CUE Content Store

Publication Design Guide

7.10.15-1

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 more **publications**. A publication, also sometimes known as a **channel** is a publishing endpoint or target for content. It may be a web site, a print publication, or a social media account (Facebook, Twitter etc.).

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. All the content to be published in a particular channel appears as content items in that channel's publication.

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.

Organizational Units

Publications can be organized into groups called **organizational units** or **OU**s for short. OUs are useful for large publishing organizations with many different publications and editorial groups. In a large news organization, an OU will typically represent a newsroom. So in this case OUs allow you to divide the organization's publications into groups and assign them to the responsible newsrooms. The main advantage of doing this is that it allows CUE users to specify which OU(s) they belong to, and thereby limit which publications they see when working in CUE.

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).

1.2 Containers and Content Inheritance

Different channels in general require very different types of content. Links, video and audio content can be an important part of online content, but are largely irrelevant for a print publication. A social media post is likely to be much shorter than either a web site or print story. Nevertheless, some content can be shared between different channels, and for most publishers it makes sense to manage a story that is to be published to many channels as a single task. CUE Content Store therefore supports single-sourcing of content for publishing to multiple channels.

Single-sourcing is provided by means of **content inheritance**: a content item in one channel is allowed to inherit its content from another content item in a different channel. In a typical **online-first** setup, content is initially added to a **base** content item in the online publication, and then inherited by **variant** content items in print and social media publications (and possibly in other online publications). In a **print-first** setup, content would first be added to a print publication and then inherited by variant content items in other publications.

Content inheritance depends on the use of storyline templates and story elements, and content is inherited at the story element level. Not all of a base content item's story elements are necessarily inherited by the variants – a Twitter content item is usually very short and may only inherit one or two story elements.

Within any one organizational unit (OU), only one level of inheritance is possible: a print variant and a facebook variant can both inherit **directly** from an online base storyline, but it is not possible for the print variant to inherit from the online base storyline, and the facebook variant to inherit from the print variant.

Multilevel inheritance **is** nevertheless possible, but only across OU boundaries. If a storyline inherits from a base storyline in another OU, then it can be used as a **local** base storyline within its own OU. So if a print storyline inherits from an online storyline in another OU, it is possible to then add a facebook storyline that inherits from the print storyline. The following diagram shows a container that spans three OUs:

The variants in each OU all inherit directly from a local base storyline in their own OU, but ultimately all of the storylines inherit from the global base storyline in OU 1.

The terms **upstream** and **downstream** are sometimes used to describe inheritance relationships between storylines. An upstream storyline is one of the storylines from which the current storyline inherits content. A downstream storyline is one of the storylines that inherit from the current storyline.

CUE supports two different styles of inheritance, called **inheritance modes**:

Inherit mode
: Inherit mode inheritance is the simplest type of inheritance. In this mode, inherited content is not editable: only the base storyline can be edited, and any changes made to the storyline are immediately reflected in all its variants. A user can, however, edit a variant by first breaking its inheritance link to the base. Its content will then no longer reflect any changes made to the base storyline.

Reconcile mode
: In reconcile mode, inheritance is controlled at the story element level rather than at the storyline level. Adding a new story element to a variant storyline, for example, will not affect the other story elements inherited from the base; they will continue to reflect changes made in the base storyline. Similarly, editing or deleting an inherited story element will only break inheritance for that story element: all the other inherited story elements will continue to reflect changes made in the base storyline.

Inherit mode was impemented before reconcile mode, and in order to ensure backwards compatibility inherit mode is the Content Store's default mode of inheritance. Reconcile mode, however, offers greater flexibility and is the recommended choice for new publications.

The following concepts are required to provide support for multi-channel publishing:

Publication type
: All publications are classified as belonging to a **publication type**. Like content types, publication types are user-definable. Publication types are much simpler than content types, but they are also defined in XML resources: for details, see section 4.1. Aside from defining the publication type's name and icon, a publication type resource simply lets you enable/disable a few publication features such as support for lists, inboxes and section 1.2.1. Default publication type resources for online, print, Facebook and Twitter channels are included in the Content Store starter pack.

Container
: A container holds all the content items comprising a particular story: the base content item to which the original content was added, and the variant content items that inherit the content. A container is identified by its slug: a working title assigned to it when it is first created, usually copied from the title or headline of the container's base content item. The slug follows all variants of the story.

Container type

Container types determine what kinds of container it is possible to create. Like content types and publication types, container types are defined in XML resources: for details, see section 4.3. Like a publication type, a container type defines a name and icon, but it also contains:

- An **`inheritance-mode`** attribute that determines the type of inheritance to be used in the container: **inherit mode** or **reconcile mode** inheritance. For example:

  ```
  inheritance-mode="reconcile"
  ```

  If **`inheritance-mode`** is not set, then the container uses **`inherit`** mode by default.

  A set of zero or more field definitions. Any fields that are defined in a container type will appear on a special tab in all content items belonging to containers of that type. Most containers have only one such field definition - the container slug.

- A set of one or more usage rules for containers of this type. Each rule consists of a publication name and a content type name. For example:

  ```
  <first-destination publication="tomorrow-online" content-type="story"/>
  ```

  The usage rules determine:

  In what contexts containers of a given type can be created:
  A container type will only appear in the CUE **Create New** dialog if at least one of its **`first-destination`** elements specifies a publication in which the user has content creation rights and which belongs to an OU that is visible to the user.

  The content type of a new container's base content item:
  When a new container is created, its base content type is taken from the first **`first-destination`** element that specifies a publication in which the user has content creation rights and which belongs to an OU that is visible to the user.

Once a base content item has been created, the CUE user can spawn variants from it by clicking on the "Add channel" (+) button in the playout bar above the content editor. A variant created in this way must:

- Be a storyline-based content type.
- Only have one allowed storyline-template.
- Not be used as a base content item itself.
- Belong to the same OU as the base content item.

Only options that satisfy the requirements above are listed by the "Add channel" function.

The Content Store links variant story elements to base story elements in accordance with built-in rules. Instances of any story element types in the variant's storyline template with names that match story elements in the base storyline are created and linked to the matching base story elements. If the variant's storyline template contains required story element types that have no matches in the base storyline, then empty instances are created in the variant.

1.2.1 Auto-publishing

Auto-publishing is a publication feature that automatically republishes variant content items whenever an upstream change is published. With auto-publishing enabled, updating and publishing a base content item can trigger a cascade of publish events in all its inheriting variants.

The auto-publishing feature only works in very tightly defined circumstances, in order to ensure that it cannot result in the unintentional publishing of content. A content item can only be automatically published if the following conditions are satisfied:

- Auto-publishing is enabled in the publication.
- The content item is a storyline content item.
- The content item is defined as a destination in a container and has **`inheritance-mode`** set to **`reconcile`**.
- The content item has already been published (auto-publishing is in fact auto-**republishing**).
- The content item is either in the state **`published`** or a staged state such as **`draft-published`**.
- If the content item is in a staged state such as **`draft-published`**, then all the changes made to the content item since the last time it was published must be inherited changes. If it contains any changes that were made directly in the content item itself, then it will not be auto-published. (This rule can alternatively be made even more restrictive, see below).

Auto-publishing can be enabled either for all publications of a particular type, by setting the **`auto-publish-storylines`** feature in a publication-type resource (see auto-publish-storylines). It can also be set for an individual publication by setting the **`autoPublishStorylines`** property in a publication's **`feature`** resource (see auto-publish-storylines).

The auto-publishing feature can either be **`disabled`** or set to one of two levels:

`cautious` **(default)**
: If a destination content item has ever been manually modified and republished since it was first published, then it will never be auto-published.

`full`
: A destination content item will be auto-published unless it is in a staged state such as **`draft-published`** and the changes it contains include any changes that were made **directly in the content item itself**. In other words it will be auto-published as long as it only contains inherited changes.

> When determining whether or not to auto-publish a content item, only changes to storyline content are considered: changes to relations or metadata such as tags will never prevent auto-publishing.

1.2.2 Cross-OU Inheritance

CUE supports cross-OU inheritance, but only a base content type can inherit from a base content item in a different OU. This creates a local base content item, from which variants can be spawned in the usual way.

