CUE

User Guide

3.3.5-3

Table of Contents

1 Introduction

Welcome to CUE, CCI Europe's new browser-based application for newsroom staff! CUE is now CUE's primary interface for all newsroom staff, and will eventually completely replace Content Studio. CUE is a joint CCI-CCI Europe project and is capable of acting as an interface to CCI Newsgate systems as well as CUE systems, so for newsrooms that use both CCI Newsgate and CUE, it provides a highly productive integrated interface to both systems.

You can use CUE to:

- Create and edit content
- Upload images and other binary content
- Insert in-line images into content items
- Drag and drop / cut and paste content from external systems such as Microsoft Word, Excel, browsers etc.
- Add relations to content items
- Edit sections
- Edit section pages
- Desk content items on section pages
- Tag content items
- Add content items to sections
- Edit lists and inboxes
- Publish content items
- etc...

CUE can in addition easily be extended with custom functionality to meet special workflow requirements, make use of external services and integrate with existing in-house systems. For more about this, see chapter 6.

> For information about using CUE with CCI Newsgate, please see "CUE-related CCI NewsGate Functionality for CUE 2.5", which you will find at http://customer.ccieurope.com/documentation/release-notes/cci-newsgate.aspx (log in using your CCI Customer Portal credentials).

2 Installing CUE

CUE 3.3.5-3 requires version 7.3 of the CUE Content Store and CUE Print 3.13.0 (3.13.0.2). Contact your customer service manager for availability of corresponding Newsgate versions on previous release tracks.

Before installing CUE, ensure that the back-end systems you are going to use it with have been upgraded accordingly:

- If CUE is to use an CUE back end, make sure your CUE Content Store is upgraded to version 7.3 or higher
- If CUE is to use a CCI Newsgate back end, make sure that Newsgate has been upgraded to a suitable version (see above)

CUE also 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, not an CUE plug-in. 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.

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, add the Escenic **`apt`** repository to your list of sources:

   ```
   # curl --silent http://user:password@apt.escenic.com/repo.key | apt-key add -
   # echo "deb http://user:password@apt.escenic.com stable main non-free" >> /etc/
   apt/sources.list.d/escenic.list
   ```

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

4. 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
   ```

5. Update your package lists:

```
# apt-get update
```

6. Download and install CUE. The recommended way to do this is:

```
# apt-get install cue-web-3.3
```

7. Download and install nginx

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

3 Configuring CUE

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 **`.yml`**. 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-3.3/`** folder. If your CUE installation includes any extensions, these extensions may have their own configurations stored in other locations.

The **`/etc/escenic/cue-web-3.3/`** folder also contains a file called **`config.yml.template`**, containing the property settings that you always need to set when installing CUE. To use this file you rename it to **`config.yml`** and then edit the contents.

To configure CUE:

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

   ```
   $ sudo su
   ```

2. Copy **`/etc/escenic/cue-web-3.3/config.yml.template`** to **`config.yml`**:

   ```
   # cp /etc/escenic/cue-web-3.3/config.yml.template /etc/escenic/cue-web-3.3/
   config.yml
   ```

3. Open the new **`/etc/escenic/cue-web-3.3/config.yml`** for editing. For example

   ```
   # nano /etc/escenic/cue-web-3.3/config.yml
   ```

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-3.3
```

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

8. 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;
}
```

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

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

10. 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.confa
# touch /etc/nginx/conf.d/request-entity-size-limit.conf
```

11. 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/**.

12. Open **/etc/nginx/default-site/webservice.conf** for editing and add the contents described in section 3.1.

13. 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 should now be able to access CUE by opening a browser and going to **http://***host***:81/cue-web**.

3.1 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';
        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.

3.2 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 ad 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.

3.2.1 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`**-3.3).

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-3.3
```

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

3.2.2 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`**-3.3). 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).

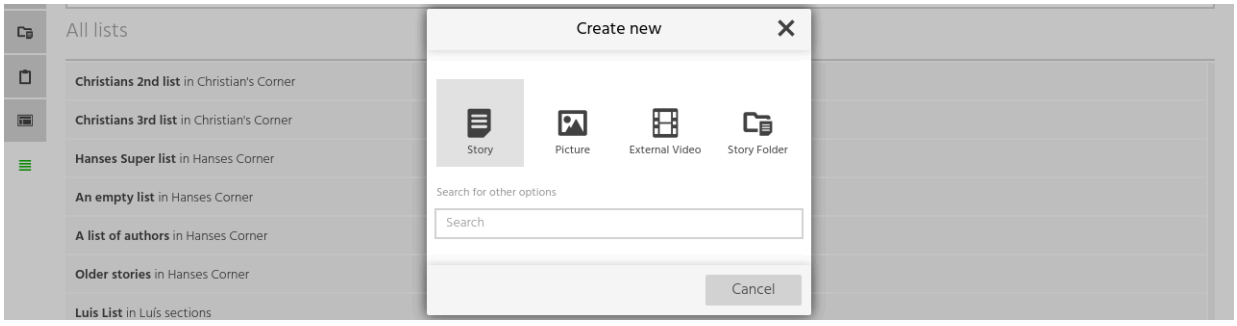
> When setting up Facebook authentication for the Content Store, you create two client IDs – one for desktop clients and one for web clients. Make sure that you use the web client ID for configuring CUE.

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

```
# dpkg-reconfigure cue-web-3.3
```

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

3.3 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-3.3/30-new-content-defaults.yml** for editing. For example

```
# nano /etc/escenic/cue-web-3.3/30-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 3.7, then any icon settings made in the **content-type** resource will override icon settings made here.

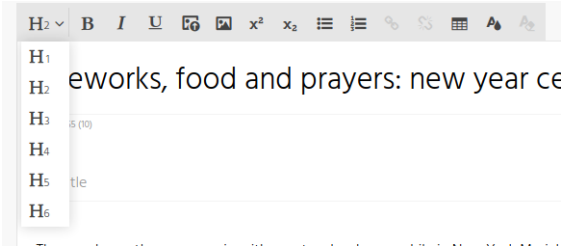
5. Save the file.
6. Enter:

```
# dpkg-reconfigure cue-web-3.3
```

This reconfigures CUE with the changes you have made.

3.4 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-3.3/plugins/internal/EscenicHeading/EscenicHeading.yml** for editing. For example:

   ```
   # nano etc/escenic/cue-web-3.3/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-3.3
   ```

   This reconfigures CUE with the changes you have made.

3.5 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-3.3/config.yml** for editing. For example

   ```
   # nano /etc/escenic/cue-web-3.3/config.yml
   ```

3. Add the following setting:

   ```
   removeDivsAutomatically: true
   ```

4. Save the file.
5. Enter:

   ```
   # dpkg-reconfigure cue-web-3.3
   ```

   This reconfigures CUE with the changes you have made.

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

3.6 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-3.3/config.yml** for editing. For example

```
# nano /etc/escenic/cue-web-3.3/config.yml
```

3. Add the following settings:

```
contentTypeSelection:
  enabled: true
```

4. Save the file.
5. Enter:

```
# dpkg-reconfigure cue-web-3.3
```

   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.

3.7 Defining Custom Content Type Icons

In most places where content items are listed in CUE, the name is accompanied by an icon. In general, the idea is that the icon represents the content item type - so stories are represented by document icons, pictures by image icons, and so on. Only a small number of generic icons are supplied with CUE, however, so in publications with many content types, different types will end up being represented by the same icons. You can, however, define your own icons and associate them with the content types defined in your publication.

To define an icon for a content type, you must add a **ui:icon** element to your publication's **content-type** resource, as a child of the appropriate **content-type** element. Suppose, for example, that you want to add an icon for your Gallery content type: you would then need to insert a **ui:icon** element as a child of the **<content-type name="gallery">** element.

For general information about the **ui:icon** element, see here. You will see that the **ui:icon** supports a number of built-in icons. These icons are, however, not suitable for use as content type icons. Instead you must:

- Make your own icon as a PNG image.
- Save the icon in a folder on the same server as CUE, or else in a location on the network that will accessible to CUE.

- Supply the URI of the PNG file in the **ui:icon** element. if you have placed the icon on the same server as CUE then you can specify a relative URI, for example:

  ```
  <ui:icon>icons/custom-gallery.png</ui:icon>
  ```

  If you have placed the icon somewhere else in the network, then you will need to specify an absolute URI, for example:

  ```
  <ui:icon>http://my-company-server/icons/custom-gallery.png</ui:icon>
  ```

The icon images you create must have the following characteristics:

- PNG format
- Monochrome: black on a transparent background. CUE automatically inverts the colors where necessary.
- 32x32 pixels in size

3.8 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-3.3/config.yml** for editing. For example:

   ```
   # nano /etc/escenic/cue-web-3.3/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. Save the file.

6. Enter:

```
# dpkg-reconfigure cue-web-3.3
```

This reconfigures CUE with the changes you have made.

3.9 Spelling Checker

CUE itself does not include a spelling checker, and depends on whatever spelling checker is provided by the browser it is running in. Chrome includes a built-in spelling checker, and there are also spelling checker plug-ins for Chrome. CUE Print, however **does** include a spelling checker, and CUE can be configured to use CUE Print's spelling checker instead of whatever is available in the browser.

By default, CUE does not make use of the CUE Print spelling checker, even if it is available. To enable it:

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

```
$ sudo su
```

2. Open **`/etc/escenic/cue-web-3.3/config.yml`** for editing. For example:

```
# nano /etc/escenic/cue-web-3.3/config.yml
```

3. Add the following settings:

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

This both enables the CUE Print spelling checker and switches 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.

4. Save the file.
5. Enter:

```
# dpkg-reconfigure cue-web-3.3
```

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.

3.10 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-3.3/config.yml** for editing. For example:

   ```
   # nano /etc/escenic/cue-web-3.3/config.yml
   ```

3. Add the following setting:

   ```
   useModificationTimeOnContentCard = true
   ```

4. Save the file.
5. Enter:

   ```
   # dpkg-reconfigure cue-web-3.3
   ```

   This reconfigures CUE with the change you made.

3.11 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 3.11.1 below.

3.11.1 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-3.3/40-search-filter.yml** for editing. For example:

   ```
   # nano /etc/escenic/cue-web-3.3/40-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-3.3
```

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.

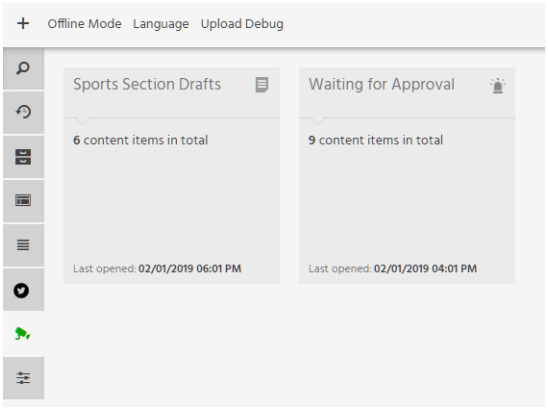
3.12 Source Monitors (Content Store only)

> 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.

Source monitors are constantly updated lists of content items maintained by CUE. They provide an easy way for editors and others to maintain control over the editorial workflow. You can, for example define one source monitor that lists all draft content items in the Sports section, one that lists all image content items in the approved state and another that lists all content items with a particular combination of tags. You can define any number of source monitors.

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

All the source monitors you define can be displayed on a dashboard by selecting from the left hand navigation menu:



Each source monitor displayed includes information about the number of content items currently in its list and also the last time the list was opened. You can display the content of the source monitor list by double-clicking on the source monitor. The list is then displayed in a new tab.

A source monitor list looks more or less identical to the CUE search panel, since source monitors are in fact predefined searches. 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 source monitor panel belong to the source monitor: you will not find it in the standard search panel or in any other search monitor panel.

Creating source monitors is a Content Store configuration task. No configuration work is required in CUE itself. For instructions on how to create source monitors, see Source Monitor Definitions.

You can, however, control how often source monitors are updated by adding some configuration settings as follows:

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

   ```
   $ sudo su
   ```

2. Open **/etc/escenic/cue-web-3.3/config.yml** for editing. For example

   ```
   # nano /etc/escenic/cue-web-3.3/config.yml
   ```

3. Add the following settings:

   ```
   sourceMonitor:
      pollingEnabled: true
      pollingInterval: interval
   ```

   where *interval* is the required update interval in seconds (the default setting is 30).

4. Save the file.

5. Enter:

   ```
   # dpkg-reconfigure cue-web-3.3
   ```

   This reconfigures CUE with the changes you have made.

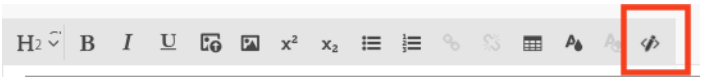
You can disable source monitor updates altogether by setting **pollingEnabled** to **false**. All source monitors and the source monitor list will then only be updated if the user explicitly forces an update by refreshing the browser tab.

3.13 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.

3.14 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.

4 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 CCI Europe 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. 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.

  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 CCI Europe. 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 CCI Europe. 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-3.3 cue-content-duplication-enrichment-service-3.3
```

