This document describes how to format Markdown for OpenADR specification documents.

We strive to use normal Markdown as much as possible. But, to make fine-looking documentation, we need some HTML post-processing and a few Markdown extensions.

[[toc]]

TODO: On studying this generated ToC - Some of the headers in this document have HTML tags. The rendered header tag is `<code>&lt;figure&gt;</code>` which correctly encodes this to safe HTML, but in the ToC the tags are rendered as raw HTML. Hence, the ToC looks strange as a result.

File names

The file names must be `.html.md` due to the design of AkashaCMS modules. This can and will be changed shortly.

Theory of Operation

The AkashaCMS tools use a Configuration object in `config.mjs`. It describes a specific project - generally speaking an AkashaCMS project takes input files and produces a rendered output directory of HTML, CSS, images, and JavaScript.

The current configuration has `documents` containing Document files, `layouts` containing Layout templates, and `partials` containing Partial templates. No `assets` directory has been created.

The rendered output is in `out`.

Document files have an extension corresponding to the document type. For Markdown the extension is `.html.md`. AsciiDoc files are also supported which would use the extension `.html.adoc`. A stylesheet is included that is written using the LESSCSS format, and its extension is `.css.less`.

Markdown and AsciiDoc files have YAML-format frontmatter.

```
---
layout: default.html.ejs
title: Article title
---
```

This is a minimal frontmatter. The `layout` field says which layout template to use, while `title` is recognized in `default.html.ejs` as the document title.

This layout template will be in EJS format. Other template engines are supported such as Nunjucks (`.njk`).

The concept for the double extension is to make it clear what format is generated from that file. The `.css.less` file generates a CSS file from LESSCSS format, while `.html.md` generates HTML from Markdown.

Custom HTML tags

AkashaCMS also supports server-side DOM processing using a jQuery-like API.

The `<img figure src=URL>` support discussed later is an example of this DOM processing. There is also HTML cleanup, a part of which is to automatically compute relative URL references for any `<a>` or `<link>` tag.

It also supports custom HTML tags that are automatically processed during page rendering, and replaced with corresponding HTML.

See the discussion of `<schema-descriptions>` below.

CSS style sheets

Stylesheets are declared in `config.mjs` as so:

```
.addStylesheet({ href: "/vendor/bootstrap/css/bootstrap.min.css" })
.addStylesheet({       href: "/style.css" });
```

That means Bootstrap v4.6.x is available, as is a custom stylesheet. The source for this is `documents/style.css.less` and it is written in LESSCSS format. This file is automatically converted to `style.css` during rendering.

Only a subset of Bootstrap is suitable for PDF rendering.

Adding ID and Class attributes for customizing the presentation

We commonly access CSS definitions using class attributes. A Markdown-IT extension, `markdown-it-attrs`, has been installed which allows setting ID values and class attributes as so:

```
# Introduction {.page_break}
```

The format is to add `{ ... }` after an element. The extension is configured to only allow `id` and `class` attributes. You set a class attribute using the `.` character, hence `.page_break` becomes `class="page_break"`. To set an ID, use the `#` character such as `{ #introduction }`

Introducing `<div>` blocks around a section of Markdown

We might want to use CSS specific to the "title page" that doesn't apply elsewhere. Such CSS would have a selector `#title-page a { ... }` to, for example, customize anchor tags on the title page.

A Markdown-IT extension, `markdown-it-div`, has been installed for this purpose. We could instead use `<div>` tags directly. In any case, this extension works as so:

```
::: #idName or .classNames

Some content

:::
```

Which would be rendered as:

```
<div id="idName" class="or classNames">
    Some content
</div>
```

Autogenerating a Table of Contents, and adding Anchors to Hn tags

Two Markdown-IT extensions have been installed for the purpose of autogenerating a Table of Contents section.

First, `markdown-it-anchor` adds ID attributes to header tags. ID attributes are what supports a link to the middle of a document. For example, the tag `<h1 id="introduction">Introduction</h1>` has an anchor named `introduction` and a link tag `<a href="#introduction>Introduction</a>` will refer to that H1 tag.

This plugin, and the `markdown-it-attrs plugin, also supports the following

```
# Introduction { #introduction }
```

This allows the document author to specify the anchor ID rather than rely on an autogenerated anchor slug.

Second, `markdown-it-table-of-contents` scans the Markdown document looking for Hn tags, and automatically generates a Table of Contents. Simply place the following in the document where the ToC is desired:

```
[[toc]]
```

Because of the `anchor` plugin, the `table-of-contents` automatically links to the Hn tag using the anchor attribute.

Adding `<section>` tags around areas started with an Hn tag

It may be useful to treat the portion of the document starting with a header tag as a *Section* of the document. In HTML, sections are delimited by the `<section>` element.

