CUE

User Guide

2.2.0-9

escenic

A CCI COMPANY

Table of Contents

1 Introduction

Welcome to CUE, Escenic's new browser-based application for newsroom staff! CUE is now Escenic's primary interface for all newsroom staff, and will eventually completely replace Content Studio. CUE is a joint CCI-Escenic project and is capable of acting as an interface to CCI Newsgate systems as well as Escenic systems, so for newsrooms that use both CCI Newsgate and Escenic, 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 5.

2 Installing CUE

> CUE 2.2.0-9 requires version 6.1 of the Escenic Content Engine and version 3.7 of CCI Newsgate with the latest patch applied. 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 Escenic back-end, make sure your Escenic Content Engine is upgraded to version 6.1 or higher
> - If CUE is to use a CCI Newsgate back-end, make sure that the version is 3.7, and that the latest patch for that version has been applied
>
> CUE also requires the use of an **SSE Proxy** to manage the delivery of Server-sent Events from the Escenic Content Engine to CUE clients. This means that an Escenic SSE Proxy must have been installed and configured to manage SSE for the Content Engine, and the Content Engine must have been configured to direct SSE connection requests to the SSE Proxy. For detailed information on how to do this, see the SSE Proxy documentation.

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 Escenic plug-in. Although it needs to be connected to an Escenic Content Engine 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 Engine 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 Escenic download credentials (the same ones you use to access the Escenic Maven repository). If you do not have any download credentials, please contact Escenic 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:

```
# apt-get install cue-web-2.2
```

7. Download and install nginx

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

3 Configuring CUE

To configure CUE:

1. If necessary, switch user to root.

   ```
   $ sudo su
   ```

2. Open **/etc/escenic/cue-web-2.2/config.yml** for editing. For example

   ```
   # nano /etc/escenic/cue-web-2.2/config.yml
   ```

   This is a new file, so it will be empty.

3. Enter the following:

   ```
   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 Engine CUE is to provide access to and *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 you should remove the **newsgate: ""** line.

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

4. Save the file.

5. Enter:

   ```
   # dpkg-reconfigure cue-web-2.2
   ```

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

6. 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;
   }
   ```

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

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

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

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

9. 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;
   }
   ```

10. Open **/etc/nginx/default-site/webservice.conf** for editing and add the contents described in section 3.1.
11. 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 Escenic Content Engine'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 Engine 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 Engine 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" 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';
        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 Escenic Content Engine 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 Engine/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 Escenic, 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**-2.2).

When you are configuring third-party authentication for the Content Engine 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 CUE 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 Engine, 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-2.2
```