1.3 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 1.4). 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).

1.4 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:

publication types
XML files defining basic information about different types of publications: name, icon and its support for features such as lists, inboxes and section 1.2.1.

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.

container types
XML files defining different types of **containers**. A container holds a **base** content item plus its **variants**: content items in different types of publications that inherit the base content item's content.

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

custom role definitions
XML files defining custom roles. A role defines a set of access rights or permissions that can be assigned to a user.

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:

Publication types
that define different types of publication.

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

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

Container types
that define different types of **containers**, used for managing content inheritance.

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

Custom search filter definitions
that either modify CUE's default search filter panel or define additional search filters for use in dashboards.

Dashboard definitions
that define CUE **dashboards**. A dashboard is a panel in CUE that displays a set of one or more custom searches.

Custom capability definitions
that can be used to control user access to CUE functionality.

Custom role definitions
that can be used to control user access to content.

Custom content card definitions
that can be used to define what appears on content cards in 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 the 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 publication types if you actually publish to different channels (online and print, for example, or online and social media).

You only need container types if you use storyline stories, publish to multiple channels **and** wish to make use of content inheritance between those channels.

You only need to upload custom workflow definitions if the default workflows supplied with the Content Store do not meet your requirements. Similarly, you only need to upload custom search filter definitions if CUE's default search functionality is insufficient or if you want to add dashboards to CUE (in which case you need to upload dashboard definitions as well).

You only need to upload custom capability definitions if the default capability definitions supplied with the Content Store do not meet your requirements. This may be the case if you have added web components to CUE and wish to restrict user access to them.

You only need to upload custom role definitions if the default roles supplied with the Content Store do not meet your requirements.

You only need to upload custom content card definitions if CUE's default content card layout does 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. If you are intending to make use of organizational units to group your publications into organizational units, then you might need to make an organizational unit first, as described here. Once you have created a publication, 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

In the past, the **content-type** resource was always a single file, but this is no longer necessarily the case. As well as a content-type file, you can also upload a **content-type** folder containing one or more **modular** content type files. Each modular content type file:

- May be called anything you like.
- May contain one or more **complete** content type definitions.
- Must be self-contained: that is, a content type in one file cannot depend on field definitions in another file.

When a publication is created, all uploaded content type definitions are included in the resulting publication, whether they are defined in an old-style **content-type** resource or in modular content type resource files. You must ensure that all the content type definitions uploaded to a publication have unique names.

For instructions on how to upload content type definitions in general, see Upload Resources. For specific instructions on how to upload modular content type definitions, see Uploading Modular Content Type Resources.

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.

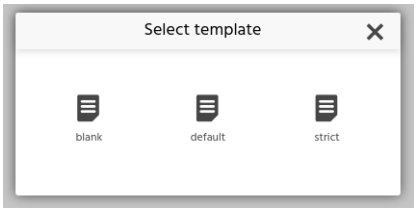
Even though the **`slug`** field is required, the CUE user will rarely need to edit it in a native CUE story: when the story is saved for the first time, the content of one of the elements in the **`storyline`** field is copied into the **`slug`** field, which then becomes the story's working title. 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 4.2.1)
- 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.

3.2.2.1 Container and Inheritance Support

The Content Store supports single-source publishing to multiple channels by means of **containers**: objects that hold links to multiple CUE-native stories. For more about this, see section 1.2.

CUE-native content types intended for use in containers must be configured slightly differently from CUE-native content types intended for standalone use. The differences are:

The slug belongs to the container
This means the content type does not need a slug field or a **`ui:title-field`** element. These are configured in the container type instead (see section 4.3). A value for the slug field is selected from a story element in the container's base content item, in the same way as for a standalone content item.

The content type must allow only one storyline template
The constraints on the content type's storyline field must include an **`allow-storyline-templates`** element that contains one and one only **`ref-storyline-template`**, as shown in bold below:

```
<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>
      <allow-storyline-templates>
        <ref-storyline-template name="strict"/>
      </allow-storyline-templates>
    </constraints>
</field>
```

A content type that does not include this constraint on its storyline field will never be used by CUE when creating a container. This is intended to improve the usability of containers in CUE: once the user has selected the base content type for the container he is creating, the container and content item are created immediately – the user is not required to select a storyline template as well.

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.

3.3.1 Supported Graphics File Types

CUE can display the following graphic file types:

image/jpeg
image/png
image/gif
image/svg+xml

As long as they are loaded as binary objects in correctly configured content types, files of these types will be displayed as images in CUE.

You can, however, make it possible for CUE to display previews of additional graphic file types by installing and configuring CUE Graphics Converter. CUE Graphics Converter is a service for converting uploaded graphics files to formats that can be displayed by CUE. It uses the Changelog Daemon to monitor the Content Store, watching for the addition of graphics files in specified formats. When a graphics file in one of these formats is added to the Content Store, the CUE Graphics Converter:

- Converts the file to a format that CUE can display (JPEG, for example)
- Loads the converted graphics file to the Content Store as a variant of the original file

Whenever the content item containing this graphics file is loaded to CUE, CUE will display the variant instead of the original file. The original file is still present in the content item, however, and will be passed unchanged to CUE Front for handling by front-end applications.

By default the CUE Graphics Converter is configured to support the following additional file formats:

image/vnd.adobe.photoshop
application/pdf
application/postscript

by creating JPEG variants for them. You can, however, configure it to support additional formats if required.

3.3.2 Using Graphic Content Types

It can in many cases make sense to define more than one graphic content type in order to enforce different workflows for different kinds of graphics. It is typically the case that you want the option of using different versions of photos in different contexts, and CUE provides a crop mechanism to support this need. Other kinds of graphic files such as diagrams, maps and infographics are probably only meaningful when displayed in full. It then makes sense to have two different graphic content types, one of which does not provide any crop functionality.

The "Tomorrow Online" demo publication has two graphic content types that illustrate this division: **`picture`** is intended for photos and is configured to provide crop functionality, while **`graphic`** is intended for diagrams, infographics and so on, and does not offer any crop functionality.

3.3.3 Image Metadata

When a JPEG image file containing embedded IPTC metadata is added to a content item, the metadata can be automatically extracted and copied to fields in the content item. In order to enable this feature, you have to set a Content Store configuration property by adding the following line to *configuration-root*`/com/escenic/storage/metadata/MetadataInjectionTransactionFilter.properties` in one of your configuration layers:

```
serviceEnabled=true
```

For general information about how to set Content Store configuration properties, see Configuring The Content Store.

Once metadata extraction is enabled, the Content Store copies the metadata as follows by default:

Extract from IPTC field	Insert into content item field
`Caption/Abstract`	`caption`
`By-line`	`byline`
`Credit`	`credit`
`Object Name`	`title`

In addition, metadata will be copied to any content item field with a name that exactly matches the name of an IPTC field.

If the target content item does not have a field with one of the above names, then that field is not copied. You can, however, override the default behavior by adding **`parameter`** elements to the field definitions in an image content type. For example:

```
<content-type name="mypicture">
  ...
  <field mime-type="text/plain" type="basic" name="title">
    <parameter name="com.escenic.metadata" value="Headline"/>
  </field>
  <field mime-type="text/plain" type="basic" name="description">
    <parameter name="com.escenic.metadata" value="Caption/Abstract"/>
  </field>
  ...
</content-type>
```

The above example will copy the the content of the IPTC **`Headline`** field into the content item **`title`** field, instead of the content of the IPTC **`Object Name`** field. It will also copy the content of the IPTC **`Caption/Abstract`** field into the content item **`description`** field.

The **`parameter`** element must appear as the child of the target **`field`** definition, which must be a **`basic`** field with the MIME type **`text/plain`**. The **`name`** attribute must always be set to **`com.escenic.metadata`** and the **`value`** attribute must be set to the name of the source IPTC field.

Should you have image metadata handling requirements that cannot be met by this functionality, you can write your own plug-in using the Java AP described in Metadata Extraction.

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.

