| **Section Metadata** | |
| --- | --- |
| style | content |

A computer screen with a blue and purple screen

AI-generated content may be incorrect.

**Content modeling for AEM authoring projects**

Learn how content modeling works for projects using AEM authoring as a content source and how to model your own content.

**Prerequisites**

Projects using AEM authoring as a content source inherit the majority of the mechanics of any other Edge Delivery Services project, independent of the content source or authoring method.

Before you begin modeling content for your project, make sure you first read the following documentation.

* [Getting Started – Universal Editor Developer Tutorial](https://www.aem.live/developer/ue-tutorial)
* [Markup, Sections, Blocks, and Auto Blocking](https://www.aem.live/developer/markup-sections-blocks)
* [Block Collection](https://www.aem.live/developer/block-collection)

It is essential to understand those concepts in order to create a compelling content model that works in a content source-agnostic way. This document provides details about the mechanics implemented specifically for AEM authoring.

**Default content**

**Default content** is content an author intuitively would put on a page without adding any additional semantics. This includes text, headings, links, and images. Such content is self-explanatory in its function and purpose.

In AEM, this content is implemented as components with very simple, pre-defined models, which include everything that can be serialized in Markdown and HTML.

* **Text**: Rich text (including list elements and strong or italic text)
* **Title**: Text, type (h1-h6)
* **Image**: Source, description
* **Button**: Text, title, url, type (default, primary, secondary)

The model of these components is part of the [boilerplate for projects with AEM authoring as the content source.](https://github.com/adobe-rnd/aem-boilerplate-xwalk/blob/main/component-models.json#L2-L112)

**Blocks**

Blocks are used to create richer content with specific styles and functionality. In contrast to default content, blocks do require additional semantics.

Blocks are essentially pieces of content decorated by JavaScript and styled with a stylesheet.

**Block model definition**

When using AEM authoring as your content source, the content of blocks must be modelled explicitly in order to provide the author the interface to create content. Essentially you need to create a model so the authoring UI knows what options to present to the author based on the block.

The component-models.json file defines the model of blocks. The fields defined in the component model are persisted as properties in AEM and rendered as cells in the table that makes up a block.

{

"id": "hero",

"fields": [

{

"component": "reference",

"valueType": "string",

"name": "image",

"label": "Image",

"multi": false

},

{

"component": "text-input",

"valueType": "string",

"name": "imageAlt",

"label": "Alt",

"value": ""

},

{

"component": "text-area",

"name": "text",

"value": "",

"label": "Text",

"valueType": "string"

}

]

}

Note that not every block must have a model. Some blocks are simply [containers](https://www.aem.live/developer/component-model-definitions#container-blocks) for a list of children, where each child has its own model.

It is also necessary to define which blocks exist and can be added to a page using the Universal Editor. The component-definitions.json [file lists the components as they are made available by the Universal Editor.](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/component-definition)

{

"title": "Hero",

"id": "hero",

"plugins": {

"xwalk": {

"page": {

"resourceType": "core/franklin/components/block/v1/block",

"template": {

"name": "Hero",

"model": "hero"

}

}

}

}

}

It is possible to use one model for many blocks. For example, some blocks may share a model that defines a text and image.

For each block, the developer:

* Must use the core/franklin/components/block/v1/block resource type, the generic implementation of the block logic in AEM.
* Must define the block name, which will be rendered in the block’s table header.
  + The block name is used to fetch the right style and script to decorate the block.
* Can define a [model ID](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/field-types#model-structure).
  + The model ID is a reference to the component’s model, which defines the fields available to the author in the properties panel.
* Can define a [filter ID](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/filtering).
  + The filter ID is a reference to the component’s filter, which allows to change the authoring behavior, for example by limiting which children can be added to the block or section, or which RTE features are enabled.

All of this information is stored in AEM when a block is added to a page. If either the resource type or block name are missing, the block will not render on the page.

**WARNING**: While possible, it is not necessary or recommended to implement custom AEM components. The components for Edge Delivery Services provided by AEM are sufficient and offer certain guard rails to ease development.

