CUE Content Store

Publication Design Guide

7.2.3-4

Table of Contents

1 Introduction

The CUE Content Store is a platform for building large, sophisticated multi-platform publishing operations. It provides editorial staff with a streamlined production environment in which they can concentrate on the production, editing and publishing of content for one or **publications**. A publication is a collection of content that may be published to a variety of online and offline targets. Within the CUE Content Store, the content for a particular target web site/print publication is stored in a **publication**.

All publications have a similar overall structure, composed of:

content items
: Stories (i.e rich text documents) of various kinds, images, videos, audio files and so on.

sections
: Containers for content items. Sections are organized in a tree structure. All content items must belong to a section. A content item can belong to mutiple sections but must always have one and only one **home section**. This section structure typically maps on to the navigation/section structure of the target web site, but it does not have to.

section pages
: A section can have one or more section pages, only one of which can be active at any given time. Section pages have an internal structure of containers called **areas** in which content items can be placed (or **desked**) by editorial staff in order to enable manual curation of a web site's front pages.

lists
: A list is an ordered list of content items that can be created and edited in CUE. Lists belong to sections and can be desked on section pages in the same way as individual content items.

inboxes
: An inbox is a list of content items that can be created and edited in CUE. Inboxes belong to sections and are mostly used as a workflow tool: they are a convenient way of organising content and passing it between users.

tags
: Tags are specially defined names that can optionally be associated with content items in order to categorize them for search and retrieval purposes. A journalist might, for example tag a travel article about Thailand with the tags `Travel` and `Thailand`. Tags can optionally be organized in hierarchies, in order to be able to represent logical associations between the concepts they represent. `Bangkok`, for example, might belong to `Thailand`, which in turn might belong to `Asia`.

All CUE publications have this overall structure, which is reflected in the CUE editor. The details of a publication's structure, however, are customer-defined:

- What types of content items are available, and the internal structure of those content types – what fields they can contain, the type and internal structure of the fields, constraints on what the fields may contain, and so on
- What sections are available, and the section tree structure
- The internal section page structure – what areas are available, how they are structured, and constraints determining what may be desked in each area

- What lists and inboxes are available in each section of a publication
- What tags are available, and how they are organized

A publication's section tree is directly editable in the CUE editor, and is therefore under editorial control. This is also the case for lists and inboxes. The tag structures available for use in publications are defined using the escenic-admin web application, and the tags in them can be edited in CUE. This manual is primarily concerned with the other two aspects of publication design: content type definition and section page definition. It will also cover the whole process of creating a publication from scratch.

1.1 Content Items and Types

Content items are the central objects in the CUE publicaton structure. They are generic containers for all the kinds of content you might want on your web site: news stories; magazine articles; theater, film, book and restaurant reviews; obituaries; interviews; stock market reports; photos; video clips; audio files; attached documents such as PDF files - the list is very long.

In addition to holding all these different kinds of content, content items also contain additional information about the content: **metadata** such as the name of the author and the article's publication history. Content items can also contain **relations** to other content items. A news story, for example, might contain relations to:

- Images to be displayed with the story
- Related news stories to be displayed as links in a "More about..." box
- A background video report to be displayed alongside the story.

Moreover, content items have an internal structure that varies according to content type, and different organizations can have very different requirements with regard to content item structure. For this reason, content items are customer-definable objects. All the **content types** required in a particular CUE installation are specified in a content type definition file called the **`content-type`** resource.

The **`content-type`** resource defines and names all the content types available to the Content Store, and for each content type it specifies:

- A set of **fields**. All the information in a content item is stored in fields. A text content item (generally referred to as a **story** in this manual) will usually have at least a **body** field for the main content and a **title** field (although different names may be used) plus a range of other fields that vary according to the content type.
- Optionally, a set of **relation types**. A relation type is simply a name used to classify an article's relations to other objects (other articles, images, multimedia objects, external links and so on). For further information about relation types, see section 3.8.

Since it defines the structure of all the articles in a CUE installation, the **`content-type`** resource is obviously of central importance. It determines:

- What is stored in the database
- What is displayed in the CUE Editor user interface
- What is made available to template developers in the CUE Front GraphQL API

For further information about this, see chapter 3.

Alongside the content-type resource files is another important resource file called the **`layout-group`** resource. This resource defines the internal structure of section pages. For further information about the **`layout-group`** resource, see chapter 5.

The **`content-type`** and **`layout-group`** resources are publication-specific: if your organization publishes multiple web sites, then each site will be defined as a separate publication, and each publication will have its own **`content-type`** and **`layout-group`** resources. All the publications can, however, also make use of a collection of shared resources called **storyline templates** and **story element types**.

Story element types define the block level elements of which stories are composed – elements such as headline, lead text, paragraph, image, pull quote and so on. Each element type is defined in a separate resource (**`story-element-headline`**, for example).

Storyline templates define different sets of story elements for use in different contexts (different publications, different types of story and so on). When a journalist creates a new story in the CUE editor, she must first select the storyline template she wants to use. This will determine which story elements are available to her while writing the story.

1.2 Workflows, States and Access Control

The progress of content through CUE is controlled by **workflows**. A workflow is:

- A sequence of **states** through which a content item passes from the point at which it is first created to the point at which it is finally published (or possibly deleted).
- The transitions allowed between those states

Each state has an associated set of permissions that determine the access users have to content in that state. Content in the draft state, for example, can usually only be transitioned to the published state by users with the editor role.

CUE has a default workflow consisting of the following states:

Draft
: The state of a newly-created content item. It remains in this state while it is being worked on by its author(s) and is only visible to editorial staff.

Submitted
: The authors are finished working on the content item and it is ready for review by others (an editor, for example). It is still only visible to editorial staff.

Approved
: The review process is completed and the content item is considered ready for publishing. It is still only visible to editorial staff.

Published
: The content item is now publicly accessible. If it is desked on an active section page then it is visible in the web publication, and even if it is not desked on an active section page, it can be found it by searching.

Draft (with published)
: The content item is published **and** modified. The original version is visible on the web site and a modified **draft** version is visible in CUE.

Submitted (with published)
The content item is published **and** modified. The original version is visible on the web site and a modified **submitted** version is visible in CUE.

Approved (with published)
The content item is published **and** modified. The original version is visible on the web site and a modified **approved** version is visible in CUE.

Deleted
The content item has been withdrawn. A deleted content item is only visible to editorial staff. A deleted content item can be "undeleted" by changing its state from deleted to one of the other states.

Optionally, the "with published" states can be omitted, to give a simpler workflow. In this simpler workflow, changes to published content are immediately published. For more about this, see Disabling Content Item Staging.

These two default workflows may be sufficient for many publications. It is, however, possible to define your own states and workflows that you can then use to control particular content types.

Workflows, like content types are defined in resource files that must be uploaded to the Content Store (see section 1.3). Once you have uploaded a set of workflows you can make use of them by associating them with content type definitions in your publication **`content-type`** resource. You do this by setting the **`content-type`** element's workflow attribute (see content-type).

1.3 The Publication Resources

The **publication resources** are files that define the underlying structure of a publication.

There are four publication-specific resources:

`content-type`
An XML file that defines the types of content items present in a publication.

`layout`
A legacy XML file that is required to be present but never needs to be edited.

`layout-group`
An XML file that defines the logical structure of the section pages in a publication.

`feature`
A plain text file containing property settings that govern the behavior of the CUE Content Store when handling a publication. This resource is optional: you only need to supply it if you want to change the default behavior of the Content Store in some way.

The publication-specific resources are stored in the following location in a publication WAR file or folder tree:

`META-INF/escenic/publication-resources/escenic`

For more detailed information about publication resources and how to define them, see section 2.2.

Publications also have access to a set of shared resources:

story element types
XML files defining the block level elements of which stories are composed – elements such as headline, lead text, paragraph, image, pull quote and so on. Each file contains the definition of one story element type.

storyline templates
XML files defining sets of story element types from which stories may be composed. A story is based on a storyline template, which determines what story element types it may contain. The storyline templates made available in a publication are determined by setting section parameters in the publication, as described in section 2.4.

custom workflow definitions
XML files defining custom workflows that may be associated with particular content types. A workflow defines:

- A set of states that a content item may be in
- A set of permissions associated with each state, determining what users may do with content items in that state
- The transitions allowed between the states in the workflow

For more detailed information about shared resources and how to define them, see section 2.1.

2 Defining a Publication

The process of defining a publication involves the following steps:

1. Define a set of shared publication resources and upload them to the Content Store.
2. Define the publication resources for this publication and upload them to the Content Store.
3. Create a publication based on the uploaded resources
4. Specify the storyline templates to be made available in the publication
5. Revise the resources if necessary

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

2.1 Defining Shared Resources

The shared publication resources are XML files of the following type:

Story element types
that define block level elements such as paragraphs, images, headlines and so on.

Storyline templates
that define particular types of story in terms of the story element types they can or must contain.

Custom workflow definitions
that may be associated with particular content types, defining how they flow through CUE.

Shared publication resources are not an absolute requirement for creating publications. You only need story elements and storyline templates if you intend to make use of storyline stories, the new story type introduced with Content Store. If you only use the rich text-based stories inherited from Escenic then you do not. For more information about the differences between storyline stories and rich text stories, see chapter 4.

The recommended choice for new publications is to use native CUE storyline stories, in which case you will need to define a set of shared resources and upload them. You might also want to create a publication that supports both kind of story, so that you can mix existing rich text stories with new storyline stories.

You only need to upload custom workflow definitions if the default workflows supplied with CUE do not meet your requirements.

The Content Store installation includes a starter pack with a set of ready-to-use shared resources. These may be a sufficient starting point for many uses, although if you use a language other than English in your CUE user interface, then you will at least want to translate the labels in the starter pack resource files.