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-3.3
```

For additional instructions regarding the installation of the **`cue-content-duplication-enrichment-service`** plug-in, see section 4.1.

4.1 cue-content-duplication-enrichment-service

This plug-in depends on nodeJS, version 6.4.0 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 6.4.0, then you need to replace it with a newer version. For advice on how to do this on Ubuntu, see (for example) this page.

4.1.1 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-3.3
```

This installs the enrichment service and starts it immediately.

4.1.2 Configuring cue-content-duplication-enrichment-service

The **cue-content-duplication-enrichment-service** is configured to run on port 8082 by default. If for some reason that won't work, you can change the port number as follows:

1. Log in as **root** if necessary.
2. Open **/etc/escenic/content-duplication-service-3.3/content-duplication-service.yaml** in an editor:

   ```
   server:
     port: 8082
   ```
3. Set the port number to whatever value you require and save the file.
4. Restart the service as follows:

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

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

1. Create a file called **/etc/escenic/cue-web-3.3/content-duplication-service.yml**, open it in an editor and add the following content:

   ```
   enrichmentServices:
     - name: "Duplicate Service"
       href: "http://myhost:8082/contentDuplicationService"
       title: "Duplicate Service"
       triggers:
   ```

```
      - name: "on-duplicate"
        properties: {}

authorizedEndpoints:
  - "http://myhost:8082/"

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.

2. If you have changed the duplication service's port number from **8082**, then replace both occurrences of this.
3. Save the configuration file.
4. Apply your configuration changes by entering:

```
# dpkg-reconfigure cue-web-3.3
```

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

5 Using CUE

This chapter contains the information existing users of Content Studio need to get started using CUE. Although CUE is intended to work on any mobile or desktop device using any modern browser, with the current version you are recommended to use the following device/browser combinations:

- Chrome on all desktop/laptop computers (including Macs) and all Android devices
- Safari on all IOS devices

5.1 Getting Started

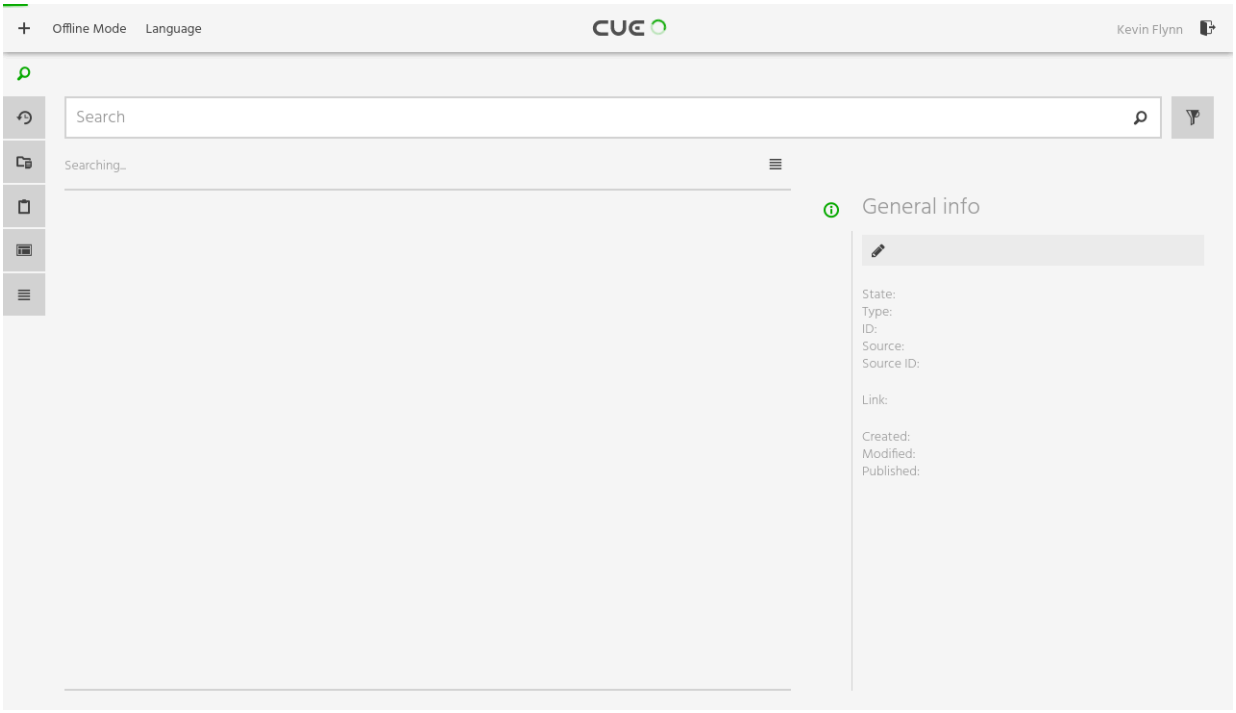
CUE is a **tab-based** webapp. What we mean by this is that CUE uses browser tabs as document containers. Every content item or section page you open or create is opened in a new browser tab, rather than everything happening in a single tab. If you've used Google Docs or Microsoft's Office Online before, then you'll know what to expect. The following video shows how CUE makes use of tabs.

Video: getting started with CUE

To get started with CUE:

1. Open a browser.
2. Go to the CUE URL you have been given by your network/system manager. You will see the CUE login window:

3. Log in using the same credentials as you would use for Content Studio. You will then see the CUE **home page**, which looks something like this:



You can now get to work. But before you do, it's probably a good idea to take look at some of the CUE tab's components.

At the top of the tab is the CUE menu bar.

This menu bar is present in all CUE tabs - document containers as well as the tab. Aside from the **Offline mode** and **Language** menus that are discussed later, the menu bar contains three important buttons:

This is the **New** button – it creates new content items.

This is the **Home** button – it opens a new CUE tab containing the home page. Not very useful when you're already on the home page, but useful if you're working on a content item or section page and don't have a home page open in your browser.

This is the **Log out** button.

Down the left hand side of the page is a column of **panel buttons** that determine what is displayed on the home page:

This button displays the CUE **Search** panel, that you can use to search for content. See section 5.3 for further information.

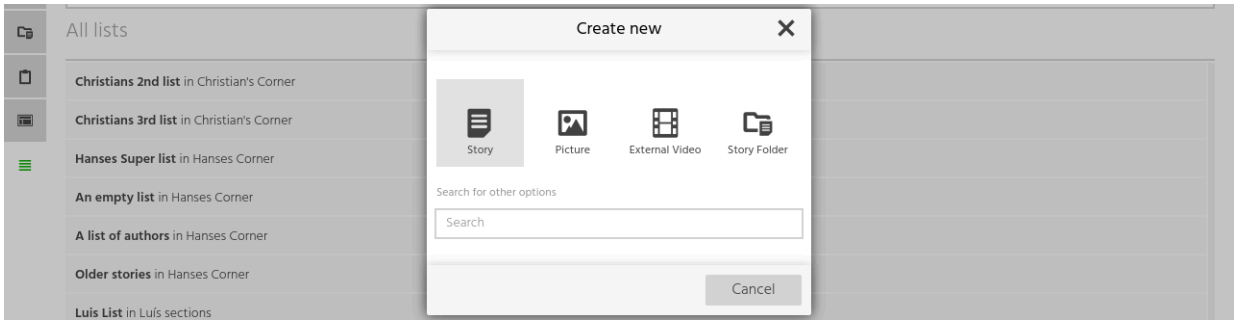
This button displays the **Recent** panel, a list of documents you have been working with recently.

This button displays the **Sections** panel, which you can use to view and work with your publications' sections and section pages.

This button displays the **Lists** panel which contains a list of all your lists.

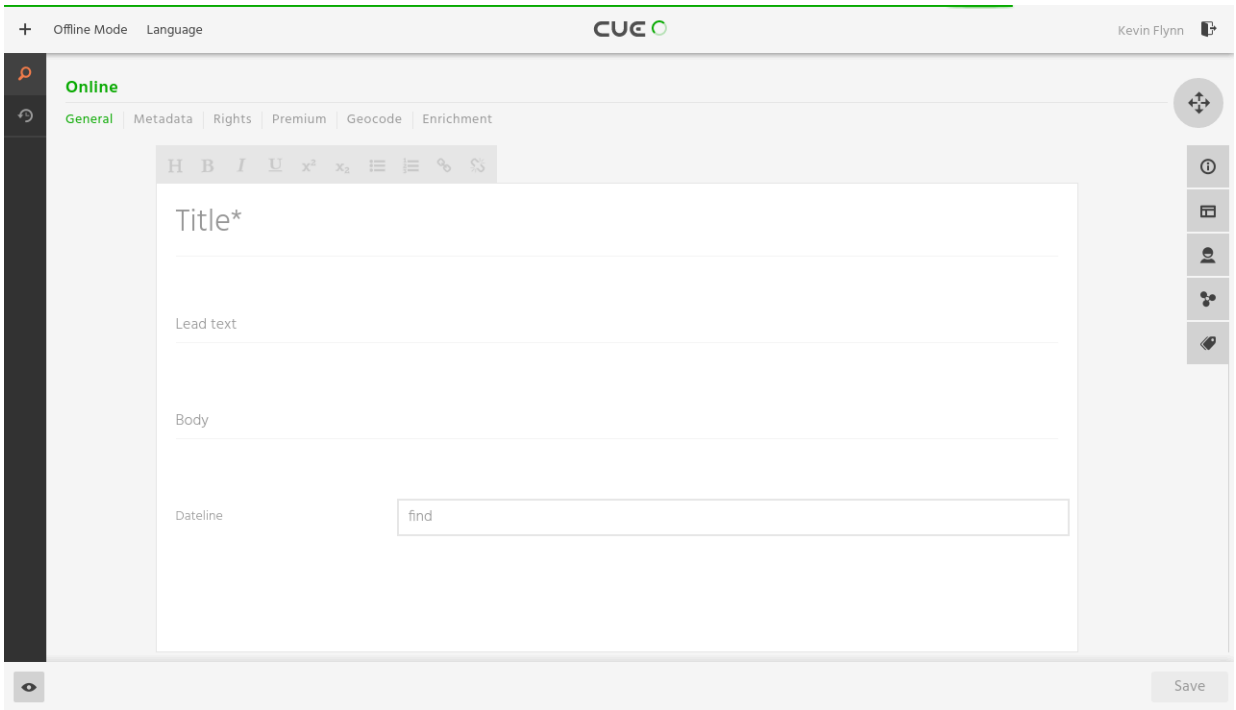
5.2 Creating Content

To create a new content item, select the + button (at the left end of the menu bar) and then select the required content type from the displayed dialog:



If you can't see a button for the content type you want to create, then use the **Search for other options** field to find the content type you want to create.

The new content item is displayed in a new browser tab and will look something like this:



Exactly what fields are included in the content item depends on its type. The fields are usually divided into groups that are displayed on separate pages. The group names are listed at the top of the content editor, and you can switch between the pages by clicking/tapping the group names. The most commonly-used fields are displayed in the first, default group, while less-used fields are relegated to the other groups.

Click / tap on these links to switch between the cards.

Now all you need to do is to fill in some fields (all the required ones marked with * at least) and click **Save** to save your content item.

If you've not used a tab-based webapp before, now is probably a good time to look closely at what has happened. You started up CUE in one tab, and when you clicked on the + button to create a new content item, the content item was created in a new tab. So now you have two CUE tabs open in your browser:

- The original CUE home page, which has the tab name **CUE**.
- Your new content item, which has the content item title as its tab name.