The components provided by AEM render a markup that can be consumed by [helix-html2md](https://github.com/adobe/helix-html2md) when publishing to Edge Delivery Services and by aem.js [when loading a page in the Universal Editor.](https://github.com/adobe/aem-boilerplate/blob/main/scripts/aem.js) The markup is the stable contract between AEM and the other parts of the system, and does not allow for customizations. For this reason, projects must not change the components and must not use custom components.

**Block structure**

The properties of blocks are defined in the component models and persisted as such in AEM. Properties are rendered as cells in the block’s table-like structure.

**Simple blocks**

In the simplest form, a block renders each property in a single row/column in the order the properties are defined in the model.

In the following example, the image is defined first in the model and the text second. They are thus rendered with the image first and text second.

With this data:

{

"name": "Hero",

"model": "hero",

"image": "/content/dam/image.png",

"imageAlt": "Helix - a shape like a corkscrew",

"text": "<h1>Welcome to AEM</h1>"

}

You get this markup:

<div class="hero">

<div>

<div>

<picture>

<img src="/content/dam/image.png" alt="Helix - a shape like a corkscrew">

</picture>

</div>

</div>

<div>

<div>

<h1>Welcome to AEM</h1>

</div>

</div>

</div>

And it will be turned into this table representation:

+---------------------------------------------+

| Hero |

+=============================================+

| ![Helix - a shape like a corkscrew][image0] |

+---------------------------------------------+

| # Welcome to AEM |

+---------------------------------------------+

You may notice that some types of values allow inferring semantics in the markup, and properties are combined into single cells. This behavior is described in the section [Type inference](https://www.aem.live/developer/component-model-definitions#type-inference).

**Key-value block**

In many cases, it is recommended to decorate the rendered semantic markup, add CSS class names, add new nodes or move them around in the DOM, and apply styles.

In other cases however, the block is read as a key-value pair-like configuration.

An example of this is the [section metadata](https://www.aem.live/developer/markup-sections-blocks#sections). In this use case, the block can be configured to render as a key-value pair table. Please see the section [Sections and Section Metadata](https://www.aem.live/developer/component-model-definitions#sections-and-section-metadata) for more information.

With this data:

{

"name": "Featured Articles",

"model": "spreadsheet-input",

"key-value": true,

"source": "/content/site/articles.json",

"keywords": ['Developer','Courses'],

"limit": 4

}

You get this markup:

<div class="featured-articles">

<div>

<div>source</div>

<div><a href="/content/site/articles.json">/content/site/articles.json</a></div>

</div>

<div>

<div>keywords</div>

<div>Developer,Courses</div>

<div>

<div>

<div>limit</div>

<div>4</div>

</div>

</div>

And it will be turned into this table representation:

+-----------------------------------------------------------------------+

| Featured Articles |

+=======================================================================+

| source | [/content/site/articles.json](/content/site/articles.json) |

+-----------------------------------------------------------------------+

| keywords | Developer,Courses |

+-----------------------------------------------------------------------+

| limit | 4 |

+-----------------------------------------------------------------------+

**Container blocks**

Both of the previous structures have a single dimension: the list of properties. Container blocks allow adding children (usually of the same type or model) and hence are two-dimensional. These blocks still support their own properties rendered as rows with a single column first. But they also allow adding children, for which each item is rendered as a row and each property as a column within that row.

In the following example, a block accepts a list of linked icons as children, where each linked icon has an image and a link. Notice the [filter ID](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/filtering) set in the data of the block in order to reference the filter configuration.

With this data:

{

"name": "Our Partners",

"model": "text-only",

"filter": "our-partners",

"text": "<p>Our community of partners is ...</p>",

"item\_0": {

"model": "linked-icon",

"image": "/content/dam/partners/foo.png",

"imageAlt": "Icon of Foo",

"link": "https://foo.com/"

},

"item\_1": {

"model": "linked-icon"

"image": "/content/dam/partners/bar.png",

"imageAlt": "Icon of Bar",

"link": "https://bar.com"

}

}

You get this markup:

<div class="our-partners">

<div>

<div>

Our community of partners is ...