You will find the starter pack in the *engine-installation*`/contrib/starter-pack` folder. If you want to use them without modification, simply upload them to the Content Store using the `escenic-admin` web application, as described in Manage Shared Resources.

If you need to translate the labels in the files first, open each file in a text editor and search for the following two strings: <**ui:label**> and <**ui:description**>. Translate the content of these elements and leave everything else unmodified.

If you want to make other changes to the shared resources, or create shared resources of your own, then you will need to read chapter 4 first.

2.2 Defining Publication Resources

Whichever kind of story content type you intend to use in your publication, you will need to upload at least three publication resource files to be able to create a publication:

- The **content-type** resource
- The **layout** resource
- The **layout-group** resource

If you have special requirements, you might also need to upload a feature resource file (see chapter 6).

You can download a default set of publication resources for the **CUE Front** demo publication **tomorrow-online** from **https://maven.escenic.com**. These publication resources contain a small number of simple content type definitions, including both a native CUE story content type (called story) which you can use together with the default shared publication resources and an Escenic legacy story content type.

As with the shared resources in the Content Store starter pack, you can use these publication resources as they are. You can upload the downloaded zip file to the Content Store using the **escenic-admin** web application, as described in Update Resources.

If you want to translate the labels in the **content-type** and **layout-group** resources before uploading them, the method is basically the same as for the shared resources:

1. Unzip the publication zip file.
2. Open the **content-type** and **layout-group** resources in a text editor.
3. Search for <**ui:label**> and <**ui:description**> elements and translate their contents.
4. Save the edited files and zip them up again (together with the **layout** resource, which does not need editing).
5. Upload the modified zip file to the Content Store using the **escenic-admin** web application, as described in Update Resources.

If you want to make other changes to the **content-type** and **layout-group** resources, or create your own from scratch, then you will need to read the relevant sections of this manual first (chapter 3 and chapter 5).

2.3 Creating a Publication

Once you have uploaded the required resources to the Content Store, you can use them to create a publication. (You can, of course, also use the same set of resources to create multiple publications. The publications will then have identical content types, relation types, layout groups and so on.)

The procedure for creating a publication is described in New publications. Once you have created a publication in this way, you can open it in CUE and start adding content.

2.4 Enabling Storyline Templates

If your publication is to include native CUE stories that depend on shared publication resources, then the first thing you must do after creating the publication is open it in the CUE editor and specify which storyline templates it is to make use of.

Unless you do this, the publication will not actually have access to any of the shared resources uploaded to the Content Store, and you will not be able to create any native CUE stories in CUE – the following error message will be displayed if you try to do so:

To enable storyline templates in a publication:

1. Open the publication in CUE.
2. Display the section tree panel.
3. Right click on the publication's root section to display its context menu.
4. Select **Edit Section**.
5. Select the **Section Parameters** tab.
6. Click the **Storyline Templates** field's **+** button and select a storyline template.
7. Repeat until you have enabled all the storyline templates you want to be available in this publication.
8. Click **Save** to save your changes.

If you have created several publications based on the same set of publication resources, you must do this in each of the publications.

3 The content-type Resource

The **content-type** resource determines (among other things):

- What kinds of content items a publication can contain
- What fields each kind of content item contains
- The type of each field
- Constraints on what you can enter in fields (maximum length, maximum and minimum values and so on)
- How the fields will be displayed in the CUE editor
- What relations each kind of article can have

The following sections provide an introduction to the **content-type** resource, and some of the things it is used for. For a full, formal description of the **content-type** resource format and all the things you can do with it, see here.

3.1 About Content Types

The primary function of the **content-type** XML file is to define the types of content item allowed in a particular publication. The content item types are defined in **content-type** elements, which can look something like this:

```
<content-type name="file">
  <ui:label>File</ui:label>
  <ui:title-field>title</ui:title-field>
  <panel name="main">
    <field mime-type="text/plain" type="basic" name="title">
      <ui:label>Title</ui:label>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
    <field mime-type="text/plain" type="basic" name="description">
      <ui:label>Description</ui:label>
    </field>
    <field name="binary" type="link">
      <ui:label>File</ui:label>
      <relation>com.escenic.edit-media</relation>
      <constraints>
        <mime-type>application/pdf</mime-type>
        <mime-type>text/*</mime-type>
      </constraints>
    </field>
  </panel>
  <summary>
    <field name="title" type="basic" mime-type="text/plain"/>
    <field name="description" type="basic" mime-type="text/plain"/>
  </summary>
</content-type>
```

This defines a simple content type for representing uploaded file attachments in CUE. It has three fields: a **`title`**, a **`description`** and a **`binary`** field used to hold an internal link to the uploaded file. The three field definitions don't belong directly to the **`content type`** element, but to a **`panel`** called **`main`**. Panels are used to organize content item fields into groups for display on separate tabs in CUE. A **`field`** element must have a name attribute and a **`type`** attribute that defines what kind of data can be store in the field. The **`mime-type`** field provides a more detailed type definition for **`basic`** fields.

The **`binary`** field's **`constraints`** element specifies what kinds of file the field supports: CUE will only allow files of the specified types to be uploaded.

The **`summary`** element defines a set of fields intended to be used when the content item appears as a relation in another content item or a teaser on a section page. It is usually a subset of the content item's main fields as in the example above.

The above content type is very simple. A content type definition can contain many more fields, spread across many panels, and the fields themselves can have complex internal structures that hold multiple values.

3.2 Defining Story Content Types

Text-heavy content types in CUE are generally referred to as **stories** or **articles**. A story typically consists of a long flow of formatted text stored in a single field, accompanied by other shorter text items such as a headline, a title, a lead text plus various items of metadata and a set of relations to other content items: images, videos, other stories and so on. CUE supports two different ways of modelling these structures:

- Escenic legacy stories: this is the way stories were defined for the Escenic Content Engine.
- Native CUE stories: this is a newer, more flexible method of defining stories, and is the recommended method.

You can freely choose which kind of story content type you want to use; both are fully supported. If you have existing publications containing Escenic legacy stories, then you can continue to use them as before. If you are starting from scratch then you are strongly recommended to use native CUE stories for the following reasons:

- Greater overall flexibility (due to extensibility)
- Better structural control (the storyline templates let you, for example, specify required elements)
- Better editing experience
- Planned functionality improvements

3.2.1 Legacy Stories

In Escenic legacy stories, the main text flow (usually referred to as the story's body) is stored in a rich text field. A rich text field is defined like this in the **`content-type`** resource:

```
<field type="basic" name="body" mime-type="application/xhtml+xml">
  ...
</field>
```

In other words, it is a **`basic`** field designed to store XHTML content. It is displayed in CUE as a rich text editor. The rich text editor provides a toolbar for formatting content, and also allows the user to insert simple inline relations to images and other stories. This rich text field, however, is only used to create the main text flow of the story. Other fields are used for storing structurally important elements such as the title/headline, lead text and byline, and most relations to other content (images, videos, audio files, other stories) are also stored outside the rich text field.

Here is a simple legacy story content type definition:

```
<content-type name="story">
  <ui:icon>news</ui:icon>
  <ui:label>Story</ui:label>
  <ui:title-field>slug</ui:title-field>
  <panel name="main">
    <ui:label>Main</ui:label>
    <field mime-type="text/plain" type="basic" name="slug">
      <ui:label>Slug</ui:label>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
    <field mime-type="text/plain" type="basic" name="leadtext">
      <ui:label>Lead text</ui:label>
    </field>
    <field mime-type="application/xhtml+xml" type="basic" name="body">
      <ui:label>Body</ui:label>
    </field>
    <field mime-type="text/plain" type="basic" name="dateline">
      <ui:label>Dateline</ui:label>
    </field>
  </panel>
  <summary>
    <ui:label>Content Summary</ui:label>
    <field name="slug" type="basic" mime-type="text/plain"/>
    <field name="leadtext" type="basic" mime-type="text/plain"/>
    <field name="dateline" type="basic" mime-type="text/plain"/>
  </summary>
</content-type>
```

For more information about how to define legacy rich text fields, see section 3.7.1.

3.2.2 Native CUE Stories

In native CUE stories, the rich text field is replaced by a **storyline editor**, which is defined like this in the **`content-type`** resource:

```
<field name="storyline" type="link">
  <relation>com.escenic.storyline</relation>
  <constraints>
    <mime-type>application/vnd.escenic.storyline+json</mime-type>
  </constraints>
</field>
```

The storyline field must be a **link** field (**`type=link`**) and must have:

- A child **`relation`** element with the content **`com.escenic.storyline`**

- A child **`constraints`** element that limits the content of the relation to the MIME type **`application/vnd.escenic.storyline+json`**

A storyline editor differs from the legacy rich text editor in the following ways:

- It does not conform to a generic predefined document format such as XHMTL. Instead, it offers a customizable set of styles defined in resource files called **story element types** and **storyline templates**.
- Because the document structure is extensible, it can include story elements for holding headlines, lead texts, bylines, images, videos, related stories. In other words, many things which have to be stored in separate fields in Escenic legacy stories can be included in the main flow of native CUE stories. It is still possible to make use of separate fields and relations where appropriate, but much more can now easily be included in the main flow of a story.

Here is a content type definition for a native CUE story:

```
<content-type name="story">
  <ui:icon>news</ui:icon>
  <ui:label>Story</ui:label>
  <ui:title-field>slug</ui:title-field>
  <panel name="main">
    <ui:label>Main</ui:label>
    <field name="storyline" type="link">
      <relation>com.escenic.storyline</relation>
      <ui:label>Storyline</ui:label>
      <constraints>
        <mime-type>application/vnd.escenic.storyline+json</mime-type>
      </constraints>
    </field>
  </panel>
  <panel name="metadata">
    <ui:label>Metadata</ui:label>
    <field mime-type="text/plain" type="basic" name="slug">
      <ui:label>Slug</ui:label>
      <ui:description>The working title of the story</ui:description>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
  </panel>
</content-type>
```

The storyline field appears in the content type's first panel, and it is the only field in the panel. This ensures that the storyline editor is displayed by default when a content item is opened, and that it has the maximum amount of screen space available, since the tab it is displayed on contains no other fields.