The `markdown-it-header-sections` plugin automatically generates the following:

```
# Header title of some kind

Some content in a section

More content in a section

# Header title of the next section

...
```

This is converted to:

```
<section>
    <h1>Header title of some kind</h1>
        <p>Some content in a section</p>
        <p>More content in a section</p>
    </h1>
</section>
<section>
    <h1>Header title of the next section</h1>
    ...
    </h1>
</section>
```

The ID or Class values that can be added are automatically copied to the surrounding `<section>` tag.

Generating `<figure>` and `<figcaption>` tags

The modern HTML way of handling images is this structure:

```
<figure>
    <img alt="alt text" src="URL"/>
    <figcaption>
        caption text
    </figcaption>
</figure>
```

Since it can be tedious, and non-Markdowny, to type all that, the `markdown-it-image-figures` plugin handles this fairly easily.

```
![alt text]{URL "caption text"}
```

This is the standard Markdown way of specifying an image.

Alternatively, the AkashaCMS tools support the following:

```
<img figure src="URL" caption="caption text"/>
```

A module in AkashaCMS recognizes the `figure` attribute as a request to add the `<figure>` structure. The `caption` attribute turns into `<figcaption>` text. But, using this means not using the Markdown notation for images.

Code block highlighting

In Markdown, there are two kinds of code blocks. An inline code block is "`inline code`". This is a pair of single back-quote characters inside of which is the code section. There are many examples in this document already.

The long-form code block is a section delimited by lines starting with three back-quote characters.

The `markdown-it-highlightjs` plugin extends the treatment of the long-form code blocks with syntax highlighting. There are many examples of this already in the document. The initial back-quote delimiter has a language specifier such as "```html" for HTML.

See: https://highlightjs.org/

Note that the Highlight.js ecosystem supports a large number of languages and themes.

Diagrams - UML or otherwise - PlantUML

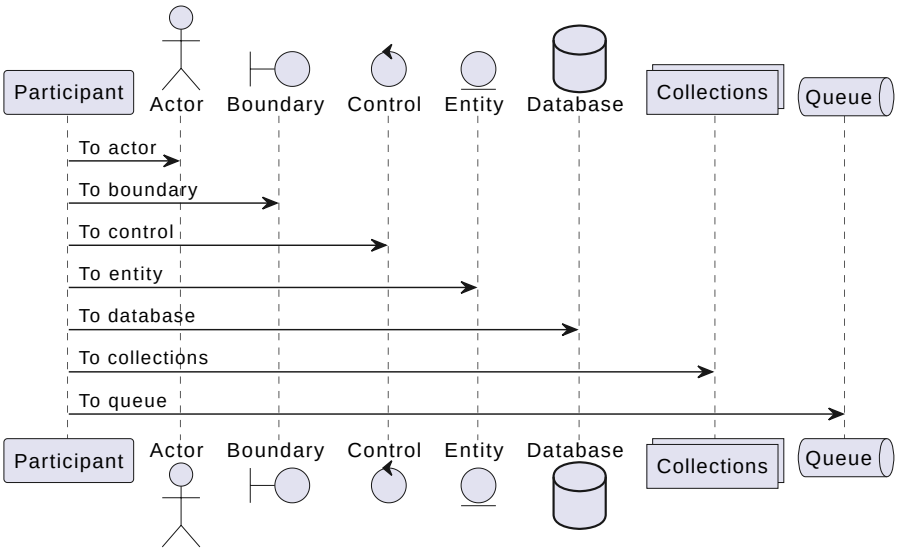
The OpenADR specification has a couple of UML diagrams. With PlantUML support the diagram description can be placed inline in the document. PlantUML converts this to SVG for display.

The PlantUML design requires a back-end server for UML rendering. By default it uses the PlantUML server, but it is preferable to install a server locally. That task has not been explored.

For example, we can add a Sequence diagram as so:

```
@startuml
participant Participant as Foo
actor       Actor       as Foo1
boundary    Boundary    as Foo2
control     Control     as Foo3
entity      Entity      as Foo4
database    Database    as Foo5
collections Collections as Foo6
queue       Queue       as Foo7
Foo -> Foo1 : To actor
Foo -> Foo2 : To boundary
Foo -> Foo3 : To control
Foo -> Foo4 : To entity
Foo -> Foo5 : To database
Foo -> Foo6 : To collections
Foo -> Foo7: To queue
@enduml
```

Which renders to the following:



Alternate approach for Diagrams - `draw.io`

At https://www.drawio.com/ - one finds a web service, and desktop application, where one can draw diagrams.

The preferred workflow is:

- Draw the diagram in `draw.io`
- Save it as a PNG making sure to include the drawio diagram inside the PNG.
- Include the PNG using normal Markdown image tags
- Use `draw.io` to edit the PNG file to edit tie diagram

Autogenerated Enumerations tables

The Definitions document has several Enumeration tables.

