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

A screenshot of a computer screen

AI-generated content may be incorrect.

**Markup, Sections, Blocks, and Auto Blocking**

To design websites and create functionality, developers use the markup and DOM that is rendered dynamically from the content. The markup and DOM are constructed in a way that allows flexible manipulation and styling. At the same time it provides out-of-the-box functionality so the developer does not have to worry about some of the aspects of modern websites.

**Structure of a Document**

The single most important aspect when structuring a document is to make it simple and intuitive for authors who will contribute the content.

This means that it is strongly recommended to involve authors very early in the process. In many cases it is good practice to just let authors put the content that needs to go on to a page into a Google Doc or Word document without having any notion of blocks and section, and then try to make small structural changes and introduce sections and blocks only where necessary.

A document follows the following structure in the abstract.

A screenshot of a document

AI-generated content may be incorrect.

A page as authored in Word or a Google Doc document uses the well-understood semantic model like headings, body text, lists, images, links, etc. that is shared between HTML, markdown, and Google Doc / Word. We call this **default content**. In an ideal situation one would leave as much of the content authored as default content as possible, since this is the natural way for authors to treat documents.  
  
In addition to default content, we have a concept of page **sections**, separated by horizontal rules or --- to group certain elements of a page together. There may be both semantic and design reasons to group content together. A simple case could be that a section of a page has a different background color.  
  
In addition to that there is concept of **blocks** which are authored as a table with a heading as the first row that identifies the type of block. This concept is the easiest approach to componentize your code.

Sections can contain multiple blocks. Blocks should never be nested as it makes things very hard to use for authors.

**DOM vs. Markup**

AEM produces a clean and easily readable semantic markup from the content that’s provided to it. You can easily access it using the **view source** feature and have a look at the markup of the page you are currently reading.

The JavaScript library used in scripts.js takes the markup and enhances it into a DOM that is then used for most development tasks, specifically to build blocks. To create a DOM that’s easy to work with for custom project code, it is best to view it as a two-step process.

In the first step, we create the markup with sections, blocks, and default content that will look similar to this.  
  
A screen shot of a computer

AI-generated content may be incorrect.

In the second step, the above mark-up is augmented into the following example DOM, which then can be used for styling and adding functionality. The differences between the markup that’s delivered from the server and the augmented DOM that is used for most of the development tasks is highlighted below.

It primarily consists of introducing a wrapper <div> for blocks and default content and dynamically adding additional helpful CSS classes and data attributes that are used by the AEM block loader.

A screen shot of a computer

AI-generated content may be incorrect.

**Sections**

Sections are a way to group default content and blocks by the author. Most of the time section breaks are introduced based on visual differences between sections such as a different background color for a part of a page.

From a development perspective, there is usually not much interaction with sections beyond CSS styling.  
Sections can contain a special block called Section Metadata, which results in data attributes to a section. The names of the data attributes can be chosen by the authors, and the only well-known section metadata property is Style which will be turned into additional CSS classes added to the containing section element.

Blocks and default content are always wrapped in a section, even if the author doesn’t specifically introduce section breaks.

**Default Content**

There is a broad range of semantics that are shared between Word documents, Google Docs, markdown, and HTML. For example there are headings of different levels (eg. <h1> - <h6>), images, links, lists (<ul>, <ol>), emphasis (<em>, <strong>) etc.

We take advantage of the intuitive understanding that authors have of how to use these semantics in the tools that they are familiar with (eg. Word/Google docs) and maps those to markdown and then renders them in the HTML markup.

All mappings should be relatively straightforward and intuitive for the developer. One area where we go a little bit further than the simplest possible translation is in handling images. Instead of a simple <img> tag, a full <picture> tag is rendered with a number of different resolutions needed for display on desktop and mobile devices as well as different formats for modern browsers that support webp and older browsers which do not.

**Blocks**

Most of the project-specific CSS and JavaScript lives in blocks. Authors create blocks in their documents and developers write the corresponding code that styles the blocks with CSS and/or decorates the DOM to take the markup of a block and transform it to the structure that’s needed or convenient for desired styling and functionality.

The block name is used as both the folder name of a block as well as the filename for the CSS and JavaScript files that are loaded by the block loader when a block is used on a page. The block name is also used as the CSS class name on the block to allow for intuitive styling.

The JavaScript is loaded as a Module (ESM) and exports a default function that is executed as part of the block loading.