3.8.1.1 Constraining Relations

Relation types are displayed as drop zones in editor metadata panels in CUE. Dropping a content item into a relation drop zone in another content item 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>
```

3.8.1.2 Customizing Relation Asset Picker Filters

You don't have to drag and drop a content item into a relation drop zone to create a relation in CUE, you can also do it by clicking on the drop zone's + button and then selecting the content item to add using the displayed **asset picker** dialog. The asset picker dialog is a standard CUE dialog for selecting content items, containing a search field and result list. The asset picker also has a filter button, that displays a pop-out filter panel containing various filters that you can use to narrow down the list of results.

By default all asset pickers have the same standard filter panel. You can, however, replace this standard filter panel with a custom set of filters. You might, for example, want to have different filters available when selecting pictures. You would then need to define a custom filter panel as described in section 4.5, and use a **ui:search-filter-name** element to add it to your relation type definition:

```
    <relation-type name="pictures">
```

```
    <ui:search-filter-name>my-picture-search-filter</ui:search-filter-name>
    <ui:label>Pictures</ui:label>
    <allow-content-types>
      <ref-content-type name="picture"/>
    </allow-content-types>
  </relation-type>
```

3.8.2 Displaying Relation Types in CUE's Main Editor Area

Relation types are by default displayed as drop zones in a CUE editor's metadata panel. In some cases, however, you may prefer to display the drop zones in the main editor. There is more space available in the main editor (useful for image and video relations), and it also increases the prominence of the relations. You can achieve this as follows:

- Add a **ui:editor-style** element to the **relation-type** definition, with a value of either **relation-gallery** or **gallery**.
- Use a **ui:ref-relation-type** element to specify where you want the **relation-type** to appear (that is, on which panel, and where).

Assume, for example, that you have a relation type group defined like this:

```
<relation-type-group name="attachments">
  <relation-type name="related-videos">
    <allow-content-types>
      <ref-content-type name="video"/>
    </allow-content-types>
  </relation-type>
  <relation-type name="related-stories">
    <allow-content-types>
      <ref-content-type name="story"/>
    </allow-content-types>
  </relation-type>
</relation-type-group>
```

The **relation-type-group** can then be included in a content type definition in the usual way by adding **ref-relation-group** elements to **content-type** elements. For example:

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

With this configuration, both the relations in the group will be displayed on the metadata panel in the usual way. You can, however, move either or both of the relations to the main editor area by adding **ui:editor-style** elements to the **relation-type** elements. The following configuration, for example:

```
<relation-type-group name="attachments">
  <relation-type name="related-videos">
    <ui:editor-style>relation-gallery</ui:editor-style>
    <allow-content-types>
      <ref-content-type name="video"/>
    </allow-content-types>
  </relation-type>
  <relation-type name="related-stories">
```

```
    <allow-content-types>
      <ref-content-type name="story"/>
    </allow-content-types>
  </relation-type>
</relation-type-group>
```

moves the **related-videos** relation to the main editor area but leaves the **related-stories** relation in the metadata panel.

Relations that are moved to the main editor in this way are by default displayed on the content type's first panel, after all the panel's ordinary fields. You can, however, move them to different panels and place them exactly where you want by making use of **ui:ref-relation-type** elements. You simply insert a **ui:ref-relation-type** element in the required location. The **ui:ref-relation-type** must have a **ref** attribute referencing the relation you want to move. To move the related-videos relation to the **story** content type's **media** panel, for example:

```
<content-type name="story">
  ...
  <ref-relation-type-group name="attachments"/>
  <panel name="media">
    ...
    <ui:ref-relation-type ref="related-videos">
    ...
  </panel>
  ...
</content-type>
```

> Note that while the **ref-relation-type-group** is an ordinary **content-type** element belonging to the **http://xmlns.escenic.com/2008/content-type** namespace, the **ref-relation-type** element belongs to the **http://xmlns.escenic.com/2008/interface-hints** namespace. That means you need to specify the namespace when you use it – either explicitly with a **ns** attribute or by using a namespace prefix such as **ui:**.

The **ui:editor-style** value **relation-gallery** can be used for all relation types, and other than allowing the relation type to be moved to the main editor area, does not affect how the relations are displayed in CUE. The **gallery** setting, on the other hand can only be used for image relation types, and enforces an image-specific gallery layout in CUE. These are the only differences between the two values: the placement rules described above are the same for both settings.

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 the following purposes:

- Defining publication types
- Defining the story element types and storyline templates used in CUE storylines
- Defining container types
- Defining custom workflows
- Defining custom search filter panels
- Defining capabilities
- Defining custom content cards

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 Publication Types

Publication type resources are relatively small XML files, each one defining a single publication type. They define the following characteristics of a publication:

- The icon and label used to represent the publication in CUE
- Which optional features the publication provides

Here is the **`default`** publication type supplied with the Content Store, used for standard online publications:

```
<?xml version="1.0" encoding="UTF-8"?>
<publication-type
                  xmlns="http://xmlns.escenic.com/2019/publication-type"
                  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
                  name="default">
  <ui:label>Default</ui:label>
  <ui:description>The default publication type</ui:description>
  <ui:icon>online</ui:icon>
  <features>
    <feature name="createDefaultInbox" enabled="true"/>
    <feature name="createDefaultFrontpage" enabled="true"/>
    <feature name="createIncomingSection" enabled="true"/>
  </features>
</publication-type>
```

All the optional features are enabled in this publication type. For other publications, these features may not be of interest and should then be disabled to save resources. Here, for example, is the **`print`** publication supplied with the Content Store:

```
<?xml version="1.0" encoding="UTF-8"?>
<publication-type xmlns="http://xmlns.escenic.com/2019/publication-type"
```

```
                  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
                  name="print">
  <ui:label>Print</ui:label>
  <ui:description>Used for print publications</ui:description>
  <ui:icon>print</ui:icon>
  <features>
    <feature name="createDefaultInbox" enabled="false"/>
    <feature name="createDefaultFrontpage" enabled="false"/>
    <feature name="createIncomingSection" enabled="false"/>
  </features>
</publication-type>
```

For a complete reference description of the **`publication-type`** format, see publication-type.

4.2 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:

`answer`
`embed`

external_link
facebook_attachments
facebook_post
gallery
headline
image
internal_link
interview
lead_text
map
note
paragraph
print_assets
print_body
print_byline
print_head_deck
print_head
pull_quote
question
relation
table
video

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.

4.2.1 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
- Complex structures such as interviews (an interview is a sequence of specially formatted questions and answers)
- 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. The following sections contain descriptions of a few different element types.

4.2.1.1 Paragraph Element Type

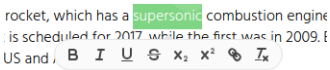
Here is a story element type definition for a paragraph:

```
<?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>
  <ui:priority>800</ui:priority>
    <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"`**)
- Will be positioned in the storyline editor's insert menus using the **`<ui:priority/>`** setting: it will appear after elements with higher priority settings, and before elements with lower priority settings.
- 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 4.2.1.7.

4.2.1.2 Image Element Type

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:element-style>image</ui:element-style>

    <ui:label>Image/Gallery</ui:label>
    <ui:icon>photo</ui:icon>
    <ui:priority>700</ui:priority>

    <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>
        <ui:search-filter-name>image-search-filter</ui:search-filter-name>
    </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 to which an image may be added.

`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. **`<ui:search-filter-name>image-search-filter</ui:search-filter-name>`** means that the CUE asset picker dialog that can be used to select the image to be added to the field will include a custom search filter called **`image-search-filter`** instead of the default search filter. For information on how to create custom search filters, see section 4.5.

The **`<ui:element-style>image</ui:element-style>`** at the top of the definition is an instruction to CUE to treat story elements of this type as image elements.

4.2.1.3 Headline Element Type

This example illustrates the use of the **`ui:keystroke`**, **`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>
    <ui:priority>900</ui:priority>
    <ui:keystroke>h</ui:keystroke>
    <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:keystroke** element must appear as a child of the **story-element-type** element. It defines a single alphabetic keystroke that can be used as part of a CUE semantic shortcut for transforming a story element of some other type into the current type (**headline** in this case). With this definition, a CUE user can convert a **paragraph** (for example) into a **heading** by pressing **Shift Shift t h**.

