert dialogs; Warning and Error.
A description of the problem can be passed to the dialog.</p>
        <div class="spacer"></div>
        <p>Type alert message:</p>
        <textarea class="text-area" id="error-msg" rows="5" >A error dialog can be
used to notify the user, who is about to do something is not permitted</textarea>
        <div class="spacer"></div>
        <button id="openWarningDialog" class="buttons">Warning Dialog</button>
        <div class="spacer"></div>
        <button id="openErrorDialog" class="buttons">Error Dialog</button>
        <div class="spacer"></div>
    </div>

    <hr>
    <h2>Advanced Dialog API</h2>
    <p>The advanced API is based on VDF models.</p>
    <div class="group">
        <h3>A dialog editor</h3>

```

```

    <p>Values can be passed to the dialog</p>
    <p>Top</p>
    <input class="text-field" type="text" id="editor-top" value="top">
    <p>Titel</p>
    <input class="text-field" type="text" id="editor-title" value="title">
    <p>Cancel label</p>
    <input class ="text-field" type="text" id="editor-cancel-label"
value="Cancel">
    <p>Ok Label</p>
    <input class ="text-field" type="text" id="editor-ok-label" value="Ok">
    <label for="editor-use-default-buttons"> Use default buttons</label>
    <input type="checkbox" id="editor-use-default-buttons" checked>
    <div class="spacer"></div>
    <button id="editor">Editor Dialog</button>
    <div class="spacer"></div>
    <p>body output:</p>
    <span class="text-field" type="text" id="editor-output"></span>
  </div>
  `;
}

connectedCallback() {
  this.shadowRoot
    .getElementById('simple-button')
    .addEventListener('click', () => {
      // Get the dialog configurations
      const top = this.shadowRoot.querySelector('#simple-title').value;
      const description = this.shadowRoot.querySelector('#simple-message')
        .value;
      const buttonLabel = this.shadowRoot.querySelector(
        '#simple-button-label'
      ).value
        ? this.shadowRoot.querySelector('#simple-button-label').value
        : undefined;
      // Call the dialog api
      this.dialog.showOneButton(description, top, buttonLabel);
    });

  this.shadowRoot
    .getElementById('two-button-button')
    .addEventListener('click', () => {
      // Get the dialog configurations
      const top = this.shadowRoot.querySelector('#two-button-title').value;
      const description = this.shadowRoot.querySelector('#two-button-message')
        .value;
      const okButtonLabel = this.shadowRoot.querySelector(
        'input[id="two-button-ok-label"]'
      ).value
        ? this.shadowRoot.querySelector('input[id="two-button-ok-label"]')
          .value
        : undefined;
      const cancelButtonLabel = this.shadowRoot.querySelector(
        'input[id="two-button-cancel-label"]'
      ).value
        ? this.shadowRoot.querySelector('input[id="two-button-cancel-label"]')
          .value
        : undefined;
      const useDefaultButtons = this.shadowRoot.querySelector(
        '#two-button-use-default-buttons'
      ).checked;

```

```
// Call the dialog api
this.dialog
  .showTwoButton(
    description,
    top,
    okButtonLabel,
    cancelButtonLabel,
    useDefaultButtons
  )
  .then(
    // if promise is resolved
    () => {
      this.shadowRoot.querySelector(
        'input[id="two-button-output"]'
      ).value = okButtonLabel ? okButtonLabel : 'OK';
    },
    //if promise is rejected
    () => {
      this.shadowRoot.querySelector(
        'input[id="two-button-output"]'
      ).value = cancelButtonLabel ? cancelButtonLabel : 'Cancel';
    }
  );
});

this.shadowRoot
  .getElementById('quiz-button')
  .addEventListener('click', () => {
    const answer = this.shadowRoot.querySelector('input[id="answer"]')
      .value;
    try {
      const possibleAnswers = ['aarhus', 'århus'];
      let correct = false;
      possibleAnswers.forEach(ans => {
        if (answer.toLowerCase().localeCompare(ans) === 0) correct = true;
      });

      if (!correct) {
        throw Error(
          'No "' +
            answer +
            '" +
            ' was not the European Capital of Culture 2017'
        );
      }
    }
    this.dialog.showOneButton(
      'Yes, it is correct that the European Capital of Culture 2017 is "Århus"',
      'Correct',
      'Yaay',
      true
    );
  } catch (error) {
    this.dialog.showOneButton(error.message, 'Incorrect', 'Oops', true);
  }
});

this.shadowRoot
  .getElementById('numValidation')
  .addEventListener('click', () => {
```

```

let x = this.shadowRoot.querySelector('input[id="numberField"]').value;
try {
  if (x == '') throw new Error('field is empty');
  if (isNaN(x)) throw new Error('input is not a number');
  x = Number(x);
  if (x < 5) throw new Error('input is too low');
  if (x > 10) throw new Error('input is too high');
  this.dialog.showOneButton(
    'Yes, number is between 5 an 10',
    'Good Job',
    'Thanks',
    true
  );
} catch (error) {
  this.dialog.showOneButton(
    error.message,
    'Invalid Input',
    "I'll try again",
    'Please try again'
  );
}
});

this.shadowRoot
  .getElementById('openWarningDialog')
  .addEventListener('click', () => {
    const description = this.shadowRoot.querySelector('#error-msg').value;
    this.dialog.showWarning(description);
  });

this.shadowRoot
  .getElementById('openErrorDialog')
  .addEventListener('click', () => {
    const description = this.shadowRoot.querySelector('#error-msg').value;
    this.dialog.showError(description);
  });

this.shadowRoot.getElementById('editor').addEventListener('click', () => {
  const top = this.shadowRoot.querySelector('#editor-top').value;
  const title = this.shadowRoot.querySelector('#editor-title').value;
  const okButtonLabel = this.shadowRoot.querySelector(
    'input[id="editor-ok-label"]'
  ).value
  ? this.shadowRoot.querySelector('input[id="editor-ok-label"]').value
  : undefined;
  const cancelButtonLabel = this.shadowRoot.querySelector(
    'input[id="editor-cancel-label"]'
  ).value
  ? this.shadowRoot.querySelector('input[id="editor-cancel-label"]').value
  : undefined;
  const useDefaultButtons = this.shadowRoot.querySelector(
    '#editor-use-default-buttons'
  ).checked;
  // Call the dialog api
  this.dialog
    .showVdf(
      `
      <vdf:payload xmlns:vdf="http://www.vizrt.com/types" model="webcomponents/
      simple-dialog/vdfEditorModel.xml">
        <vdf:field name="title">

```

```

        <vdf:value>` +
        title +
        `</vdf:value>
        </vdf:field>
        <vdf:field name="body">
            <vdf:value>
                <div xmlns="http://www.w3.org/1999/xhtml">
                    <p>this is body</p>
                </div>
            </vdf:value>
        </vdf:field>
    </vdf:payload>`,
    top,
    okButtonLabel,
    cancelButtonLabel,
    useDefaultButtons
)
.then(
    // if promise is resolved
    res => {
        this.shadowRoot.querySelector('#editor-output').innerHTML =
            res.values.body;
    },
    //if promise is rejected
    res => {
        console.log(res);
    }
);
});
}
}
customElements.define('simple-dialog', SimpleDialog);

class SimpleDialogIcon extends cue.core.webcomponents
    .StorylineEditorMetadataPanel {
    constructor() {
        super();

        this.attachShadow({ mode: 'open' });
        this.shadowRoot.innerHTML = `
            <style>
                :host { margin: 0 0px 0 0px; width: 26px; display: inline; float: left;
margin-right: 18px; }
                img { width: 20px; position: relative; }
            </style>
            <img class="icon">
        `;

        this.activeIconPath = 'simple-dialog-icon.png';
        this.inactiveIconPath = 'simple-dialog-icon.png';
    }
    connectedCallback() {
        this.activeStateChanged(this.active);
        this.addActiveWatcher(active => {
            this.activeStateChanged(active);
        });
    }
}

activeStateChanged(active) {
    let img = this.shadowRoot.querySelector('img.icon');

```



```
    if (active) {
      img.src = this.getAbsolutePath(this.activeIconPath);
    } else {
      img.src = this.getAbsolutePath(this.inactiveIconPath);
    }
  }
  getAbsolutePath(path) {
    const baseURI = import.meta.url;
    return baseURI.substring(0, baseURI.lastIndexOf('/') + 1) + path;
  }
}
customElements.define('simple-dialog-icon', SimpleDialogIcon);
```

## 4.2 Enrichment Services

It is not feasible for CUE to meet every user's requirements out of the box – particularly when it comes to integration with external systems. Such integrations are increasingly important as organizations adapt their workflows to make use of popular online productivity tools, publish content to social media and so on. When you publish a story in CUE, for example, you might also want to:

- Connect it to a Slack channel
- Create a card in Trello
- Push it to a Wordpress site
- Share it on one or more social media
- Send it to a legacy print system

CUE's enrichment services provide the means for you to satisfy such requirements for yourself, in a surprisingly straightforward way.

An enrichment service is a simple HTTP service that has a defined workflow. When it receives a request it recognizes from CUE, it responds in such a way as to guide CUE through the workflow, providing CUE with explicit instructions on what it should do next. You can, for example, configure CUE so that when the user clicks on **Publish** to publish a story, the story is not immediately published, but instead sent to an enrichment service you have created. The enrichment service can then perform some check on the story – count the related links, for example – and return a response to CUE. In this case the response could either be an "OK, continue", allowing the story to be published, or an instruction to display a message saying "please add 3 related links" and cancel the publish operation.

It is also possible to define much more complex interactions though: you can instruct CUE to display a sequence of dialogs for the user to fill in, and use the supplied data to modify the content of the submitted story. You can also trigger enrichment services in different ways, not only when the **Publish** button is pressed.

Here is an example workflow for publishing to social media that could be implemented using an enrichment service:

1. The user selects **Publish**.

2. CUE displays a dialog containing:

- A **Title** text field (max 140 characters). It is pre-filled with either the story's title or the first 140 characters of its lead text field if available, but the user can edit it if required.
- A **Social media** drop-down field, containing the names of supported social media.
- Three buttons:
  - **Share**: publishes the story and then shares it on the selected medium, using the specified title.
  - **Don't share**: publishes the story without sharing it.
  - **Cancel**: cancels both operations – the story is neither published nor shared.

If the user selects **Share**, then the enrichment service will make an appropriate HTTP request to a back-end server that will take care of sharing a link to the published story, using the specified **Title**.

Despite the fact that this additional functionality is implemented in an enrichment service completely outside CUE, it appears to be fully integrated from the user's point of view: the dialog is constructed and displayed by CUE and looks just like any other CUE dialog.

Enrichment services can be created to handle a number of different CUE structures, not only content items. You can, for example, create enrichment services to handle section pages. Enrichment services for the following items are, however, not supported at present:

- Lists
- CUE Live events
- CUE Print-related structures such as assignments and story folders

To create an enrichment service you need to:

- Configure CUE to access a service.
- Create the service. It must be an HTTP service that accepts specific kinds of requests from CUE, and supplies specific kinds of response.

#### 4.2.1 Configuring Enrichment Services in CUE

Configuring CUE to access an enrichment service is very straightforward – all you need to do is add a few entries to the CUE configuration file, `/etc/escenic/cue-web/config.yml`. Open this file for editing. If it does not already contain an `enrichmentServices` entry, then add one:

```
| enrichmentServices:
```

Underneath this entry, you can add sub-entries for all the enrichment services you want to define. An enrichment service configuration contains the following entries:

```
| - name: service-name  
  href: http://host:port/service-url  
  title: service-title  
  contentTypes: content-type-filter  
  mimeTypes: mime-type-filter  
  publications: publication-filter  
  triggers:  
    - name: trigger-name
```

where:

**name**

Is the name of the enrichment service. The name must be unique since CUE identifies the services by their name. Any service definition with a duplicate name will be ignored.

**href**

Is the URI of the enrichment service. CUE will **POST** the current content item to this URI as an Atom entry.

**title**

Is the title of the enrichment service. This title is displayed by CUE in headers and labels as appropriate.

**contentTypes**

Is an optional filter, specified as an array of content type names. For example:

```
| contentTypes: ['story', 'storyline']
```

By default, an enrichment service is applied to all content types. If you specify a **contentTypes** filter, however, then it is only applied to the specified content types.

**mimeTypes**

Is an optional filter, specified as an array of MIME type names. For example:

```
| mimeTypeTypes: ['x-ece/picture', 'x-ece/video', 'x-ece/gallery']
```

By default, an enrichment service is applied to all MIME types. If you specify a **mimeTypeTypes** filter, however, then it is only applied to the specified MIME types. The possible MIME types that may be specified in the array are:

<b>x-ece/story</b>	CUE story-type content item
<b>x-ece/picture</b>	CUE image content item
<b>x-ece/video</b>	CUE video content item
<b>x-ece/gallery</b>	CUE gallery content item
<b>x-ece/new-content</b>	New CUE content that has not yet been saved
<b>x-ece/section</b>	CUE section
<b>x-ece/section-page</b>	CUE section page
<b>x-ece/*</b>	All kinds of CUE content

**publications**

Is an optional filter, specified as an array of publication names. For example:

```
| publications: ['tomorrow-online', 'tomorrow-sport']
```

By default, an enrichment service is applied to content items irrespective of which publication they belong to. If you specify a **publication** filter, however, then it is only applied to content items from the specified publications.

**triggers**

Is a list of one or more triggers defining when CUE is to **POST** a content item to the enrichment service.

A trigger may optionally have the same kinds of filters as can be specified for enrichment services: **contentType**s, **mimeType**s and **publications**. They work in exactly the same way as enrichment service filters, but apply only to the specific trigger. In addition, they override any corresponding filters set at the enrichment service level. In the following case, for example:

```
- name: "myservice"
  ...
  contentType: [ "story", "storyline" ]
  ...
  triggers:
    - name: trigger1
    - name: trigger2
    - name: trigger3
      contentType: [ "story" ]
```

**myservice** will be applied to both **story** and **storyline** content items when *trigger1* and *trigger2* fire, but will only be applied to **story** content items when *trigger3* fires.

Some triggers may have properties that need to be specified, in which case the service configuration will also include a **properties** value consisting of a sequence of one or more property settings.

Here is an example trigger definition with both a **mimeType** setting and a list of properties.

```
triggers:
  - name: trigger-name
    mimeType: [mimetype-list]
    properties:
      property-name: property-value
      property-name: property-value
```

See [section 4.2.1.1](#) for further information.

#### 4.2.1.1 Enrichment Service Triggers

CUE supports a number of different triggers that make it possible to call enrichment services at different points in the editing/publishing process. There is also a timer-based trigger that will call an enrichment service repeatedly at a specified interval. The triggers vary slightly according to the type of enrichment service.

In addition to the specific triggers described in the following sections, all triggers may have a **timeout** property. This is a timeout specified in seconds, for example:

```
properties:
  timeout: 10
```

If a triggered environment service does not respond within the timeout period, then the request is abandoned. A timeout failure of this kind is handled by CUE in the same way as an error response from the service (see [section 4.2.1.5](#)).

##### 4.2.1.1.1 Content Item Triggers

The available triggers for content item enrichment services are:

###### **before-save**

Before saving, when the user presses the **Save** button. No properties required.

**after-save**

After saving, when the user presses the **Save** button. No properties required.

**before-save-state-state-name**

Before saving, when the user changes the state to *state-name*. If CUE has an Escenic Content Engine back end, then *state-name* can only be the name of one of the CUE default states (**draft**, **submitted**, **approved**, **published** or **deleted**). If CUE has a CUE Content Store back end, then *state-name* can either be one of these standard names or the name of a custom state defined in a custom workflow (see [Custom Workflow Definitions](#)). No properties required. Not supported for section page enrichment services.

**after-save-state-state-name**

After saving, when the user changes the state to *state-name*. If CUE has an Escenic Content Engine back end, then *state-name* can only be the name of one of the CUE default states (**draft**, **submitted**, **approved**, **published** or **deleted**). If CUE has a CUE Content Store back end, then *state-name* can either be one of these standard names or the name of a custom state defined in a custom workflow (see [Custom Workflow Definitions](#)). No properties required. Not supported for section page enrichment services.

**editor-opened**

A specified number of seconds after the content item is opened for editing. You must specify the number of seconds to wait as a property: **delay**: *n*.

**editor-recurring**

At specified intervals for as long as the content item is open for editing. You must specify the length of the interval (in seconds) as a property: **interval**: *n*.

**on-click**

When the user clicks a button in the content item. No properties required. For more information about this kind of trigger see the last example in [section 4.2.1.3](#).

#### 4.2.1.1.2 Section Page Triggers

The available triggers for section page enrichment services are:

**before-save**

Before saving/publishing, when either:

- The section page is in the **published** state and the user presses the **Publish** button.
- The section page is in the **draft published** state and the user presses the **Save** button.

**after-save**

After saving/publishing, when either:

- The section page is in the **published** state and the user presses the **Publish** button.
- The section page is in the **draft published** state and the user presses the **Save** button.

**before-save-state-save**

Before saving, when the section page is in the **published** state and the user presses the **Save** button. No properties required.

**after-save-state-save**

After saving, when the section page is in the **published** state and the user presses the **Save** button. No properties required.

**before-save-state-publish**

Before saving, when the section page is in the **draft published** state and the user presses the **Publish** button. No properties required.

**after-save-state-publish**

After saving, when the section page is in the **draft published** state and the user presses the **Publish** button. No properties required.

**editor-opened**

A specified number of seconds after the section page is opened for editing. You must specify the number of seconds to wait as a property: **delay: n**.

**editor-recurring**

At specified intervals for as long as the section page is open for editing. You must specify the length of the interval (in seconds) as a property: **interval: n**.

#### 4.2.1.2 Enrichment Service Authentication

In order for an enrichment service to be able to make Content Store web service calls on behalf of the user, it must be able to authenticate itself. You can make this possible by defining **authorized endpoints**.

Any enrichment service deployed on an authorized endpoint (or on the same origin as CUE) is given the user's credentials. This enables the enrichment service to make requests to the web service on the user's behalf and thereby perform tasks such as publishing related content items or creating new content items.

To define authorized endpoints, add an **authorizedEndpoints** entry to your **config.yml** file. This entry can contain an array of authorized endpoint URLs (each preceded by a hyphen). For example:

```
authorizedEndpoints:
  - "http://my-enrichment-service.info:1234/"
  - "http://some-other-enrichment-service.info:1234/"
```

#### 4.2.1.3 Configuration Examples

Here are a few example enrichment service configurations:

- Check that a content item has at least three tags before it is saved:

```
- name: check-minimum-tag
  href: http://host:port/checkMinimumTag
  title: Minimum Tags
  triggers:
    - name: before-save
```

- Check that a content item has no unpublished relations before it is published:

```
- name: check-unpublished-related-content
  href: http://host:port/checkUnpublishedContent
  title: Unpublished Related Content
  triggers:
    - name: before-save-state-published
```

- Check a content item's spelling at regular intervals:

```
- name: check-spelling
  href: http://host:port/spellChecker
  title: Spell Checker
  triggers:
```

```
- name: editor-recurring
  properties:
    interval: 20
```

- **Print a content item:**

```
- name: print-article
  href: http://host:port/printArticle
  title: Print Article
  triggers:
    - name: on-click
```

In order for this configuration to work there must not only be an enrichment service to print the content item at `http://host:port/printArticle`, the content item being edited must also contain a **Print** button for the user to click. Such buttons must be defined in content type definitions in the publication content-type resource. An enrichment service trigger button is defined by a content item `field` element with a `ui:editor` child element. The `ui:editor` child element must have a `type` attribute with the value `enrichment-service` and a name attribute that matches the name of the CUE enrichment service. For example:

```
<field name="enrichmentbutton" type="basic" mime-type="text/plain">
  <ui:label>Print</ui:label>
  <ui:editor type="enrichment-service" name="print-article"/>
</field>
```

#### 4.2.1.4 Enrichment Service Context Menu Entries

In some cases you may want users to be able to send a content item to an enrichment service by selecting a menu entry. You can achieve this by adding menu entries to content item context menus. A context menu can be displayed by right clicking on the content cards displayed in search results and other lists of content item, or by clicking on the "hamburger" button displayed in the top right corner of a content editor. To add an enrichment service to the context menus, you need to add a configuration like this to `/etc/escenic/cue-web/config.yml`, as well as the main enrichment service configuration:

```
extendedContextMenuItems:
  - name: "print-service"
    title: "Print"
    trigger: "on-print-menu-item-click"
    publication: "tomorrow-online"
    mimeTypes: ["x-ece/story"]
```

The `extendedContextMenuItems` property can contain any number of children, each defining a menu entry for a different enrichment service. Each menu entry definition should consist of the following properties:

**name**

A name for the menu entry definition.

**title**

The label to appear on the menu entry.

**trigger**

A trigger name for the menu item. The name(s) you specify here must also appear in the enrichment service's list of triggers. If you have specified `on-print-menu-item-click` as in the example shown above, then the same trigger name would need to appear in the print enrichment service configuration:

```

- name: print-article
href: http://host:port/printArticle
title: Print Article
triggers:
  - name: on-click
  - name: on-print-menu-item-click

```

**publication (optional)**

The publication with which the menu entry is to be associated. The menu entry will only appear in the context menu of content items that belong to the specified publication. If you omit this property then the menu entry will appear in the context menu of content items belonging to any publication.

**mimeTypes (optional)**

The MIME types with which the menu entry is to be associated. The menu entry will only appear in the context menu of content items of the specified MIME types. If you omit this property then the menu entry will appear in the context menu of content items of all MIME types.

**contentTypes (optional)**

The content types with which the menu entry is to be associated. The menu entry will only appear in the context menu of content items of the specified types. If you omit this property then the menu entry will appear in the context menu of content items of all types.

**states (optional)**

The states with which the menu entry is to be associated. The menu entry will only appear in the context menu of content items in the specified states. If you omit this property then the menu entry will appear in the context menu of content items in all types.

**selection (optional)**

This property must be set to one of the following values:

**["single"] (default)**

This menu entry is only displayed for single item selections.

**["multi"]**

This menu entry is only displayed for multiple item selections. You should only specify this option if the target enrichment service has been designed to handle multiple content items. For details see [section 4.2.3](#).

**["single", "multi"]**

This menu entry is displayed for both multiple and single item selections. You should only specify this option if the target enrichment service has been designed to handle multiple content items. For details see [section 4.2.3](#).

It doesn't make sense to specify both a **mimeTypes** and a **contentTypes** property. If you do, then the **contentTypes** property is ignored, and only the **mimeTypes** property is used.

**4.2.1.5 Handling Enrichment Service Errors**

An enrichment service may sometimes fail and return an **HTTP 5xx** response. An enrichment service request may also time out, if the service fails to respond quickly enough (see the **timeout** description in [section 4.2.1](#)). By default, either kind of failure stops the execution of the current workflow. If, for example, the enrichment service was started by a content item **before-save** trigger, then the content item in question will not be saved if the enrichment service fails or times out. This may not, however always be what you want. If the task performed by the enrichment service is regarded as non-essential, you may want the content item to be saved anyway.



To achieve this, set the enrichment service's **stopOnFailure** property to **false**. For example:

```
- name: check-minimum-tag
  href: http://host:port/checkMinimumTag
  title: Minimum Tags
  stopOnFailure: false
  triggers:
    - name: before-save
```

If **stopOnFailure** is set to **false** in this way, then a failure will not interrupt the flow of events - the content item will continue to be processed by any other configured enrichment services, and it will be saved as requested. An info message reporting the failure will be sent to the CUE notification center, but otherwise everything will proceed as normal.

If **stopOnFailure** is not specified, its default value is **true**.

If **stopOnFailure** is not specified or set to **true**, then by default failures are notified as followed:

Type of failure	Error Message / Notification
Enrichment service not reachable	A "service unreachable" error dialog is displayed
Enrichment service timed out	A "service timed out" error dialog is displayed
<b>HTTP 500</b> response	The custom message included in the response object is used to display an error dialog
Other <b>HTTP 5xx</b> responses	A generic HTTP 5xx error message is displayed in an error dialog

You can, however, set **notifyOnFailure** to **true**, for example:

```
- name: check-unpublished-related-content
  href: http://host:port/checkUnpublishedContent
  title: Unpublished Related Content
  stopOnFailure: true
  notifyOnFailure: true
  triggers:
    - name: before-save-state-published
```

If you do this, then the error messages displayed if the enrichment service is either unreachable or times out are also sent to the CUE notification center. **notifyOnFailure** does not, however, have this effect in the case of **HTTP 5xx** responses. **HTTP 5xx** messages are only displayed in dialogs, irrespective of the **notifyOnFailure** setting

## 4.2.2 Creating an Enrichment Service

An enrichment service is a standard web service that accepts HTTP **POST** requests from CUE, and responds with a specific subset of HTTP responses understood by CUE. You can create an enrichment service using any web technology or platform you like so long as it conforms to CUE's enrichment service protocol requirements.

When an enrichment service is triggered, CUE sends an HTTP **POST** request to the enrichment service URL, with an Atom entry in the body of the request. The Atom entry will contain the content item

currently being edited in CUE, packaged as a VDF payload document. This is the same packaging that is used to send content items to the Content Store web service - for details, see the [Content Engine Integration Guide](#).

The enrichment service can then examine the supplied content item, apply tests to it, modify it, modify other content in the Content Store (via the Content Store's web service), make use of external web services such as spelling or grammar checkers, publish the content item in external channels and so on. It must, however, finally send one of the following HTTP responses back to CUE:

#### 500 (Internal Server Error)

The enrichment service can give this response to indicate that an error of some kind has occurred. Unless **stopOnFailure** has been set to **false**, CUE would then cancel the trigger operation and either display an error dialog or notification containing the response message. The response body can be one of the following types:

##### **text/plain**

A plain text response message to be displayed in an error dialog.

##### **text/html**

A formatted HTML response message to be displayed in an error dialog.

##### **application/vnd.cue.notification+json**

JSON data containing a notification to be sent to the CUE notification center. The JSON data must have the following form:

```
{
  "title" : "Notification title",
  "body"  : "Notification body"
}
```

#### 400 (Bad Request)

The enrichment service can give this response to indicate that the **POST**ed content item has failed some test or other, and the response can contain an explanatory message that is displayed by CUE or sent to the notification center. A service that checks for unpublished relations, for example, could send a **400** response if it found any unpublished relations. Unless **stopOnFailure** has been set to **false**, CUE would then cancel the trigger operation (publish, presumably) and either display an error dialog or notification containing the response message. The response body can be one of the following types:

##### **text/plain**

A plain text response message to be displayed in an error dialog.

##### **text/html**

A formatted HTML response message to be displayed in an error dialog.

##### **application/vnd.cue.notification+json**

JSON data containing a notification to be sent to the CUE notification center. The JSON data must have the following form:

```
{
  "title" : "Notification title",
  "body"  : "Notification body"
}
```

#### 204 (No Content)

The enrichment service can give this response to indicate that CUE should just carry on as normal. A service that checks for unpublished relations, for example, could send a **204** response if it did **not** find any unpublished relations: Unless **stopOnFailure** has been set to **false**,

CUE would then simply complete the publish operation that triggered the enrichment service call, and take no further action.

#### 200 (OK)

The enrichment service can give this response in a number of different circumstances. Exactly what it means, and how it is used by CUE depends on the content returned in the body of the response. This can be one of the following types:

##### **text/plain**

This response is functionally the same as a **204 (No Content)** response from CUE's point of view: the only difference is that CUE displays the text content of the response in an information dialog. An enrichment service that automatically adds the content item to an automatically selected list might, for example, return information about which list the content item has been added to: "Item added to list 'urgent'".

##### **text/html**

This response is similar to a **200** response with **text/plain** content except that the information dialog displayed by CUE will contain formatted HTML.

##### **application/vnd.cue.notification+json**

In this case, the response body is JSON data containing a notification to be sent to the CUE notification center. The JSON data must have the following form:

```
{
  "title" : "Notification title",
  "body"  : "Notification body"
}
```

##### **application/atom+xml**

The Atom entry is expected to contain a content item. How the content item is handled depends on the entry's `<link rel="self"/>` element:

- If the **link** element contains the same **self** URL as the Atom entry originally **POSTed** by CUE, then it is assumed to be a modified version of the **POSTed** content item (returned, for example, from a grammar correction service). If the Atom entry contains all the fields originally **POSTed** by CUE, CUE overwrites the current content item with the returned version and then continues with the operation that triggered the enrichment service call (saving or publishing, for example). If, on the other hand, the Atom entry only contains **some** of the fields originally **POSTed** by CUE, then CUE only updates these fields before continuing with the operation that triggered the enrichment service call.
- If the **link** element contains a different **self** URL from the original Atom entry, then CUE opens the referenced content item in a new editor and completes the operation that triggered the enrichment service call (saving or publishing the original content item, for example).

This type of response cannot be sent by a multi-select enrichment service (see [section 4.2.3](#)).

##### **application/vnd.vizrt.payload+xml**

This response is a VDF payload document (described in [Content Engine Integration Guide](#)). It is expected to contain a sequence of field definitions. CUE constructs and displays a dialog box containing the fields specified in the VDF document, plus an **OK** and **Cancel** button. The expectation is that the user will fill in the form and click **OK**, or else click **Cancel**.

If the user clicks **OK**, then the content of the filled form is submitted to the enrichment service URL. The enrichment service can then process the content of the form and respond again in any of the ways listed above. It could, for example, return a **204 (No Content)** response, or it could get CUE to display another dialog by returning another **200 (OK)** response with a different `application/vnd.vizrt.payload+xml`. In this way the enrichment service can, if necessary, force CUE to display a long sequence of dialogs before finally performing some operation and terminating the operation that initially triggered the enrichment service.

If the user clicks **Cancel**, then the operation that triggered the enrichment service is cancelled.

### 4.2.3 Multi-select Enrichment Services

The ability to submit content items to an enrichment service by selecting a context menu entry opens the possibility of submitting multiple content items in one go. You can enable this possibility by specifying `selection: ["multi"]` or `selection: ["single", "multi"]` when configuring a context menu entry, as described in [section 4.2.1.4](#). However, in order for this to work the enrichment service must be able to handle multiple selections correctly. It must therefore differ from a single-select enrichment service in the following ways:

- It must be designed to accept a `text/uri-list` holding the URIs of the selected content items rather than an Atom entry holding the selected content item itself.
- If its purpose is to modify the selected content items, then it must do so by submitting **GET** and **PUT** requests to the Content Store web service for each URI in the list. It cannot include the modified content items in an HTTP **200 (OK)** response (which is what a single-select enrichment service does).

These are the only differences between a multi-select enrichment service and a single-select enrichment service.

If your enrichment service needs to handle **both** single and multiple selections then you must design it as a multi-select service that handles `text/uri-lists` rather than Atom entries, and add a `requestContentType` property setting to the enrichment service's trigger configurations as follows:

```
- name: print-article
  href: http://host:port/printArticle
  title: Print Article
  triggers:
    - name: on-click
      properties:
        requestContentType: text/uri-list
    - name: on-print-menu-item-click
      properties:
        requestContentType: text/uri-list
```

This property setting forces CUE to send a `text/uri-list` to the enrichment service rather than an Atom entry even for single content items.

### 4.2.4 Some Examples

This provides a couple of examples of how enrichment services can be used:

- A "text analysis" enrichment service that makes use of the "update content" action triggered by an **application/atom+xml** response.
- A "post to Slack" enrichment service that makes use of the dialog sequence triggered by an **application/vnd.vizrt.payload+xml** response.

Both example descriptions assume that an Atom entry like this is **POST**ed to the enrichment service:

```
<entry xmlns="http://www.w3.org/2005/Atom"
  xmlns:app="http://www.w3.org/2007/app"
  xmlns:metadata="http://xmlns.escenic.com/2010/atom-metadata"
  xmlns:dcterms="http://purl.org/dc/terms/">
  <id>http://host-ip-address/webservice/escenic/content/43</id>
  <title type="text">Test</title>
  <app:edited>2010-06-23T09:09:50.654Z</app:edited>
  <dcterms:created>2010-06-22T10:22:20.000Z</dcterms:created>
  <author>
    <name>demo Administrator</name>
    <uri>http://host-ip-address/webservice/escenic/person/2</uri>
  </author>
  <dcterms:identifier>4</dcterms:identifier>
  <metadata:reference source="ece-auto-gen" sourceid="6d7203c9-27d5-4fce-b14a-
a466ead83875"/>
  <link rel="http://www.vizrt.com/types/relation/home-section"
    href="http://host-ip-address/webservice/escenic/section/4"
    title="New Articles"
    type="application/atom+xml; type=entry"/>
  <link href="http://wrk-ermo:12345/publication-id/incoming/article4.ece"
    rel="alternate"/>
  <link href="http://host-ip-address/webservice/escenic/lock/article/43"
    rel="http://www.vizrt.com/types/relation/lock"/>
  <link rel="http://www.vizrt.com/types/relation/publication"
    href="http://host-ip-address/webservice/escenic/publication/demo"
    title="demo"
    type="application/atom+xml; type=entry"/>
  <metadata:creator>
    <name>demo Administrator</name>
  </metadata:creator>
  <metadata:publication href="http://host-ip-address/webservice/escenic/publication/
demo">
    <link rel="http://www.vizrt.com/types/relation/home-section"
      href="http://host-ip-address/webservice/escenic/section/4"
      title="New Articles"
      type="application/atom+xml; type=entry"/>
    <link rel="http://www.vizrt.com/types/relation/section"
      href="http://host-ip-address/webservice/escenic/section/4"
      title="New Articles"
      type="application/atom+xml; type=entry"/>
  </metadata:publication>
  <link href="http://host-ip-address/webservice/escenic/content/43" rel="edit"/>
  <link href="http://host-ip-address/webservice/escenic/content/43" rel="self"/>
  <content type="application/vnd.vizrt.payload+xml">
    <vdf:payload xmlns:vdf="http://www.vizrt.com/types"
      model="http://host-ip-address/webservice/escenic/model/another">
      <vdf:field name="TITLE">
        <vdf:value>Test</vdf:value>
      </vdf:field>
      <vdf:field name="BODY">
        <vdf:value>
          <div xmlns="http://www.w3.org/1999/xhtml">
```

```

        <p>This is a test</p>
      </div>
    </vdf:value>
  </vdf:field>
  <vdf:field name="ANALYSIS"></vdf:field>
</vdf:payload>
</content>
</entry>

```

#### 4.2.4.1 A "Text Analysis" Enrichment Service

This service sends the content of a story to an external text analysis service which returns some kind of results (a list of keywords, for example). One way of handling this would be to include a hidden "analysis" field in all the content types you want to be analyzed, to be used as a container for the keywords. Your enrichment service could then forward the content of all the visible fields to the analysis service, and add the keywords returned from the service to the hidden "analysis" field.

Here are the configuration settings for such a service:

```

enrichmentServices:
  - name: "Analyze text"
    href: http://my-web-service-host/analysis-service
    title: "Analyze text"
    triggers:
      - name: after-save-state-published
        properties: {}
        mimeTypes: ["x-ece/story"]

```

This configuration specifies that any "story-type" content items (content items that don't contain any binary fields such as video or images, and aren't live events or Newsgate stories) will be posted to the enrichment service at <http://my-web-service-host/analysis-service> when they are published.

When the enrichment service receives such a content item, it forwards the content from all the visible fields to a text analysis service. When it gets the results back from the text analysis service, it sends an **HTTP 200** response back to CUE with an **application/atom+xml** body containing a copy of the original Atom entry posted by CUE. The only part of the Atom entry that is modified is the VDF payload. All the fields except the **ANALYSIS** field have been removed, and the **ANALYSIS** field now contains the keywords returned from the text analysis service:

```

<vdf:payload xmlns:vdf="http://www.vizrt.com/types"
  model="http://host-ip-address/webservice/escenic/model/another">
  <vdf:field name="ANALYSIS"></vdf:field>
  <vdf:value>sport,football,brazil</vdf:value>
</vdf:field>
</vdf:payload>

```

When CUE receives this response from the enrichment service, it overwrites the **ANALYSIS** field of the content item with the value supplied by the enrichment service and publishes the content item. No other fields are modified.

#### 4.2.4.2 A "Post to Slack" Enrichment Service

This enrichment service posts a link to the [Slack](#) messaging service whenever a story is published. Before it posts the link, however, it needs to prompt the CUE user to enter a short name for the story, and the name of the Slack channel in which it is to be posted.

Here are the configuration settings for such a service:

```
enrichmentServices:
  - name: "Post to Slack"
    href: http://my-web-service-host/slack-service
    title: "Post to Slack"
    triggers:
      - name: after-save-state-published
        properties: {}
        mimeTypes: ["x-ece/story"]
```

This configuration specifies that any "story-type" content items (content items that don't contain any binary fields such as video or images, and aren't live events or Newsgate stories) will be posted to the enrichment service at **http://my-web-service-host/slack-service** when they are published.

When the Slack enrichment service receives such a content item, it returns an **HTTP 200** response with an **application/vnd.vizrt.payload+xml** body containing a VDF payload document. The VDF document contains the prompts to be displayed in the dialog, for example:

```
<vdf:payload xmlns:vdf="http://www.vizrt.com/types" model="http://web-service-host/
slack-channel-description.xml">
  <vdf:field name="slack-name">
    <vdf:value>red-herring</vdf:value>
  </vdf:field>
  <vdf:field name="channel">
    <vdf:value>#sports</vdf:value>
  </vdf:field>
</vdf:payload>
```

Note the following about this document:

- The **vdf:payload** element's **model** attribute must contain the URI of a **VDF model document** defining the structure of the payload. You must create this model document yourself and make it available somewhere (most likely on the same host as the enrichment service itself). The VDF model document for the example payload shown above might look like this:

```
<model xmlns:vdf="http://www.vizrt.com/types">
  <schema>
    <fielddef name="slack-name" label="Story name in Slack" xsdtype="string"/>
    <fielddef name="channel" label="Slack channel" xsdtype="string"/>
  </schema>
</model>
```

For a description of the VDF model document format, see [here](#).

- The values in the fields are defaults. If you do not want to supply defaults to the fields in the form, you can omit the values.

When CUE receives the payload document, it looks up the referenced model document and uses the information to construct and display a dialog containing the specified fields. The user can then enter the required values. When the user selects **OK**, CUE will **POST** the payload document (with any changes made by the user) back to the enrichment service. The enrichment service can then post the story to Slack and return **HTTP 204** (No Content) to CUE, allowing CUE to complete the operation that initiated the enrichment service call. Alternatively, the enrichment service could return **HTTP 200** (OK) with a **text/plain** Content-Type header in order for CUE to display a message indicating that the story has been posted to Slack. If the user selects **Cancel** instead of **OK**, then nothing is sent to the

enrichment service and CUE just carries on and completes the operation that initiated the enrichment service call.

## 4.2.5 Learning More About Enrichment Services

If you want to learn more about CUE enrichment services, take a look at this series of articles on <http://blogs.escentic.com>:

[Diving into enrichment services](#)

## 4.3 Drop Resolvers

A **drop resolver** is an HTTP service that reacts to objects dropped into CUE relation drop zones. A drop resolver is invoked when an object that matches a specified MIME type or URL pattern is dropped in a drop zone. The drop resolver is passed a **drop context** containing the information needed to be able to upload content to the CUE Content Store or the CCI Newsgate back end. A drop resolver can therefore be used to seamlessly import external objects dropped into CUE.

A typical use of a drop resolver would be to handle the import of images dragged into CUE from a Digital Asset Management (DAM) system. When such an image is dropped into a story relation in CUE, CUE checks the dropped image's MIME type or URI and calls the appropriate drop resolver. The drop resolver then uploads the dropped image in the background to the appropriate back end, and return the URI of the uploaded content to CUE. This allows CUE to continue with the drop operation using the URL of the uploaded copy rather than the object that was originally dropped.

Drop resolvers can also be used to customize what happens when relations are created by dropping existing content items into other content items' relation drop zones. You can use a drop resolver, for example, to copy an image dragged from a foreign publication into the publication where it is being dropped, so that the resulting dropped image is not cross-published.

### 4.3.1 Configuring Drop Resolvers in CUE

All you need to do make CUE call a drop resolver is add a few entries to the CUE configuration file, `/etc/escentic/cue-web/config.yml`. Open this file for editing. If it does not already contain a `dropTriggers` entry, then add one:

```
dropTriggers:
```

Underneath this entry, you can add sub-entries for all the drop resolvers you want to define. A drop resolver configuration contains the following entries:

```
- name: resolver-name
  href: http://host:port/service-url
  resultMimeType: mime-type
  attributes:
    custom-resolver-attribute
    ...
  triggers:
    trigger-specification
```

where:



**name**

Is the name of the resolver. The name must be unique since CUE identifies the resolvers by their names. Any resolver definition with a duplicate name will be ignored.

**href**

Is the URI of the resolver service. The resolver service **can** run in a different domain from CUE, but will then need to be specified as an **authorized endpoint** in order to be granted access to CUE's endpoints (see [section 4.2.1.2](#)).

**resultMimeType**

Is a CUE MIME type, identifying the type of the content returned from the drop resolver.

**attributes**

Is an optional property that you can use to send custom parameters to the drop resolver. For example:

```
attributes:
  my-resolver-param-1: "value1"
  my-resolver-param-2: "value2"
```

In general, you can choose any names that you like for these attributes. There is, however, one attribute name that you must avoid, since it is reserved by CUE. This reserved name is **serviceUri**. In the situation where the MIME type of a dropped object is supported by more than one content type in the publication, CUE can automatically display a dialog asking the user to choose which content type to use for the dropped object. CUE then creates a **serviceUri** attribute from the name of the selected content type (for example **http://content-store-host/web-service/publication/publication-name/binary/content-type-name**) and adds it to the object passed to the drop resolver. This content type selection dialog is only displayed if CUE is configured to display it (see [section 2.2.5](#)).

**triggers**

A specification of the conditions that will trigger CUE to send a dropped object to the drop resolver. The specification can either consist of an array of MIME types or an array of URI patterns (but not both). For example:

```
triggers:
  mimeType: [mime-type, ...]
```

or:

```
triggers:
  urlPatterns: [url-pattern, ...]
```

If **mimeType** is specified, then the drop resolver will be called whenever an object with a MIME type that matches one of the specified MIME types is dropped into CUE.

If **urlPatterns** is specified, then the drop resolver will be called whenever an object with a URL that matches one of the specified URL patterns is dropped into CUE.

Here is an example configuration for a Google image import drop resolver:

```
dropTriggers:
- name: "GoogleImageImport"
  href: "http://my-server/GoogleImageImport"
  resultMimeType: "x-ece/picture"
  triggers:
    urlPatterns: ['^https?:\\/\www\.google\.\.*\imgres\?.*']
```

This configuration will cause CUE to forward any dropped object with a URL that matches the regular expression `^https?:\/\/\www\.google\.\.*/imgres\?.*` to the drop resolver `http://my-server/GoogleImageImport`.

### 4.3.2 Drop Resolver Parameters

When CUE triggers a drop resolver, it passes an object to the resolver containing the following parameters:

**uri**

The URI of the dropped object.

**endpoints**

CUE's configured endpoints (one or more of Content Store, CCI Newsgate and the bridge).

**attributes**

Any parameters supplied in the **attributes** object of the drop resolver configuration (see [section 4.3.1](#)).

**accessTokens**

Access tokens that can be used to authenticate any requests the drop resolver sends to CUE's endpoints. These access tokens will only be passed to the drop resolver if it:

- Either belongs to the same domain as CUE
- Or is listed as an **authorized endpoint** (see [section 4.3.1](#)).

**context**

The context of the drop operation. For an CUE publication, this consists of the following structure:

```
publication:
  name: publication-name
  uri: publication-uri
```

where:

- *publication-name* is the name of the CUE publication in which the drop operation occurred.
- *publication-uri* is the URI of the CUE publication in which the drop operation occurred.

For a Newsgate publication, the context is:

```
storyId: story-folder-id
```

where *story-folder-id* is the ID of the story folder in which the drop operation occurred.

### 4.3.3 Drop Resolver Return Values

A drop resolver must return one of the following HTTP responses on termination:

**HTTP 200 (OK)**

The drop resolver has terminated successfully. The body of the HTTP response must contain the URI of a resource in the Content Store or CCI Newsgate back end (usually this will be an imported version of the object that was dropped by the user). CUE will then complete the drop operation using the supplied URI.

**HTTP 204 (No Content)**

The drop resolver has terminated successfully, but does not have a URI to return. CUE will then:

- If the dropped object was external, abandon the drop operation.
- If the dropped object was an CUE content item, complete the drop operation with the original content item.

#### **HTTP 4xx Or 5xx**

An error of some kind has occurred. The body of the HTTP response must contain an error message text. CUE will then abandon the drop operation and display the supplied error message in a dialog box.

## 4.4 URL-based Content Creation

CUE lets you create a draft content item by simply passing a URL to a browser. A script running in some other application such as Trello, Google Sheets or Slack can simply construct a CUE URL containing the details of a new content item and pass the URL to a browser. CUE will then start in the browser and create the requested content item, ready for the user to continue editing (if required), save and publish.

If the user is currently logged in to CUE then the new content item is created immediately. If the user is not logged in, then the CUE login screen is displayed in the browser. Once the user has logged in, the content item is created.

### 4.4.1 Content Creation URL Structure

A content creation URL must have the following overall structure:

```
| https://your-cue-host/cue-web/#/main?parameter-list"
```

where *your-cue-host* is the host name (and possibly the port number) of your CUE host and *parameter-list* is a sequence of three URL parameters separated by **&** characters:

```
| uri=source-id&mimetype=mime-type&extra=content-definition
```

These parameters must contain the following values:

#### **uri=source-id**

A source ID is a unique string used to identify a content item. The [section 4.4.2](#) generates an ID from the current date and time, but you can use whatever method you choose to supply a unique string.

#### **mimetype=mime-type**

You must specify the MIME type **x-ecel/new-content**; **type=story**.

#### **extra=content-definition**

*content-definition* is a JSON value defining the content you want to create. The structure of the JSON data depends on whether you want to create a [storyline container \(section 4.4.1.1\)](#) or a [classic XHTML-based story \(section 4.4.1.2\)](#).

Note that all the field names and values in the JSON structure **must** be enclosed in quotes, otherwise the URL will not be accepted by CUE.

The values of the three URL parameters must all be [URL-encoded](#).

#### 4.4.1.1 Defining a Storyline Container

To create a storyline container, your **extra** parameter JSON data must be structured as follows:

```
{
  "container": true,
  "containerSlug": "container-slug",
  "homePublication": "publication-name",
  "modelURI": {
    "$class": "URI",
    "string": "model-uri"
  },
  "storyElements": story-elements,
  "tags": tag-references
}
```

where:

##### ***container-slug***

Is an optional slug for the container. If you do not want to supply a slug, then omit the entire **containerSlug** field.

##### ***publication-name***

Is the name of the publication in which the container is to be created.

##### ***model-uri***

Is the web service URI of the content model for the storyline type you want to create.

##### ***story-elements***

Is an array of field definitions defining the content you want to insert into the new storyline, for example:

```
"storyElements": [
  {
    "storyElement": "headline",
    "value": "Example Headline"
  },
  {
    "storyElement": "lead_text",
    "value": "Example lead text"
  },
  {
    "storyElement": "paragraph",
    "value": "Example para"
  },
  {
    "storyElement": "interview",
    "elements": [
      {
        "storyElement": "interview_question",
        "value": "Example question"
      }
    ]
  }
]
```

The field definitions in the array must:

- Comply with the content model referenced in the **modelURI** field.

- Be either plain text story elements such as **headline** and **paragraph**, or complex story elements such as **interview** (that must also only contain plain text story elements). Graphic / video story elements are not supported.

### ***tag-references***

Is an optional array of tag references to be added to the story (see [section 4.4.1.3](#) for details). If you do not want to tag the story, then omit the entire **tags** field.

#### **4.4.1.2 Defining a Classic Story**

To create a classic XHTML-based story, your **extra** parameter JSON data must be structured as follows:

```
{
  "modelURI": {
    "string": "model-uri",
    "$class": "URI"
  },
  "homeSectionUri": "home-section-uri",
  "values": content-item-field-values,
  "tags": tag-references
}
```

where:

#### ***model-uri***

Is the web service URI of the content model for the content item you want to create.

#### ***home-section-uri***

Is the web service URI of the section to which you want to add the new content item.

#### ***values***

Is an object containing a series of field values defining the content you want to add to the new content item. You can leave this object empty if you don't want any of the fields in the new content item to be predefined, for example:

```
"values": {}
```

The fields must be identified by their names as specified in the CUE **content-type** resource, not by the labels displayed in CUE. To predefined values for the **title** and **body** fields of a content item, you would need to specify:

```
"values": {
  "title": "This is the title",
  "body": "<p>This is the body.</p>"
}
```

### ***tag-references***

Is an optional array of tag references to be added to the story (see [section 4.4.1.3](#) for details). If you do not want to tag the story, then omit the entire **tags** field.

#### **4.4.1.3 Tagging the Story**

You can add tags to both storylines and classic stories by including a **tags** containing an array of tag references like this:

```
"tags": [
  {
```

```

    "$class": "URI",
    "string": "https://host/webservice/escenic/classification/tag/
tag:entity@escenic.com,2017:iPad"
  },
  {
    "$class": "URI",
    "string": "https://host/webservice/escenic/classification/tag/
tag:location@escenic.com,2017:home"
  }
]

```

The tag references must be URLs referencing tags that are already defined in the Content Store - it is not possible to create new tags.

#### 4.4.2 Example Script

The following example bash script shows how to construct a content creation URL and submit it to CUE. It creates a storyline container with tags.

```

#!/bin/bash
urlencode() {
  # urlencode <string>
  old_lc_collate=$LC_COLLATE
  LC_COLLATE=C
  local length=${#1}
  for (( i = 0; i < length; i++ )); do
    local c=${1:i:1}
    case $c in
      [a-zA-Z0-9._~_-]) printf "%c" ;;
      *) printf '%%%02X' "'$c" ;;
    esac
  done
  LC_COLLATE=$old_lc_collate
}
cue="https://your-cue-host/cue-web"
webservice="https://your-escenic-webservice-host/webservice"
homepublication="publication-name"
homesection="$webservice/escenic/section/section-id"
modeluri="$webservice/escenic/shared/model/container/regular-news-story"
mimetype="x-ece/new-content; type=story"
sourceid=`date '+%y%m%d-%H%M%S'`
container=true
containerslug="test container slug"

storyElements="[{"storyElement": {"headline", "value": "example plain text"},
  {"storyElement": {"image", "value": "example plain text"}}, {"storyElement": {"paragraph", "value": "example plain text"}, {"storyElement": {"paragraph", "value": "example plain text"}, {"storyElement": {"interview", "elements": [{"storyElement": {"interview_question", "value": "example plain text"}}]}}]"

tags="[{"$class": "URI", "string": "https://host/webservice/escenic/classification/tag/tag:entity@escenic.com,2017:iPad"}, {"$class": "URI", "string": "https://host/webservice/escenic/classification/tag/tag:nationality@escenic.com,2017:German"}]"

```