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**-2.2). 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 Engine, 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-2.2
```

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

4 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

4.1 Getting Started

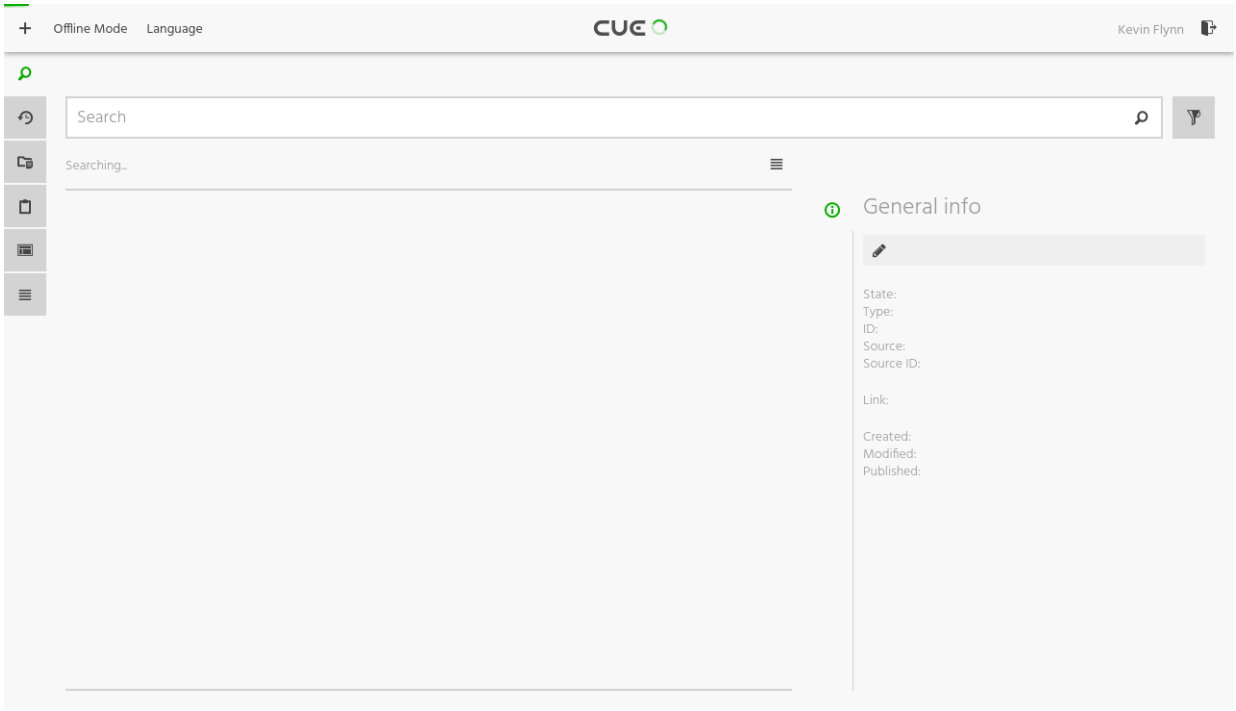
CUE is a **tab-based** webapp. What we mean by this is that CUE uses browser tabs as document containers. Every content item or section page you open or create is opened in a new browser tab, rather than everything happening in a single tab. If you've used Google Docs or Microsoft's Office Online before, 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 4.3 for further information.

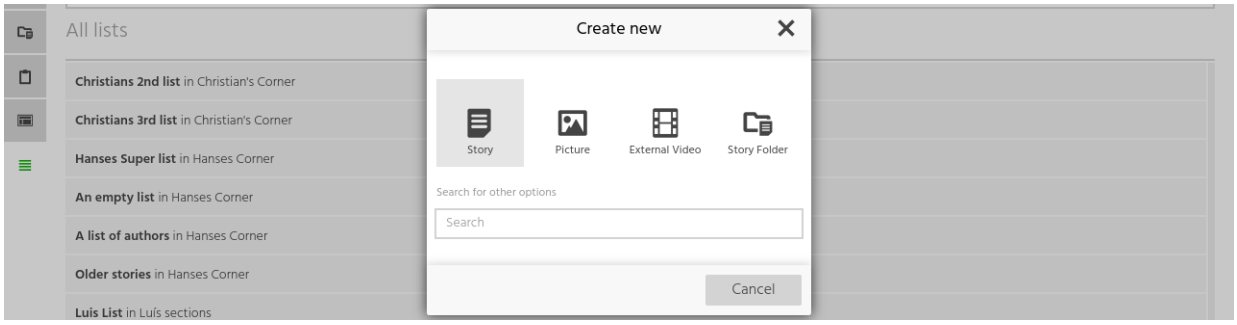
This button displays the **Recent** panel, a list of documents you have been working with recently.

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.

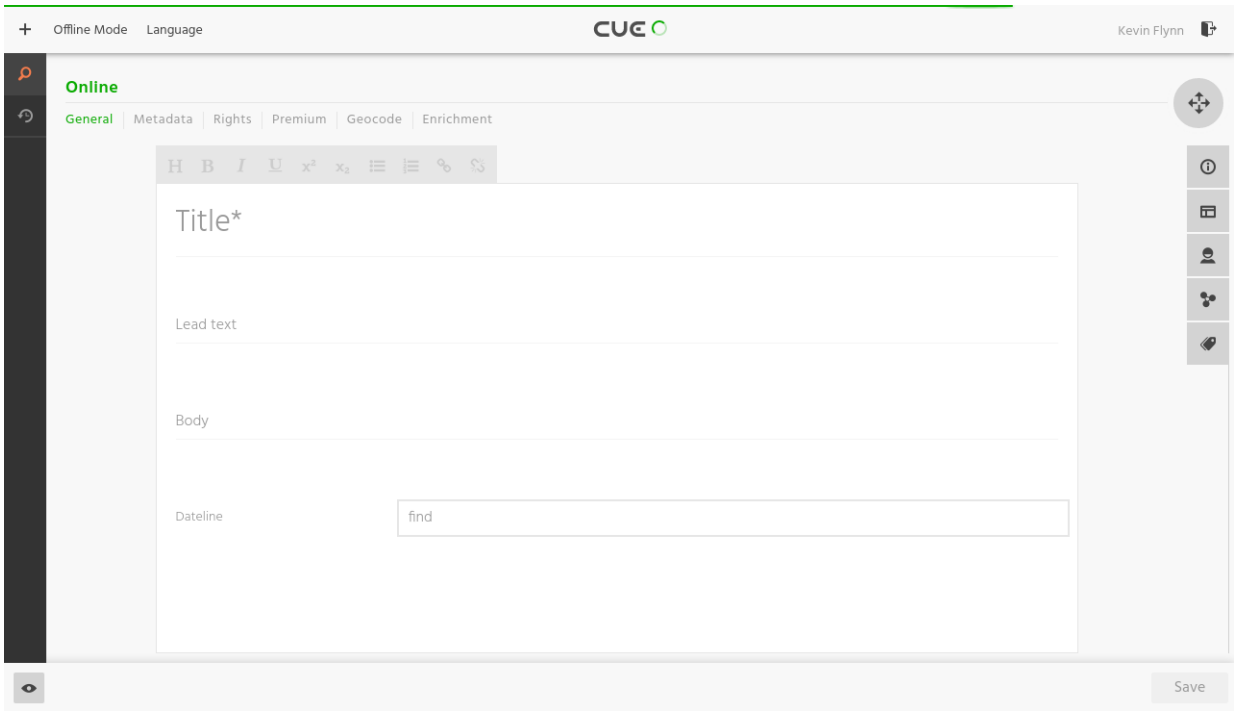
4.2 Creating Content

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).

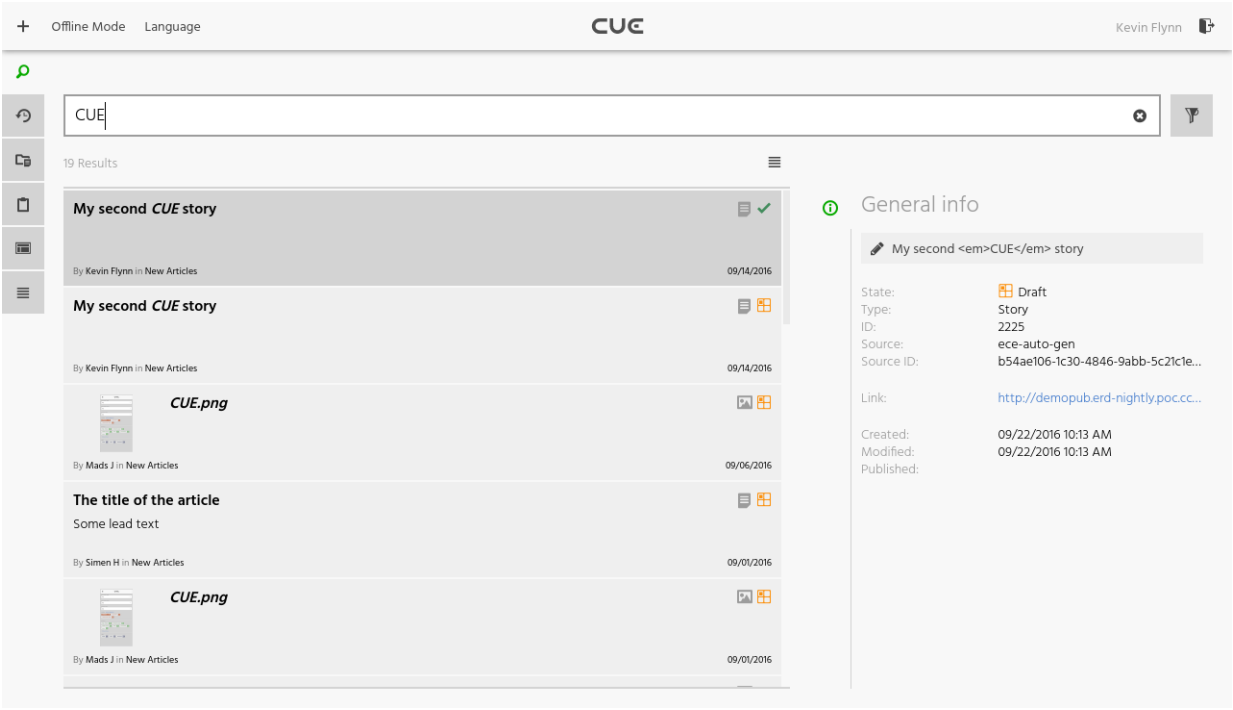
When you create a content item in this way using the menu bar + button, it is added to your publication's default section (usually called **New Articles** or something similar). You can move it, or add it to other sections as well if you want. You can, however, also create content items from the home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

4.3 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.

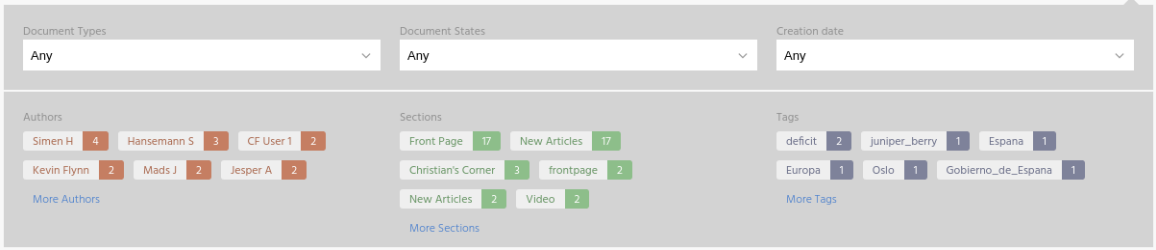
For more information about searching for content, see section 4.4.

4.4 Searching for Content