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.

4.2.1.4 Interview Element Type

The **interview** element type is an example of a complex element type.

```
<?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="interview">
  <ui:label>Interview</ui:label>
  <ui:icon>interview</ui:icon>
  <ui:priority>450</ui:priority>
  <elements>
    <constraints>
      <allowed-story-elements>
        <ref-story-element-type name="question"/>
        <ref-story-element-type name="answer"/>
      </allowed-story-elements>
    </constraints>
    <base-story-element>
      <ref-story-element-type name="question"/>
    </base-story-element>
    <default-story-elements>
      <ref-story-element-type name="question"/>
    </default-story-elements>
    <ui:element-flow element="question" next="answer"/>
  </elements>
</story-element-type>
```

A complex element type is perhaps best regarded as a "mini-storyline". Its internal structure is exactly the same as the internal structure of a **storyline-template** element. It defines a constrained sequence of elements that can be inserted as block into a storyline: In this case a sequence of **question** and **answer** elements (each of which has its own element type definition).

For a more detailed description of the internal structure of both storyline templates and complex element types, see section 4.2.2.

One thing to note from this example, however, is the use of the **`ui:element-flow`** element. This element modifies the behavior of CUE. The default behavior of CUE is to create a new element of the base element type when the user presses **`Enter`** in the storyline editor. The **`ui:element-flow`** element changes this behavior so that if the insertion caret is in a **`question`** element, then pressing **`Enter`** creates an **`answer`** element.

4.2.1.5 Table Element Type

The **`table`** element type illustrates two features you can use for creating more complex types of story elements:

- Field editors
- Settings fields

```
<?xml version="1.0" encoding="UTF-8"?>
<story-element-type xmlns="http://xmlns.escenic.com/2008/content-type"
                    xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
 name="table">
  <ui:label>Table</ui:label>
  <ui:priority>550</ui:priority>
  <ui:icon>table</ui:icon>
  <field mime-type="application/json" type="basic" name="tableeditor">
    <ui:editor type="web-component" name="data-visualization" />
  </field>

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

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

The **`tableeditor`** field above illustrates how you associate a field with a CUE **field editor**. Field editors are custom-built editors for displaying and editing specialized field types in CUE. You can, for example, create specialized date picker components for editing date fields, map components for editing location coordinates, and so on. Field editors can either be implemented as web components (created using HTML, Javascript and so on) or as enrichment services (that is, web services). In this case the **`tableeditor`** field is associated with a CUE web component extension called **`data-visualization`**. All of the **`table`** element type's table editing functionality is provided by this web component, not by CUE itself.

For further information about CUE web components and enrichment services, see the CUE documentation. For further information about the **`ui:editor`** element used to associate the field with the **`data-visualization`** web component, see ui:editor.

The **rowheader** and **columnheader** fields (boolean switches that determine whether a table should have row or column headings are **settings fields**. A settings field doesn't appear in the story element itself in CUE, but in a separate **Settings** dialog, displayed by clicking on a **Settings** button:

The **rowheader** and **columnheader** fields are displayed in the **Settings** dialog because of the **ui:editor-style** elements included in their definitions. The **ui:editor-style** element must contain the value **settings**. For further information about this element, see ui:editor-style.

4.2.1.6 List Element Types

The **list_bulleted** element type shown below is a complex element type for handling bulleted lists. A bulleted list is basically a complex element consisting of a sequence of **paragraph** story elements. Because the **paragraph** story elements are wrapped inside a **bulleted-list** story element, they are given a special appearance by CUE – they are indented and prefixed with a bullet character. The **list_numbered** element type is more or less identical, but uses a numerical prefix instead of a bullet.

```
<story-element-type xmlns="http://xmlns.escenic.com/2008/content-type"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints" name="list_bulleted">
  <ui:label>Bulleted List</ui:label>
  <ui:group-prefix-label>List</ui:group-prefix-label>
  <ui:element-style>list</ui:element-style>
  <ui:keystroke>b</ui:keystroke>
  <ui:count show="true" for="body total"></ui:count>
  <field name="list_style" type="enumeration">
    <ui:editor-style>settings</ui:editor-style>
    <ui:label>List style</ui:label>
    <ui:value-if-unset>disc</ui:value-if-unset>
    <enumeration value="disc">
      <ui:label>Bullet</ui:label>
    </enumeration>
    <ui:list-style-field/>
    <ui:hidden/>
  </field>
  <elements>
    <ui:label>List</ui:label>
    <constraints>
     <allowed-story-elements>
        <ref-story-element-type name="list_bulleted"/>
        <ref-story-element-type name="list_numbered"/>
        <ref-story-element-type name="paragraph"/>
      </allowed-story-elements>
    </constraints>
    <base-story-element>
      <ref-story-element-type name="paragraph"/>
    </base-story-element>
    <default-story-elements>
      <ref-story-element-type name="paragraph"/>
    </default-story-elements>
    <ui:element-wrap element="paragraph" in="list_bulleted">
      <ui:keystroke>TAB</ui:keystroke>
    </ui:element-wrap>
    <ui:element-wrap element="paragraph" in="list_bulleted" content="-">
      <ui:keystroke>SPACE</ui:keystroke>
    </ui:element-wrap>
    <ui:element-wrap element="paragraph" in="list_numbered" content="1.">
      <ui:keystroke>SPACE</ui:keystroke>
```

```
    </ui:element-wrap>
    <ui:element-unwrap element="paragraph" unwrap-mode="keep-in-group">
      <ui:keystroke>shift TAB</ui:keystroke>
    </ui:element-unwrap>
    <ui:element-unwrap element="paragraph" content="">
      <ui:keystroke>ENTER</ui:keystroke>
    </ui:element-unwrap>
  </elements>
</story-element-type>
```

The list element types have the following interesting features:

- **<ui:element-style>list</ui:element-style>** is an instruction to CUE to treat story elements of this type as list elements. This means, among other things that the +, settings and delete buttons normally displayed for each story element are suppressed inside the list (but not for the list as a whole). List items can only be added and removed using the keystrokes defined in **ui:element-wrap** and **ui:element-unwrap** elements (see below).
- The **<ui:list-style-field/>** element identifies its parent field as a list style field (that is, a field used to define the style of the list). The field can contain any CSS **list-style-type** property value (**disc**, **circle**, **square** etc. for bulleted lists, **decimal**, **upper-alpha**, **lower-alpha** etc. for numbered lists) Currently, the list style fields in the supplied story element types each contain only a single alternative (**disc** for bulleted lists and **decimal** for numbered lists). The field is therefore hidden. You can however add alternative values and remove the **ui:hidden** element. Note, however, that this setting only supports the display of alternative list styles in CUE: you may need to make corresponding changes in your presentation/print applications to support the additional list styles.
- The **allowed-story-elements** list includes not only **paragraph** (representing list items) but also **list_bulleted** and **list_numbered**. In other words, a list is allowed to contain sublists.
- **ui:element-wrap** defines a function for wrapping a story element inside another story element (in this case, wrapping a paragraph story element inside a **list_bulleted** or **list_numbered** story element. This, for example:

  ```
  <ui:element-wrap element="paragraph" in="list_bulleted">
    <ui:keystroke>TAB</ui:keystroke>
  </ui:element-wrap>
  ```

  specifies that pressing the **TAB** key with the caret marker inside a **paragraph** story element (that is, a list item), will wrap a **list_bulleted** story element around it, effectively moving it into a sublist.

  Adding a **content** attribute to a **ui:element-wrap** element makes the function dependent on the current content of the target story element:

  ```
  <ui:element-wrap element="paragraph" in="list_numbered" content="1.">
    <ui:keystroke>SPACE</ui:keystroke>
  </ui:element-wrap>
  ```

  Here, pressing the **SPACE** key will only wrap the **paragraph** element in a **list_numbered** element if its current content is "**1.**". Effectively, this means you can start a numbered sublist by simply typing "**1.SPACE**". A third **ui:element-wrap** uses the same mechanism to let users start a bulleted list by typing "**-SPACE**"
- **ui:element-unwrap** defines a function for unwrapping a wrapped story element. This for example:

  ```
  <ui:element-unwrap element="paragraph" unwrap-mode="keep-in-group">
  ```

```
    <ui:keystroke>shift TAB</ui:keystroke>
  </ui:element-unwrap>
```

specifies that pressing **shift TAB** with the caret marker inside a wrapped **paragraph** story element, will unwrap it. In the case of a list item, in other words it will move it from the sublist to the parent list. The **unwrap-mode="keep-in-group"** attribute specifies that this function will never unwrap the target element completely (that is, move it to the root level of the storyline). In the context of a list item, this means you can use **shift TAB** to move a list item from its current sublist to a parent list, but you cannot use it to change a list item into an ordinary paragraph.

This **ui:element-unwrap** element does allow the user to convert a list item into an ordinary paragraph, since it has no **unwrap-mode="keep-in-group"**:

```
<ui:element-unwrap element="paragraph" content="">
  <ui:keystroke>ENTER</ui:keystroke>
</ui:element-unwrap>
```

However, the **content=""** attribute means that this unwrap function will only work on empty list items.

- **ui:group-prefix-label** defines a string to be prepended to the labels of any child story elements. In the case of these lists, The prefix **List** is used to turn the **Paragraph** labels of the list item **paragraph** elements into **List Paragraph** labels.

4.2.1.7 Custom Annotations

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

4.2.1.7.1 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).

4.2.1.7.2 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>
```

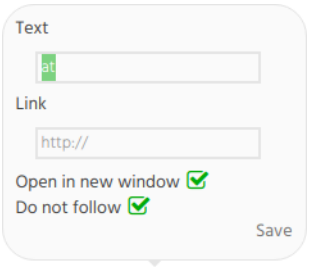
The **allow-story-element-type** element specifies that an annotation may include a reference to a single story element, and the **ref-story-element-type** element specifies that the story element must be of the type **external_link**.

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.

4.2.2 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"?>
<storyline-template
          xmlns="http://xmlns.escenic.com/2008/content-type"
          xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
          name="strict">
  <elements>
    <constraints>
      <required-story-elements>
        <ref-story-element-type name="kicker" allow-empty="true"/>
        <ref-story-element-type name="headline"/>
        <ref-story-element-type name="lead_text"/>
        <ref-story-element-type name="image"/>
        <ref-story-element-type name="paragraph"/>
      </required-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="pull_quote"/>
        <ref-story-element-type name="image"/>
        <ref-story-element-type name="gallery"/>
        <ref-story-element-type name="embed"/>
        <ref-story-element-type name="relation"/>
        <ref-story-element-type name="map"/>
        <ref-story-element-type name="table"/>
```

```
        <ref-story-element-type name="interview"/>
      </allowed-story-elements>
    </constraints>

    <base-story-element>
      <ref-story-element-type name="paragraph"/>
    </base-story-element>
  </elements>
</storyline-template>
```

The root **storyline-template** element must contain a single **elements** element that defines the elements from which a storyline of this type is constructed. The **elements** element:

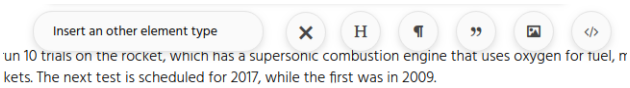
- Must contain one **base-story-element** and one **constraints** element
- May contain one **default-story-elements** element

These elements have the following usage and meaning:

default-story-elements
If present, this element 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.

constraints
This element specifies rules about what the storyline may contain. The primary constraint is specified with an **allowed-story-elements** element. Additional constraints may be specified with **required-story-elements**, **min** and **max** elements. These elements have the following usage and meaning:

allowed-story-elements
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.

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.

By default, the required elements are not allowed to be empty, thereby enforcing a strict sequence of specified elements at the top of the storyline. It is, however, possible to introduce a degree of flexibility by specifying **allow-empty="true"** on some of

the required elements. In the example above, for example, the **kicker** element above the headline is allowed to be empty. The publication templates can then be configured to ensure that the kicker disappears completely if it is empty, making the the kicker effectively an optional element.

minimum, **maximum**
The minimum/maximum number of story elements the storyline may contain.

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

Storyline templates 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 **storyline-template** element. So for a complete, formal description of the file format, start here.

4.3 Container Types

A container is a structure for managing content inheritance between content items. It defines the relationships between a **base** content item and a number of **variant** content items that have no content of their own but inherit content from the base content item and present it in a different form for use in a different context or **channel**. A typical container might hold a base content item belonging to an online publication (or **channel**), plus variants aimed for publication in print, Facebook and Twitter channels. Containers are publication-independent objects. The content items they contain all belong to (different) publications, but the containers themselves do not.

Container types define the types of container that can be created. The defined container types show up in CUE's **Create new** dialog in the same way as content types, allowing users to create containers rather than directly creating content items.

One container type resource file contains the definition of one container type.

A typical container type resource file looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<container-type
    xmlns="http://xmlns.escenic.com/2019/container-type"
    xmlns:ct="http://xmlns.escenic.com/2008/content-type"
    xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
    xmlns:doc="http://xmlns.vizrt.com/2010/documentation"
    name="news">
  <ui:label>News story</ui:label>

  <ui:title-field>com.escenic.container.slug</ui:title-field>
  <ct:field mime-type="text/plain" type="basic" name="com.escenic.container.slug">
    <ct:constraints>
      <ct:required>true</ct:required>
    </ct:constraints>
  </ct:field>

  <ct:relation-type name="com.escenic.container.items">
    <ui:label>Content items in this container</ui:label>
  </ct:relation-type>
```

```
    <first-destination content-type="storyline" publication="tomorrow-online">
      <constraints>
        <allowed-destination content-type="print-story" publication="tomorrow-today"
  max="1"/>
        <allowed-destination content-type="facebook" publication="tomorrow-facebook"/>
        <allowed-destination content-type="twitter" publication="tomorrow-twitter"/>
      </constraints>
    </first-destination>
    <first-destination content-type="storyline" publication="tomorrow-sport"/>

    <copy-fields>
      <copy-field source-content-type="storyline"
                  target-content-type="print-story"
                  source-field="metadata.authors"/>
    </copy-fields>

  </container-type>
```

This defines a container type called **news**. The **ui:label** element ensures that it will show up in CUE's **Create new** dialog as "News story" rather than just "news".

According to the formal syntax definition of a container type resource, the inclusion of **field** and **relation-type** elements is optional, and you can include as many as you like of each. In reality you should include one of each, defined exactly as shown in the example above. The **field** element is used to define a slug (that is, working title) field for the container, and the **relation-type** element defines a relation drop zone used to hold relations to all the content items in the container.

Both the slug field and the content items relation are displayed on an automatically created tab called **Main** that is added to all the container's content items in CUE. This means that whenever you open one of the content items belonging to a container:

- You can see that it belongs to a container (because it has a **Main** tab).
- You can see and edit the container's slug.
- You can see what other content items belong to the same container and open them.

The **relation-type** element must be called **com.escenic.container.items**. The field element can be called anything you like although **com.escenic.container.slug** is recommended. Whatever you call it, however, you must declare it as the slug field by including a **ui:title-field** element as shown in the example.

The most important elements in a container type resource are the **first-destination** elements, as these determine which content types can be used as base content items. A container type resource must contain at least one **first-destination** element, but may contain many. Each element specifies a **publication** and a **content-type**. When the CUE **Create new** dialog is displayed, these **first-destination** elements determine which container types are displayed as options. A container type will only be displayed if at least one of its **first-destination** elements specifies a publication that is visible to the current user. A publication is visible to a CUE user if:

- The user has the right to create content in the publication.
- The user has configured CUE to show the organizational unit to which the publication belongs.

> In addition, a container type will only be displayed if the content types referenced in its **first-destination** elements are correctly configured for use in containers. Specifically, a content

type must have a storyline field that is constrained to use only one specific storyline template. For further information about this, see section 3.2.2.1.