This content type has fewer field definitions than the legacy story definition shown in section 3.2.1. The **`leadtext`** and **`dateline`** fields are no longer required because they can be defined as story elements inside the storyline field. The **`slug`** field, however, is retained as a separate field because it is used by CUE as a "working title" for the story. As in any other content type, The **`slug`** field must be a plain text field, and must be referenced by a **`ui:title-field`** element. It must also be defined as a required field.

Even though the **`slug`** field is required, the CUE user will rarely need to edit it in a native CUE story: when the story is saved for the first time, the content of one of the elements in the **`storyline`** field is

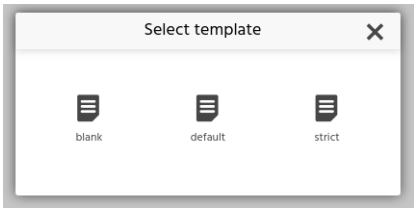
copied into the **slug** field, which then becomes the story's working title. The story element used to fill the **slug** field is selected as follows:

- If available, the first non-empty element in the storyline that has a **ui:title-field** element in its story element type (see section 4.1.1)
- Otherwise, the first non-empty element in the storyline.

The typical case is that there is a **headline** story element type that has a **ui:title-field**, and that this is a required element in the storyline, with the result that the **slug** field gets its content from the **headline**.

If the source story element is subsequently changed, the change is not copied over to the **slug** field, so the slug remains the same throughout the life of the story. A newly-created story cannot be saved until either the first element in the **storyline** field or the **slug** field itself has some content.

When a story content item is created in CUE, the user is immediately prompted to select a storyline template:



The content item is then created and the storyline field is populated with all the required and default element types specified in the selected template, ready for editing.

3.3 Defining Media Content Types

Content types are not only used to define stories. A publication will usually also include content type definitions for a variety of media types: images, videos, audio tracks, documents such as Word files and PDFs and so on. A media content type must have at least two fields:

- One link field (**type=link**)
- One slug field

The link field in this case is used to hold a link to the media file. The link field must have:

- A child **relation** element with the content **com.escenic.edit-media**
- A child **constraints** element that limits the content of the relation to the appropriate MIME types. So for an image content type, this would typically be:

```
<constraints>
  <mime-type>image/jpeg</mime-type>
  <mime-type>image/png</mime-type>
  <mime-type>image/gif</mime-type>
</constraints>
```

or for a video content type:

```
<constraints>
```

```
    <mime-type>video/*</mime-type>
    <mime-type>application/x-troff-msvideo</mime-type>
    <mime-type>application/mxf</mime-type>
  </constraints>
```

and so on.

As with story content types, media content types may also contain many other fields, grouped into different panels. An image content type, for example, usually contains a **`crop`** panel, with a field that can be used to set up alternative image versions called representations.

3.4 Defining Summaries

A **summary** is a subset of the fields or storyline elements in a content item. Summaries are used as teasers for content items desked on section pages or dragged in as relations to other content items. (Whether or not they are in fact used in this way depends, however, upon your presentation layer application.)

Summaries are defined by including a **`summary`** element in the content type definition.

The **`summary`** element must contain one or more **`field`** elements.

For legacy stories and media content types, these elements are usually references to fields that are already defined in the main part of the content type, for example:

```
<content-type name="file">
  <ui:label>File</ui:label>
  <ui:title-field>title</ui:title-field>
  <panel name="main">
    <field mime-type="text/plain" type="basic" name="title">
      <ui:label>Title</ui:label>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
    <field mime-type="text/plain" type="basic" name="description">
      <ui:label>Description</ui:label>
    </field>
    <field name="binary" type="link">
      <ui:label>File</ui:label>
      <relation>com.escenic.edit-media</relation>
      <constraints>
        <mime-type>application/pdf</mime-type>
        <mime-type>text/*</mime-type>
      </constraints>
    </field>
  </panel>
  <summary>
    <field name="title" type="basic" mime-type="text/plain"/>
    <field name="description" type="basic" mime-type="text/plain"/>
  </summary>
</content-type>
```

For native CUE stories, the summary's **`field`** elements usually refer to storyline elements instead of fields:

```
<content-type name="story">
  <ui:icon>news</ui:icon>
  <ui:label>Story</ui:label>
  <ui:title-field>slug</ui:title-field>
  <panel name="main">
    <ui:label>Main</ui:label>
    <field name="storyline" type="link">
      <relation>com.escenic.storyline</relation>
      <ui:label>Storyline</ui:label>
      <constraints>
        <mime-type>application/vnd.escenic.storyline+json</mime-type>
      </constraints>
    </field>
  </panel>
  <panel name="metadata">
    <ui:label>Metadata</ui:label>
    <field mime-type="text/plain" type="basic" name="slug">
      <ui:label>Slug</ui:label>
      <ui:description>The working title of the story</ui:description>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
  </panel>
    <summary>
      <field name="headline" type="basic" mime-type="text/plain"/>
      <field name="leadtext" type="basic" mime-type="text/plain"/>
    </summary>
</content-type>
```

When a content item is desked on a section page or added to another content item as a relation in CUE, the content of the referenced field or storyline element is copied into the summary field as default content. The CUE user can, however, edit this default content. This makes it possible for summary fields to have different content to their source fields/storyline elements, and also for content item that appear in multiple locations in a publication to have different summary fields in each location.

A summary field does not have to reference an existing field or storyline element. If it doesn't, then when the content item is added to a section page or related to another content item, the field is blank by default.

Note that:

- You cannot use **`ref-field-group`** inside **`summary`** elements: you must directly specify the fields to be included in the summary.
- You cannot use rich text fields (that is, **`basic`** fields with the MIME type **`application/xhtml+xml`**) in summaries.
- A storyline may contain more than one instance of the storyline element referenced by a summary field. In this case, the first instance is used.
- Any annotations in the content of a storyline element are removed when the content is copied to a summary field.

3.5 Controlling CUE Content Card Fields

CUE uses **content cards** to represent content items in the user interface. Search results, for example, are displayed as a list of content cards. A content card consists of three fields: a title, a summary and a binary. The fields are filled as follows:

Title
For legacy stories and media content types, the content card title is taken from the content type field referenced by the **`ui:title-field`** element. For native CUE stories it is taken from the first storyline element containing a **`ui:title-field`** element. If there is no such storyline element, then it is taken from the content type field referenced by the **`ui:title-field`** element.

Summary
For legacy stories and media content types, the content card summary is taken from the content type fields referenced by the **`com.escenic.index.summary.fields`** parameter. For native CUE stories it is taken from the first storyline element containing a **`ui:summary-field`** element. If there is no such storyline element, then it is taken from the content type fields referenced by the **`com.escenic.index.summary.fields`** parameter.

Binary
For legacy stories and media content types, the content card binary is taken from the first related binary content item. For native CUE stories it is taken from the first storyline element containing a link field that references a binary content item. If there is no such storyline element, then it is taken from the first related binary content item.

This means that for legacy stories and media content types, you can control the content of the content card title and summary by adding a **`ui:title-field`** element and a **`parameter`** element to the content type definition, for example:

```
<content-type name="story">
  ...
  <ui:title-field>title</ui:title-field>
  <parameter name="com.escenic.index.summary.fields" value="leadtext"/>
  ..
</content-type>
```

You can use the same method for native CUE stories, but you also have the option of marking specific storyline elements to be used as the title or the summary. For example:

```
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="headline">
  ...
  <ui:title-field />
  ...
</story-element-type>
```

or

```
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="leadtext">
  ...
  <ui:summary-field />
  ...
```

```
</story-element-type>
```

If you do this, then it will take precedence over anything specified in the content-type definition.

3.6 Controlling Content Item URLs

By default, the URL assigned to a published content item is generated from the content item's id, prefixed by the string **`article`** and followed by the suffix **`.ece`** - for example:

`article1234.ece`

You can, however, replace these with more informative (or "pretty") URLs by adding **`url`** elements to your publication's **`content-type`** elements. This element allows you to specify a template from which more meaningful or "pretty" URLs are generated. The template is composed from any text you want, plus a set of standard place-holders representing parts of a content item's publication date (**`{dd}`**, **`{MM}`**, **`{YY}`**), the content of selected content item fields (typically the title field) and so on.

The **`url`** element must be specified as the child of a **`content-type`** element, and determines the URLs assigned to content items of thahrt type. It affects:

- Newly-created content items belonging to its parent **`content-type`**.
- Any existing content items belonging to its parent **`content-type`** that are updated after the addition of the **`url`** element. The original URL of an updated content item is retained as an alternative URL, and attempts to access the original URL are redirected to the new URL (the Content Store returns an HTTP **`301 Moved Permanently`** response).

For detailed information and examples, see here.

3.7 More About Defining Fields

The **`content-type`** element you will use most frequently is almost certainly the **`field`** element. It is also one of the more complicated elements in the content-type resource, so it is probably worth looking at it more closely.

The **`field`** element has a **`type`** attribute that determines what type of data the field will accept. The main **`type`** values are:

`basic`
: Accepts **any** string data, including XHTML fragments. This is the default field type.

`number`
: Accepts only numbers.

`boolean`
: Accepts only Boolean (true/false) values.

`enumeration`
: Accepts only values from a predefined list.

`uri`
: Accepts only URIs.

`date`
Accepts only date/time values.

`schedule`
Accepts only schedule definitions.

`collection`
Accepts only values from an atom feed.

The type of a field determines not only what kind of data it is capable of accepting, but also:

- What kind of constraints can be placed on the input data
- What kind of child elements the **`field`** element may contain
- What kind of data the field returns when accessed from a template, and therefore how the field is best accessed.