This means that if you have a number of CUE tabs open in your browser, you can easily see what is what.

> The advantage of displaying each content item in its own browser tab is that each CUE content item effectively becomes a web page, with a unique and unchanging URL, just like any other web page. This means you can bookmark content items using your browser's bookmark functions if you want. And if you copy the URL of a content item and send it to another CUE user by mail or chat, then that person can open the content item in CUE by simply clicking on the link (and logging in if necessary).

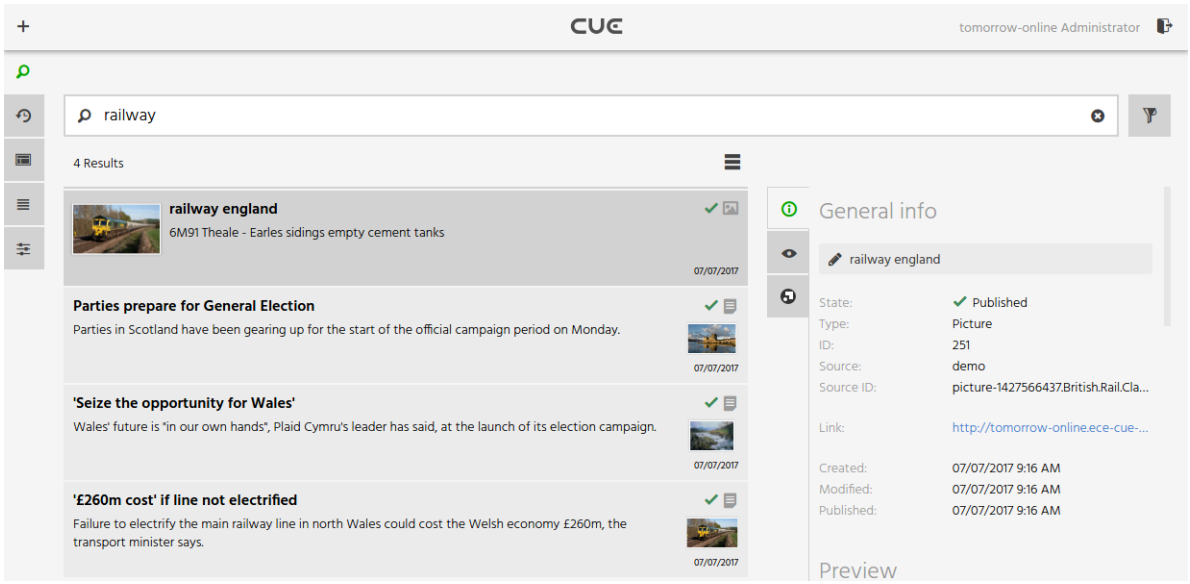
When you create a content item in this way using the menu bar + button, it is added to your publication's default section (usually called **New Articles** or something similar). You can move it, or add it to other sections as well if you want. You can, however, also create content items from the home page's **Sections** ( ) panel: this allows you to create the content items directly in the correct section.

5.3 Finding and Opening Content

To find an existing content item:

1. Open the CUE home page.
2. Make sure that the search panel is is displayed (select the button if necessary).
3. Enter a search string in the field at the top of the panel.

The search results are displayed below the search field:



There are several different ways to open a content item:

Desktop	**Mobile**
Double-click the content item	Double-tap the content item
Right-click the content item, then select **Open** from the displayed menu	Long-press the content item, then select **Open** from the displayed menu
Select the content item, then press **`Enter`** on the keyboard	—

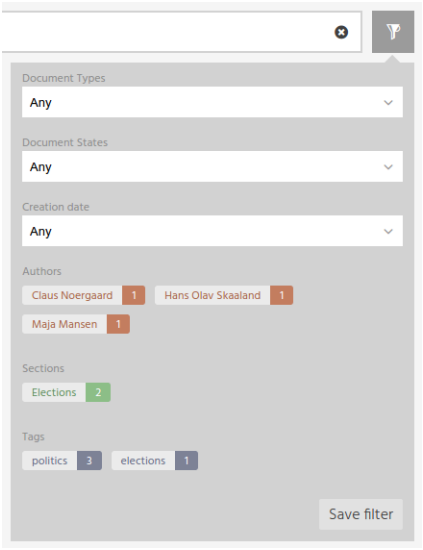
The selected content item is opened in a new tab, just the same as when you create a new content item.

For more information about searching for content, see section 5.4.

5.4 Searching for Content

You've already seen in section 5.3 how you can use the search field at the top of the CUE home page to find content items you are interested in. Sometimes, however, simple text string searches aren't enough to find what you want. CUE therefore also includes a powerful set of filters that you can use to narrow your search.

To display the search filters, click on the button at the right-hand end of the search field:



You can now narrow down your results using up to six different criteria in addition to a search string: document type, document state, creation date, author, section (that is, the publication section content belongs to) and tags. The first three criteria are drop-down selections, and you can only choose one of the available options: if you select the "Story" document type, then you can't choose "Picture" as well. **Author** is also a single choice criterion: if you select one author label, then all the other options disappear. **Sections** and **Tags**, however are multiple choice criteria: if you pick several section labels, then the results will be narrowed down to include only documents that belong to all of the selected sections. Similarly, if you pick several tags, then the results will be narrowed down to include only documents that have all the selected tags.

> Note how every selection you can make is followed by a number – this is the number of results that match all the criteria you have already specified **plus** that particular selection. This makes it easy to see how you can best narrow down the search results.

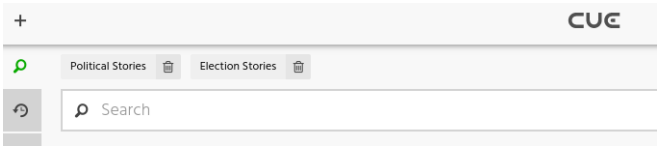
By default, CUE shows a small amount of content and some information about each document in the results list. If you want a more compact list, click on the button at the top right to display titles only. Click a second time to expand the list again.

You don't have to go back to the CUE home page to search for content: document tabs have a side bar search function. To open the search side bar, click on the button on the left side of the document tab. The search side bar contains all the same search and filtering functions as the home page.

5.4.1 Saving Search Filters

If you frequently use the same combination of filter selections to search for content, you can save time by saving search filters. To do this, just select the **Save filter** button at the bottom of the search filter

panel and enter a name for the filter. The filter will then appear as a button above the search field. You can create as many search filters as you like in this way.



You can instantly execute complex searches at any time by selecting these buttons. To cancel a selected filter, just select it a second time. To delete a filter when you no longer have any use for it, just select it's trashcan icon.

Search filters are saved on the server, not on the device you are using. This means that if you use multiple devices and computers, the filters you create will always be available on all of them.

5.4.2 Sharing Searches

When you define a search or select a saved search filter, you may notice that the URL in the browser's URL field changes. This is because the search you have defined is added to the URL. This means that you can easily share a search you have defined with other CUE users by simply sending the URL to them. Copy the content of the URL field, paste it into a mail or messaging application and send it to any CUE user. The recipient will be able to see your search results by simply clicking on the link.

5.4.3 More About Search

This section contains more detailed information about how searches are performed.

Ignored characters

The following special characters in a search string are ignored (or to be precise, they are removed from the search string and replaced by space characters):

% ^ " / \ : ; ( ) + ? ! { } [ ] ~ | -

In other words, if you enter the search string **`Test%Search\String`** it will be converted to **`Test Search String`** before it is submitted to the search engine.

Wild cards added

Once any special characters have been removed, wild cards are added to each word in the resulting search string as follows:

```
Test Search String => (Test Search String) OR ((Test OR Test*) (Search OR Search*)
  (String OR String*))
```

This means it will find not only occurrences of the specified words, but also occurrences of longer words that start with the same character sequence ("**Test**ed **search**ing **string**s", for example).

How filters are combined

Multiple filters **of the same type** are combined with an **`OR`** operator, so that items matching **any** of the specified criteria will be found. Adding more filters of the same type therefore potentially **increases** the number of results. Specifying the **Document Types** filters **Story** and **Picture**, for

example, will return both stories and pictures, whereas if you specify only **Story** then only stories will be returned.

Multiple filters of different types are combined with an **AND** operator, so that only items matching **all** of the specified criteria will be found. Adding more filters of the same type therefore potentially **decreases** the number of results. Specifying just the **Document Types** filter **Story**, for example, will return all all stories, whereas specifying both the **Document Types** filter **Story** and the **Authors** filter **Alice** will only return those stories that have Alice as an author. Specifying the **Document Types** filters **Story** and **Picture** and the **Authors** filters **Alice** and **Bob** will return both stories and pictures that have either Alice or Bob (or both) among their authors.

5.5 Editing Content

Once you have created or opened a content item, editing is simply a matter of clicking in the fields in the content editor and entering/editing content in the usual way. Different types of content items have different fields: they are customer-defined, so the content items in your particular system may have different fields from the examples you see here.

Content item fields are usually divided into groups that are displayed on separate pages. The group names are listed at the top of the content editor, and you can switch between the pages by clicking on the group names. The most commonly-used fields are displayed in the first, default group, while less-used fields are relegated to the other groups.

A content item can contain many different kinds of fields, designed to hold different kinds of values (plain text, formatted text (called **rich text**), numbers, specific values such as keywords, and so on. Many fields have constraints that limit what you can enter, such as maximum string lengths or minimum/maximum numerical values. Any fields that are marked with an asterisk (*) are required fields: until you have specified an allowed value in all required fields, the **Save** button is disabled so that you cannot save the content item.

When you edit a field, you'll see that a lock symbol is displayed below the field, indicating that the field is now locked by you – in order to prevent conflicts, nobody else is allowed to modify this field until you have saved your changes. Once you save your changes, the lock is removed, and other people can make changes to the field. Occasionally, you may notice that a field you want to edit is locked by somebody else, and you cannot change it.

Above all the fields in a group is a formatting toolbar:

This toolbar is only enabled when you are editing a rich text field.

You can copy text into a content item from other content items, from other browser tabs/windows or from other applications such as Word or Excel either by copying and pasting or by dragging and dropping.

You can preview the current state of the content item at any time by clicking or pressing the button below the editor.

To save your edits select the **Save** button below the editor.

On the left hand side of a CUE content tab are two buttons, (**Search**) and (**Recent**). They have the same purpose as these buttons do on the home page (displaying search and recently opened panels) but in this case, the panels they display are just side bars. This allows you to search for other content while you are editing a content item – so that you can, for example, add them to the content item you are working on as **relations**. For more about this, see section 5.5.3.

On the right side of a document tab is a column of buttons that you can use to display the document's **attributes panel**. Each button displays a different part of the attributes panel:

Displays **General info** – general information about the document.

Displays **Sections** – the sections to which the document belongs. You can add/remove the document to/from sections here: see section 5.5.1 for details.

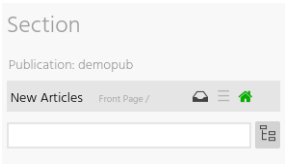
Displays **Authors** – the authors of the document. You can edit the document's list of authors here: see section 5.5.2 for details.

Displays **Related** – the documents related to this document. You can add and remove relations here: see section 5.5.3 for details.

Displays **Tags** – the document's tags. You can add and remove tags here: see section 5.5.4 for details.

5.5.1 Adding Content to Sections

The sections to which a content item belongs are displayed in the attributes panel on the right of the content editor, in a division called **Sections**. To display the sections, click/tap the button.



When you save a new content item it is automatically added to your publication's "new articles" section. To include it in other sections as well:

1. Make sure the attributes panel's **Section** division is expanded.
2. Click in the **Section** division's search field and start typing the name of the required section.
3. When the required section appears, select it to include the content item in the section.

Each section the content item belongs to is represented by a section bar that contains the name of the bar, plus the following buttons:

Moves the content item between this section's inboxes. A content item can only be in one inbox at a time.

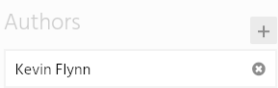
Adds/removes the content item to/from this section's lists. A content item can be in several lists at the same time.

Sets the content item's home section. Just click on the button of the section that you want to be the content item's home section.

Removes the content item from this section. You cannot remove a content item from its home section.

5.5.2 Changing a Document's Authors

A document's authors are displayed in the attributes panel on the right of the content editor, in a division called **Authors**. To display the list of authors, click/tap the button.



By default, a document has one author: the user who initially created it. You can, however, add more names to the list of authors, or replace the default author.

To add an author:

1. Make sure the **Authors** division of the attributes panel is displayed.
2. Select the button above the author field. This displays a **Select authors** dialog.
3. Select the content item you want to add as a relation. There is a search field at the top of the dialog to help you find the content item you want.
4. Click /tap **Add**.

To remove an author from the list of authors, select its button.

5.5.3 Managing Relations

Relations between documents are very important in CUE systems. They are used to both represent links between documents (links to related articles, for example), and inclusions of one document in another (the inclusion of images and/or videos in articles, for example).

Relations are displayed in the attributes panel on the right of the document tab, in a division called **Related**. To display the relations, click/tap the button.

You can then add a relation to the document in either of the following ways:

- 1. Open the **Search** ( ) or **Recent** ( ) side panel (on the left side of the document).
  2. Find the document you want in the side panel.
  3. Drag your chosen document across to the **Related** section.
  4. Drop it in the appropriate relation field.

- 1. Select the + button above one of the relation fields. This displays a **Pick relation** dialog.
  2. Select the document you want to add as a relation. There is a search field at the top of the dialog to help you find the document you want.
  3. Click /tap **Add**.

You can also add the document you are currently working on as a relation in another document. The other document must already be open in CUE as well. To do this:

1. Select the **drag handle** of the document you are currently working on. A document's drag handle is the icon displayed in the top right corner of the document.
2. Drag the handle to the tab of the document you want to relate it to.
3. Hold the drag handle over the tab until the target document comes into view.
4. Drag the handle to the required relations field and drop it there. (You might need to first drag the handle to the button and hold it there until the attributes panel opens and displays the **Related** section.)

To remove a relation, select its button.

5.5.4 Tagging Content

A content item's tags are displayed in the attributes panel on the right of the content editor, in a division called **Tags**. To display the tags, click/tap the button.

You can then add a tag to the content item as follows:

1. Start typing the name of the tag you want to insert in the **Find Tags** field. A list of matching tags will be displayed below the field.
2. Select tag you want to add.
3. The five bars to right of the tag represent the **relevance** of the tag to the content item. All tags you add are initially assigned a relevance of five, but you can change the relevance by clicking/tapping these bars. Relevance can be used to control the display of tags in your publications and to fine tune tag-based searching. If relevance is used in your publications, then you will most likely have been given some rules for how to set it - otherwise, you can just ignore it.

If the tag you enter does not already exist, then an option to create it is offered below the field instead of a list of matches.

To remove a tag, select its button.

5.6 Editing Persons and Users

Content Store person and user objects appear in CUE as special content items called **Person** content items. **Person** content items differ from other content items in the following ways:

- They are predefined by the system, not user-defined as are all other types of content item.
- They have a fixed structure, with a fixed set of fields that cannot be modified.

- They are identified internally by the special name **`com.escenic.person`**, and this is the content type name you have to use in CUE's **Document types** search filter.
- They are subject to a different workflow and cannot be published.

The difference between a person and a user is that a user has a Content Store login. When you open a Person content item you can see whether it represents a person or user by checking the **User name** field under **General Info** in the metadata panel.

Person content items cannot themselves be published, as they are considered to contain private information such as addresses and telephone numbers. Person content items do, however, have a **Profiles** relation, which you can used to associate ordinary, publishable content items with Persons. Profile content items can be used to hold images, biographical details and so on for use in bylines.

The recommended approach is to define a special "Profile" content type that contains fields/relations for all the information you want to be able to publish about authors: biography, email address, Twitter handle, headshot and so on. A Profile content item is then created for each Person object and dropped in its **Profiles** relation.

Assuming you have administrator access rights then you can use CUE to edit Person content items (for both persons and users). You cannot, however, create or delete them. To do these things you need to use Web Studio (see User Administration).

If you do not have administrator access rights then you can only edit your own user's Person content item or persons (not users) that you have created yourself.

5.6.1 Changing User State

A user is always in one of the following states:

Enabled
This is the default state. In this state, a user has access to CUE, Content Studio, Web Studio and all other editorial applications.

Limited
In this state, a user only has access to Mobile Studio, and will not be able to log in to any other editorial applications. This state is useful for external contributors such as bloggers.

Suspended
Not currently used. Has the same effect as **Disabled**.

Banned
Not currently used. Has the same effect as **Disabled**.

Disabled
In this state a user has no access to any editorial applications.

When editing a user's Person content item, you can switch between these states using the workflow buttons at the bottom of the content editor.

5.7 Publishing Online Content

You can publish content items from CUE and make other state changes in much the same way as you do it in Content Studio.

5.8 Multimedia Publishing

If CUE is connected to a CCI NewsGate system as well as an CUE Content Store, then **ONLINE**, **PRINT** and **TABLET** options are displayed at the top of content tabs, allowing you to distribute content to print and tablet publications as well as to online publications.

To publish to print or tablet, select the corresponding view (**PRINT** or **TABLET**): you will then see the settings needed to publish to the selected medium. The settings and publishing process for print and tablet are similar to the settings and procedures in Newsgate.

If there is a print or tablet version of the current Escenic content item (or **online package** in Newsgate terminology) then the **PRINT** or **TABLET** option is displayed in black. If there is no such version, then the corresponding option is grayed out. You can however create a version of that type by clicking on the option.

5.9 NewsGate Story Folders

If CUE is connected to a CCI NewsGate system as well as an CUE Content Store, then an extra **Story folders** button is included in the column of **panel buttons** displayed on the left hand side of the home page. Selecting this button displays a panel containing all your Newsgate story folders.

You can filter the list of story folders by typing in the **Filter** field at the top of the navigation pane: only story folders containing the string you enter will be displayed.

Select a folder from the list to open it in a content tab.

A story folder contains the sections **Stories**, **Assets**, **Assignments**, **Team** and **Contacts**. All these sections have similar contents to the corresponding sections in CCI NewsGate, except for the **Stories** section. In NewsGate, stories are added to different packages for output to different media, whereas In CUE, a story can exist in different media versions, all of which share a common text.

To create a new story in CUE, select the **+** button **on the menu bar**. This creates a story in a new story folder, containing an online package (that is, an Escenic content item).

To simply add a new text to a story folder, select the **+** button **in the story folder**. The process of creating a new text is the same as in NewsGate.

The metadata properties displayed on the right of the story folder are the same as those displayed in CCI Newsgate and have the same meaning/functionality.

5.10 Exploring CUE

There's a lot more to CUE than we've covered in this short introduction. Most of the things you can do in Content Studio, you can now also do using CUE. You can use it, for example, to create and edit sections, section pages, lists and inboxes, crop images and so on.

Now you are familiar with CUE's basic content editing functionality, you should find it relatively easy to learn how to use CUE to accomplish other objectives, especially if you are already a Content Studio user.

6 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 6.1 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 6.2 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 6.3 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 6.4 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 6.6.

6.1 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 storyline elements 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`-3.3). 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.

6.1.1 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.
  }
}
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.
  }
}

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.

The cueInterface object

In most cases, CUE passes a **`cueInterface`** object to the web component. This is not the case, however, for storyline element field editors. **`cueInterface`** always has the following basic properties:

`escenic.endpoint`
: The URI of the Content Store web service as defined with the **`endpoints/escenic`** setting in the CUE configuration file (see chapter 3).

`escenic.credentials`
: The authorization token needed to access the **`escenic.endpoint`** URI. This value must be included as the **`Authorization`** header in requests sent to the **`escenic.endpoint`** URI.

`currentUser.userName`
: The user name of the current user.

currentUser.realName
The real name of the current user.

currentUser.email
The email address of the current user.

newsgate
An object providing various functions for interacting with the Newsgate server.

selection
An object for manipulating the current rich text selection (if any). This object exposes the following functions:

getCurrentSelection()
Returns the current rich text selection.

replaceSelection(newContent: string, selection: Selection)
Replaces **selection** with **newContent**. **newContent** must be a valid HTML fragment.

forEachBlockInSelection(range: Range, forEachBlock: Function)
Loops through every top-level node in the selection and applies the **forEachBlock** function. This is particularly useful for text selections that cross paragraph boundaries.

replaceElement(element: Element, elementName: string, className: string)
Replaces **element** with a new element of type **elementName** and class **className**. All the children of **element** are preserved. In other words, after the function has executed, the children of **element** will be the children a new element of type **elementName**.

replaceBlockElement(element: Element, elementName: string, className: string)
Replaces the closest **<p>** or ** ** ancestor of **element** with a new element of type **elementName** and class **className**. All the children of the ancestor element are preserved. In other words, after the function has executed, all the children of the ancestor element will be the children of a new element of type **elementName**.

addSelectionWatcher(watcher: Function)
Adds a watcher that will be notified when the current rich text selection changes. The watcher will be invoked with the new selection.

removeSelectionWatcher(watcher: Function)
Removes a previously added watcher.

For an example of how to use these functions, see section 6.1.3.4.

For content items, the **cueInterface** object's properties will also include article URL, article ID and content type. For sections it will also include section URL, section ID, and section name.

In addition, the **cueInterface** object has other context-dependent properties that vary according to the type of web component created. These properties are described along with the different web component types.

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 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`**.

See section 6.1.2.3 for an example of how to construct a draggable object.

6.1.2 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.

An editor side panel is typically used to display information from some external system or web site – the example supplied in section 6.1.2.3 displays images from the Pixabay web site.

6.1.2.1 Editor Side Panel Configuration

The following properties must be defined to configure an editor side panel:

`- id`
The tag 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 (**`-`**).

`name`
The display name of the component. The name is only actually displayed when the mouse is held over the side panel button.

`directive`
Must be set to **`"cf-custom-panel-loader"`**.

`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.

`mimeTypes`
An array of MIME types specifying the editors for which this panel should be made available. The following MIME types may be specified:

x-ece/story	CUE content editor for items other than images, videos, galleries and events
x-ece/picture	CUE image editor
x-ece/video	CUE video editor
x-ece/gallery	CUE gallery editor
x-ece/event	CUE event editor
x-ece/new-content	Editor containing new CUE content that has not yet been saved
x-ece/section	CUE section editor
x-ece/section-page	CUE section page editor
x-ece/list	CUE list editor
x-ece/*	All CUE editor types
x-cci/assignment	Newsgate assignment editor
x-cci/storyfolder	Newsgate story folder editor
x-cci/*	All Newsgate editor types
/	All editor types

homeScreen
: Set this to **true** if you want your panel to function as a home page panel as well as an editor side panel. Note that home page panels are full width panels, whereas editor side panel width is restricted. Dual-purpose panels must therefore be made to display nicely in both contexts.

metadata
: Not used. Should be specified as an empty array – **[]**.

active
: Set to **false**.

order
: Determines the position of this panel button in the column of side panel/home page buttons. The buttons are arranged in numerical order from lowest to highest.

All the properties must be entered as a list item belonging to a **sidePanels** 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:

```
sidePanels:
  - id: "image-search"
    name: "Image Search"
    directive: "cf-custom-panel-loader"
    webComponent:
      modulePath: "http://www.example.com/webcomponents/image-search/image-search.js"
      icon: "image-search-icon"
    mimeTypes: ['*/*']
    homeScreen: false
```

```
    metadata: []
    active: false
    order: 802
```

6.1.2.2 Editor Side Panel cueInterface Properties

A **`cueInterface`** object is passed to both the main side panel component and to the icon component. In addition to the basic properties described in section 6.1.1, this **`cueInterface`** object has the following properties in each case:

For the editor side panel component

`active`
A boolean property that indicates whether or not the panel is currently active.

`addActiveWatcher(watcher: Function)`
A function that adds a watcher that will be notified when the active state of the editor side panel changes. The watcher will be invoked with the new state.

`removeActiveWatcher(watcher: Function)`
A function that removes a previously added watcher.

For the icon component

`active`
A boolean property that indicates whether or not the panel is currently active.

`addActiveWatcher(watcher: Function)`
A function that adds a watcher that will be notified when the active state of the editor side panel changes. The watcher will be invoked with the new state.

`removeActiveWatcher(watcher: Function)`
A function that removes a previously added watcher.

6.1.2.3 Editor Side Panel Example

This example searches the Pixabay web site for images.

```
(function (imageSearch) {
  imageSearch.dragStart = function (event) {
    const imageData = JSON.parse(event.currentTarget.getAttribute('data-hit'));
    const dataUrl = this.getDataUrlFromImage(event.currentTarget,
 imageData.webformatWidth, imageData.webformatHeight);
    const jsonData = JSON.stringify({
      files: [
        {
          dataUrl: dataUrl,
          name: imageData.tags.replace(/, /g, '-') + '-' + imageData.id,
          mimeType: 'image/jpeg'
        }
      ]
    });
    event.dataTransfer.setData('application/x-web-component-data', jsonData);
  };

  imageSearch.getDataUrlFromImage = function (image, width, height) {
    let canvas = document.createElement('canvas');
    canvas.width = width;
```

```
    canvas.height = height;

    const context = canvas.getContext('2d');
    context.drawImage(image, 0, 0);

    return canvas.toDataURL('image/jpeg');
  };
})(window.imageSearch || (window.imageSearch = {}));

class ImageSearch extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          margin: 0;
          padding: 0;
          width: 100%;
          height: 100%;
          display: flex; }

        .content-wrapper {
          flex-grow: 1;
          flex-direction:
              column; margin: 0 10px 0 10px;
          height: calc(100vh - 60px);
        }

        #wrapper {
          flex-grow: 1;
          flex-direction: column;
          position: relative;
          overflow: auto;
          height: calc(100% - 60px);
          margin-top: 28px;
        }

        .search {
          position: relative;
        }

        #images {
          overflow-y: auto;
          overflow-x: hidden;
          position: absolute;
          top: 10px;
          bottom: 10px;
          left: 0;
          right: 0;
          scrollbar-face-color: #000000;
          scrollbar-track-color: transparent;
        }

        #images::-webkit-scrollbar {
          width: 9px;
          height: 9px;
        }
```

```
#images::-webkit-scrollbar-thumb {
  background: #333333;
  border-left: 2px solid #444444;
}

#images::-webkit-scrollbar-track {
  background: transparent;
}

#images img {
  display: block;
  max-height: 250px;
  max-width: 80%;
  padding: 2px;
  margin: 0 auto 10px auto;
  cursor: pointer;
  border: 2px solid #9c9c9c;
  opacity: 0.8;
  transition: all 250ms ease-in-out;
}

#images img:hover {
  border: 2px solid #ffffff;
  opacity: 1;
}

#search-term {
  position: absolute;
  z-index: 1;
  width: 100%;
  height: 28px;
  border: 2px solid #000000;
  background: #333333;
  font-family: "Hind", Helvetica Neue, Helvetica, Arial, Sans-serif;
  color: #ffffff;
  font-weight: 300;
  font-size: 14px;
  padding-left: 10px;
  padding-right: 28px;
  box-sizing: border-box;
}



#search-term:focus,  .searchButton:focus {
  outline: none;
}

.searchButton {
  position: absolute;
  z-index: 2;
  top: 2px;
  right: 2px;
  width: 24px;
  height: 24px;
  border: 0;
  cursor: pointer;
  background: transparent;
  color: #9c9c9c;
}
```

```
        .searchButton:before {
          font: 16px 'cf';
          font-style: normal;
          font-weight: normal;
          padding: 2px 0 0 4px;
          line-height: 24px;
          content: '\\e878';
          -webkit-font-smoothing: antialiased;
          -moz-osx-font-smoothing: grayscale;
        }

        .searchButton:hover{
          color: #ffffff;
        }

        h1 {
          display: block;
          width: 100%;
          height: 44px;
          margin: 0;
          padding: 0;
          background: url("webcomponents/image-search/pixabay_logo.png") no-repeat
center 10px;
        }

      </style>
      <div class="content-wrapper">
        <h1></h1>
        <div class="search">
          <input id="search-term" type="textbox" value="" placeholder="Search in
Pixabay free images">
          <div id="search" class="searchButton"></div>
        </div>

        <div id="wrapper">
          <div id="images"></div>
        </div>
      </div>
    `;
  }

  connectedCallback() {
    if (this.cueInterface.editor && this.cueInterface.editor.getTags) {
      const tags = this.cueInterface.editor.getTags();
      if (tags && tags.length > 0) {
        this.shadowRoot.querySelector('#search-term').value = tags[0].value;
      }
    }
    this.search();
    $(this.shadowRoot.querySelector('#search-term')).keypress(event => {
      if (event.which === 13 /* ENTER */) {
        this.search();
      }
    });
    $(this.shadowRoot.querySelector('#search')).click(this.search.bind(this));
  }

  ImageSearchProto.search = function() {
    var searchTerm = panelShadowRoot.querySelector('#search-term').value;
```

```
    searchTerm = searchTerm.replace(' ', '+');
    this.loadImages(searchTerm);
  };

  loadImages(searchTerm) {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://pixabay.com/api/?key=<YOUR_API_KEY>&q=' + searchTerm +
'&image_type=photo&safesearch=true', true);

    xhr.onload = function () {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          this.showImages(xhr.responseText);
        }
        else {
          console.error(xhr.statusText);
        }
      }
    }.bind(this);

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

    xhr.send(null);
  };

  ImageSearchProto.showImages = function(responseData) {
    var responseJson = JSON.parse(responseData);
    var images = '';
    responseJson.hits.forEach(function (hit) {
      images += "<img src='" + hit.webformatURL + "' data-hit='" + JSON.stringify(hit)
+ "' crossorigin='anonymous' ondragstart='imageSearch.dragStart(event);'>";
    });

    images += '<a href="https://pixabay.com/" target="_blank"><img src="https://
pixabay.com/static/img/public/medium_rectangle_a.png" alt="Pixabay"></a>';

    panelShadowRoot.querySelector('#images').innerHTML = images;
  }
;

customElements.define('image-search', ImageSearch);

class ImageSearchIcon extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        :host { margin: 0; padding: 6px; display: block; }
        img { max-width: 80%; position: relative; top: 3px; left: 3px; }
      </style>
      <img class="icon">
    `;

    ImageSearchIconProto.createdCallback = function () {
      var template = thisDoc.querySelector('template#icon').content;
    iconShadowRoot = this.attachShadow({ mode: 'open' });
```

```
    iconShadowRoot.appendChild(template.cloneNode(true));
  };

  ImageSearchIconProto.attachedCallback = function() {
    if (this.cueInterface.homeScreen) {
      this.activeStateChanged(this.cueInterface.isActive());
      this.cueInterface.addActiveWatcher(function (active) {
        this.activeStateChanged(active);
      }.bind(this));
    }
    else {
      var img = iconShadowRoot.querySelector('img.icon');
      img.src = this.getAbsolutePath(this.editorIconPath);
    }
  };

  ImageSearchIconProto.activeStateChanged = function(active) {
    var img = iconShadowRoot.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('image-search-icon', ImageSearchIcon);
```

6.1.3 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.

An editor metadata section is used to display information about the object being edited – the example supplied in section 6.1.2.3 displays the history of the content item being edited.

6.1.3.1 Editor Metadata Section Configuration

The following properties must be defined to configure an editor metadata section:

- **`name`**
  The display name of the component. The name is only actually displayed when the mouse is held over the metadata section button. Remember also that the **`name`** property name must be preceded by a hyphen (`-`).

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

`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.

mimeTypes
An array of MIME types specifying the editors for which this metadata section should be made available. The following MIME types may be specified:

x-ece/story	CUE content editor for items other than images, videos, galleries and events
x-ece/picture	CUE image editor
x-ece/video	CUE video editor
x-ece/gallery	CUE gallery editor
x-ece/event	CUE event editor
x-ece/new-content	Editor containing new CUE content that has not yet been saved
x-ece/section	CUE section editor
x-ece/section-page	CUE section page editor
x-ece/list	CUE list editor
x-ece/*	All CUE editor types
x-cci/assignment	Newsgate assignment editor
x-cci/storyfolder	Newsgate story folder editor
x-cci/*	All Newsgate editor types
/	All editor types

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 an **editors/metadata** 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:

```
editors:
  metadata:
    - name: "Content History"
      directive: "content-history"
      webComponent:
        modulePath: "http://www.example.com/webcomponents/history/history.js"
        icon: "content-history-icon"
      mimeTypes: ["x-ece/story"]
```