When the user selects one of the displayed container types, the **first-destination** elements are again used to determine exactly what is created. If the user has the right to create content in only one of the publications named in the **first-destination** list, then that **first-destination** element is used to create a container. In the case of the container type example shown above, a user with creation rights in Tomorrow Online only would get a container holding a base **storyline** content item in **tomorrow-online**. A user with creation rights in Tomorrow Sport only, on the other hand, would get a container holding a base **storyline** content item in **tomorrow-sport**.

The **copy-fields** element is optional. It is used to control the **copying** (not inheritance) of metadata, fields and relations from base content items to variants. It contains a sequence of one or more **copy-field** elements. Each **copy-field** element specifies the circumstances in which a particular field should be copied from the base content item to a variant: **if** the base content item is of type **source-content-type** **and** the variant is of **target-content-type** **then** copy the field, metadata field or relations specified in **source-field**. An optional **target-field** attribute allows fields and relation types to be copied to different fields/relation types in the target.

A **first-destination** element may contain a **constraints** element, which in turn must contain one or more **allowed-destination** elements. If a **constraints** element is specified, then the list of **allowed-destination**s determines what kinds of variants the base content item is allowed to have: that is, it limits the publication/content type combinations offered by CUE when the user makes a variant. If no **constraints** are specified, then CUE will offer all destinations to which the user has access, if constraints are specified then CUE will only offer the allowed destinations to which the user has access. It is also possible to limit how many times a destination is used by specifying the **allowed-destination** elements **max** attribute. In the example shown above, each **storyline** in Tomorrow Online may only be used to create one **print-story** in Tomorrow Today, while they can be used to create any number of Facebook and Twitter posts.

For a complete reference description of the **container-type** format, see container-type.

4.4 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 4.4.1 and section 4.4.2 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.

4.4.1 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**.

4.4.2 The Workflow Definition Elements

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

4.4.2.1 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.

4.4.2.2 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>
```

4.4.2.3 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 4.4.1.

4.4.2.4 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.

4.4.2.5 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. The following roles are available by default:

- **`reader`**
- **`journalist`**
- **`editor`**
- **`useradmin`**
- **`sectionadmin`**
- **`tagadmin`**
- **`administrator`**
- **`articleWithContentTypeReader`**
- **`articleWithContentTypeWriter`**

You can, however, define additional roles if needed. See section 4.8 for further information.

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

4.4.2.6 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/>
```

4.4.2.7 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 one optional attribute:

`time-control`
: Specifies that this permission is only granted for time-controlled content items that are in a specified time control sub-state. It only makes sense to use this attribute in combination with the **`published`** state, since time control is not relevant for other states. It may be set to one of the following values:

  `scheduled`
  : The specified permission is only granted for content items that are scheduled for publishing but not yet actually published.

  `expired`
  : The specified permission is only granted for content items that were published but have now expired.

  `none`
  : No effect: specifying this value is the same as omitting the **`time-control`** attribute.

The **`permission`** element's content must be one of the following permission names:

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

4.4.3 Modifying Workflow Definitions

Once a workflow definition has been taken into use, you must think hard before making changes to it.In particular, **you must never delete a workflow state if you have content items in that state**: if you do, the resulting "stateless" content items will become unusable.

The correct way to take a content state out of use is to remove all transitions to that state from the workflow definition. You can then open all content items in the unwanted state and transition them to a different state. Once you are certain you have transitioned all your content out of the unwanted state, you can delete it from the workflow definition.

4.5 Custom Search Filter Definitions

Custom search filters are defined in XML files, one per file. One sample search filter definition is supplied in the Content Store starter pack: **`contrib/starter-pack/search-filter/main.xml`**. This file is a copy of the default search filter definition included in the Content Store, so uploading it will have no effect: it's already there. You can however, use it as a starting point for any modifications you want to make to CUE search functionality. You can use search filter definitions in two ways:

- You can modify the default search filter definition used in CUE's search panel
- You can create one or more additional search filters to be used in dashboards (see section 4.6) and/or in asset picker dialogs (see (section 3.8.1.2) and section 4.2.1.2)

The overall process is much the same in both cases:

1. Copy **`contrib/starter-pack/search-filter/main.xml`** to a new file.
2. Make the required changes to the copied file.
3. Upload the modified file to the Content Store as described in Upload a Resource.

If you are modifying the the default search filter definition used in CUE's search panel then you need to make sure of the following points:

- That you do not change the value of the **`ui:panel`** element while editing the file: it must have the value **`cue-search-sidepanel`**.
- That you enter the path **`/escenic/search-filter/main`** in the **URI or resource** field when uploading the modified file. This ensures that it overwrites the default definition.

If you are creating an additional search filter to be used in a dashboard, then you need to make sure of the following points:

- That you remove the **`ui:panel`** element while editing the file.
- That you **do not** enter the path **`/escenic/search-filter/main`** in the **URI or resource** field when uploading the modified file (otherwise you will break the default main search panel). You should enter a path of the form **`/escenic/search-filter/`***filter-name* - **`/escenic/search-filter/my-search-filter`** for example.

4.5.1 Editing a Search Filter Definition

A search filter definition resource is an XML file with the following overall structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<search-filter xmlns="http://xmlns.escenic.com/2018/search-filter"
               xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
               name="main">
  <filter name="field-name">
    ...
  </filter>
  ...

  <sort-by name="sort-name">
    ...
  </sort-by>
</search-filter>
```

In other words:

- The root element is a **`search-filter`** element belonging to the namespace **`http://xmlns.escenic.com/2018/search-filter`**.
- The **`search-filter`** element contains a sequence of **`filter`** and **`sort-by`** elements. The **`filter`** elements define the fields that will appear in the CUE search panel's filter. The **`sort-by`** elements define the options that will be provided by the CUE search results sort menu.

In addition, the **`search-filter`** element **may** also contain:

- A single **`ui:panel`** element specifying the ID of the CUE panel in which the the search page is to be displayed. This should **only** be included in the **`main`** filter used for CUE's default search pane, and it must then always be set to **`cue-search-sidepanel`**.
- A single **`default-term`** element specifying a series of terms that will always be included when searching with this search filter. This should **never** be included in the **`main`** filter used for CUE's default search pane, only in search filters that are intended to be used in dashboards. The **`default-term`** element makes it possible to "specialize" the results returned by the search filter. Specifying a **`default-term`** value of **`+state:draft`**, for example, ensures that only draft content items are found, no matter what other filters the user sets in CUE.

For more detailed information about search filter definitions, see search-filter.

4.5.2 Indexing Fields

You can only use a field for filtering or sorting search results if it is indexed in the Content Store's Solr schema. Here are some of the most important fields that are indexed by default.

`activatedate`
`activepublishdate`
`ancestorsection`
`author`
`author_username_s`
`contenttype`
`creationdate`
`creator`
`expiredate`
`firstpublishdate`
`home_section`
`home_section_name`
`last_edited_by`
`lastmodifieddate`
`org_unit`
`publication`
`publication_type`
`publishdate`
`section`
`section_uniquename`
`source`
`sourceid`
`state`

For a more complete description of the default schema, including a list of how content-dependent index fields are generated, see What Gets Indexed. Or you can examine the schema itself, which you will find in the **/etc/escenic/solr/***solr-core***/schema.xml** folder on your Content Store host.

If you want to index additional fields so that you can search their contents and use them for filtering and sorting, then you can do so by editing the Solr schema. To index an enumeration field called **story_type**, for example, you might add an entry like this to the schema:

```
<field name="story_type_enum" type="string" indexed="true" stored="true"
 omitNorms="true"/>
```

To make the indexed field appear as a facet in your search filter, you need to do as follows:

- Include a **facet** element in the filter definition:

  ```
  <filter name="story_type">
    <ui:label>Story type</ui:label>
    <facet field="story_type_enum"/>
  </filter>
  ```

- Add a property to the **/com/escenic/webservice/search/RelationTypes.properties** configuration file in one of the Content Store's configuration layers:

  ```
  STORY_TYPE_ENUM=story_type_enum
  ```

  For general information about the the Content Store's configuration layers and how to work with them, see Configuring The Content Store.

For more information about customizing the index schema, see Customizing the Index Schema.

4.6 Dashboard Definitions

CUE dashboards are defined in in XML files, one per file. One sample dashboard definition is supplied in the Content Store starter pack: **contrib/starter-pack/dashboard/dashboard.xml**. You can use it as a starting point for your own dashboard definitions.

To make your own dashboard:

1. First create one or more custom search filters on which to base your dashboard (see section 4.5).
2. Copy **contrib/starter-pack/dashboard/dashboard.xml** to a new file.
3. Make the required changes to the copied file (see below).
4. Upload the modified file to the Content Store as described in Upload a Resource.

4.6.1 Editing a Dashboard Definition

A dashboard definition resource is an XML file with the following basic structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<dashboard
  xmlns="http://xmlns.escenic.com/2020/dashboard"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  name="dashboard-name">

  <ui:label>dashboard-label</ui:label>
```

```
  <ui:icon>dashboard-icon</ui:icon>

  <ref-search-filter name="search-filter-name">
    <ui:label>search-filter-label</ui:label>
  </ref-search-filter>
</dashboard>
```

All you need to do is set the following values correctly:

dashboard-name
The internal name of the dashboard (used as part of its URL). It must be unique among all dashboards you create. For example, **`my-dashboard`**.

dashboard-label
The name of the dashboard as you want it to appear in CUE.

dashboard-icon
The name or URL of the icon you want to represent this dashboard in CUE. For example **`text`**, or **`http://my-company-server/icons/my-icon.png`**. For more information about the use of the **`ui:icon`** element, see icon.

search-filter-name
The name of a search filter to be used in this dashboard. Note that the name of a search filter is the name specified inside the search filter resource file, not the name given to it when it is uploaded to the Content Store (although they may well be identical). For example, **`drafts`**.

search-filter-label
The name of a search filter as you want it to appear in CUE. For example **`Drafts`**. If you omit the **`ui:label`** element, then the search filter name will be used as a label.

You can include as many **`ref-search-filter`** elements as you want, allowing you to define dashboards containing several custom searches. You can also optionally include the following additional elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<dashboard
  xmlns="http://xmlns.escenic.com/2020/dashboard"
  xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
  name="dashboard-name">

  <ui:label>dashboard-label</ui:label>
  <ui:icon>dashboard-icon</ui:icon>

  <required-capability>capability-name</required-capability>
  <update-interval>interval</update-interval>

  <ref-search-filter name="search-filter-name">
    <ui:label>search-filter-label</ui:label>
  </ref-search-filter>
</dashboard>
```

The **`required-capability`** element (if specified) must contain the name of a CUE **capability**. A capability is a string representing a unit of CUE functionality that can be enabled or disabled for individual users. If you include a **`required-capability`** element in a dashboard definition, then the dashboard is only available to users who "have" the specified capability. A user "has" a capability if that capability is enabled for the user or its owning user group. If no **`required-capability`** element is specified, then the dashboard is made available to all CUE users.

The **update-interval** element (if specified) must contain an interval specified in seconds. It determines how frequently the dashboard's search results are updated. It is specified in seconds. If the element is omitted, then CUE's default interval of 30 seconds is used.

For a complete description of the dashboard resource format, see http://docs.escenic.com/ece-resource-ref/7.10/dashboard.html.

4.7 Capability Definitions

A capability represent a CUE user interface component and its associated functionality. A capability can be enabled or disabled for individual users or groups of users. By default, CUE / CUE Content Store are delivered with a standard set of capabilities that represent all the standard CUE sidepanels and metadatapanels. This means that CUE can be configured to deny some users (such as bloggers and freelance journalists, for example) access to certain side panels and metadata data panels. Capabilities are enabled / disabled when creating and editing user definitions in Web Studio. For details, see Editing Users and Persons.

The default capabilities are sufficient for controlling access to a standard CUE installation. If however, you extend CUE by adding your own web components, then you may wish to control access to these additions as well, and to be able to do that you need to create and upload your own capability definition resources.

Here is a very simple capability definition file:

```
<?xml version="1.0" encoding="UTF-8"?>
<capabilities xmlns="http://xmlns.escenic.com/2020/capability"
              xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">

  <capability-group name="myextension">
    <ui:label>My extension</ui:label>
  </capability-group>

  <capability name="my.sidepanel.myextension" ref-capability-group="myextension">
    <ui:label>My extension side panel</ui:label>
    <ui:description>My extension side panel</ui:description>
    <ui:value-if-unset>true</ui:value-if-unset>
  </capability>
  <capability name="my.metadata.myextension" ref-capability-group="myextension">
    <ui:label>My extension metadata panel</ui:label>
    <ui:description>My extension metadata panel</ui:description>
    <ui:value-if-unset>true</ui:value-if-unset>
  </capability>
</capabilities>
```

This file creates two capabilities, one for a custom side panel component and one for a custom metadata panel component. The name attributes must match the capability names specified in the configurations of the CUE web components you have created (see the **CUE User Guide** for details). You are free to name capabilities as you choose, but you should take care to avoid the names of the default capabilities delivered with CUE. The default capabilities all have names that start with either **cue.sidepanel**, **cue.metadata** or **cue.search**, so you can easily avoid name clashes by using your own prefix instead of **cue**.

You can control the capabilities' labels in Web Studio with the **ui:label** elements, and their default settings with the **ui:value-if-unset** elements.

The capability **capability-group** element at the start of the file groups together the capabilities that reference it. Since all three capabilities reference the same group, they are displayed as a group in Web Studio and can be enabled / disabled either as a group or individually.

You can actually create a hierarchy of capability groups, making it possible for the Web Studio user to enable and disable groups at different levels:

```
<?xml version="1.0" encoding="UTF-8"?>
<capabilities xmlns="http://xmlns.escenic.com/2020/capability"
              xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">

  <capability-group name="myextension">
    <ui:label>My extension</ui:label>
      <capability-group name="mypanels">
        <ui:label>My panels</ui:label>
      </capability-group>
  </capability-group>

  <capability name="my.sidepanel.myextension" ref-capability-group="mypanels">
    <ui:label>My extension side panel</ui:label>
    <ui:description>My extension side panel</ui:description>
    <ui:value-if-unset>true</ui:value-if-unset>
  </capability>
  <capability name="my.metadata.myextension" ref-capability-group="mypanels">
    <ui:label>My extension metadata panel</ui:label>
    <ui:description>My extension metadata panel</ui:description>
    <ui:value-if-unset>true</ui:value-if-unset>
  </capability>
</capabilities>
```

You will find the default capability definitions in the Content Store starter pack *installation-root*/**contrib/starter-pack/capabilities/cue-defaults.xml**. These capabilities are already defined, so you cannot remove any of them. You can, however, change the default capabilities by editing this file in the following ways and uploading your modified version:

- Change or translate the labels of capabilities and capability groups
- Change the default values (**ui:value-if-unset**) of capabilities and capability groups
- Reorganize the group structure (thereby changing the layout / structure in Web Studio).

In general it is recommended that you create one or more separate capability files for your own web components. You can, however add them to default capability groups defined in **cue-defaults.xml** if you wish. For example:

```
  ...
  <capability name="my.sidepanel.myextension" ref-capability-group="cue-side-panel">
    <ui:label>My extension side panel</ui:label>
    <ui:description>My extension side panel</ui:description>
    <ui:value-if-unset>true</ui:value-if-unset>
  </capability>
  ...
```

4.8 Custom Role Definitions

A **role** represents a type of CUE user such as a journalist, reader or editor. The purpose of roles is to carry a set of access rights or **permissions**. When a CUE user is assigned a role, he is granted all the permissions associated with that role. A user can be assigned multiple roles, and will then be granted the sum of all the permissions carried by the assigned roles. Roles are assigned to users when creating and editing user definitions in Web Studio. For details, see Editing Users and Persons.

A standard set of roles is supplied with CUE that are likely to meet most requirements. Should you need to create an additional role, you can do so by creating and uploading a role definitions as a shared resource. You will find the default role definitions in the Content Store starter pack's **`roles`** folder (*installation-root*`/contrib/starter-pack/roles/`). An easy way to create a new role is to copy one of these and modify it. You can also modify the supplied roles to match your corporate procedures if required.

Here is the content of the supplied journalist role definition:

```
<role xmlns="http://xmlns.escenic.com/2020/role"
      xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
      type="section" name="journalist">
  <ui:label>Journalist</ui:label>
  <permission type="user">
    <action>read</action>
  </permission>
  <permission type="person">
    <action>create</action>
    <action>delete</action>
    <action>read</action>
    <action>write</action>
  </permission>
  <permission type="publication">
    <action>read</action>
  </permission>
  <permission type="usergroup">
    <action>read</action>
  </permission>
  <permission type="pool">
    <action>add_to_inbox</action>
    <action>remove_from_inbox</action>
    <action>read</action>
  </permission>
  <permission type="section">
    <action>read</action>
  </permission>
</role>
```

Note that this role provides permissions for all the objects that make up a publication **except** for content items. There is in fact a permission type for content items (**`<permission type="article">`**), but it is not used much because content item permissions are primarily associated with roles in workflow definitions (see section 4.4). This allows a user's content item permissions to depend on content item state.

For a detailed description of the role definition resource format, see role.

4.9 Custom Content Card Definitions

Custom content card definitions can be used to create customized layouts for the **content cards** that CUE uses to represent content items in lists (search results lists, for example). CUE has a standard content card layout that is used if nothing else is specified, so you only need to upload content card definitions if the standard layout does not meet your requirements.

A custom content card definition can only be used to define what items of information appear on the cards and where they appear. You cannot use it to change the graphical appearance of the cards. A content card has four modifiable areas: **`title`**, **`summary`**, **`date`** and **`thumbnail`**, and all you can change is what appears in them. You cannot change their size, shape, colour or position.

4.9.1 Content Card Definition Format

This is what a content card definition looks like:

```
?xml version="1.0" encoding="UTF-8"?>
<content-card xmlns="http://xmlns.stibodx.com/2020/content-card" name="story">
    <field name="title">
        <template>{% if item.containerSlug is defined %}{{ item.containerSlug }}{%
 else %}{{ item.title }}{% endif %}</template>
    </field>

    <field name="summary">
        <template>{{ item.summary }}</template>
    </field>

    <field name="date">
        <template>{{ item.creationDate | date("D j M Y")}}</template>
    </field>
</content-card>
```

Its root element is called **`content-card`**, and must have a **`name`** attribute. It may contain between one and three **`field`** elements, each defining the content of a modifiable area on the card:

`title`
The title area at the top of the card.

`summary`
The card's main area.

`date`
The date area in the bottom right corner of the card.

You do not have to supply all three **`field`** elements: any area for which you don't supply a definition will get the default content.

Each **`field`** element must contain a single **`template`** element containing a Twig template. This template defines the content to be displayed in this area of the card.

For a formal description of the content-card definition format, see content-card

4.9.2 Twig Template Processing

The Twig templates in the content card definitions are processed by CUE. When rendering the templates, CUE has access to a set of values provided as properties of an object called **`item`**. The label

of a content item's owning publication, for example can be included in one of the content card areas by including the following Twig code in the relevant template:

```
{{ item.homePublicationLabel }}
```

For a complete description of Twig syntax, visit the Twig web site.

> Note that CUE does not render the output from the Twig templates as HTML, it just inserts it as a string in the appropriate area of the content card. Therefore you cannot format output by including HTML code in the templates: if you do this, the HTML code will be passed straight through and displayed in the content card.

The **item** object provides the following properties for use in the Twig templates:

title
The content item title. This is the value displayed in the **title** area by default. When a content card is displayed in a section page or relation area, the local summary title will be used if defined.

originalTitle
The content item title. In this case, the original title defined in the content item itself is always used, even on content cards displayed in section pages and relation areas.

label
The label of the content item's type. This property is only present if the content type has been defined with a label.

summary
The content item summary. This is the value displayed in the **summary** area by default.

activePublishedDate
The content item's "active/published" date, in the format **YYYY-MM-DDThh:mm:ssTZD** (e.g. **2020-08-14T12:41:30+02:00**). This date is **either** the date and time at which the content item was/will be activated (if an activate date has been set) **or** the user-editable published date.

publishedDate
The content item's "published" date, in the format **YYYY-MM-DDThh:mm:ssTZD** (e.g. **2020-08-14T12:41:30+02:00**).

lastModifiedDate
The content item's "last modified" date, in the format **YYYY-MM-DDThh:mm:ssTZD** (e.g. **2020-08-14T12:41:30+02:00**).

creationDate
The content item's "creation" date, in the format **YYYY-MM-DDThh:mm:ssTZD** (e.g. **2020-08-14T12:41:30+02:00**).

dateStr
Either the content item's "last modified" date or its "creation" date (depending on how CUE is configured). The date is presented in "human readable" format date (e.g. **Today at 12:56 PM**). This is the value shown in the **date** area by default.

author
The full name of the content item's first author.

state
The label of the content item's current state (e.g. **Published**).

containerSlug
The **slug** field of the content item's container (storyline stories only).

homePublication
The **title** of the content item's home publication (e.g. **tomorrow-online**).

homePublicationLabel
The label of the content item's home publication (e.g. **Tomorrow Online**).

section
The name of the content item's home section (e.g. **New Articles**)

values
An object containing additional content item values. All content item fields that are returned as search results are available in the **values** object. You can therefore add any missing fields you need by adding the attribute **search="store"** to the field's definition in the **content-type** resource. To add a content item's **subtitle** field to the **values** object, for example, you would need to change the field's definition as follows:

```
<content-type ...>
  ...
  <field mime-type="text/plain" type="basic" name="subtitle" search="store">
    ...
  </field>
  ...
</content-type>
```

Note that you will need to include the **subtitle** field in the content item's **summary** definition in order to ensure that it is available for content cards displayed on section pages and relation drop zones as well for content cards displayed in search lists etc. If you don't want the **subtitle** field to actually appear in your summaries, you can achieve that by using the **ui:hidden** element to hide it.

```
<content-type ...>
  ...
  <field mime-type="text/plain" type="basic" name="subtitle" search="store">
    ...
  </field>
  ...
  <summary>
    ...
    <field mime-type="text/plain" type="basic" name="subtitle">
      <ui:hidden/>
    </field>
  </summary>
</content-type>
```

You do not need to repeat the **search="store"** attribute when adding the **subtitle** field to the summary.

4.9.3 Using Content Card Definitions

You can upload as many content card definitions as you like, each in its own file. The uploaded definitions are not used unless they are explicitly requested. To use a content card definition you must add a **ui:content-card** element to a content type definition, as a child of the root **content-type** element:

```
<content-type name="storyline">
    <ui:content-card>story</ui:content-card>
    ...
</content-type>
```

The specified content card definition will then be used for content items of that type (but not for any other types).

4.9.4 Controlling Content Card Thumbnail Selection

You can determine where CUE looks for images to display as thumbnails in content cards by adding **`parameter`** elements like this to the **`content-type`** definitions in the **`content-type`** resource:

```
<content-type name="storyline">
    <parameter name="com.escenic.index.thumbnail.order"
    value="relations:top, relations, storyline"/>
    ...
</content-type>
```

The name of the parameter must be **`com.escenic.index.thumbnail.order`** and its value must be a comma-separated list containing one or more of the following keywords:

`relations`
`inline`
`storyline`

The **`relations`** keyword may be followed by a colon-separated parameter. The parameter must be the name of a relation type. In the above example it is **`top`**.

CUE will look for images in the specified locations, in the order specified, and use the first image it finds as the content item thumbnail. In the example given above, for example, CUE will first look in the content item's relation group **`top`** for an image, and if it doesn't find one there, it will look in the rest of the item's relations, and finally it will look in the storyline.

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 various objects such as content items, story element types, publications, workflow states and dashboards. CUE provides a range of built-in icons that you can specify by name:

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

Alternatively you can specify various kinds of custom icons such as PNG images:

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

SVG drawings or Unicode characters.

The specific icon options available vary according to what kind of CUE object the icon is to represent. For a detailed description of the options available in each context, see here.

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.