These types are described in more detail in the following sections. Also discussed are **field arrays**, **complex fields** and **hidden fields**.

> Changing field type for any given field that is currently in use, is not supported. The behaviour if done is undefined.

For a complete list of all available field types, see here.

3.7.1 Basic Fields

The following **`field`** element defines a **`basic`** field with the name **`title`**:

```
<field mime-type="text/plain" type="basic" name="title">
  <constraints>
    <required>true</required>
  </constraints>
</field>
```

Note the following points:

- The **`name`** attribute may not contain spaces and must start with a letter (not a number).
- The **`mime-type`** attribute specifies more precisely the type of data allowed in the field. Currently, the following **`mime-type`** values are supported:

  `text/plain` **(default)**
  Any text. A simple text editing field is displayed in CUE.

  `application/xhmtl+xml`
  XHTML. A rich text editor with a formatting toolbar is displayed in CUE.
- The optional **`constraints`** child element specifies that a value is required.

3.7.2 Number Fields

The following **`field`** element defines a **`number`** field with the name **`age`**:

```
<field type="number" name="age">
  <constraints>
    <minimum>18</minimum>
    <maximum>99</maximum>
  </constraints>
```

```
    <format>##</format>
  </field>
```

Note the following points:

- The optional **constraints** child element specifies the allowed value range for the field.
- The optional **format** child element controls formatting in the input field in CUE. **format** syntax is based on the Java **DecimalFormat** class. See https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html for details.

3.7.3 Boolean Fields

The following **field** element defines a **boolean** field with the name **debug**:

```
<field type="boolean" name="debug"/>
```

Note the following points:

- A **boolean** field is displayed as a checkbox in CUE.
- It can contain only two values, **true** (checked) or **false** (not checked).

3.7.4 Enumeration Fields

The following **field** element defines an **enumeration** field with the name **review-type**:

```
<field type="enumeration" name="review-type">
  <enumeration value="film"/>
  <enumeration value="play"/>
  <enumeration value="book"/>
  <enumeration value="game"/>
</field>
```

Note the following points:

- in CUE an **enumeration** field is displayed either as a drop-down list from which the user can make a single selection or as multi-select list from which the user can make multiple selections.
- The example above displays a **drop-down** list. To display a multi-select list you must add a **multiple="true"** attribute to the field element.

3.7.5 URI Fields

The following **field** element defines a **uri** field with the name **homepage**:

```
<field type="uri" name="homepage"/>
```

A **uri** field will only accept a valid URI (Uniform Resource Identifier) as input in CUE. URI syntax is defined in the IETF's RFC 2396 (http://www.ietf.org/rfc/rfc2396.txt) and RFC 2732 (http://www.ietf.org/rfc/rfc2732.txt).

3.7.6 Link Fields

Link fields are used to contain references to:

- Binary objects such as images, video and audio files and so on.

- Storyline objects.

The following element defines a **link** field in an image content type:

```
<field name="binary" type="link">
  <relation>com.escenic.edit-media</relation>
  <constraints>
    <mime-type>image/jpeg</mime-type>
    <mime-type>image/png</mime-type>
  </constraints>
</field>
```

Note the following points:

- The child **relation** element defines the relationship between the field and the objects it used to reference. In this case, the value **com.escenic.edit-media** indicates that the field is to be used to reference binary media objects.
- The child **constraints** element lists the types of media objects the field is allowed to reference (in this case, JPEG and PNG image files)

This is a storyline **link** field definition:

```
<field name="storyline" type="link">
  <relation>com.escenic.storyline</relation>
  <constraints>
    <mime-type>application/vnd.escenic.storyline+json</mime-type>
  </constraints>
</field>
```

In this case the child **relation** element contains the value **com.escenic.storyline**, indicating that the field is to reference a storyline object, and the **constraints** element specifies a corresponding MIME type used to identify CUE storylines.

3.7.7 Date Fields

The following **field** element defines a **date** field with the name **startdate**:

```
<field type="date" name="startdate"/>
```

A **date** field is displayed in CUE as two specialized fields, one for the date and one for time of day. The content of a date field is stored as a UTC time in ISO-8601 format (that is, *YYYY-MM-DD*`'T'`*HH:mm:ss*`'Z'`) and is indexed as a date.

3.7.8 Schedule Fields

A schedule field is a specialized date/time field that contains a schedule start and end date, an event start and end time and an optional set of recurrence rules. Together, they define a sequence of date/time values. Schedule fields are typically used in articles describing events such as concerts, meetings etc.

A schedule field is defined as follows:

```
<field name="when" type="schedule">
  <ui:label>Schedule</ui:label>
</field>
```

A schedule is defined by:

- A schedule start date
- Either a schedule end date or a specified number of occurrences
- An event start and end time
- A recurrence specification (daily, weekly on Fridays, etc.)

When a content item containing a schedule field is stored, all of the event occurrences defined by the schedule are indexed, and can be searched by their start date. You can search for a list of content items contain scheduled events that start within a certain start date range.

You can limit the maximum number of occurrences users are allowed to specify by setting the **`maxOccurrences`** property in **`/com/escenic/schedule/OccurrenceHelper.properties`** file in one of you installation's **configuration layers**. If you do not specify this, then a default occurrence limit of **`100`** is used. For information about configuration layers, see Configuring The Content Engine.

3.7.9 Collection Fields

A collection field is special field that can contain value from an atom feed.

A collection field is defined as follows:

```
<field name="collection" type="collection" mime-type="text/plain" src="http://j.mp/
cwaXJM" select="title">
  <ui:label>Collection</ui:label>
</field>
```

3.7.10 Field Arrays

The following **`field`** element defines an array of **`basic`** fields with the name **`cities`**:

```
<field type="basic" name="cities">
  <array default="3" max="10"/>
</field>
```

Note the following points:

- You can make arrays of any field type.
- The **`array`** element's **`default`** attribute specifies the number of fields that are displayed by default, and **`max`** specifies the maximum number of fields that can be displayed.

3.7.11 Complex Fields

A **`complex field`** is an array of related **`field`**s that are displayed as a group in CUE. The component **`field`**s can be of any type except **`complex`**.

The following code defines a complex **`field`** with the name **`details`**:

```
<field type="complex" name="details">
  <complex>
    <field type="basic" name="availability"/>
    <field type="basic" name="colors"/>
    <field type="number" name="price"/>
  </complex>
```

```
</field>
```

You can create arrays of complex fields.

3.7.12 Hidden Fields

Any **`field`** can be hidden by adding a **`ui:hidden`** element to it.

The following **`field`** element defines a hidden field with the name **`result`**:

```
<field type="basic" name="result">
  <ui:hidden/>
</field>
```

Note the following points:

- A hidden field is not displayed in CUE.
- Hidden fields are intended to be filled by application code.

3.8 Relations

A content item can be **related to** a number of other content items. A story, for example, might have relations to:

- Images to be displayed with the article
- An image to be displayed with the article summary on section pages
- Other articles on the same subject, to be displayed as a list of links
- Related media objects, such as audio and video files
- Links to resources such as external web pages

Relations can be handled in two different ways in CUE:

- As structured relations defined in the content-type resource using the **`relation-type`** element.
- As inline relations in stories, where related content items are dropped directly in the text flow of a story. In-line relations are supported by both legacy stories based on rich text fields and native CUE stories based on story element types and storyline templates.

This section covers the use of structured relations based on use of the **`relation-type`** element. Structured relations may be used less in new publications that make use of native CUE stories, since you can use story element types to define richly structured relation story elements. It is nevertheless expected that the **`relation-type`** element will continue to be used for some kinds of relations.

3.8.1 Defining Relations

The Content Store's **relation** concept allows related items to be managed in an organized and standardized way.

Relations are added to content type definitions by first defining a **`relation-type-group`** (as a child of the root **`content-types`** element):

```
<relation-type-group name="attachments">
  <relation-type name="pictures">
    <ui:label>Pictures</ui:label>
  </relation-type>
  <relation-type name="stories">
    <ui:label>Stories</ui:label>
  </relation-type>
</relation-type-group>
```

The **relation-type-group** element contains a list of one or more **relation-type** elements. This group of relation types can then be included in content type definitions by adding **ref-relation-group** elements to **content-type** elements. For example:

```
<content-type name="legacy-story">
  ...
  <ref-relation-type-group name="attachments"/>
  ...
</content-type>
```

This would add the relation types **images** and **stories** to the content type **legacy-story**. You can define as many **relation-type-group**s as you wish in a **content-type** resource, and you can re-use the same **relation-type-group** in many content type definitions if you wish.

Relation types are displayed as drop zones in editor metadata panels in CUE. Dropping a content item in one of another content item's relation drop zones creates a relationship between them. You can restrict what types of content item may be dropped in a particular relation type by specifying allowed content types as follows:

```
<relation-type-group name="attachments">
  <relation-type name="pictures">
    <ui:label>Pictures</ui:label>
    <allow-content-types>
      <ref-content-type name="picture"/>
    </allow-content-types>
  </relation-type>
  <relation-type name="stories">
    <ui:label>Stories</ui:label>
    <allow-content-types>
      <ref-content-type name="story"/>
      <ref-content-type name="legacy-story"/>
    </allow-content-types>
  </relation-type>
</relation-type-group>
```

3.9 Changing Content Types That Are in Use

Most changes you make to content type definitions will be made during the publication design phase before it is in active use. Occasionally, however, you may need to update content types that are in active use in a live system. For publications that are actively updated 24/7 there may be little or no opportunity to stop all editing activity while the change is made.

The precise consequences of changing a content type that is in use depends on which of the following three classes the change belongs to:

- Compatible change

- Incompatible change
- Unsupported change

3.9.1 Compatible Changes

A **compatible change** to a content type causes no problems or side effects in CUE. The following changes are compatible changes:

- Changes to a field's **`ui:label`**.
- Changes to a field's **`ui:description`**.