Video: searching for content in CUE

You've already seen in section 4.3 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. You can drag the side bar out across the page in order to make more space to work if necessary.

4.5 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.

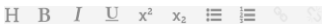
Content item fields are usually divided into groups that are displayed on separate pages. The group names are listed at the top of the content editor, and you can switch between the pages by clicking on the group names. The most commonly-used fields are displayed in the first, default group, while less-used 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 4.5.3.

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 4.5.1 for details.

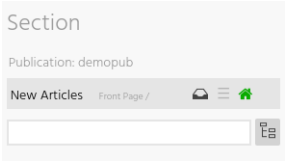
Displays **Authors** – the authors of the document. You can edit the document's list of authors here: see section 4.5.2 for details.

Displays **Related** – the documents related to this document. You can add and remove relations here: see section 4.5.3 for details.

Displays **Tags** – the document's tags. You can add and remove tags here: see section 4.5.4 for details.

4.5.1 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.

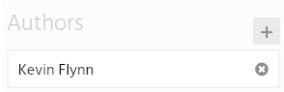
Adds/removes the content item to/from this section's lists. A content item can be in several lists at the same time.

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.

4.5.2 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.

4.5.3 Managing Relations

Relations between documents are very important in Escenic 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.

4.5.4 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.

4.6 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.

4.7 Multimedia Publishing

If CUE is connected to a CCI NewsGate system as well as an Escenic Content Engine, 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.

4.8 NewsGate Story Folders

If CUE is connected to a CCI NewsGate system as well as an Escenic Content Engine, 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.

4.9 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.

5 Extending CUE

CUE is more than a simple editor for Content Engine - 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 5.1 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 5.2 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 5.3 for further information.

5.1 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.

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 an HTML 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`**-2.2). 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.

5.1.1 Creating a Web Component

A web component is a specially structured HTML file. It contains:

- One or more **`template`** elements containing the HTML needed to define the required component. A **`template`** element can contain a **`style`** element that defines **local** CSS styles. These styles are **only** applied to HTML elements inside that **`template`** – they will not affect any elements in documents where the web component is displayed or the elements contained in other **`template`** elements.
- A **`script`** element containing the Javascript code needed to control the component.

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