```
        order: 803
```

6.1.3.2 Editor Metadata Section cueInterface Properties

A **`cueInterface`** object is passed to both the main metadata section component and to the icon component. In addition to the basic properties described in section 6.1.1, this **`cueInterface`** object has the following properties in each case:

For the editor metadata section component

`editor`
An object representing the current editor. The properties and functions exposed by this object vary according to the type of editor as follows:

Story editor
Actually, any editor with an **`x-ece/*`** MIME type for which there is no more specific match. This, is in other words, the fallback case. The following properties and functions are exposed:

`type`
The editor type: always **`story`**.

`getLink()`
Returns the link object opened in the editor. A **`GET`** request to the URI contained in the object will return an XML document describing the content. CUE credentials must be included in the request as an Authorization header (see general web component information). These credentials are provided in the **`cueInterface`** object (see section 6.1.1).

`getTitle()`
Returns the title of the editor.

`getArticleId()`
Returns the ID of the content item opened in the editor.

`getContentType()`
Returns the type of the content item opened in the editor.

`getState()`
Returns the state (draft, published etc.) of the content item opened in the editor.

`getPublishedDate()`
Returns the published date of the content item opened in the editor. The date is returned as a **`moment`** object.

`getTags()`
Returns an array containing any tags assigned to the content item opened in the editor.

`getContent()`
Returns the entire content item opened in the editor.

`updateContentValue(key: string, value: string)`
Updates the content item opened in the editor by assigning **`value`** to the field identified by **`key`**.

`addContentWatcher(watcher: Function)`
Adds a watcher that will be notified when the content item changes.

List editor

An editor with the MIME type **`x-ece/list`**. The following properties and functions are exposed:

`type`
The editor type: always **`list`**.

`getLink()`
Returns the link object opened in the editor. A **`GET`** request to the URI contained in the object will return an XML document describing the list. CUE credentials must be included in the request as an Authorization header (see general web component information). These credentials are provided in the **`cueInterface`** object (see section 6.1.1).

`getTitle()`
Returns the title of the editor.

`addListWatcher(watcher: Function)`
Adds a watcher that will be notified when the list changes.

Section editor

An editor with the MIME type **`x-ece/section`** or **`x-ece/section-page`**. The following properties and functions are exposed:

`type`
The editor type: always **`section`**.

`getLink()`
Returns the link object opened in the editor. A **`GET`** request to the URI contained in the object will return an XML document describing the section. CUE credentials must be included in the request as an Authorization header (see general web component information). These credentials are provided in the **`cueInterface`** object (see section 6.1.1).

`getTitle()`
Returns the title of the editor.

`getSection()`
Returns the section that the opened section page belongs to.

`getSectionPage()`
Returns the section page opened in the editor.

`addSectionWatcher(watcher: Function)`
Adds a watcher that will be notified when the section page changes.

Assignment editor

An editor with the MIME type **`x-cci/assignment; type=picture`**. The following properties and functions are exposed:

`type`
The editor type: always **`assignment`**.

`getLink()`
Returns the link object opened in the editor.

`getTitle()`
Returns the title of the editor.

`getAssignment()`
Returns the assignment object opened in the editor.

`getStory()`
Returns the story folder object that the assignment belongs to.

Story folder editor
An editor with the MIME type **`x-cci/storyfolder`**. The following properties and functions are exposed:

`type`
The editor type: always **`storyFolder`**.

`getLink()`
Returns the link object opened in the editor.

`getTitle()`
Returns the title of the editor.

`getStoryFolder()`
Returns the story folder object opened in the editor.

`addStoryFolderWatcher(watcher: Function)`
Adds a watcher that will be notified when the story folder changes.

For the icon component

`active`
A boolean property that indicates whether or not the metadata section is currently active.

`addActiveWatcher(watcher: Function)`
A function that adds a watcher that will be notified when the active state of the editor metadata section changes. The watcher will be invoked with the new state.

`removeActiveWatcher(watcher: Function)`
A function that removes a previously added watcher.

6.1.3.3 Editor Metadata Section Example

This example displays the history of the content item being edited.

```
class HistoryElement extends HTMLElement {
  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>
    `;
  }

  HistoryProto.attachedCallback = function() {
    this.fetchHistory();
    this.cueInterface.editor.addContentWatcher(function () {
      this.fetchHistory();
    }.bind(this));
  };

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

  HistoryProto.displayHistory = function(document) {
    var entriesDiv = panelShadowRoot.querySelector('.entries');
    var parsedEntries = this.parseDocument(document);
    entriesDiv.innerHTML = '';
    parsedEntries.forEach(function (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>';
    });
  };

  HistoryProto.parseDocument = function(document) {
```

```
    var 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';
      }
    };

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

document.registerElement('content-history', {
      prototype: HistoryProto
    });

/**
 * Creating the icon (if required)
 */
var HistoryIconProto = Object.create(HTMLElement.prototype);
  var iconShadowRoot;

    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>
    `;
  }

  HistoryIconProto.attachedCallback = function() {
    this.activeStateChanged(this.cueInterface.isActive());
    this.cueInterface.addActiveWatcher(function (active) {
      this.activeStateChanged(active);
    }.bind(this));
  };

  HistoryIconProto.activeStateChanged = function(active) {
    var icon = iconShadowRoot.querySelector('.icon');
    if (active) {
      $(icon).addClass('active');
    }
    else {
      $(icon).removeClass('active');
    }
  };

document.registerElement('content-history-icon', {
      prototype: HistoryIconProto
    });
```

6.1.3.4 Selection API Example

This second example illustrates the use of the rich text editing functionality provided by the **`cueInterface`** object's **`selection`** property:

```
class TextModification extends HTMLElement {
  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.cueInterface.selection) {
      this.addButtonEventListeners();
      this.selection = this.cueInterface.selection.getCurrentSelection();
      this.cueInterface.selection.addSelectionWatcher(newSelection => {
        this.selection = newSelection;
        this.setButtonStates(!!newSelection);
      });
    }
    this.setButtonStates(!!this.selection);
  }

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

  addCharacterTagEventListener() {
    const button = this.shadowRoot.querySelector('.character-tag');
    $(button).on('click', () => {
      if (this.selection.rangeCount) {

 this.cueInterface.selection.forEachBlockInSelection(this.selection.getRangeAt(0),
 element => {
          this.cueInterface.selection.replaceElement(element, 'span', 'quote_attrib');
        });
      }

    });
  }

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

  addEmDashEventListener() {
```

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

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

  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 HTMLElement {
  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.cueInterface.active);
    this.cueInterface.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);
```

6.1.4 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 (see section 6.1.4.4). 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.

6.1.4.1 Custom Field Editor Configuration

The following properties must be defined to configure a custom field editor:

- 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"
```

6.1.4.2 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 6.1.4.1, 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>
```

6.1.4.3 Custom Field Editor cueInterface Properties

A **cueInterface** object is passed to the component. In addition to the basic properties described in section 6.1.1, this **cueInterface** object has a number of properties describing the content item currently loaded to the editor:

articleId
The ID of the current content item.

articleUri
The published URI of the current content item.

contentType
The content type of the current content item.

state
The state of the current content item.

publishedDate
The date on which the current content item was published

6.1.4.4 Custom Field Editor Example

This example creates field editor that allows integer values to be set and displayed using a slider. Note that the value of the field is written/read using the component's **value** attribute.

```
// Creates an object based in the HTML Element prototype
class MySlider extends HTMLElement {
  // Fires when an instance of the element is created
  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>
    `;
  }

  readOnlyCallback() {
    this.shadowRoot.querySelector('#thefield').readOnly = this.cueInterface.readOnly;
  }

  static get observedAttributes() {
    return ['value'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'value') {
      this.shadowRoot.querySelector('input').value = newValue;
    }
  }

  connectedCallback() {
    // getting initial value of value attribute
    const value = this.hasAttribute('value') ?
        this.getAttribute('value') : '';

    // setting value property of input element
    this.shadowRoot.querySelector('input').value = value;
    this.shadowRoot.querySelector('input').addEventListener('input', e => {
      this.setAttribute('value', e.target.value);
    });

    // Add a watcher that will watch on if the custom field is locked.
    // If locked then invoke the callback function and make the field readonly.
    if (this.cueInterface) {
      this.cueInterface.addReadOnlyWatcher(() => this.readOnlyCallback());
    }
  }
}
customElements.define('my-slider', MySlider);
```

6.1.5 Custom Storyline Element Field Editor (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 custom storyline element field editor extension changes the appearance and behavior of a storyline element field. You can, for example, configure CUE to display an integer field in a storyline element as a graphical slider instead of displaying a simple text field. You can also use it to display much more complex components. Examples of two such components, a map component and a data visualization (i.e. table) component are included in the Content Store 7.3 distribution. For details of how to make use of these components, see ??.

> Note that no **`cueInterface`** object is passed to a storyline element field editor. In this respect, storyline element field editors are different from all other CUE Web Component extensions.

6.1.5.1 Custom Storyline Element Field Editor Configuration

The following properties must be defined to configure a custom storyline element field editor:

- 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 (**-**).

modulePath
The URI of the component.

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: "google-map"
    modulePath:  "http://www.example.com/webcomponents/google-map/google-map.js"
```

6.1.5.2 Custom Storyline Element Field Editor Invocation

To use a custom editor for a particular storyline element field, you need to add a **ui:editor** to the field's definition in the **story-element-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 storyline element field editor configuration file.

To use the table editor defined in section 6.1.5.1, for example, you would need to add the following **ui-editor** element to your **story-element-type** definition:

```
<?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="table">
  <ui:label>Table</ui:label>
  <field mime-type="application/json" type="basic" name="tableeditor">
    <ui:editor type="web-component" name="data-visualization" />
  </field>

  <field name="rowheader" type="boolean">
    <ui:label>Use first row as labels</ui:label>
    <ui:value-if-unset>false</ui:value-if-unset>
    <ui:editor-style>settings</ui:editor-style>
  </field>

  <field name="columnheader" type="boolean">
    <ui:label>Use first column as labels</ui:label>
    <ui:value-if-unset>false</ui:value-if-unset>
    <ui:editor-style>settings</ui:editor-style>
  </field>
</story-element-type>
```

6.1.5.3 Custom Storyline Element Field Editor Example

This example is the table editor included in the CUE Content Store's starter pack. The web component looks like this

```
const maxCols = 10;
const maxRows = 6;

class DataVisualization extends HTMLElement {

  static get observedAttributes() {
    return ['value'];
  }

  constructor() {
    super();

    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
        <-- CSS omitted -->
      </style>
      <div class="container grid-placeholder" id="container">
        <div id="showSize" class="show-size"></div>
        <table id="tableContainer" class="grid-table layout-select"></table>

        <div id="contextMenu" class="dropdown context-menu" style="display: none;">
          <ul class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu"
 style="position:static;margin-bottom:5px;">
            <li><a id="insertRowAbove">Insert row above</a></li>
            <li><a id="insertRowBelow">Insert row below</a></li>
            <li><a id="insertColumnLeft">Insert column left</a></li>
            <li><a id="insertColumnRight">Insert column right</a></li>
            <li class="dropdown-divider"></li>
            <li><a id="deleteRow">Delete row</a></li>
            <li><a id="deleteColumn">Delete column</a></li>
          </ul>
        </div>
      </div>
    `;
  }

  connectedCallback() {
    this.resolveData();
  }

  updateEditor(data) {
    this.setAttribute('value', JSON.stringify(data));
  }

  attributeChangedCallback(name, oldValue, newValue) {
    const table = this.shadowRoot.getElementById('tableContainer');

    if (this.isConnected && name === 'value' && newValue !== '' && newValue !==
 oldValue) {
      $(table)
          .find('tr')
          .remove();
      this.resolveData();
    }
  }

  resolveData() {
    const value = this.hasAttribute('value') ? this.getAttribute('value') : '';
```

```
    if (value !== '') {
      const tableData = $.parseJSON(value);
      const rows = tableData.length;
      const cols = tableData[0].length;

      const extraDataString = this.hasAttribute('extra-data')
          ? this.getAttribute('extra-data')
          : '';
      const extraData = extraDataString ? $.parseJSON(extraDataString) : {};
      this.createDataTable(rows, cols, tableData, extraData);
    } else {
      this.createGridSelector();
    }
  }

  createGridSelector() {
    const that = this;
    const table = that.shadowRoot.getElementById('tableContainer');
    const showSizeElement = that.shadowRoot.querySelector('#showSize');

    const clickHandler = function() {
      const target = $(this);
      const cols = target.index() + 1;
      const rows = target.parent().index() + 1;

      that.createDataTable(rows, cols);
      that.prepareTableData();
    };

    for (let i = 0; i < maxRows; i++) {
      const row = document.createElement('tr');
      table.append(row);

      for (let j = 0; j < maxCols; j++) {
        const cell = document.createElement('td');

        cell.append(document.createElement('div'));
        row.append(cell);
      }
    }

    showSizeElement.innerHTML = '0 x 0';

    function clearHighlight(target) {
      target
          .parent()
          .find('div')
          .removeClass('grid-select-active');
    }

    function addHighlight(target) {
      var cellIndex = target.index();
      var rowIndex = target.parent().index();
      var rows = target
          .parent()
          .parent()
          .children();

      for (var i = 0; i <= rowIndex; i++) {
        $(rows[i])
```

```
            .children()
            .eq(cellIndex)
            .prevAll()
            .addBack()
            .find('div')
            .addClass('grid-select-active');
      }

      showSizeElement.innerHTML = rowIndex + 1 + ' x ' + (cellIndex + 1);
    }

    $(table)
        .find('td')
        .on('click', clickHandler);