All block level CSS should be scoped to the block to make sure that there are no side-effects for other parts of your project, which means that all selectors in a block should be prefixed with the corresponding block class. In certain cases it makes sense to use the block’s wrapper or containing section for the selector as well.

There is a balance of DOM manipulation in JavaScript and complexity of the CSS selectors. Complex brittle CSS selectors are not recommended and at the same time adding classes to every element makes your code more complex and disregards the semantics of elements.

One of the most important tenets of a project is to keep things simple and intuitive for authors. Complicated blocks make it hard to author content, so it is important that developers absorb the complexity of translating an intuitive authoring experience into the DOM that is needed for layout or application logic. It is often tempting to delegate complexity to the author. Instead, developers should make sure that blocks do not become unwieldy to create for authors. An author should always be able to simply copy/paste a block and intuitively understand what it is about.

A simple example is the [Columns Block](https://github.com/adobe/helix-project-boilerplate/tree/main/blocks/columns). It adds additional classes in JavaScript based on how many columns are in the respective instance created by the author. This allows it to be able to use flexible styling of content that is in two columns vs. three columns.

Blocks can be very simple or contain full application components or widgets and provide a way for the developer to componentize their codebase into small chunks of code that can be managed easily and can be loaded on to the web pages as needed.

A block’s content is rendered into the markup as nested <div> tags for the rows and columns that the author entered. In the simplest case, a block has only a single cell.

<div class=”blockname”>

<div>

<div>

<p>Hello, World.</p>

</div>

</div>

</div>

Authors can add blocks to their pages in an ad-hoc manner by simply adding a table with the block name in the first row or table heading. Some blocks are also loaded automatically. header and footer blocks that need to be present on every page of a site are a good example of that.

**Block Options**

If you need a block to look or behave slightly differently based on certain circumstances, but not different enough to become a new block in itself, you can let authors add **block options** to blocks in parentheses. These options add modified classes to the block. For example Columns (wide) in a table header will generate the following markup.

<div class=”columns wide”>

Block options can also contain multiple words. For example Columns (super wide) will be concatenated using hyphens.

<div class=”columns super-wide”>

If block options are comma-separated, such as Columns (dark, wide), they will be added as separate classes.

<div class=”columns dark wide”>

**Auto Blocking**

In an ideal scenario the majority of content is authored outside of blocks, as introducing tables into a document makes it harder to read and edit. Conversely blocks provide a great mechanism for developers to keep their code organized.

A frequently-used mechanism to get the best of both worlds is called **auto blocking**. Auto blocking turns default content and metadata into blocks without the author having to physically create them. Auto blocking happens very early in the page decoration process before blocks are loaded and is a practice that programmatically creates the DOM structure of a block as it would come as markup from the server.

Auto blocking is often used in combination with metadata, particularly the template property. If pages have a common template, meaning that they share a certain page design or functionality, that’s usually a good opportunity for auto blocking.

A good example is an article header of a blog post. It might contain information about the author, the title of the blog post, a hero image, as well as the publication date. Instead of having the author put together a block that contains all that information, an auto block (e.g. article-header block) would be programmatically added to the page based on the <h1>, the first image, the blog author, and publication date metadata.

This allows the content author to keep the information in its natural place, the document structure outside of a block. At the same time, the developer can keep all the layout and styling information in a block.

Another very common use case is to wrap blocks around links in a document. A good example is an author linking to a YouTube video by simply including a link, while the developer would like to keep all the code for the video inline embed in an embed block.

This mechanism can also be used as a flexible way to include both external applications and internal references to video, content fragments, modals, forms, and other application elements.

The code for your projects auto blocking lives in buildAutoBlocks() in your scripts.js.

Please see the following examples of auto blocking.

* [Adobe Blog](https://github.com/adobe/blog/blob/3f4318d622e2626aca65c239ad0bf2b4b45012a2/scripts/scripts.js#L823-L852)
* [AEM Boilerplate](https://github.com/adobe/aem-boilerplate)

| **Pagination (contained)** | |
| --- | --- |
| Previous  [**Keeping it 100**](/developer/keeping-it-100) | Up Next  [**Spreadsheets**](/developer/spreadsheets) |

| **Metadata** | |
| --- | --- |
| Title | Markup, Sections, Blocks, and Auto Blocking |
| Image | A screenshot of a computer screen  AI-generated content may be incorrect. |
| Description | To design websites and create functionality, developers use the markup and DOM that is rendered dynamically from the content. The markup and DOM are constructed ... |
| Template | guides |
| Category | Build |