Compatible changes do not necessarily become visible in CUE immediately after the changed **`content-type`** resource is uploaded to the Content Store: they become visible the next time CUE refreshes the affected content type for some reason or a content item of the affected type is opened for editing.

3.9.2 Incompatible Change

All the following types of change are classed as **incompatible changes:**

- Adding new optional field
- Making a mandatory field optional
- Adding a new option to an enumeration field
- Removing any constraint
- Adding a new field to a complex field

If a content item is open for editing in CUE when an incompatible change is made to its content type, then when the user saves his changes, a message is displayed stating that the content type has changed in an incompatible way. When the user clicks **OK**, the change is saved in the usual way but the editor tab is then immediately closed and reopened, incorporating the content type change. The user can then continue editing with the content type change in effect.

3.9.3 Unsupported Change

All other types of change to content types are **unsupported changes**. This means CUE cannot save a content item that is affected by such a change.

If you need to make an unsupported change to a content type, you should inform all CUE users so that they can avoid editing content items of the affected type while the updated **`content-type`** resource is uploaded.

4 Shared Resources

Shared publication resources serve two purposes:

- Defining the story element types and storyline templates used in CUE storylines
- Defining custom workflows

They are described in greater detail in the following sections.

Shared resources must be uploaded to the Content Store before they can be used in a publication. For information on how to upload shared resources, see Manage Shared Resources.

4.1 Storyline Resources

Content types defined using **`content-type`** definitions alone can be large and complex, but they are also quite rigidly defined. In order to be able to write the text content of a news story or article, a more flexible structure is required. In CUE, that flexibility can be provided in two different ways:

- By including a **rich text field** in the content type definition. This is the old way of creating a story content type. The rich text field is rendered as a rich text editor in CUE, and allows the user to write and format HTML content in a relatively free way. A rich text field is defined as a basic field with the MIME type **`application/xhtml+xml`**, for example:

  ```
  <field mime-type="application/xhtml+xml" type="basic" name="body">
    ...
  </field>
  ```

  This kind of content type is still supported, since there are many existing publications containing stories of this type. It is not the recommended method of creating story content types, however.
- By using **storylines** and **story elements**. This is the recommended method for new publications.

A story element is a block-level story component such as a paragraph, a heading or an image. Different types of story element are defined in XML resource files called **story element types**. A storyline is a particular type of story structure that makes use of a specified set of story element types in a specified way. Storylines are defined in XML resource files called **storyline templates**. When a user creates a new story in CUE, the first thing he must do is select which storyline template the story is to be based on.

Story element types and storyline templates are closely related to **`content-type`** resources, and use many of the same elements and namespaces. They differ, however, in the following ways:

- They are global resources that can be shared between many publications, and are not tied to a single publication.
- Although users may create their own story element types and storyline templates, and often will do so, it is not absolutely necessary: a starter pack of standard story element types and storyline templates is included in the Content Store distribution.

The following standard story element types are included in the Content Store distribution:

- Headline

- Lead text
- Paragraph
- Pull quote
- Image
- Gallery
- Video
- Relation
- Embed

together with the following standard storyline templates:

- Blank (a permissive template that imposes no predefined structure)
- Default (must start with a headline)
- Strict (must start with the sequence headline, lead text, image, paragraph)

These storyline resources can be found in in the Content Store distribution's **`contrib/starter-pack`** folder. If you want to use them in your publications, then all you need to do is upload them to the Content Store, as described in Manage Shared Resources.

If the standard storyline resources do not meet your requirements, then you can create your own and upload them in the same way.

4.1.1 Story Element Types

A story element is a block-level component of the text flow in a story. Examples of story elements include:

- Headings
- Ordinary paragraphs
- Specially-formated paragraphs such as fact boxes and pull quotes
- Images, videos and audio recordings
- Embeds
- Related stories

Story element types are defined in XML files, one type per file. Standard story element types are included in the Content Store distribution, but you can modify them and/or create additional ones of your own. Here is a story element type definition for a paragraph:

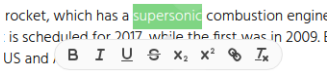
```
<?xml version="1.0" encoding="UTF-8"?>
<story-element-type xmlns="http://xmlns.escenic.com/2008/content-type"
                    xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
                    name="paragraph">
  <ui:label>Paragraph</ui:label>
  <ui:icon>paragraph</ui:icon>
    <field name="paragraph" type="basic" mime-type="text/plain">
      <annotation name="bold"/>
      <annotation name="italic"/>
      <annotation name="underline"/>
      <annotation name="strike"/>
      <annotation name="subscript"/>
```

```
      <annotation name="superscript"/>

      <annotation name="inline_link">
        <allow-story-element-type>
          <ref-story-element-type name="external_link"/>
        </allow-story-element-type>
        <ui:style>
          {
            color: blue;
            text-decoration: underline;
          }
        </ui:style>
      </annotation>

      <annotation name="internal_link">
        <allow-story-element-type>
          <ref-story-element-type name="internal_link"/>
        </allow-story-element-type>
        <ui:style>
          {
            color: green;
            text-decoration: underline;
          }
        </ui:style>
      </annotation>

      <annotation name="note">
        <allow-story-element-type>
          <ref-story-element-type name="note"/>
        </allow-story-element-type>
        <ui:style>{ background-color: yellow; }</ui:style>
      </annotation>

      <ui:summary-field/>
    </field>
</story-element-type>
```

The above definition says that a **paragraph** element:

- Will be called **Paragraph** in CUE (**<ui:label>Paragraph</ui:label>**)
- Will be represented by the **paragraph** icon in CUE
- Has one field (called **paragraph**) that may contain plain text (**type="basic" mime-type="text/plain"**)
- Can be formatted by applying various **annotations** (inline formatting instructions) to the contained text. The **ui:annotation** elements specify which formats are allowed in paragraphs.

In CUE the specified annotations appear as buttons on a formatting bar. This bar appears whenever the user selects some of the text in the paragraph:



> The **bold**, **italic**, **underline**, **strike**, **subscript** and **superscript** annotations are built in to CUE, with predefined rendering and button icons. You can, however, define your own additional annotations, like the **inline_link**, **internal_link** and **note** annotations in the above example. For more about this, see section 4.1.1.1.

Here is another story element type, this one for images:

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

    <field name="alignment" type="enumeration" >
      <ui:label>Alignment</ui:label>
      <ui:editor-style>buttons</ui:editor-style>
      <enumeration value="left">
        <ui:label>Left</ui:label>
      </enumeration>
      <enumeration value="center">
        <ui:label>Center</ui:label>
      </enumeration>
      <enumeration value="right">
        <ui:label>Right</ui:label>
      </enumeration>
    </field>

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

    <field name="relation" type="link">
        <relation>com.escenic.content-item</relation>
        <constraints>
            <allow-content-types>
                <ref-content-type name="picture"/>
            </allow-content-types>
          <required>true</required>
        </constraints>
    </field>
</story-element-type>
```

Elements of this type consist of three fields:

- The first field is labelled **Alignment** and is rendered as three buttons, one for each possible value.
- The second field is labelled **Caption** and may contain plain text (**`type="basic" mime-type="text/plain"`**), which in this case may not be formatted since no **`ui:annotation`** elements are specified.
- The third field has no label and is a rendered in CUE as a drop zone into which an image may be dragged.

`type="link"`, means that the third field can only contain a link to an object. **`<relation>com.escenic.content-item</relation>`** means that it can only contain a link to a CUE content item (not an image dragged from the desktop, for example), and the **`constraints`** element narrows things down even further, so that CUE users are only allowed to drop content items of the type **`picture`** in this field.

This example illustrates the use of the **ui:style** and **ui:title-field** elements in story element types:

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

If used, the **ui:style** element must appear as a child of the **story-element-type** element. It controls the appearance of the story element type in CUE. For more information on how to use the **ui:style** element, see section 7.3.

If used, the **ui:title-field** element must appear as a child of the **story-element-type** element. It indicates that this story element type can be used by CUE to automatically fill a story's slug field (the field pointed to by a story content type's **ui:title-field** element. For more information about this, see section 3.2.2.

Story element type definitions are defined using XML elements from the same namespace as the **content-type** resource. Whereas the root of a **content-type** resource is a **content-types** element, the root element of a story element type definition is a **story-element-type** element. So for a complete, formal description of the file format, start here.

4.1.1.1 Custom Annotations

You can define annotations of your own to supplement the built-in annotations provided with CUE.

4.1.1.1.1 Simple Annotations

The simplest kind of annotation definition consists of just a name and some CSS defining how the annotation is to be rendered by CUE:

```
<annotation name="shock">
  <ui:style>{ color: red; }</ui:style>
</annotation>
```

If you added the above annotation to your **paragraph** story element type, then when editing paragraph elements in CUE, you would see that the formatting bar now included a new style button. Since your annotation definition doesn't include an icon, the button would be formed from the first two characters of the annotation name: **sh** in this case. You can, however, include a CSS icon definition as follows:

```
<annotation name="shock">
  <ui:style>{ color: red; }</ui:style>
  <ui:icon>
    {
```

```
      -moz-osx-font-smoothing: grayscale;
      -webkit-font-smoothing: antialiased;
      font-family: "Storylines";
      font-size: 90%;
      content: "\E889";
    }
  </ui:icon>
</annotation>
```

CUE actually includes built-in icons for the commonly-used custom annotations supplied in the Content Store starter pack: **`inline_link`**, **`internal_link`** and **`note`**. So you don't need to define your own icons for these annotations (although you can override the default ones if you choose).

4.1.1.1.2 Complex Annotations

You can also define more complex annotations that contain an information structure as well as a name and a presentation. The Content Store starter pack includes three examples of this kind of annotation: **`inline_link`**, **`internal_link`** and **`note`**. The **`internal_link`** annotation, for example has the name **`internal_link`**, an associated rendering (blue text with an underline), but in addition it needs to hold the URL and link text plus a couple of other values. In order to achieve this, the annotation definition includes a reference to a special story element type used to define the annotation's data structure:

```
<annotation name="inline_link">
  <allow-story-element-type>
    <ref-story-element-type name="external_link"/>
  </allow-story-element-type>
  <ui:style>
    {
      color: blue;
      text-decoration: underline;
    }
  </ui:style>
</annotation>
```

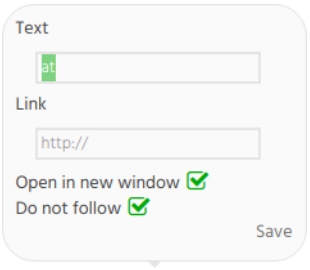
The **`allow-story-element-type`** element specifies that an annotation may include a reference to a single story element, and the **`ref-story-element-type`** element specifies that the story element must be of the type **`external_link`**.

Here is the definition of the **`external_link`** story element type (which is also supplied in the Content Store starter pack).

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

  <field name="uri" type="uri">
    <ct:constraints
      xmlns:ct="http://xmlns.escenic.com/2008/content-type">
      <ct:required>true</ct:required>
    </ct:constraints>
```

```
    <ui:label>Link</ui:label>
  </field>

  <field name="newWindow" type="boolean">
    <ui:label>Open in new window</ui:label>
    <ui:value-if-unset>true</ui:value-if-unset>
  </field>

  <field name="noFollow" type="boolean">
    <ui:label>Do not follow</ui:label>
    <ui:value-if-unset>true</ui:value-if-unset>
  </field>

</story-element-type>
```

This story element type defines the items of information that will be stored with an **`inline_link`** annotation, and the fields in the dialog that is displayed by CUE when the user inserts an inline link:



There are two things to bear in mind when defining a story element type for use with an annotation:

- There is no point including any **`ui:style`** or annotation elements – they will be ignored.
- The definition should always include a **`ui:hidden`** element. This ensures that the story element type will not appear anywhere in CUE other than as an annotation dialog.

4.1.2 Storyline Templates

A storyline template defines rules regarding which story element types may (or must) appear in a storyline. Standard storyline templates are included in the Content Store distribution, but you can modify them and/or create additional ones of your own. Here is an example storyline template:

```
<?xml version="1.0"?>
<template
  xmlns="http://xmlns.escenic.com/2017/storyline-templates"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="online">
  <base-story-element>
    <ref-story-element-type name="paragraph"/>
  </base-story-element>
  <required-story-elements>
    <ref-story-element-type name="headline"/>
    <ref-story-element-type name="image"/>
  </required-story-elements>
  <default-story-elements>
    <ref-story-element-type name="paragraph"/>
  </default-story-elements>
  <allowed-story-elements>
    <ref-story-element-type name="headline"/>
    <ref-story-element-type name="lead_text"/>
```

```
    <ref-story-element-type name="paragraph"/>
    <ref-story-element-type name="image"/>
    <ref-story-element-type name="video"/>
    <ref-story-element-type name="embed"/>
    <ref-story-element-type name="pull_quote"/>
    <ref-story-element-type name="gallery"/>
  </allowed-story-elements>
</template>
```

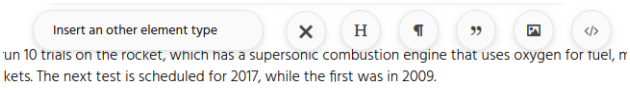
The root **`template`** element must contain one **`base-story-element`**, and may contain one each of **`required-story-elements`**, **`default-story-elements`** and **`allowed-story-elements`**. These groups of story element type references have the following meaning:

`base-story-element`
: Must contain one **`ref-story-element-type`** referencing the storyline's **base** story element type – the one that gets inserted if the user just presses Enter in the CUE storyline editor. It is typically set to **`paragraph`**.

`required-story-elements`
: If present, must contain one or more **`ref-story-element-type`**s referencing elements that **must** be present in the storyline. When a storyline is created in CUE, these elements are inserted by default. They are inserted at the top of the story in the order specified and cannot be deleted. Nor is it possible to insert other elements above or between them.

`default-story-elements`
: If present, must contain one or more **`ref-story-element-type`**s referencing default elements. When a storyline is created in CUE, these elements are inserted by default. They are inserted immediately after any required story elements, in the order specified. Unlike required story elements, default elements can be deleted. It is also possible to insert other elements above and between them.

`allowed-story-elements`
: If present, must contain one or more **`ref-story-element-type`**s referencing the elements that **may** be inserted using the CUE storyline editor's insert element menu:



This menu is displayed by clicking on one of the + buttons displayed after and between story elements (anywhere insertion of a new element is allowed). It allows you to select the element type to be inserted.

> The appearance of entries in the insert element menu depend upon how the element types are defined. Elements that have an icon (specified with **`ui:icon`** in the **`story-element-type`** file) appear as buttons on the right side of the menu bar, those that do not are displayed in the drop down list at the left hand end of the menu bar.

The **`name`** attribute of a **`ref-story-element-type`** element must exactly match the name of a **`ref-story-element-type`**.

For a complete description of the storyline template format, see storyline-templates.

4.2 Custom Workflow Definitions

Custom workflows are defined in XML files, one workflow per file. The workflows are defined as state charts, using the W3C standard State Chart XML format. These state chart definition files are conventionally given the extension **.scxml**.

The Content Store has two built-in workflows (**standard** and **standard-staging**). If these are insufficient for your requirements, some workflow state charts are supplied in the Content Store distribution's **contrib/starter-pack** folder. Here you will find state charts for the two built-in workflows, plus a few additional ones:

image.scxml
standard.scxml
standard-staging.scxml
storylines.scxml
wire.scxml

If you want to use one of the additional workflows in your publications, all you need to do is upload it to the Content Store, as described in Manage Shared Resources.

If none of the supplied workflows meet your requirements, then you can create your own and upload them in the same way.

The overall procedure for creating and using a new workflow is:

1. Create a state chart (**.scxml** file) defining the states and transitions that make up your new workflow. Read section 4.2.1 and section 4.2.2 to find out how to do this.
2. Upload the workflow definition as described in Manage Shared Resources.
3. Open your publication's **content-type** resource for editing.
4. Add the name of the new workflow to the required content type definitions in your **content-type** resource. The new workflow will only be used for the specified content types. The name is specified using the **content-type** element's **workflow** attribute (see content-type).
5. Upload the modified **content-type** resource as described in Update Resources.

4.2.1 Example Workflow

Here is an example workflow definition called **wire.scxml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
    xmlns:cs="http://xmlns.escenic.com/2013/content-state"
    xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
    xmlns:acl="http://xmlns.escenic.com/2018/acl"
    name="wire" initial="new" version="1.0" >

  <state id="new">
    <ui:label>New</ui:label>
    <ui:label lang="de">Neu</ui:label>
    <transition target="read" event="read"/>
    <acl:content>
      <acl:role name="reader">
        <acl:permission>read</acl:permission>
      </acl:role>
```

```
      <acl:role name="journalist">
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="editor">
        <acl:permission>read</acl:permission>
        <acl:permission>write</acl:permission>
      </acl:role>
    </acl:content>
  </state>

  <state id="read">
    <ui:label>Read</ui:label>
    <ui:label lang="de">Lesen</ui:label>
    <ui:icon>read</ui:icon>
    <transition target="new"  event="new"/>
    <transition target="used" event="used"/>
    <acl:content>
      <acl:role name="reader">
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="journalist">
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="editor">
        <acl:permission>read</acl:permission>
        <acl:permission>write</acl:permission>
      </acl:role>
    </acl:content>
  </state>

  <state id="used">
    <ui:label>Used</ui:label>
    <transition target="read" event="read">
      <ui:description>Are you sure you would like to mark the wire as read?</
ui:description>
      <cs:choice>
        <cs:name>read</cs:name>
        <ui:label>Mark as read</ui:label>
        <cs:action>read</cs:action>
      </cs:choice>
    </transition>
    <transition target="deleted" event="deleted"/>
    <acl:content>
      <acl:role name="reader">
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="journalist">
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="editor">
        <acl:permission>read</acl:permission>
        <acl:permission>write</acl:permission>
      </acl:role>
    </acl:content>
  </state>

  <state id="deleted">
    <transition target="new" event="new"/>
    <acl:content>
      <acl:role name="reader">
```

```
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="journalist">
        <acl:permission>read</acl:permission>
      </acl:role>
      <acl:role name="editor">
        <acl:permission>read</acl:permission>
        <acl:permission>write</acl:permission>
      </acl:role>
    </acl:content>
  </state>

</scxml>
```

This file defines a custom workflow for handling wire feeds. Wire feeds have some special workflow requirements because they are only intended to be used as source material for stories, they should not ever be published. So the purpose of this workflow is to allow wire feeds to be imported into CUE as a special content type that can be read and copied from, but not published.

In detail, this state chart specifies that:

- Wire feed content items must always be in one of the states **`new`**, **`read`**, **`used`** or **`deleted`**. A newly-created content items will always start out in the state **`new`**.
- All four states have the same permissions, which specify that the content can be accessed by any users, but only modified by editors. Since a change of state is a modification, this means that only editors can transition the content to a different state.
- New content can only be transitioned to the state **`read`** (which indicates that the content has been seen and read by a member of the editorial team).
- Read content can either be transitioned forward to the state **`used`** (which indicates that the content has been made use of) or back to the state **`new`**.
- Used content can either be transitioned forward to the state **`deleted`** or back to the state **`read`**.
- If an editor tries to transition content back to the state **`read`**, a confirmation dialog will be displayed in CUE, with the prompt "Are you sure you would like to mark the wire as read?".
- Deleted content can only be transitioned back to the state **`new`**.

4.2.2 The Workflow Definition Elements

The elements used in workflow definitions are described in the following sections.

4.2.2.1 scxml

The root element of a workflow definition must be an **`scxml`** element, and it must belong to the namespace **`http://www.w3.org/2005/07/scxml`**:

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
    xmlns:cs="http://xmlns.escenic.com/2013/content-state"
    xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
    xmlns:acl="http://xmlns.escenic.com/2018/acl"
    name="workflow-name" initial="state-id" version="1.0" >
  ...
</scxml>
```

The **`scxml`** element has the following attributes:

name
The name of the work flow.

new
The ID of the first state in the workflow. It must match the **id** attribute of a child **state** element. A new content item that uses this workflow will be created in this state.

version
Must be set to **"1.0"**.