    $(table)
        .find('td')
        .on('mouseover', function() {
          const target = $(this);
          addHighlight(target);
        });

    $(table).on('mouseout', function() {
      const target = $(this);
      clearHighlight(target);
    });
  }

  prepareTableData() {
    const table = this.shadowRoot.getElementById('tableContainer');
    // A few jQuery helpers for exporting only
    jQuery.fn.pop = [].pop;
    jQuery.fn.shift = [].shift;

    const rows = $(table).find('tr:not(:hidden)');
    const data = [];

    // Convert all existing rows into a loopable array
    rows.each(function() {
      var cells = $(this).find('td input');
      var cellData = [];

      cells.each(function(i, cell) {
        cellData[i] = $(cell).val() || '';
      });

      data.push(cellData);
    });
    this.updateEditor(data);
  }

  createDataTable(rows, cols, tableData = [], extraData = {}) {
    const that = this;
    const table = that.shadowRoot.getElementById('tableContainer');
    const showSizeElement = that.shadowRoot.querySelector('#showSize');
    const dataTableContextMenu = that.shadowRoot.querySelector('#contextMenu');

    $(table)
        .find('tr')
        .remove();
```

```
$(table)
    .parent()
    .removeClass('grid-placeholder');
$(table)
    .addClass('data-table')
    .removeClass('layout-select');

if (showSizeElement) {
  showSizeElement.remove();
}

const createNewRow = function(rowValues = []) {
  const tr = $('<tr>');

  for (let c = 0; c < cols; c++) {
    const cellValue = rowValues[c] || '';
    $(
        '<td><input type="text" class="data-cell" value="' +
        cellValue +
        '"></td>'
    ).appendTo(tr);
  }
  return tr;
};

const createNewColumn = function(position) {
  $(table)
      .find('tr')
      .each(function(index) {
        this.insertCell(position).outerHTML =
            '<td><input type="text" class="data-cell" value=""></td>';
      });
};

function setCaretPosition(element, position) {
  element.childNodes[0].focus();
  element.childNodes[0].setSelectionRange(position, position);
}

const dataUpdateHandler = function(event) {
  const activeCellElement = event.target.parentElement;
  const rowIndex =
      activeCellElement && activeCellElement.parentElement
          ? activeCellElement.parentElement.rowIndex
          : -1;
  const cellIndex =
      activeCellElement && isNaN(activeCellElement.cellIndex)
          ? -1
          : activeCellElement.cellIndex;
  const position = event.target.selectionStart;

  that.prepareTableData();

  if (rowIndex !== -1 && cellIndex !== -1) {
    const cellElement = $(table).find(
        `tr:eq(${rowIndex}) td:eq(${cellIndex})`
    );
    setCaretPosition(cellElement.get(0), position);
  }
};
```

```
const dataTableContextMenuHandler = function(event) {
  const dataElement = $(event.target).parent();
  event.preventDefault();
  dataTableContextMenu.style.display = 'block';
  $(dataTableContextMenu).css({
    left: dataElement.get(0).offsetLeft,
    top: dataElement.get(0).offsetTop,
  });

  $(dataTableContextMenu.querySelector('#insertRowAbove'))
      .unbind()
      .click(function(e) {
        dataElement.parent().before(createNewRow());
        dataUpdateHandler(e);
      });

  $(dataTableContextMenu.querySelector('#insertRowBelow'))
      .unbind()
      .click(function(e) {
        dataElement.parent().after(createNewRow());
        dataUpdateHandler(e);
      });

  $(dataTableContextMenu.querySelector('#deleteRow'))
      .unbind()
      .click(function(e) {
        dataElement.parent().remove();
        dataUpdateHandler(e);
      });

  $(dataTableContextMenu.querySelector('#insertColumnLeft'))
      .unbind()
      .click(function(e) {
        createNewColumn(dataElement[0].cellIndex);
        dataUpdateHandler(e);
      });

  $(dataTableContextMenu.querySelector('#insertColumnRight'))
      .unbind()
      .click(function(e) {
        createNewColumn(dataElement[0].cellIndex + 1);
        dataUpdateHandler(e);
      });

  $(dataTableContextMenu.querySelector('#deleteColumn'))
      .unbind()
      .click(function(e) {
        const columnIndex = dataElement[0].cellIndex;

        $(table)
            .find('tr')
            .each(function(index) {
              this.deleteCell(columnIndex);
            });

        dataUpdateHandler(e);
      });

  $(document).click(function(event) {
```

```
        dataTableContextMenu.style.display = 'none';
      });
      $(document).mousedown(function(event) {
        if (event.which === 3) {
          dataTableContextMenu.style.display = 'none';
        }
      });

      return false;
    };

    for (let r = 0; r < rows; r++) {
      createNewRow(tableData[r]).appendTo(table);
    }

    $(table)
        .find('tr td input')
        .on('input', dataUpdateHandler);
    $(table)
        .find('tr td input')
        .contextmenu(dataTableContextMenuHandler);

    if (extraData.rowheader) {
      $(table)
          .find('tr:eq(0)')
          .addClass('row-heading');
    }

    if (extraData.columnheader) {
      $(table)
          .find('tr')
          .each(function(index, row) {
            $(row)
                .find('td:eq(0)')
                .addClass('column-heading');
          });
    }
  }
}

customElements.define('data-visualization', DataVisualization);
```

As with the custom field editor (see section 6.1.4.4), the value of the field is written/read using the component's **`value`** attribute. In a storyline element field editor, however, additional information may also be supplied via an **`extra-data`** attribute:

- The **`value`** attribute contains the value of the storyline element field that is marked as a web component in the storyline element definition (the **`tableeditor`** field in this case)
- The **`extra-data`** attribute is a JSON structure containing the values of all the other fields in the storyline element (**`rowheader`** and **`columnheader`** in this case)

6.1.5.4 Configuring the Sample Storyline Element Field Editors

The Content Store 7.3 distribution incorporates two sample storyline element types that make use of storyline field editors:

- **`map`**, a storyline element type for maps, plus a supporting web component called **`google-map`**.

- **table**, a storyline element type for editing tables, plus a supporting web component called **data-visualization**.

You will find the storyline element definitions and the supporting web components in the Content Store's **contrib** folder. The storyline element definitions are in the **contrib/starter-pack/storyline-element-types** folder and the web components are in the are in the **contrib/starter-pack/story-element-web-components** folder.

To make use of these storyline element types, in your publications you need to:

1. Upload to the Content Store:
   - The storyline element types
   - A storyline template that contains references to the new storyline element types (for example, the sample template **contrib/starter-pack/storyline-templates/blank.xml**)

   For a description of how to do this, see Manage Shared Resources.
2. Upload the web components to a web location that is accessible to CUE.
3. Add a configuration file to your CUE config folder (**/etc/escenic/cue-web-3.3/**) containing the following definitions:

```
customComponents:
  - name: "google-map"
    modulePath:  "http://your-server-location/google-map/google-map.js"
  - name: "data-visualization"
    modulePath:  "http://your-server-location/data-visualization/data-
visualization.js"
```

   where *your-server-location* is the location to which you uploaded the web components.

6.1.6 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.

A home page panel is typically used to display information from some external system or web site – the example supplied in section 6.1.6.3 displays a Twitter timeline.

6.1.6.1 Home Page Panel Configuration

The following properties must be defined to configure a home page panel:

- id
The tag 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 (**-**).

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

directive
Must be set to **"cf-custom-panel-loader"**.

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.

mimeTypes
For a pure home page panel, this property should be specified as an empty array – `[]`. You can, however, specify values in which case the component will also function as an editor panel. For details of the values that may be specified, see section 6.1.2.1. Note that while home page panels are full width panels, editor side panel width is restricted. Dual-purpose panels must therefore be made to display nicely in both contexts.

homeScreen
Set to **true**.

metadata
Not used. Should be specified as an empty array – `[]`.

active
Set to **false**.

order
Determines the position of this panel button in the column of side panel/home page buttons. The buttons are arranged in numerical order from lowest to highest.

All the properties must be entered as a list item belonging to a **sidePanels** 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:

```
sidePanels:
  - id: "twitter-home-panel"
    name: "Twitter Timelines"
    directive: "cf-custom-panel-loader"
    webComponent:
      modulePath: "http://www.example.com/webcomponents/twitter/twitter-home-panel.js"
      icon: "twitter-home-panel-icon"
    mimeTypes: []
    homeScreen: true
    metadata: []
    active: false
    order: 801
```

6.1.6.2 Home Page Panel cueInterface Properties

A **cueInterface** object is passed to both the main home page panel component and to the icon component. In addition to the basic properties described in section 6.1.1, this **cueInterface** object has the following properties in each case:

For the home page panel component

active
A boolean property that indicates whether or not the panel is currently active.

addActiveWatcher(watcher: Function)
A function that adds a watcher that will be notified when the active state of the home page panel changes. The watcher will be invoked with the new state.

removeActiveWatcher(watcher: Function)
A function that removes a previously added watcher.

For the icon component

active
A boolean property that indicates whether or not the panel is currently active.

addActiveWatcher(watcher: Function)
A function that adds a watcher that will be notified when the active state of the home page panel changes. The watcher will be invoked with the new state.

removeActiveWatcher(watcher: Function)
A function that removes a previously added watcher.

6.1.6.3 Home Page Panel Example

This example displays a Twitter timeline:

```
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
 */
var TwitterTimelineProto = Object.create(HTMLElement.prototype);
  var timelineShadowRoot;

    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>
    `;
  }

  TwitterTimelineProto.attachedCallback = function() {
    twttr.ready(function () {
      twttr.widgets.load();
```

```
      twttr.widgets.createTimeline(
          {
            sourceType: 'profile',
            screenName: 'escenic'
          },
          timelineShadowRoot.querySelector('#timeline'),
          {
            height: 1000
          }
      );
    });
  }
;

document.registerElement('twitter-home-panel', {
      prototype: TwitterTimelineProto
    });

/**
 * Twitter icon
 */
var TwitterIconProto = Object.create(HTMLElement.prototype);
  var iconShadowRoot;

    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">
    `;

    TwitterIconProto.createdCallback = function () {
      var template = thisDoc.querySelector('template#icon').content;
    iconShadowRoot = this.attachShadow({ mode: 'open' });
      iconShadowRoot.appendChild(template.cloneNode(true));
  };

  TwitterIconProto.attachedCallback = function() {
    this.activeStateChanged(this.cueInterface.isActive());
    this.cueInterface.addActiveWatcher(function (active) {
      this.activeStateChanged(active);
    }.bind(this));
  };

  TwitterIconProto.activeStateChanged = function(active) {
    var img = iconShadowRoot.querySelector('img.icon');
    if (active) {
      img.src = this.getAbsolutePath(this.activeIconPath);
    }
    else {
      img.src = this.getAbsolutePath(this.inactiveIconPath);
    }
  };

  TwitterIconProto.getAbsolutePath = function(path) {
    var baseURI = import.meta.url;
    return baseURI.substring(0, baseURI.lastIndexOf('/') + 1) + path;
```

```
    }
;

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

6.1.7 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 **Pages** and **Lists** 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.