</div>

</div>

<div>

<div>

<picture>

<img src="/content/dam/partners/foo.png" alt="Icon of Foo">

</picture>

</div>

<div>

<a href="https://foo.com">https://foo.com</a>

</div>

</div>

<div>

<div>

<picture>

<img src="/content/dam/partners/bar.png" alt="Icon of Bar">

</picture>

</div>

<div>

<a href="https://bar.com">https://bar.com</a>

</div>

</div>

</div>

And it will be turned into this table representation:

+------------------------------------------------------------ +

| Our Partners |

+=============================================================+

| Our community of partners is ... |

+-------------------------------------------------------------+

| ![Icon of Foo][image0] | [https://foo.com](https://foo.com) |

+-------------------------------------------------------------+

| ![Icon of Bar][image1] | [https://bar.com](https://bar.com) |

+-------------------------------------------------------------+

**Columns Block**

The columns block is a bit different from the standard block types. Unlike those block types where content modeling is all about creating semantic HTML, the columns block is about defining a layout. If you set the number of rows and/or columns, it accordingly renders an appropriate table you can style with the block option classes.

For this reason, columns blocks have the following limitations.

* They offer no content modeling.
* They can only have rows, columns and classes (or classes\_\*) properties.
* You can only add default content (text, title, image, link/button) to the cells.

**Creating semantic content models for blocks**

With the mechanics of block structure explained, it is possible to create a content model that maps content persisted in AEM one-to-one to the delivery tier.

Early in every project, a content model must be carefully considered for every block. It must be agnostic to the content source and authoring experience in order to allow authors to switch or combine them while reusing block implementations and styles. More details and general guidance can be found in [David’s Model (take 2).](https://www.aem.live/docs/davidsmodel) More specifically, [the block collection](https://www.aem.live/developer/block-collection) contains an extensive set of content models for specific use cases of common user interface patterns.

For AEM authoring as your content source, this raises the question how to serve a compelling semantic content model when the information is authored with forms composed of multiple fields instead of editing semantic markup in-context like rich text.

To solve this problem, there are three methods that facilitate creating a compelling content model:

* Type Inference
* Field Collapse
* Element Grouping

**NOTE**: Block implementations can deconstruct the content and replace the block with a client-side-rendered DOM. While this is possible and intuitive for a developer, it is not the best practice for Edge Delivery Services.

**Type inference**

For some values we can infer the semantic meaning from the values itself. Such values include:

* **Images** - If a reference to a resource in AEM is an asset with a MIME type starting with image/, the reference is rendered as <picture><img src="${reference}"></picture>.
* **Links** - If a reference exists in AEM and is not an image, or if the value starts with https?:// or #, the reference is rendered as <a href="${reference}">${reference}</a>.
* **Rich** **text** - If a trimmed value starts with a paragraph (p, ul, ol, h1-h6, etc.), the value is rendered as rich text.
* **Class** **names** - The classes property is treated as [block options](https://www.aem.live/developer/universal-editor-blocks#block-options) and rendered in the table header for [simple blocks](https://www.aem.live/developer/component-model-definitions#simple-blocks), or as value list for items in a [container block](https://www.aem.live/developer/component-model-definitions#container-blocks). It is useful if you want to style a block differently, but don’t need to create an entirely new block. It is possible to use multiple properties as block options using Element Grouping.
* **Value** **lists** - If a value is a multi-value property and the first value is none of the previous, all values are concatenated as a comma-separated list.

Everything else will be rendered as plain text.

**Field collapse**

Field collapse is the mechanism to combine multiple field values into a single semantic element based on a naming convention using the suffixes Title, Type, MimeType, Alt, and Text (all case sensitive). Any property ending with any of those suffixes will not be considered a value, but rather as an attribute of another property.