This element should also contain namespace declarations for all the namespaces that will be used in the workflow definition. You will usually need to include these three declarations:

```
xmlns:cs="http://xmlns.escenic.com/2013/content-state"
xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
xmlns:acl="http://xmlns.escenic.com/2018/acl"
```

The **scxml** element must contain a series of **state** elements, one for each state in the workflow.

4.2.2.2 state

A **state** element in a workflow definition represents a content item state. It must belong to the namespace **http://www.w3.org/2005/07/scxml**. If you have declared this as the default namespace on the root element, then no further declaration or prefix is required:

```
<state id="state-id">
  ...
</state>
```

The **state** element has one attribute:

id
The state's ID. It must be unique within the workflow.

A **state** element should contain the following child elements:

- Zero or more **ui:label** elements defining the label(s) to be used to represent the state in the CUE editor. If more than one **ui:label** is specified, then each should have a different **lang** attribute. If no **ui:label** is specified then the state's **id** attribute is used as label. For further information, see ui:label.
- Zero or more **transition** elements defining the state transitions possible from this state.
- Zero or one **ui:icon** element, defining an icon to be used to represent the state in the CUE editor. For further information, see ui:icon.
- One **acl:content** element defining the permissions for content items in this state.

```
<ui:label>New</ui:label>
<ui:label lang="de">Neu</ui:label>
<ui:icon color="green">draft</ui:icon>
<transition target="state-id" event="event-id"/>
<acl:content>
  ...
</acl:content>
```

4.2.2.3 transition

A **transition** element in a workflow definition represents a possible transition from the state represented by its parent element to the state identified in the target attribute. It must belong to the namespace **http://www.w3.org/2005/07/scxml**.

```
<transition target="state-id" event="event-id"/>
```

The **transition** element has the following attributes:

target
The **id** of the transition's target **state** (which must be defined in the same workflow).

event
The **id** of the event that will trigger this transition. In a custom workflow this must always be identical to the target **id**.

The **transition** element may contain a child **ui:description** and **cs:choice** element defining a confirmation dialog to be displayed by CUE, should that be required. For an example, see the **used** state's **read** transition in section 4.2.1.

4.2.2.4 content

A container for a set of access control permissions that will be enforced for content items in this state. It must belong to the namespace **http://xmlns.escenic.com/2018/acl**. Usually this namespace is assigned the prefix **acl** in the root **scxml** element:

```
<acl:content>
  ...
</acl:content>
```

The **content** element has no attributes.

The **content** element must contain a series of **role** elements, specifying the permissions to be granted to various roles. If a role is not specified in this series of **role** elements, then users with that role are granted no access to content items in this state.

The content element may also contain one **private** element.

4.2.2.5 role

Defines the permissions to be granted to users with a specified role. This element must belong to the namespace **http://xmlns.escenic.com/2018/acl**. Usually this namespace is assigned the prefix **acl** in the root **scxml** element:

```
<acl:role name="role-name">
  ...
</acl:role>
```

The **role** element has one attribute:

name
The name of a CUE user role, one of the following:

- **reader**
- **journalist**

- **editor**

The **role** element must contain a series of 0 or more **permission** elements defining the access permissions to be granted to the role.

4.2.2.6 private

Specifies that the parent state to which this element belongs is a **private** state. This means that content items in this state are only visible to content items author(s). In practice, this element is mostly used for states representing newly created content. It enables a newly created content item to remain private until the author chooses to make it more widely available by transitioning it to a different state. This element must belong to the namespace **http://xmlns.escenic.com/2018/acl**. Usually this namespace is assigned the prefix **acl** in the root **scxml** element:

```
<acl:private/>
```

4.2.2.7 permisson

Represents a permission. This element must belong to the namespace **http://xmlns.escenic.com/2018/acl**. Usually this namespace is assigned the prefix **acl** in the root **scxml** element:

```
<acl:permission>
  permission-name
</acl:permission>
```

The **permission** element has no attributes. Its content must be one of the following permission names:

- **read**
- **write**

5 The layout-group Resource

The **layout-group** publication resource defines the logical structure of the layouts available for use on a publication's section pages.

The following sections provide an introduction to the **layout-group** resource, and some of the things it is used for. For a full, formal description of the **layout-group** resource format and all the things you can do with it, see here.

5.1 Defining Section Page Layouts

A **section page** in the CUE publication model is a "front page" for a publication section: it displays links and teasers to the most recent/interesting/relevant content in the section. Links and teasers can be placed on a section page in two different ways: they can be automatically selected by means of queries defined in the presentation layer, or they can be manually placed (or **desked**) on the section pages by editorial staff.

The section page layouts defined in the **layout-group** resource define the structure displayed in CUE to support manual placement of teasers on section pages. This means that if your publication front end is designed to manage all selection and placement of content automatically, then you don't need to worry about the **layout-group** resource at all, as it will not be used. Most organizations want to exercise some manual control over the placement of content on their publications' section pages, however, so you will most likely need to edit your l**ayout-group** resource.

5.1.1 Controlling Page Structure

Depending on how much control editorial staff are allowed over the details of section page layout, the structure defined in the **layout-group** resource may either be very close to the physical layout of section pages, with area/group names that reflect rows, columns of different widths, page positions and so on, or may not reflect the layout at all and have area/group names that just reflect logical categories ("Top stories", "Features" and so on).

Whichever "style" you use, there is no automatic, direct relationship between the structure you define in the **layout-group** resource and the pages displayed on the web site: what gets displayed on the site is determined by the presentation layer software and how it chooses to interpret the layout information provided to it. The **layout-group** structure provides a mechanism by which editorial staff can, to a greater or lesser degree, specify layout requirements. The presentation layer software is then responsible for interpreting those requirements and producing the final pages.

The root element of the **layout-group** resource must be a **groups** element, which may contain one or more child **group** elements. A **group** may contain one or more **area** elements. The groups and areas are displayed as nested rectangles on CUE section pages, the areas being drop zones for stories. Editorial staff desk stories on a section page in CUE by dropping them in the areas of their choice.

Areas can contain references to groups, allowing the CUE user to build complex nesting structures representing multi-column layouts if required.

Here is a very simple **layout-group** that defines just two areas, one called **Top story** and one called **Main stories**:

```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
  <group name="front" root="true">
    <area name="top">
      <ui:label>Top story</ui:label>
    </area>
    <area name="main">
      <ui:label>Main stories</ui:label>
    </area>
  </group>
</groups>
```

The **group** element's **root="true"** attribute indicates that this is a **root layout group** - that is, a group that defines an entire section page layout. **group** elements without this attribute are only used if they are referenced in an **area** element.

If the **groups** element contains more than one **group** element with **root="true"** set, then these groups represent **alternative** page layouts, and the CUE user can select which layout is to be used on a particular section page. This choice is made using the **Current layout** option displayed in the **General info** section of the CUE metadata panel.

area elements can be given labels using the **ui:label** element, in the same way as panels and fields can be labelled in the **content-type** resource. If specified, these labels are used in CUE. If no label is specified for an area, then CUE uses its **name** attribute instead. Since the **label** element belongs to the **interface-hints** namespace rather than the **layout-group** namespace, the name must include a prefix declared on the **groups** root element (**xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"** in the example shown above).

CUE users can desk whatever content they like in the **Top story** and **Main stories** areas. You can, however, limit the allowed content types for an area by adding an **allow-content-types** element:

```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
  <group name="front" root="true">
    <area name="top">
      <ui:label>Top story</ui:label>
      <allow-content-types>
        <ref-content-type-group name="content"/>
      </allow-content-types>
    </area>
    <area name="main">
      <ui:label>Main stories</ui:label>
    </area>
  </group>
  <content-type-group name="content">
    <ref-content-type name="story"/>
    <ref-content-type name="video"/>
  </content-type-group>
</groups>
```

Now CUE will only allow content types specified in the **content-type-group** called **content** to be dropped in the **Top story** area.

The following example shows a more complex section page design in which the area names indicate something about their location on the page:

```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
  <group name="main" root="true">
    <ui:label>Main stories</ui:label>
    <area name="top">
      <ui:label>Top</ui:label>
    </area>
    <area name="rightcolumn">
      <ui:label>Right Column</ui:label>
    </area>
    <area name="center">
      <ui:label>Center Column</ui:label>
      <ref-group name="two-col"/>
      <ref-group name="three-col"/>
    </area>
  </group>
  <group name="two-col">
    <area name="left"/>
    <area name="right"/>
  </group>
  <group name="three-col">
    <area name="left"/>
    <area name="center"/>
    <area name="right"/>
  </group>
</groups>
```

If an **area** contains **ref-group**, elements, this indicates that it **may** contain subdivisions (the children of the referenced groups). The CUE user can insert the referenced groups into the area by right-clicking on the area and selecting the required group from the displayed menu. In the case of the **Center Column** area in the above example, the user can insert two-col and three-col groups as well as (or instead of) desking content items in the area. The user can insert any number of such groups into the area, in any combination.