```
extra="{\"modelURI\":{\"string\":\"\${modeluri}\",\"class\":\"URI\"},
\"homePublication\":\"\${homepublication}\", \"container\":\"\${container}\",
\"storyElements\": \${storyElements}, \"containerSlug\":\"\${containerslug}\", \"tags
\": \${tags}}"

url=$cue/#/main?uri=$(urlencode "$sourceid")\&mimetype=$(urlencode
"$mimetype")\&extra=$(urlencode "$extra")

google-chrome $url &
```

If you edit this script to match your installation, then running it should start the Chrome browser and create a draft content item. You would need to replace *your-cue-host* and *your-escenic-webservice-host* with the correct host names, replace *homepublication* with the name of one of your publications, and replace *section-id* with the ID of a section in that publication before running it. Otherwise, as long as you have a content type called **regular-new-story**, with a suitable content model, it should work.

## 4.5 URL-based Content Editing

CUE lets you open a content item by simply passing a URL to a browser. A script running in some other application such as Trello, Google Sheets or Slack can simply construct a CUE URL containing ID of an existing content item and pass the URL to a browser. CUE will then start in the browser and open the requested content item, ready for the user to continue editing (if required), save and publish.

If the user is currently logged in to CUE then the new content item is opened immediately. If the user is not logged in, then the CUE login screen is displayed in the browser. Once the user has logged in, the content item is opened.

### 4.5.1 Content Editing URL Structure

A content editing URL must have the following overall structure:

```
http://your-cue-host/cue-web/#/main?escenicid=content-item
```

where:

- *your-cue-host* is the host name (and possibly the port number) of your CUE host
- *content-item* is the ID of the content item to be edited. The ID can be supplied in the following three forms:

- Just the ID itself. For example:

```
http://mycueserver.com:81/cue-web/#/main?escenicid=1234
```

- As a Content Store web service URL, specified relative to the CUE installation's **escenic** end point:

```
http://mycueserver.com:81/cue-web/#/main?escenicid=escenic/content/1234
```

- As a complete Content Store web service URL:

```
http://mycueserver.com:81/cue-web/#/main?escenicid=http://
mycontentstore.com:8080/webservice/escenic/content/1234
```

## 4.6 Logout Triggers

A logout trigger is a simple HTTP **GET** request that is sent to a specified URL when the user logs out from CUE. It provides a mechanism for integrators to automatically perform other actions (such as logging out of a VPN) on logout from CUE. You can define multiple logout triggers. In this case, a **GET** request will be sent to each specified URL when the user logs out.

The CUE logout process does not wait for any response from the defined trigger URLs – it simply makes the requests and then performs the logout operation.

To define logout triggers:

1. If necessary, switch user to **root**.

```
| $ sudo su
```

2. Open **/etc/escenic/cue-web/config.yml** for editing. For example:

```
| # nano /etc/escenic/cue-web/config.yml
```

3. Add a **logoutTriggers** property containing a list of trigger URLs to which **GET** requests are to be sent:

```
|   logoutTriggers:
|     - http://my-vpn-service/logout
|     - http://my-other-service/logout
```

4. Save the file.

5. Enter:


```
| # dpkg-reconfigure cue-web-3.16
```

This reconfigures CUE with the changes you have made.

## 4.7 CUE Safe Mode

CUE safe mode lets you easily disable some or all extensions of you have installed to help you track down the cause of any problems that may arise.

To disable all web components, enrichment services and drop resolvers you have installed:

1. Open the **Settings** panel by selecting the  panel button (on the left).
2. Double-click **System Settings and Information** to display the **System Settings** page.



3. Check the Enable safe mode option at the top of the page:

**CUE**

### System Settings

**Safe Mode**

Manual control of custom components  Enable Safe Mode

**Enrichment Services**  Enable for all

kitchen sink  Enable

Edit Summary Fields  Enable

Use Wire Service  Enable

**Web Components**  Enable for all

Twitter Timelines  Enable

Image Search  Enable

General info  Enable

Text Modification  Enable

Storyline Stat  Enable

4. Select **Apply Changes**.
5. Refresh any open CUE browser tabs to see the effects of the change.

All extensions are now disabled. You can re-enable extensions individually or in groups by selecting the individual check boxes on the rest of the page, selecting **Apply Changes** and once again refreshing any open CUE browser tabs. By enabling and disabling extensions and then testing you can often determine whether or not an extension is the cause of your problem and if so, which one.

## 4.8 Custom Capabilities (Content Store only)

A CUE **capability** is a unit of CUE functionality that can be enabled or disabled for individual users. This makes it possible for different users to see different versions of CUE, customized to match their **role**. A house journalist, for example, may be granted access to different functionality from an editor or a freelance journalist. All the standard side panels and metadata panels in CUE are defined as capabilities and can therefore be either hidden or shown based on a user's role.

If you have extended CUE with your own web components, then you can also define custom capabilities that will allow them to be enabled and disabled in the same way as the built-in functionality. A custom capability in CUE is simply a name that you assign to a web component by setting a property in its configuration file. Here, for example, is a side panel web component configuration that includes a capability definition:

```
sidePanels:
  - id: "twitter-home-panel"
    name: "Twitter Timelines"
    directive: "cue-custom-panel-loader"
    isAngular: true
    webComponent:
      modulePath: "webcomponents/twitter/twitter-home-panel.js"
      icon: "twitter-home-panel-icon"
    mimeTypes: []
    homeScreen: true
```

```
metadata: []
active: false
order: 705
capability: "twitter-panel"
```

Here is a metadata panel web component configuration with a capability definition:

```
editors:
  metadata:
    - name: "storyline-stat"
      directive: "storyline-stat"
      cssClass: "storyline-stat"
      title: "Storyline Stat" #translate
      webComponent:
        modulePath: "webcomponents/storyline/storyline-stat.js"
        icon: "storyline-stat-icon"
      mimeTypes: ["x-ece/story", "x-ece/new-content; type=story"]
      order: 731
      capability: "storyline-stat"
```

You can add a **capability** property like this to any side panel or metadata panel configuration (but not to a custom field editor configuration). Your capability name must not clash with any of the built-in capability names. All the built-in capabilities have names that start with **cue-**, so just avoid this prefix in your names. If you have a group of extensions that are so closely related that they can be seen as a single piece of functionality, then you can give them all the same capability name: it will then be possible to enable/disable them as a group.

The management of user access to CUE capabilities is a Content Store responsibility, so once you have defined your custom capabilities in CUE, you will need to add some corresponding configurations to the Content Store and then grant selected users access to the capabilities using Web Studio. For information about how to do these things, see [Capability Definitions](#) and [Capabilities](#).

## 5 DC-X Integration

This section provides a preliminary description of the DC-X extension for CUE. DC-X is a Digital Asset Management system offering simple but sophisticated functionality for the creation, management and storage of digital assets such as text, images, video and audio.

The main components of the CUE DC-X extension are:

- A DC-X side panel in CUE that allows users to search DC-X for images, videos and so on, for import into CUE.
- A related **drop resolver** (an HTTP service that reacts to objects dropped into CUE). This drop resolver listens for drops of DC-X resources, dragged either from the DC-X side panel or from a DC-X client running on the same device as CUE. On detecting such an event, the drop resolver imports the dropped resource(s) into CUE. For general information about drop resolvers, see [section 4.3](#).
- A DC-X Wires side panel in CUE that allows users to browse and search for wire stories in DC-X, and import selected wire stories for use in CUE.

The extension includes three features that depend on the use of the CUE Zipline extension. These are:

- Reporting back to DC-X on the use of DC-X assets in CUE
- Copying back to DC-X assets that are uploaded to CUE
- Import of wire stories from DC-X

### 5.1 DC-X Drop Resolver Installation

The DC-X extension requires the installation of the DC-X drop resolver. Currently this component is not available for download via the usual channels. Contact Stibo DX through your sales or support representative to obtain a copy of the DC-X drop resolver and instructions on how to install it.

### 5.2 DC-X Extension Configuration

These instructions are based on the assumption that the DCX drop trigger has been installed and is accessible from CUE.

1. Switch user to **root** (if necessary)  
| `$ sudo su`
2. Open `/etc/escenic/cue-web/config.yml` for editing. For example  
| `# nano /etc/escenic/cue-web/config.yml`
3. Edit the file as described in the following sections.
4. Save the file.
5. Enter the following to reconfigure CUE:  
| `# dpkg-reconfigure cue-web-3.16`

## 5.2.1 Endpoint Configuration

The DC-X system with which CUE is to communicate must be configured as an endpoint, in the same way as the Content Store and CUE Print back ends. Add the URI of the DC-X endpoint as a new property under **endpoints**. For example:

```
endpoints:
  escenic: "http://escenic-host:81/webservice/index.xml"
  newsgate: "http://newsgate-host/newsgate-cf/"
  dc-x: "http://dcx-host/dcx/"
```

## 5.2.2 Side Panel Configuration

Add **endpointServices**, **sidePanels** and **useDCXWirePanel** configurations, which should look something like this:

```
endpointServices:
  dc-x:
    - serviceName: "dcx-login.service"

sidePanels:
  - id: "dcx-assets"
    isAngular: true
    name: "DC-X Assets" #translate
    cssClass: "dcx dcx-assets"
    directive: "dam-datasource"
    mimeTypes: ['x-ece/story', 'x-ece/container', 'x-ece/new-content; type=story', 'x-
ece/event', 'x-ece/gallery',]
    homeScreen: false
    active: true
    requires: ["dc-x"]
    order: 302
    attributes:
      dcxChannels: ['ch020dcxsystempoolapict', 'ch060dcxsystempoolvideo',
'ch050dcxsystempoolnative']

  - id: 'dcx-wires'
    name: 'DC-X Wires' #translate
    directive: 'dam-datasource'
    homeScreen: true
    order: 301
    attributes:
      dcxChannels: ['channel_pool_story']
      updateInterval: 30

useDCXWirePanel: true
```

The DC-X Wire panel configuration is optional. If you do not intend to make use of DC-X wires from CUE, then you can omit the **dcx-wires** side panel configuration and the **useDCXWirePanel** property (or set **useDCXWirePanel** to false).

There is an example configuration file called **DCX.yml** included in the CUE distribution that you can copy and uncomment.

Make sure that the **dcxChannels** property is correctly set. This property must contain a list of the DC-X channels that CUE should have access to. You must use the channel IDs to specify the channels, not their names.

You can use the **updateInterval** property to specify how frequently the DC-X Wire panel is to be updated. The interval is specified in seconds.

### 5.2.3 Drop Resolver Configuration

Add a drop resolver (or drop trigger) configuration, which should look something like this:

```
dropTriggers:
  - name: DCXToContentStoreImport
    href: "http://drop-resolver-host/DCXToContentStoreImport"
    triggers:
      mimeTypees: ['x-dcx/image', 'x-dcx/postscript', 'x-dcx/illustrator', 'x-dcx/pdf',
        'x-dcx/video']
      urlPatterns: ['^https?:\\//dcx-host\\dcx\\api\\document\\.*', '^https?:\\//dcx-
        host\\dcc\\document\\.*']
      resultMimeType:
        - sources: ['x-dcx/image', 'x-dcx/postscript', 'x-dcx/illustrator', 'x-dcx/
        pdf']
          results: ["x-ece/picture"]
        - sources: ["x-dcx/video"]
          results: ["x-ece/video"]
        - sources: ['^https?:\\//dcx-host\\dcx\\api\\document\\.*', '^https?:\\//dcx-
        host\\dcc\\document\\.*']
          results: ["x-ece/picture", "x-ece/video"]
    attributes:
      defaultState:
        - dcxTypes: ["image", "postscript", "illustrator", "pdf", "video"]
          state: "published"
      fieldMapping:
        - dcxTypes: ["image", "postscript", "illustrator", "pdf"]
          mapping:
            - name: "title"
              value:
                - dcxField: "Filename"
            - name: "caption"
              value:
                - dcxField: "body"
                - dcxField: "Headline"
            - name: "credit"
              value:
                - dcxField: "Provider"
            - name: "byline"
              value:
                - dcxField: "Creator"
        - dcxTypes: ['video']
          mapping:
            - name: "title"
              value:
                - dcxField: "Filename"
            - name: "description"
              value:
                - dcxField: "body"
                - dcxField: "Headline"
            - name: "credit"
              value:
                - dcxField: "Provider"
            - name: "byline"
              value:
                - dcxField: "Creator"
```

where *drop-resolver-host* and *dcx-host* are replaced by the appropriate host names.

The **defaultState** attribute determines which states will be assigned to dropped assets. You can differentiate the dropped assets by their DC-X type and assign different states to different types of asset. In the example shown above, all types of asset are assigned the state **published**.

The **fieldMapping** defines how DC-X field values are to be mapped on to Content Store fields. You can differentiate the dropped assets by their DC-X type and set different field mappings for different types of asset. In the example shown above, video assets are assigned different field mappings from the other asset types. You can map several DC-X fields onto one Content Store field:

```
- name: "description"
  value:
    - dcxField: "body"
    - dcxField: "Headline"
```

In this case the DC-X fields must be specified in priority order: the first non-empty field is used to fill the Content Store field.

## 5.2.4 Content Type Configuration

In order to support the automatic upload of assets from the Content Store to DC-X, a special field must be added to the definition of all relevant content types (i.e, graphics and video content types). This field is used to hold DC-X status information about the content items / assets. The field can have any name, but it must satisfy the following requirements:

- Be a **basic** field with the MIME type **application/json**
- Be read-only
- Be identified as a DC-X status field with a **ui:dam-status** child element

It should also ideally be hidden. Here is an example of a suitable field definition:

```
<field name="_dam_status" type="basic" mime-type="application/json">
  <ui:hidden/>
  <ui:read-only/>
  <ui:dam-status/>
</field>
```

## 5.2.5 Zipline Configuration

CUE Zipline is a CUE extension that was originally designed to support CUE Print integration, but is now also used to support DC-X integration. For general information about CUE Zipline and how to configure it, see [here](#).

CUE Zipline supports data transfer between DC-X and CUE. Specifically, it enables the transfer of:

- Reports on the use of DC-X assets in CUE
- Assets uploaded to CUE
- Wire stories from DC-X to CUE

To make CUE Zipline support basic DC-X integration (excluding the import of wire stories), add a DC-X processor definition to the **processors** entry in the CUE Zipline configuration file:

```
processors:
  ...
  # Reporting usage information for DC assets
  - type: dcx
    endpoint:
      # URL of DCX API endpoint
      url: dcx-integration-endpoint
      user: dcx-integration-user
      password: dcx-integration-password
    cache:
      # Override default cache capacity (10000)
      max_size: dcx-integration-cachesize

  # Base URL of CUE web installation (e.g., http://server:port/cue-web/)
  cue_web: cue-web-endpoint

  # Configuration of usage info block
  info:
    view:
      label: View
      link_text: Browse
    edit:
      label: Edit
      link_text: Open in CUE
    upload:
      upload-configuration
```

The DC-X processor definition contains the following properties:

**type**

Must be set to **dcx**.

**endpoint**

Must contain properties specifying the DC-X endpoint URL plus a valid DC-X user name and password.

**cache**

May optionally be used to specify cache settings.

**cue\_web**

Must contain the CUE endpoint URL.

**upload**

May optionally be used to define what kinds of uploaded assets should be copied back to DC-X, and how they should be handled by DC-X. For details see [section 5.2.5.1](#).

### 5.2.5.1 Upload Configuration

```
processors:
  ...
  - type: dcx
    ...
    upload:
      - filter:
          publications:
            - tomorrow-online
          content-types:
            - picture
            - graphic
          states:
```

```
- approved
- published
content:
  tags:
    - name: Creator
      meta: creator
folder: native
```

The **upload** property controls the upload of assets from the Content Store to DC-X. It can contain a list of upload specifications, each of which consists of the following three properties:

**filter**

This property contains a set of criteria that determine which of the assets uploaded to the Content Store should be copied to DC-X. The criteria are:

**publications**

A list of publication names. Only assets uploaded to these publications will be copied to DC-X.

**content-types**

A list of content type names. Only these content types will be copied to DC-X.

**states**

A list of workflow state names. Only content items in one of these states will be copied to DC-X

**content**

This property contains a **tags** property that defines the mapping between content item fields and DC-X tags. For details, see [section 5.2.5.2](#).

**folder**

This property specifies the DC-X upload folder to use.

### 5.2.5.2 Tag Mapping

The tag mappings specified in a **tags** property consist of:

- A **name** property identifying a DC-X tag
- A second property specifying how the DC-X tag is to be set

The following variations are possible:

- ```
- name: Creator
  field: byline
```

Assign the value of the uploaded content item's **byline** field to the DC-X **Creator** tag.

- ```
- name: Creator
  meta: creator
```

Assign the value of the uploaded content item's **creator** metadata field to the DC-X **Creator** tag.

- ```
- name: Creator
  first-of:
    - field: byline
    - meta: author
```



```
- meta: creator
```

Read the fields listed under **first-of** in the specified order. Use the first one that contains a value to set the DC-X **Creator** tag.

- ```
- name: body
  template: >
    <p>{{caption}}</p>
  context:
    - name: caption
      field: caption
```

Use the result of executing the specified [Jinja2](#) template to set the DC-X **body** tag. The **context** property can be used to define the variables that will be available to the template. These variables can be assigned values in exactly the same way as values are assigned to DC-X tags. So in this example, the `{{caption}}` variable will be replaced with the content of the uploaded content item's **caption** field.

### 5.2.5.3 Wire Stories Configuration

In order for CUE Zipline to support the import of wire stories from DC-X, you need to add a **dcx-converters** top-level section like this to the CUE Zipline configuration file:

```
dcx-converters:
  # Configuration to convert DC-X wire story to CUE storyline in a container
  wire:
    # Relative path where templates for DC-X wire live
    # template-dir: /etc/cue/zipline/conversion-templates
  target:
    publications:
      - text: Tomorrow Online # Text to show in a label
        value: tomorrow-online # Name of the publication
        containers:
          - text: Regular News Story
            value: regular-news-story
            # Map between CUE container fields and DC-X document fields
            fields:
              - name: Headline
                meta: com.escenic.container.slug
    content-types:
      - text: Storyline
        value: storyline
        # Map between CUE content type fields and DC-X document fields
        fields:
          - name: Title
            meta: title
        # Map between CUE story element type fields and DC-X document fields
        story-elements:
          - value: headline # Name of the story element type
            fields:
              - name: Headline
                meta: headline
          - value: lead_text
            fields:
              - name: SubHeadline
                meta: lead-text
            extract-substrings: False # Attribute used to separate out a DC-X
            XHTML field value
          - value: paragraph
```