**Images**

With this data:

{

"image": "/content/dam/red-car.png",

"imageAlt": "A red card on a road"

}

You get this markup:

<picture>

<img src="/content/dam/red-car.png" alt="A red car on a road">

</picture>

And it will be turned into this table representation:

![A red car on a road][image0]

**Links and buttons**

With this data:

{

"link": "https://www.adobe.com",

"linkTitle": "Navigate to adobe.com",

"linkText": "adobe.com",

"linkType": "primary"

}

You get this markup:

<a href="https://www.adobe.com" title="Navigate to adobe.com">adobe.com</a>

And it will be turned into this table representation:

[adobe.com](https://www.adobe.com "Navigate to adobe.com")

\*\*[adobe.com](https://www.adobe.com "Navigate to adobe.com")\*\*

\_[adobe.com](https://www.adobe.com "Navigate to adobe.com")\_

**Headings**

With this data:

{

"heading": "Getting started",

"headingType": "h2"

}

You get this markup:

<h2>Getting started</h2>

And it will be turned into this table representation:

## Getting started

**Element grouping**

While [field collapse](https://www.aem.live/developer/component-model-definitions#field-collapse) is about combining multiple properties into a single semantic element, element grouping is about concatenating multiple semantic elements into a single cell. This is particularly helpful for use cases where the author should be restricted in the type and number of elements that they can create.

For example, a teaser component may allow the author to only create a subtitle, title, and a single paragraph description combined with a maximum of two call-to-action buttons. Grouping these elements together yields a semantic markup that can be styled without further action.

Element grouping uses a naming convention, where the group name is separated from each property in the group by an underscore. Field collapse of the properties in a group works as previously described.

If you add a group and there is already a field that has the group name, the field will become part of the group as well, so that element grouping can be added to blocks that didn’t previously use it, without having to migrate content.

With this data:

{

"name": "teaser",

"model": "teaser",

"image": "/content/dam/teaser-background.png",

"imageAlt": "A group of people sitting on a stage",

"teaserText\_subtitle": "Adobe Experience Cloud"

"teaserText\_title": "Meet the Experts"

"teaserText\_titleType": "h2"

"teaserText\_description": "<p>Join us in this ask me everything session...</p>"

"teaserText\_cta1": "https://link.to/more-details",

"teaserText\_cta1Text": "More Details"

"teaserText\_cta2": "https://link.to/sign-up",

"teaserText\_cta2Text": "RSVP",

"teaserText\_cta2Type": "primary"

}

You get this markup:

<div class="teaser">

<div>

<div>

<picture>

<img src="/content/dam/teaser-background.png" alt="A group of people sitting on a stage">

</picture>

</div>

</div>

<div>

<div>

<p>Adobe Experience Cloud</p>

<h2>Meet the Experts</h2>

<p>Join us in this ask me everything session ...</p>

<p><a href="https://link.to/more-details">More Details</a></p>

<p><strong><a href="https://link.to/sign-up">RSVP</a></strong></p>

</div>

</div>

</div>

And it will be turned into this table representation:

+-------------------------------------------------+

| Teaser |

+=================================================+

| ![A group of people sitting on a stage][image0] |

+-------------------------------------------------+

| Adobe Experience Cloud |

| ## Meet the Experts |

| Join us in this ask me everything session ... |

| [More Details](https://link.to/more-details) |

| [RSVP](https://link.to/sign-up) |

+-------------------------------------------------+

**Element Grouping for Block Options**

As previously described, the classes property can be used to specify the block options of a block and children in a container block. Usually a [select](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/field-types#select) or [multiselect](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/field-types#multiselect) field is used to author the block options as a single field. However, if there are multiple options and if some of them are mutually exclusive, it is easier for authors to use multiple fields to select from for individual block options. This is possible using the [element grouping naming convention](https://www.aem.live/developer/component-model-definitions#element-grouping) as described previously.

With the data:

{

"name": "teaser",

"model": "teaser",

"classes": "variant-a",

"classes\_background": "light",

"classes\_fullwidth": true,

}

You get this markup:

<div class="teaser variant-a light fullwidth">

...

</div>

Each field in the classes group can be text, an array of texts, or a boolean. In the example above, fields with a boolean value are handled slightly differently than fields with texts. For a boolean field, the property name excluding the classes group name will be added as a block option. This is particularly helpful when a [toggle](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/field-types#boolean) is used to turn a block option on or off.