```
<!-- Template for the actual web component -->
<template id="component">
  <style>
    :host { width: 100%; display: block; } /* Styles the web component tag */
  </style>

  <!-- Add your web component HTML here -->

</template>

<!-- Template for the web component icon (if required) -->
<template id="icon">

  <style>
    :host { margin: 0; padding: 2px; display: block; } /* Styles the web component
 icon tag */
  </style>
```

```
  <!-- Add your web component icon HTML here -->

</template>

<script>
  (function(window, document) {

    var thatDoc = document;
    var thisDoc = (thatDoc._currentScript || thatDoc.currentScript).ownerDocument;

    /**
     * Creating the web component
     */
    var MyComponentProto = Object.create(HTMLElement.prototype);
    var myComponentShadowRoot;

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

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

    document.registerElement('my-component', {
      prototype: MyComponentProto
    });


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

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

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

    document.registerElement('my-component-icon', {
      prototype: MyComponentIcon
    });

  })(window, document);
</script>
```

Field editor web components have no use for an icon, so in this case the icon **`template`** element and corresponding Javascript code can be omitted.

The cueInterface object

CUE always passes a **`cueInterface`** object to the web component. **`cueInterface`** always has the following basic properties:

`escenic.endpoint`
: The URI of the Content Engine 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.

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

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 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`
: The content of this object, encoded as a data URL

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 5.1.2.3 for an example of how to construct a draggable object.

5.1.2 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 5.1.2.3 displays images from the Pixabay web site.

5.1.2.1 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:

`htmlImport`
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`	Escenic content editor for items other than images, videos, galleries and events
`x-ece/image`	Escenic image editor
`x-ece/video`	Escenic video editor
`x-ece/gallery`	Escenic gallery editor
`x-ece/event`	Escenic event editor
`x-ece/new-content`	Editor containing new Escenic content that has not yet been saved
`x-ece/section`	Escenic section editor
`x-ece/section-page`	Escenic section page editor
`x-ece/list`	Escenic list editor
`x-ece/*`	All Escenic 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:
      htmlImport: "http://apps.poc.ccieurope.com/webcomponents/image-search/image-
search.html"
      icon: "image-search-icon"
    mimeTypes: ['*/*']
    homeScreen: true
    metadata: []
    active: false
    order: 802
```

5.1.2.2 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 5.1.1, 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.