The OpenADR project will contain JSON Schema files that correspond to these Enumeration tables.

In the spirit of *one source of truth* it is useful if the Enumeration tables are automatically generated from the Schema files. That way the Enumeration tags and descriptions need only be written once, and be maintained in the Schema files.

To support this use:

```
<schema-descriptions
        id="event-payload-enumeration"
        schemahref="/home/david/Projects/openadr/openapi-3.0.0/enumerations/event-interval-payloads.schema.yaml">
</schema-descriptions>
```

The implementation for this tag is in `mahafuncs.mjs`, and is an example of custom HTML tags discussed earlier.

This tag reads the named Schema file - it only supports YAML format - then uses the following template to render the schema into the enumerations table.

```
<table {% if id %}id="{{ id | safe }}"{% endif %}>
    {% for def in defs -%}
    <tr><td>{{ def.key }}</td><td>{{ def.description }}</td>
    {%- endfor %}
</table>
```

This template is in Nunjucks format. The `id` attribute you see above is added to the `<table>`, then each `definitions` schema item is rendered as a table row.

Rendering Markdown to HTML, generating PDF from HTML

The other tools for rendering Markdown to PDF typically have a middle step where the Markdown is rendered to HTML.

This setup makes the HTML step explicit and it can be inspected in the `out` directory. That makes it easier to diagnose issues with generating the PDF.

Currently there is a script, `build.mjs`, and a related script `build-core.mjs`, that drives the process. It is written in JavaScript as a ZX script. (https://google.github.io/zx/getting-started) The `build-core.mjs` script uses Puppeteer-core, which requires installing a separate Chrome instance.

Puppeteer is a tool containing a Chrome instance that can be run headless. The primary target for Puppeteer is automated UI testing of web application front ends. But, because it is a real web browser, it includes a Print to PDF function.

Script execution:

```
$ time npx zx build.mjs

real    0m10.467s
user    0m7.597s
sys     0m1.575s
```

Rendering to HTML

This portion of the script renders HTML from the documents:

```
import { default as config } from './config.mjs';

const akasha = config.akasha;
await akasha.setup(config);

// await data.removeAll();
await config.copyAssets();
let results = await akasha.render(config);

// ...

await akasha.closeCaches();
```

This is how one reads an AkashaCMS configuration, then renders its documents to the output directory. There is a command-line option which is:

```
$ npx akasharender render --copy-assets config.mjs
```

This portion of the script could be replaced with the above.

Rendering HTML to PDF

With the HTML in the `out` directory the script moves on to launching Puppeteer.

```
import puppeteer from 'puppeteer';
// ...

const browser = await puppeteer.launch({
    headless: true,
    userDataDir: './tmp',
    args: [
        // ...
    ]
});

const page = await browser.newPage();
await page.goto(`file://${__dirname}/out/Definition.html`, { waitUntil: 'networkidle0' });

// Generate PDF at default resolution
const pdf = await page.pdf({
    format: 'A4',
    margin: { top: '20mm', right: '20mm', bottom: '20mm', left: '20mm' },
    displayHeaderFooter: true,
    headerTemplate: '<div class="title">TITLE GOES HERE</div>',
    footerTemplate: '<div>Page <span class="pageNumber"></span> of <span class="totalPages"></span></div>',
    printBackground: true
});

// Write PDF to file
fs.writeFileSync('PDF/Definition.pdf', pdf);

await browser.close();
```

The `headerTemplate` and `footerTemplate` shown above does not make visible headers and footers. After a lot of searching, entries in the Puppeteer issue queue said that the context where headers and footers are render have zero CSS styles, and therefore it's necessary to add more styling.

```
    headerTemplate: `
        <div class="text-center title" style="margin-left: auto; margin-right: auto; font-size: 12px;">
            TITLE GOES HERE
        </div>
    `,
    // headerTemplate: `
    //     <div class="text-center title" style="margin: 0 15mm 5mm; font-size: 12px;">TITLE GOES HERE</div>
    // `,
    footerTemplate: `
        <div class="text-left"  style="margin: 0 auto 0 20mm; text-align: left; font-size: 12px;">
```

```
            Copyright © OpenADR Alliance (2023-24). All Rights Reserved
        </div>
        <div class="text-right" style="margin: 0 20mm 0 auto; text-align: right; font-size: 12px;">
            Page <span class="pageNumber"></span> of <span class="totalPages"></span>
        </div>
    `,
```

These templates do the following

- Header has the document title, centered
- Footer has an OpenADR copyright statment on the left, and the page number on the right.

Also note that Puppeteer looks for the `class` attribute to find places to inject values. The `<span class="pageNumber">` therefore is injected with the page number.