**Sections and section metadata**

The same way a developer can define and model multiple blocks, they can define different sections.

The content model of Edge Delivery Services deliberately allows only a single level of nesting, which is any default content or block contained by a section. This means in order to have more complex visual components that can contain other components, they have to be modelled as sections and combined together using auto-blocking client side. Typical examples of this are tabs and collapsible sections like accordions.

A section can be defined in the same way as a block, but with the resource type of core/franklin/components/section/v1/section. Sections can have a name and a [filter ID](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/filtering), which are used by the [Universal Editor](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/introduction) only, as well as a [model ID](https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/implementing/developing/universal-editor/field-types#model-structure), which is used to render the section metadata. The model is in this way the model of the section metadata block, which will automatically be appended to a section as a key-value block if it is not empty.

The model ID and filter ID of the default section is section. It can be used to alter the behavior of the default section. The following example adds some styles and a background image to the section metadata model.

{

"id": "section",

"fields": [

{

"component": "multiselect",

"name": "style",

"value": "",

"label": "Style",

"valueType": "string",

"options": [

{

"name": "Fade in Background",

"value": "fade-in"

},

{

"name": "Highlight",

"value": "highlight"

}

]

},

{

"component": "reference",

"valueType": "string",

"name": "background",

"label": "Image",

"multi": false

}

]

}

The following example defines a tab section, which can be used to create a tabs block by combining consecutive sections with a tab title data attribute into a tabs block during auto-blocking.

{

"title": "Tab",

"id": "tab",

"plugins": {

"xwalk": {

"page": {

"resourceType": "core/franklin/components/section/v1/section",

"template": {

"name": "Tab",

"model": "tab",

"filter": "section"

}

}

}

}

}

**Page metadata**

Documents can have a [page metadata block](https://www.aem.live/developer/block-collection/metadata), which is used to define which <meta> elements are rendered in the <head> of a page. The page properties of pages in AEM as a Cloud Service map to those that are available out-of-the-box for Edge Delivery Services, like title, description, keywords, etc.

Before further exploring how to define your own metadata, please review the following documents to understand the concept of page metadata first.

* [Metadata](https://www.aem.live/developer/block-collection/metadata)
* [Bulk metadata](https://www.aem.live/docs/bulk-metadata)

It is also possible to define additional page metadata in two ways.

**Metadata spreadsheets**

It is possible to define metadata on a per path or per path pattern basis in a table-like way in AEM as a Cloud Service. There is an authoring UI for table-like data available that is similar to Excel or Google Sheets.

For further details, please see the document [Using Spreadsheets to Manage Tabular Data](https://www.aem.live/docs/authoring-tabular-data) for more information.

**Page properties**

Many of the default page properties available in AEM are mapped to the respective page metadata in a document. That includes for example title, description, robots, canonical url or keywords. Some AEM-specific properties are available as well:

* cq:lastModified as modified-time in ISO8601 format
* The time the document was last published as published-time in ISO8601 format
* cq:tags as cq-tags as a comma-separated list of the tag IDs.

It is also possible to define a component model for custom page metadata, which will be made available to the author in the Universal Editor.

To do so, create a component model with the ID page-metadata.

{

"id": "page-metadata",

"fields": [

{

"component": "text",

"name": "theme",

"label": "Theme"

}

]

}

You can also create component models per template. To do so, create entries named <template>-metadata, with template being the value stored in the template metadata property. For details on metadata, please see the document [Metadata Block.](https://www.aem.live/developer/block-collection/metadata)

| **Pagination (contained)** | |
| --- | --- |
| Previous  [**Creating Blocks Instrumented for use with the Universal Editor**](https://www.aem.live/developer/universal-editor-blocks) | Up Next  [**Path Mapping**](https://www.aem.live/developer/authoring-path-mapping) |

| **Metadata** | |
| --- | --- |
| Title | Content modeling for AEM authoring projects |
| Image | A computer screen with a blue and yellow computer screen  AI-generated content may be incorrect. |
| Description | Learn how content modeling works for projects using AEM authoring as a content source and how to model your own content. |
| Template | guides |
| Category | Build |