```

        fields:
          - name: body
            meta: paragraph
            extract-substrings: True
binary-content-types:
  - text: Picture
    value: picture
    # Map between CUE binary content type fields and DC-X image fields
    fields:
      - name: ImageCaption
        meta: caption
      - name: _display_title
        meta: title
    summary-fields: # Binary content type summary fields
      - name: ImageCaption
        meta: caption
  - text: Binary
    value: binary
    fields:
      - name: _display_title
        meta: title
    summary-fields:
      - name: _display_title
        meta: title
  - text: Graphics
    value: graphics
    fields:
      - name: ImageCaption
        meta: caption
      - name: _display_title
        meta: title
    summary-fields:
      - name: ImageCaption
        meta: caption
# Map between DC-X text formatters and CUE storyline annotations
annotations:
  - name: bold # Name of the css class that DC-X uses to format a text in story
    meta: bold # Name of the annotation that CUE uses to annotate a text in
storyline
  - name: italic
    meta: italic
  - name: underline
    meta: underline

```

The overall purpose of this section is to define:

- Which CUE publications DC-X wire stories may be imported to
- Which containers DC-X wire stories may be imported to
- Mappings between DC-X fields and the fields / story elements in CUE containers and content types
- Mappings between DC-X character styles and CUE annotations

The **dcx-converters/wire/target** property contains the following settings:

**publications**

Contains an array of entries, one for each supported publication. Each array entry can contain the following properties:

**text**

The display name (label) of a CUE publication.

**value**

The internal name of the publication.

**containers**

An array of entries, one for each container that may be used for imported wire stories. Each array entry can contain the following properties:

**text**

The display name (label) of the container.

**value**

The internal name of the container.

**fields**

An array of entries, one for each container field that is to be used. Each array entry can contain the following properties:

**name**

The name of a DC-X field to be imported.

**meta**

The name of the container field into which the content of **name** is to be imported.

**content-types**

An array of entries, one for each content type that may be used for imported wire stories. Each array entry can contain the following properties:

**text**

The display name (label) of the content type.

**value**

The internal name of the content type.

**fields**

An array of entries, one for each content type field that is to be used. Each array entry can contain the following properties:

**name**

The name of a DC-X field to be imported.

**meta**

The name of the content type field into which the content of **name** is to be imported.

**story-elements**

An array of entries, one for each story element in this content type that is to be used. Each array entry can contain the following properties:

**value**

The internal name of the story element.

**fields**

An array of entries, one for each story element field that is to be used. Each array entry can contain the following properties:

**name**

The name of a DC-X field to be imported.

**meta**

The name of the story element field into which the content of **name** is to be imported.

**extract-substrings**

If the source DC-X field contains rich text (XHTML), then setting this property to **True**, instructs CUE Zipline to extract the text content of each block element in the source field and create a separate story element of this type for each of them. In other words, you can use it to transform a sequence of **p** elements in the source field to a corresponding sequence of **paragraph** story elements in CUE.

**binary-content-types**

An array of entries, one for each binary content type that may be used for imported wire stories. Each array entry can contain the following properties:

**text**

The display name (label) of the content type.

**value**

The internal name of the content type.

**fields**

An array of entries, one for each content type field that is to be used. Each array entry can contain the following properties:

**name**

The name of a DC-X field to be imported.

**meta**

The name of the content type field into which the content of **name** is to be imported.

**summary-fields**

An array of entries, one for each summary field that is to be used. Each array entry can contain the following properties:

**name**

The name of a DC-X field to be imported.

**meta**

The name of the summary field into which the content of **name** is to be imported.

**annotations**

An array of entries containing mappings between DC-X CSS classes and CUE storyline annotations. Note that these mappings are global, not publication-specific. Each array entry must contain the following properties:

**name**

The name of a CSS class used in DC-X

**meta**

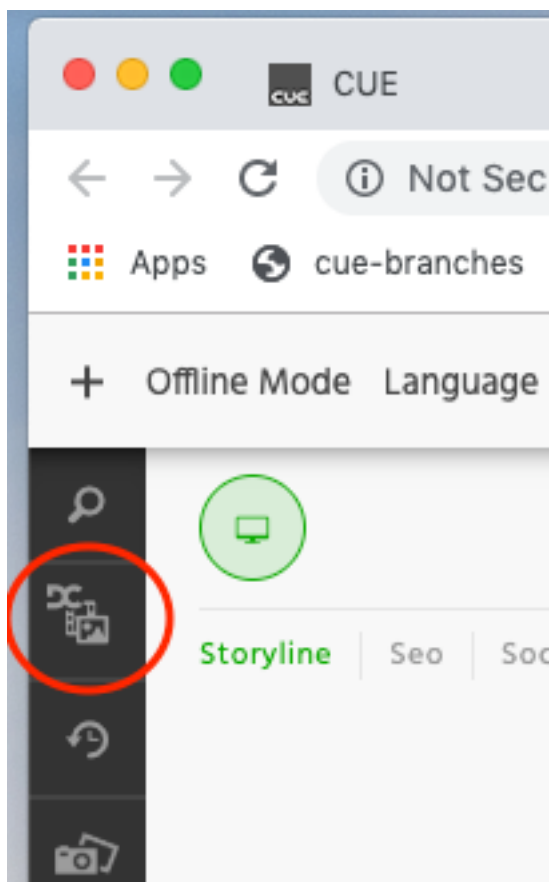
The name of a corresponding CUE annotation.

## 5.3 Login Credentials

The DC-X extension uses the credentials supplied when you log in to CUE as login credentials for DC-X as well. You must therefore use the same username/password combinations in both systems for the DC-X extension to work.

## 5.4 Using The Main DC-X Integration

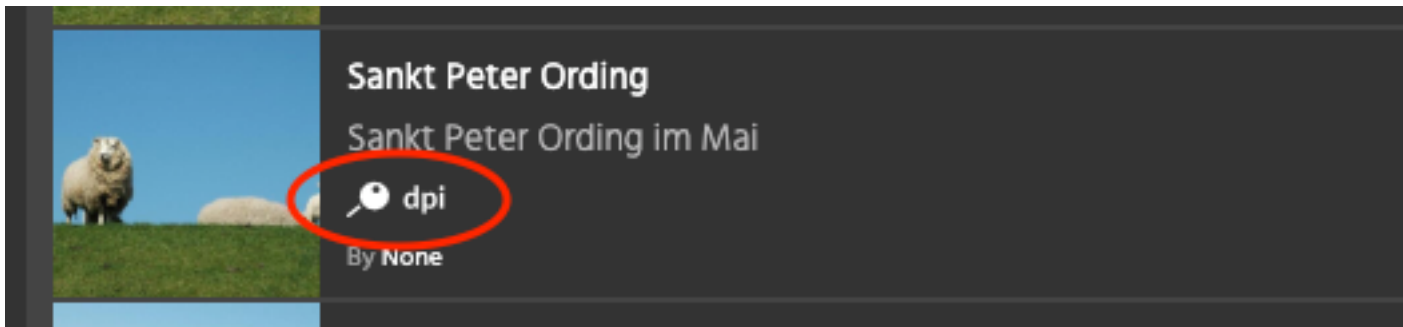
Once the DC-X integration is correctly installed and configured, DC-X appears as a secondary search panel on the left side of the CUE window:



When expanded, the DC-X panel looks and behaves like the main CUE search panel with the following differences:

- It offers access to DC-X assets, rather than ordinary CUE content
- The search and filtering options reflect the search and filtering functionality provided by DC-X

Each DC-X asset is represented in the panel search results by an entry consisting of a thumbnail preview image, a title, description, owner and a series of DCX **flags** describing attributes of the asset such as its resolution, whether it is an online-only asset, whether it is expensive and so on:



To use DC-X assets in a publication, you simply drag them from the search results list and drop them in an appropriate location in a content item, in exactly the same way as you would drop images and videos selected from the ordinary search panel. When an asset is dropped in this way, a content item of the appropriate type is created for it, and a copy of the binary object is stored in the Content Store. The copied object is marked to indicate it is a DC-X asset, and the new content item is set to a state defined in the DC-X drop trigger configuration (or **draft** if the drop trigger configuration does not specify a state). If the dropped asset has been dropped before and already exists as a content item in CUE, then that content item is used and its state is not modified.

If you have a DC-X client running on the same machine as CUE, you can also drag assets directly from the DC-X client into CUE. Assets added in this way behave in exactly the same way as assets dragged from the DC-X search panel in CUE.

Information about the use of DC-X assets in CUE is reported back to DC-X. The usage information is recorded in DC-X as follows:

- Assets related to a published story are assigned a usage entry with the state **Published**, along with the publication date and URL of the story.
- Assets related to a story that is not published (or to the working copy of a published story) are assigned a usage entry with the state **Planned**.

When assets are removed from a story, the corresponding usage entry is removed from DC-X. This usage information is displayed in the DC-X **Usage** tab.

Depending on how the system has been configured, video and graphic content uploaded to CUE from other sources may be automatically passed on to DC-X for storage. The system can be configured to only store certain content types in DC-X, and only if they are added to specific content items. In addition, they may not actually be uploaded to DC-X until they reach a specified state in the workflow.

CUE's **General Info** metadata panel section contains a **DAM info** field that shows information about the current status of content items that meet the requirements for upload to DC-X. It can contain the following status messages:

**Upload not initiated**

The content item has not yet reached the workflow state that triggers upload.

**Uploading**

Upload is in progress.

**error-code**

Upload failed.

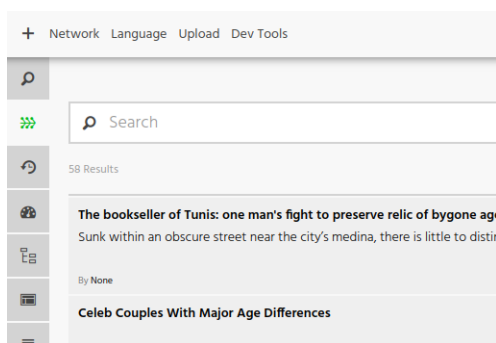
**dex-document-id (a long random string)**

Upload has succeeded.

## 5.5 Using The DC-X Wire Integration

The DC-X wire integration makes it possible to use DC-X as a source of wire stories for use in CUE publications. Wire stories managed by DC-X can be browsed and searched from CUE, and selected for use in CUE publications. Selected wire stories are imported into CUE and can then be edited and published to multiple channels in the normal way.

If CUE and CUE Zipline have been configured to support the use of DC-X wires, then you will see a button for a DC-X Wires panel on the left side of the CUE window:



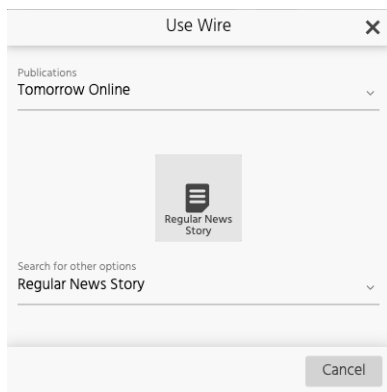
When expanded, the DC-X Wires panel looks and behaves like the main CUE search panel with the following differences:

- It offers access to wire stories in DC-X, rather than ordinary CUE content
- The search and filtering options reflect the search and filtering functionality provided by DC-X

You can access a listed wire story in three different ways:

- Double-click to open it in DC-X
- Press the space bar to display a quick view of the story in CUE
- Right click to display the context menu and select **Use Wire** to import the story into CUE.

Selecting **Use Wire** usually displays the following dialog, allowing you to choose what kind of story to create, and in which publication:



This dialog is effectively the same as the **Create new** dialog, and works in the same way. The values available for selection may, however, be more constrained - you may not be able to import wires into all of the publications that you are allowed to create stories in, for example, and you may not have the same choice of containers/content types. The values available for each of these options are determined by the configuration settings specified in the `dcx-converters/wire/target` section of the CUE Zipline configuration file (see [section 5.2.5.3](#)).

The Use Wire Service dialog shown above is only displayed if it is actually needed (that is, if there are actual choices to be made). If the configuration in the `dcx-converters/wire/target` section of the CUE Zipline configuration file only specifies one publication and container type, then it will not be displayed and the wire story will be imported to the configured destination immediately. Note also that the **Publication** option in the dialog will be automatically set to your default publication if possible (that is, of you have specified a default publication in your personal preferences and if that publication is one of the publications specified in the CUE Zipline wire service configuration).

Any binary resources such as images or videos that are referenced in the wire story are imported along with it as related content.

Once a wire story has been imported it can be used in exactly the same way as any other CUE content item.