5.1.2.3 Editor Side Panel Example

This example searches the Pixabay web site for images.

```
template id="panel">
  <style>
    :host { margin: 0; padding: 0; width: 100%; height: 100%; display: flex; }
    .content-wrapper { flex-grow: 1; flex-direction: column; margin: 0 20px 0 20px;
 height: calc(86vh); }
    #wrapper { flex-grow: 1; flex-direction: column; position: relative; overflow:
 auto; height: 100%; }
    .search { padding-bottom: 20px; }
    #images { overflow: auto; position: absolute; top: 0; bottom: 0; left: 0; right:
 0; }
    #images img { max-height: 300px; max-width: 150px; padding-right: 20px; cursor:
 pointer; }
  </style>
  <div class="content-wrapper">
    <h1>Image Search</h1>

    <div class="search">
      <input id="search-term" type="textbox" value="">
      <input id="search" type="button" value="Search">
    </div>

    <div id="wrapper">
      <div id="images"></div>
    </div>
  </div>
</template>

<template id="icon">
  <style>
    :host { margin: 0; padding: 6px; display: block; }
    img { max-width: 80%; position: relative; top: 3px; left: 3px; }
  </style>
  <img class="icon">
</template>

<script>
  (function(window, document) {

    var thatDoc = document;
    var thisDoc = (thatDoc._currentScript || thatDoc.currentScript).ownerDocument;

    /**
     * Image search
     */
    var ImageSearchProto = Object.create(HTMLElement.prototype);
    var panelShadowRoot;

    ImageSearchProto.createdCallback = function () {
```

```
      var template = thisDoc.querySelector('template#panel').content;
      panelShadowRoot = this.attachShadow({ mode: 'open' });
      panelShadowRoot.appendChild(template.cloneNode(true));
    };

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

    ImageSearchProto.search = function () {
      var searchTerm = panelShadowRoot.querySelector('#search-term').value;
      searchTerm = searchTerm.replace(' ', '+');
      this.loadImages(searchTerm);
    };

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

    document.registerElement('image-search', {
      prototype: ImageSearchProto
    });

    /**
     * Image search icon
     */
    var ImageSearchIconProto = Object.create(HTMLElement.prototype);
    var iconShadowRoot;

    ImageSearchIconProto.activeIconPath = 'image-search-icon-active.png';
    ImageSearchIconProto.inactiveIconPath = 'image-search-icon.png';
    ImageSearchIconProto.editorIconPath = 'image-search-icon-editor.png';

    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);
      }
    };

    ImageSearchIconProto.getAbsolutePath = function (path) {
      var baseURI = thisDoc.querySelector('template#icon').baseURI;
      return baseURI.substring(0, baseURI.lastIndexOf('/') + 1) + path;
    };

    document.registerElement('image-search-icon', {
      prototype: ImageSearchIconProto
    });

  })(window, document);
</script>
```

```
<script>
  var imageSearch;
  (function (imageSearch) {
    imageSearch.dragStart = function (event) {
      var imageData = JSON.parse(event.currentTarget.getAttribute('data-hit'));
      var dataUrl = this.getDataUrlFromImage(event.currentTarget,
 imageData.webformatWidth, imageData.webformatHeight);
      var 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) {
      var canvas = document.createElement('canvas');
      canvas.width = width;
      canvas.height = height;

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

      return canvas.toDataURL('image/jpeg');
    };
  })(imageSearch || (imageSearch = {}));
</script>
```

5.1.3 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 5.1.2.3 displays the history of the content item being edited.

5.1.3.1 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:

htmlImport
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	Escenic content editor for items other than images, videos, galleries and events
x-ece/image	Escenic image editor
x-ece/video	Escenic video editor
x-ece/gallery	Escenic gallery editor
x-ece/event	Escenic event editor
x-ece/new-content	Editor containing new Escenic content that has not yet been saved
x-ece/section	Escenic section editor
x-ece/section-page	Escenic section page editor
x-ece/list	Escenic list editor
x-ece/*	All Escenic 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:
        htmlImport: "http://apps.poc.ccieurope.com/webcomponents/history/history.html"
        icon: "content-history-icon"
      mimeTypes: ["x-ece/story"]
```

```
        order: 803
```

5.1.3.2 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 5.1.1, 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. Escenic 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 5.1.1).

`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. Escenic 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 5.1.1).

`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. Escenic 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 5.1.1).

`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.

5.1.3.3 Editor Metadata Section Example

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

```
<template id="panel">
  <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>
</template>

<template id="icon">
  <style>
    :host {
      margin: 0;
      display: block;
    }
    .icon:before {
      font: 16px 'cf';
      font-style: normal;
      font-weight: normal;
      color: #444444;
      padding: 0 0 0 14px;
      line-height: 44px;
      content: '\e8b9';
      -webkit-font-smoothing: antialiased;
      -moz-osx-font-smoothing: greyscale;
    }
    .icon.active:before {
      color: #09ab00;
    }
  </style>
  <span class="icon"></span>
</template>