A home page metadata section is used to display information about whatever object is currently selected in the home page – the example supplied in section 6.1.7.3 displays a preview of the currently selected image or video.

6.1.7.1 Home Page Metadata Section Configuration

The following properties must be defined to configure an editor metadata section:

- 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:

cf-escenic-search-sidepanel	CUE home screen search panel
cf-latest-opened-leftpanel	CUE home screen latest-opened panel
cf-sections-sidepanel	CUE home screen sections panel
cf-lists-sidepanel	CUE home screen lists 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: ["cf-escenic-search-sidepanel", "cf-latest-opened-leftpanel"]
    directive: "content-preview"
    name: "Preview"
    webComponent:
      modulePath: "http://www.example.com/webcomponents/preview/preview.js"
      icon: "content-preview-icon"
    order: 804
```

6.1.7.2 Home Page Metadata Section cueInterface Properties

A **cueInterface** object is passed to both the main metadata section component and to the icon component. In addition to the basic properties described in section 6.1.1, this **cueInterface** object has the following properties in each case:

For the home page metadata section component

homePanel
An object representing the current home screen panel. The properties and functions exposed by this object vary according to the type of panel as follows:

Search panel
The following functions are exposed:

addFocusedResultWatcher(watcher: Function)
Adds a watcher that will be notified whenever the focus moves to a new search result. The watcher will be invoked with the currently selected result.

Latest opened
The following functions are exposed:

addFocusedResultWatcher(watcher: Function)
Adds a watcher that will be notified whenever the focus moves to a new item.

Section editor
The following functions are exposed:

addFocusedSectionWatcher(watcher: Function)
Adds a watcher that will be notified whenever the focus moves to a section.

For the icon component

active
A boolean property that indicates whether or not the metadata section is currently active.

addActiveWatcher(watcher: Function)
A function that adds a watcher that will be notified when the active state of the home page metadata section changes. The watcher will be invoked with the new state.

removeActiveWatcher(watcher: Function)
A function that removes a previously added watcher.

6.1.7.3 Home Page Metadata Section Example

This example displays a preview of the currently selected image or video.

```
class PreviewPanel extends HTMLElement {
  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>
    `;

    PreviewPanelProto.attachedCallback = function () {
      this.cueInterface.homePanel.addFocusedResultWatcher(function (result) {
        this.focusedResultChanged(result);
      }.bind(this));
  };

  PreviewPanelProto.blobUrl = undefined;

  PreviewPanelProto.focusedResultChanged = function (result) {
    if (this.blobUrl) {
      window.URL.revokeObjectURL(this.blobUrl);
    }
    var hide = true;
    if ( result && result.link.mimeType) {
      hide = !_.includes(['x-ece/picture', 'x-ece/video'],
 result.link.mimeType.format());
    }
    this.info.hidden = hide;
    if (!hide) {
      this.fetchPreview(result);
      }

      connectedCallback() {
        this.cueInterface.homePanel.addFocusedResultWatcher((result, content) => {
          this.focusedResultChanged(result, content);
        });
      }

    PreviewPanelProto.getBinaryLinkFromDocument = function (document) {
      var resolver = function (namespace) {
```

```
        if (namespace === 'atom') {
          return 'http://www.w3.org/2005/Atom';
    }
  };

  return document
          .evaluate('./atom:entry/atom:link[@rel="edit-media"]', document,
 resolver).iterateNext()
          .attributes.getNamedItem('href').value;
    };

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

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

    xhr.send(null);
  };

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

document.registerElement('content-preview', {
      prototype: PreviewPanelProto
    });
/**
 * Creating the icon (if required)
 */
var PreviewIconProto = Object.create(HTMLElement.prototype);
  var iconShadowRoot;

    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>
    `;
  }

  PreviewIconProto.attachedCallback = function() {
    this.activeStateChanged(this.cueInterface.isActive());
    this.cueInterface.addActiveWatcher(function (active) {
      this.activeStateChanged(active);
    }.bind(this));
  };

  PreviewIconProto.activeStateChanged = function(active) {
    var icon = iconShadowRoot.querySelector('.icon');
    if (active) {
      $(icon).addClass('active');
    }
    else {
      $(icon).removeClass('active');
    }
  };

document.registerElement('content-preview-icon', {
      prototype: PreviewIconProto
    });
```

6.1.8 Content Summary Extension

A content summary extension is a web component that is used to display additional information in the content cards used to represent content items. A summary extension is only displayed when the content card is in its expanded state. It is displayed above the summary fields on the content card.

6.1.8.1 Summary Extension Configuration

The following properties must be defined to configure a content summary extension:

- **`name`**
  The name of the web component that will provide the summary extension The name you specify **must** contain a hyphen. Remember also that the **`name`** property name must be preceded by a hyphen (`-`).

`modulePath`
The URI of the component.

attributes
An optional set of *key:value* pairs to be passed to the 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: "my-summary-extension"
     modulePath: "http://myhost:8180/webcomponent/my-summary.js"
     attributes:
       one: "first"
       two: "second"
```

Content summary extensions are only actually displayed if they are included in a content type's content summary definition in the **content-type** resource. To make my-summary-extension appear on the content cards of your story content items, for example, you would need to include the following element in your publication's **content-type** resource as a child of the **summary** element in your **story** content type definition.

```
<ui:additional-editor>my-summary-extension</ui:additional-editor>
```

6.1.8.2 Summary Extension cueInterface Properties

A **cueInterface** object is passed to the component. Unlike other CUE web components, a summary extension's **cueInterface** object does **not** expose the basic properties described in section 6.1.1. The properties exposed by this **cueInterface** object are:

item
The content summary data displayed on the section page, as a JSON object.

content
The entire content summary as a JSON object.

link
The CUE content editor tab URL for the content item represented by the summary.

6.1.8.3 Summary Extension Example

This example simply displays some example text plus a link to the CUE content editor tab for the content item represented by the summary:

```
class MyAdditionalEditor extends HTMLElement {
  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;
        }

        .wc-additional-editor-group {
          with: 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="" class="wc-
additional-editor-link" id="wc-additional-editor-editor-link">Open content</a></span>
        </div>

      </div>
    `;
  }

  connectedCallback() {
    if (this.cueInterface) {
      const title = this.cueInterface.item.values.title;
```

```
        this.shadowRoot.querySelector('#wc-additional-editor-title').innerHTML = title;
        const type = this.cueInterface.content.descriptor.uiProperties.label;
        this.shadowRoot.querySelector('#wc-additional-editor-type').innerHTML = type;
        const editorLink = this.cueInterface.link;
        this.shadowRoot.querySelector('#wc-additional-editor-editor-link').href =
 editorLink;
      }
    }
}
customElements.define('my-additional-editor', MyAdditionalEditor);
```

6.2 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.

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.

6.2.1 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-3.3/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
    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.

triggers

Is a list of one or more triggers defining when CUE is to **POST** the content item to the enrichment service.

Optionally, a trigger may include a **mimeType** setting. This is a list of MIME type for which the trigger is to fire. If you specify this property, then the trigger will only fire for the MIME types specified in the list. The possible MIME types that may appear are:

x-ece/story	CUE story-type content item
x-ece/picture	CUE image content item
x-ece/video	CUE video content item
x-ece/gallery	CUE gallery content item
x-ece/event	CUE event content item
x-ece/new-content	New CUE content that has not yet been saved
x-ece/section	CUE section
x-ece/section-page	CUE section page
x-ece/list	CUE list
x-ece/*	All kinds of CUE content
x-cci/assignment	Newsgate assignment
x-cci/storyfolder	Newsgate story folder
x-cci/*	All kinds of Newsgate content
/	All kinds of CUE content

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 6.2.1.1 for further information.

6.2.1.1 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 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.

`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.

`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 6.2.1.3.

6.2.1.2 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/"
```

6.2.1.3 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>
```

6.2.1.4 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-3.3/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 you specify here must also appear in the enrichment service's list of triggers. For example:

```
  - name: print-article
    href: http://host:port/printArticle
    title: Print Article
    triggers:
      - name: on-click
      - name: on-print-menu-item-click
```

publication
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. This property is optional: if you omit it then the menu entry will appear in the context menu of content items belonging to any publication.

mimeTypes
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. This property is optional: if you omit it then the menu entry will appear in the context menu of content items of all MIME types.

contentTypes
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. This property is optional: if you omit it then the menu entry will appear in the context menu of content items of all types.

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.

6.2.2 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:

`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. A service that checks for unpublished relations, for example, could send a **`400`** response if it found any unpublished relations. CUE would then cancel the trigger operation (publish, presumably) and display an error dialog containing the response message – "Cannot publish. There are unpublished related articles."

`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: 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/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 **`POST`**ed by CUE, then it is assumed to be a modified version of the **`POST`**ed content item (returned, for example, from a grammar correction service). If the Atom entry contains all the fields originally **`POST`**ed 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 **`POST`**ed 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).

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.

6.2.3 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>
```

6.2.3.1 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.

6.2.3.2 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.

6.2.4 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.escenic.com:

Diving into enrichment services

6.3 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.

6.3.1 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/escenic/cue-web-3.3/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 6.2.1.2).

`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*`/webservice/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 3.6).

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:
  mimeTypes: [mime-type,...]
```

or:

```
triggers:
  urlPatterns: [url-pattern,...]
```

If **mimeTypes** 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**.

6.3.2 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 6.3.1).

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 6.3.1).

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.

6.3.3 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.

6.4 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.

6.4.1 Content Creation URL Structure

A content creation URL must have the following overall structure:

```
http://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 example script 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-ece/new-content; type=story**.

extra=*content-definition*
content-definition is a JSON value with the following structure:

```
{
  "modelURI": {
    "string": "model-uri",
    "$class":"URI"
  },
  "homeSectionUri":"home-section-uri",
  "values": "content-item-field-values"
}
```

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 array of field values defining the content you want to add to the new content item. You can leave this array 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 predefine 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>"
}
```

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.

6.4.2 Example Script

The following example bash script shows how to construct a content creation URL and submit it to CUE.

```
#!/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="http://your-cue-host/cue-web"
webservice="http://your-escenic-webservice-host/webservice"
homesection="$webservice/escenic/section/section-id"
modeluri="$webservice/escenic/publication/demopub/model/content-type/story"
relations=("Content_ID1|group1" "Content_ID2|group2") # eg. ("2015|images" "2016|
images")

mimetype="x-ece/new-content; type=story"
sourceid=`date '+%y%m%d-%H%M%S'`
title="My Title"
body="<p>My first paragraph.</p><p>My second paragraph.</p>"

relationsJSON="["
for i in "${relations[@]}"
do :
   contentUri=`echo $i | cut -d'|' -f1`
   contentUri="$webservice/escenic/content/$contentUri"
   group=`echo $i | cut -d'|' -f2`
   relationsJSON="$relationsJSON{\"group\":\"${group}\",\"contentUri\":
\"${contentUri}\"},"
done
if [ ${#relationsJSON} -gt 1 ]
then
    relationsJSON="${relationsJSON%?}]"
else
    relationsJSON="$relationsJSON]"
fi
extra="{\"modelURI\":{\"string\":\"${modeluri}\",\"\$class\":\"URI\"},\"homeSectionUri
\":\"${homesection}\",\"values\":{\"title\":\"${title}\",\"body\":\"${body}\"},
\"relations\":${relationsJSON}}"

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 with the title "My Title". You would need to replace *your-cue-host* and *your-escenic-webservice-host* with the correct host names and replace *section-id* with the ID of a section in one of your publications before running it. Otherwise, as long as you have a content type called **`story`**, it should work.

6.5 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.

6.5.1 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
    ```

6.6 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-3.3/config.yml** for editing. For example:

   ```
   # nano /etc/escenic/cue-web-3.3/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-3.3
   ```

   This reconfigures CUE with the changes you have made.