5.1.2 Group, Area and Teaser Options

Both **group** elements and **area** elements can have options associated with them, defined in a child **options** element:

```
    <area name="main">
      <ui:label>Main stories</ui:label>
      <ct:options scope="current">
        <ct:field name="teaserStyle" type="enumeration">
          <ui:label>Teaser Style</ui:label>
         <ct:enumeration value="default">
            <ui:label>Default</ui:label>
          </ct:enumeration>
          <ct:enumeration value="breakingNews">
            <ui:label>Breaking News</ui:label>
          </ct:enumeration>
          <ui:value-if-unset>default</ui:value-if-unset>
        </ct:field>
      </ct:options>
    </area>
```

An **options** element that is a child of an **area** element can have a **scope** attribute, the purpose of which is to specify whether the options belong to the **area** itself (**scope="current"** as in the example above) or to the individual content items desked in the area (**scope="items"**), in which case they are referred to as **teaser options** rather than area options.

Options are displayed in the right hand panel when a group, area or desked content item is selected in CUE, allowing the CUE user to make choices about how the selected component is to be presented. Exactly how such choices are interpreted and the effect they have on layout is determined in the presentation layer.

The **options** element is not actually a member of the **layout-group** namespace, it is "borrowed" from the **content-type** namespace. If you use this element, therefore, its name must include a prefix declared on the **groups** root element, as follows:

```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
...
</groups>
```

6 The feature Resource

Unlike the other publication resources, the **`feature`** resource is not an XML file. It is a plain text file containing a series of simple property settings like this:

```
allowFrontPageAsHomeSection=true
```

All the property settings consist of a single *keyword=value* pair like the one above, and modify the behavior of the publication in some way. For a full, formal description of the **`feature`** resource format and all the things you can do with it, see feature.

7 The interface-hints Namespace

You have probably noticed that the resource file examples in chapter 3 and chapter 5 contain some elements with the prefix **ui:**. These elements are user interface hints, and the **ui:** prefix identifies them as belonging to the **http://xmlns.escenic.com/2008/interface-hints** namespace.

If you look at the syntax diagrams for the content-type and content-type resources you will see that many of them include the placeholder *ANY-FOREIGN-ELEMENT*. This placeholder is used to indicate that an element can contain elements from any foreign namespace, but is primarily intended to indicate that you can insert elements from the **interface-hints** namespace.

Use of the **interface-hints** elements is entirely optional - you can create a working **content-type** or **layout-group** resource without using them. By using them, however, you can create a more user-friendly interface for your publication in CUE.

The following sections discuss the use of some of the most frequently used **interface-hints** elements. For full details about all elements in this namespace, see interface-hints.

7.1 label

The **interface-hints** element you will probably make most use of is **label**. By default, CUE generates labels for user interface components from the **name** attribute of the resource file elements they are based on, by simply capitalizing the first letter of the name. A **field** element called **title** in the **content-type** resource, for example, will result in the field label **Title** in CUE. If, however, you want the field to be called **Headline** in CUE, then you can achieve this by adding a **ui:label** element as follows:

```
<field type="basic" name="title">
  <ui:label>Headline</ui:label>
</field>
```

The other important function of the label element is to enable multilingual user interfaces. An element can have several child **label** elements, each with a different **xml:lang** attribute identifying its language. For example:

```
<field type="basic" name="title">
  <ui:label xml:lang="fr">Titre</ui:label>
  <ui:label xml:lang="de">Titel</ui:label>
</field>
```

7.2 value-if-unset

This is a very useful element that you can use to specify default values for fields:

```
<field type="uri" name="homepage">
  <ui:value-if-unset>http://www.escenic.com/</ui:value-if-unset>
</field>
```

7.3 style

You can use the **style** element to control the appearance of content in CUE, both in the rich text fields of Escenic legacy stories and in the story element types used in native CUE stories.

7.3.1 Styling Rich Text Fields

The **style** element lets you style the rich text fields in Escenic legacy stories using CSS. You can put any standard CSS in the body of the element, giving you detailed control over the appearance and layout of rich text field content in CUE. To set the color of **h1** and **h2** headings in a field, for example, you could specify:

```
<field mime-type="application/xhtml+xml" type="basic" name="body">
  <ui:style>
    h1 {color:red}
    h2 {color:green}
  </ui:style>
</field>
```

You can also use this element to style in-line relations so that CUE users can easily distinguish between relations to different content types. To do this, you must create CSS classes with names of the form:

escenic-*content-type-name*

To make in-line links to **news** content items green and in-line links to **blog** content items red, for example, you could specify:

```
<field mime-type="application/xhtml+xml" type="basic" name="body">
  <ui:style>
    .escenic-news { color: green;}
    .escenic-blog {color: red;}
  </ui:style>
</field>
```

> The **style** element can **only** be used with **basic** fields where **mime-type** is set to **application/xhtml+xml**. It has no effect if used with any other elements.

For hints and examples about more advanced uses of the **style** element, see style.

7.3.2 Styling Story Element Types

The **style** element lets you style the story element types of native CUE stories using CSS. You can put any standard CSS in the body of the element, giving you detailed control over the appearance and layout of story elements in CUE. To set the font size of a heading story element type, for example, you could enter:

```
<story-element-type
  name="headline"
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
  ...
  <ui:style>
    .story-element-headline [contenteditable='true'] {
      font-size: 2.3em;
    }
  </ui:style>
```

```
</story-element-type>
```

You can use the following CSS class names to select different parts of the HTML generated for a story element:

`.story-element-`*name*
where *name* is the name of the story element type. This selects the whole story element.

`.fields`
This selects all story element fields, so

```
.story-element-pull_quote .fields
```

will select all the fields in a pull_quote story element.

.field-name
where *field-name* is the name of a field in a story element. This selects the specific field, so:

```
.story-element-pull_quote .fields .quote
```

or

```
.story-element-pull_quote .quote
```

will select the **`quote`** field in a **`pull_quote`** story element.

You can also use the **`[contenteditable='true']`** predicate to select only the editable content in a field.

For a simple story element type, the hints given above may be enough to enable you to style elements as required, but for more complicated cases, the best method is probably to wait with styling until you have a working publication, and then use the following method:

1. Open the publication in CUE, using the Chrome browser.
2. Create a story content item, selecting a storyline that includes this story element type.
3. Insert a story element of this type.
4. Start the Chrome developer tools and display the Elements pane.
5. Locate the HTML code representing the story element.
6. Use the Chrome style editing functionality to get the appearance you want.
7. Copy the CSS you created back to a **`ui:style`** element in the story element type resource.
8. Upload the modified story element type resource to the server in the usual way (see Manage Shared Resources)
9. Restart CUE and check the results.

> You should thoroughly test your styling changes before committing them. In the worst case, inappropriate styling can render fields unusable in CUE.

7.4 icon

This element lets you set the icons used in CUE to represent content items (for example, in search result lists, inboxes and so on) and story element types. You can only specify this element as the child

of a **content-type** or **story-element-type** element: it is ignored in all other contexts. You must either specify the name of a built-in icon provided by CUE:

```
<ui:icon>news</ui:icon>
```

or the absolute URI of an image that you want to use as an icon:

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

The icon images must have the following characteristics:

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

7.5 macro

The **macro** element can be used to provide a means of inserting custom HTML code into the content of a rich text field. If a rich text field definition in the **content-type** resource contains one or more **ui:macro** elements, then a macro button is added to the rich text toolbar in CUE. Clicking this macro button displays a drop-down menu containing one menu item for each **ui:macro** element. The following macro definition, for example, inserts a placeholder for a fact box (to be replaced by the presentation layer during page rendering):

```
<ui:macro name="insert-factbox">
  <ui:step action="insert" text="Fact box" wrap-element="eceplaceholder"
 class="factbox"/>
  <ui:keystroke>alt F</ui:keystroke>
  <ui:description>Insert fact box</ui:description>
</ui:macro>
```

7.6 tag-scheme

The **tag-scheme** element can be used to enable tagging for a particular content type. By default, content types do not support tagging, and you enable it by adding **tag-scheme** elements to the required content type definitions. Adding a **ui:tag-scheme** element to a **content-type** element enables access to one tag structure. You can enable access to several tag structures by adding multiple **ui:tag-scheme** elements.

For example:

```
<content-type name="story">
    <ui:tag-scheme>tag:concept@escenic.com,2017</ui:tag-scheme>
    <ui:tag-scheme>tag:location@escenic.com,2017</ui:tag-scheme>
    <ui:tag-scheme>tag:organization@escenic.com,2017</ui:tag-scheme>
    <ui:tag-scheme>tag:entity@escenic.com,2017</ui:tag-scheme>
    <ui:tag-scheme>tag:person@escenic.com,2017</ui:tag-scheme>
...
</content-type>
```

The content of a **ui:tag-scheme** element must be the **scheme** of one of the site's tag structures. A tag structure scheme is a URI that uniquely identifies the tag structure. The schemes of all tag structures defined on a Content Store site are listed on the **escenic-admin** application's tag management page (see Manage Tag Structures).

7.7 title-field

The **title-field** element is used for two purposes:

- To identify which of a content type's fields CUE is to be used as the internal title or **slug**. It is this field that will be used as the "name" of content items in CUE:

```
<content-type name="story">
  <ui:title-field>slug</ui:title-field>
  ...
  <panel name="metadata">
    <field mime-type="text/plain" type="basic" name="slug">
      <constraints>
        <required>true</required>
      </constraints>
    </field>
  </panel>
</content-type>
```

- To indicate that the content of a story element type may be used as a story title or **slug**. It is typically used in the definition of a headline story element type, so that the headline entered when a content item is first created becomes the slug used to identify it in CUE:

```
<story-element-type
  xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="headline">
  <field name="headline" type="basic" mime-type="text/plain">
  </field>
  <ui:title-field/>
  ...
</story-element-type>
```

You can only specify this element as the child of a **content-type** or **story-element-type** element: it is ignored in all other contexts.