<script>
  (function(window, document) {

    var thatDoc = document;
    var thisDoc = (thatDoc._currentScript || thatdoc.currentScript).ownerDocument;

    /**
     * History panel
     */
    var HistoryProto = Object.create(HTMLElement.prototype);
    var panelShadowRoot;

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

    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
      });

      /**
       * History icon
       */
      var HistoryIconProto = Object.create(HTMLElement.prototype);
      var iconShadowRoot;

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

      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
      });

    })(window, document);
  </script>
```

5.1.4 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 5.1.4.4). 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.

5.1.4.1 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.

htmlImport
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"
    htmlImport:  "http://apps.poc.ccieurope.com/webcomponents/my-slider.html"
```

5.1.4.2 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 5.1.4.2, 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>
```

5.1.4.3 Custom Field Editor cueInterface Properties

A **cueInterface** object is passed to the component. In addition to the basic properties described in section 5.1.1, 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

5.1.4.4 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.

```
<template>
  <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>
</template>

<script>
(function() {
    var thatDoc = document;
    var thisDoc =  (thatDoc._currentScript || thatDoc.currentScript).ownerDocument;
    // Creates an object based in the HTML Element prototype
    var MySliderProto = Object.create(HTMLElement.prototype);
    // Fires when an instance of the element is created
    MySliderProto.createdCallback = function() {
       // getting initial value of value attribute
       var value = this.hasAttribute('value') ?
                   this.getAttribute('value') : '';
       var template = thisDoc.querySelector('template').content;
       // setting value property of input element
       template.querySelector('input').value = value;
       var shadowRoot = this.attachShadow({ mode: 'open' });
       shadowRoot.appendChild(template.cloneNode(true));
       shadowRoot.querySelector('input').addEventListener('input', function (e) {
         this.setAttribute('value', e.target.value);
       }.bind(this));
    };
   MySliderProto.attributeChangedCallback = function (name, oldValue, newValue) {
     if (name === 'value') {
       this.shadowRoot.querySelector('input').value = newValue;
     }
    };
    document.registerElement('my-slider', {
        prototype: MySliderProto
```

```
    });
})();
</script>
```

5.1.5 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 5.1.5.3 displays a Twitter timeline.

5.1.5.1 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:

`htmlImport`
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 5.1.2.1. 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:
      htmlImport: "http://apps.poc.ccieurope.com/webcomponents/twitter/twitter-home-
panel.html"
      icon: "twitter-home-panel-icon"
    mimeTypes: []
    homeScreen: true
    metadata: []
    active: false
    order: 801
```

5.1.5.2 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 5.1.1, 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.

5.1.5.3 Home Page Panel Example

This example displays a Twitter timeline:

```
<template id="panel">
  <style>
    :host { margin: 0 20px 0 20px; padding: 0; width: 100%; display: block; }
  </style>
  <h1>Twitter Timelines</h1>
  <div id="timeline"></div>
```

```
</template>

<template id="icon">
  <style>
    :host { margin: 0; padding: 2px; display: block; }
    img { max-width: 80%; position: relative; top: 3px; left: 3px; }
  </style>
  <img class="icon">
</template>

<script>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"));</script>

<script>
  (function(window, document) {

    var thatDoc = document;
    var thisDoc = (thatDoc._currentScript || thatDoc.currentScript).ownerDocument;

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

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

    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;

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

    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 = thisDoc.querySelector('template#icon').baseURI;
      return baseURI.substring(0, baseURI.lastIndexOf('/') + 1) + path;
    };

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

  })(window, document);
</script>
```

5.1.6 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 5.1.6.3 displays a preview of the currently selected image or video.

5.1.6.1 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`	Escenic home screen search panel
`cf-latest-opened-leftpanel`	Escenic home screen latest-opened panel
`cf-sections-sidepanel`	Escenic home screen sections panel
`cf-lists-sidepanel`	Escenic 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:

`htmlImport`
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:
      htmlImport: "http://apps.poc.ccieurope.com/webcomponents/preview/preview.html"
      icon: "content-preview-icon"
    order: 804
```

5.1.6.2 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 5.1.1, 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.

5.1.6.3 Home Page Metadata Section Example

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

```
<template id="panel">
  <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>

</template>

<template id="icon">
  <style>
    :host {
      margin: 0;
      display: block;
    }
    .icon:before {
      font: 16px 'cf';
      font-style: normal;
      font-weight: normal;
      color: #444444;
      padding: 0 0 0 14px;
      line-height: 44px;
      content: '\e879';
      -webkit-font-smoothing: antialiased;
      -moz-osx-font-smoothing: greyscale;
    }
    .icon.active:before {
      color: #09ab00;
    }
  </style>
  <span class="icon"></span>
</template>

<script>
  (function(window, document) {

    var thatDoc = document;
    var thisDoc = (thatDoc._currentScript || thatDoc.currentScript).ownerDocument;

    /**
     * Preview panel
     */
    var PreviewPanelProto = Object.create(HTMLElement.prototype);
    var panelShadowRoot;

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

    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);
      }
    };

    PreviewPanelProto.fetchPreview = function (result) {
      var addCredentials = function (xhr) {
        xhr.setRequestHeader('Authorization', this.cueInterface.escenic.credentials);
      }.bind(this);

      $.ajax({
        url: result.link.uri.toString(),
        type: 'GET',
        beforeSend: addCredentials
      })
          .done(function (document) {
            var binaryLink = this.getBinaryLinkFromDocument(document);
            this.showPreview(binaryLink, result.link.mimeType.format());
          }.bind(this))
          .fail(function (error) {
            console.error(error);
          });
    };

    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
    });

    /**
     * Preview icon
     */
    var PreviewIconProto = Object.create(HTMLElement.prototype);
    var iconShadowRoot;

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

    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
    });

  })(window, document);
```

```
</script>
```

5.2 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.

5.2.1 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-2.2/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. 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. For example:

```
  - name: service-name
    href: http://host:port/service-url
    title: service-title
    triggers:
      - name: trigger-name
        properties:
          property-name: property-value
```

```
            property-name: property-value
```

See section 5.2.1.1 for further information.

5.2.1.1 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-draft`
Before saving, when the user changes the state to **Draft**. No properties required.

`after-save-state-draft`
After saving, when the user changes the state to **Draft**. No properties required.

`before-save-state-submitted`
Before saving, when the user changes the state to **Submitted**. No properties required.

`after-save-state-submitted`
After saving, when the user changes the state to **Submitted**. No properties required.

`before-save-state-approved`
Before saving, when the user changes the state to **Approved**. No properties required.

`after-save-state-approved`
After saving, when the user changes the state to **Approved**. No properties required.

`before-save-state-published`
Before saving, when the user changes the state to **Published**. No properties required.

`after-save-state-published`
After saving, when the user changes the state to **Published**. No properties required.

`before-save-state-deleted`
Before saving, when the user changes the state to **Deleted**. No properties required.

`after-save-state-deleted`
After saving, when the user changes the state to **Deleted**. 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 interval (in seconds) to wait as a property: **`interval:`** *n*.

5.2.1.2 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
```

5.2.2 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 Engine 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 Engine (via the Content Engine'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 this link contains the same 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). 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 the **<link rel="self"/>** element contains a different URL, then CUE opens the referenced content item in a new editor and completes the operation that triggered the enrichment service call (saving 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 and the URL of a new enrichment service endpoint. 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 new 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.

5.2.3 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

5.3 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.

5.3.1 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. For more about this, see ??

home-section-uri
: Is the web service URI of the section to which you want to add the new content item. For more about this, see ??

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 Escenic **`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.

5.3.2 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"

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>"

extra="{\"modelURI\":{\"string\":\"${modeluri}\",\"\$class\":\"URI\"},\"homeSectionUri
\":\"${homesection}\",\"values\":{\"title\":\"${title}\",\"body\":\"${body}\"}}"
```

```
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.