Adobe

Adobe Summit

LAB WORKBOOK

L423 - Getting started with AEM authoring and Edge Delivery Services

Adobe Summit 2024

Table of Contents

Lab overview

In this hands-on lab you will learn how Adobe Experience Manager as a Cloud Service with Edge Delivery Services and the Adobe Experience Manager based authoring with the new Universal Editor work together - or in short, how AEM Sites CS with Universal Editor authoring publishes to Edge Delivery Services, resulting in fast development and deployment while driving exceptional site performance with Lighthouse score 100 and green Core Web Vitals.

You will learn how to create an Edge Delivery Services project setup for AEM authoring from scratch using our new GitHub boilerplate, how to configure AEM with Edge Delivery Services, how to build an AEM site, how to author AEM using Universal Editor, and how to publish pages to Edge Delivery.

Additionally, you will learn how to add new Edge Delivery Services blocks to extend your authoring experience.

Key Takeaways

1. how to use the AEM Sites with Edge Delivery GitHub project template
2. configuring AEM CS working with Edge Delivery
3. create your first AEM page from a template
4. experience the new Universal Editor authoring
5. Instant preview and publishing to Edge Delivery
6. Adding new authoring Edge Delivery features in form of Edge Delivery blocks

Prerequisites

For authors and developers

- No prerequisites other than being curious about AEM's new authoring and delivery capabilities.

Additionally for the developer tasks:

- You have a personal GitHub account, and understand git basics
- You understand basic HTML, CSS and JavaScript

An Introduction to Adobe Experience Manager based authoring with Edge Delivery Services

Objectives

1. Understand how AEM authoring and Edge Delivery Services work together.
2. Understand the architecture and how projects are set up.
3. Know about the limitations and guard rails.

Overview

AEM, Adobe Experience Manager based authoring with Edge Delivery Services is a natural extension of Edge Delivery Services with document-based authoring. It combines AEM as content storage and source, a WYSIWYG authoring experience, and features like MSM Multi-site-manager, Translations, Workflows, Launches, etc. with the web performance-optimized Edge Delivery Services delivery tier.

For the authoring experience, the authored pages are served from the AEM authoring environment and are instrumented for Universal Editor. The markup is identical to the one that would be served on a project's preview or live URL[1]. All scripts, styles and configurations are loaded from the Edge Delivery Services GitHub project and no server-side deployment to AEM is needed. Publishing from AEM to Edge Delivery Services is enabled by a configuration to link a site to an Edge Delivery Services project, and by linking the Edge Delivery Services project to AEM. With that set, publishing a page or asset will automatically preview and publish to Edge Delivery Services additionally to publishing to the AEM publish tier [2].

In general, everything that you can do with document-based authoring is possible with AEM authoring as well, but also any limitation or guardrail, that applies to Edge Delivery Services projects with document-based authoring also applies to AEM authoring.

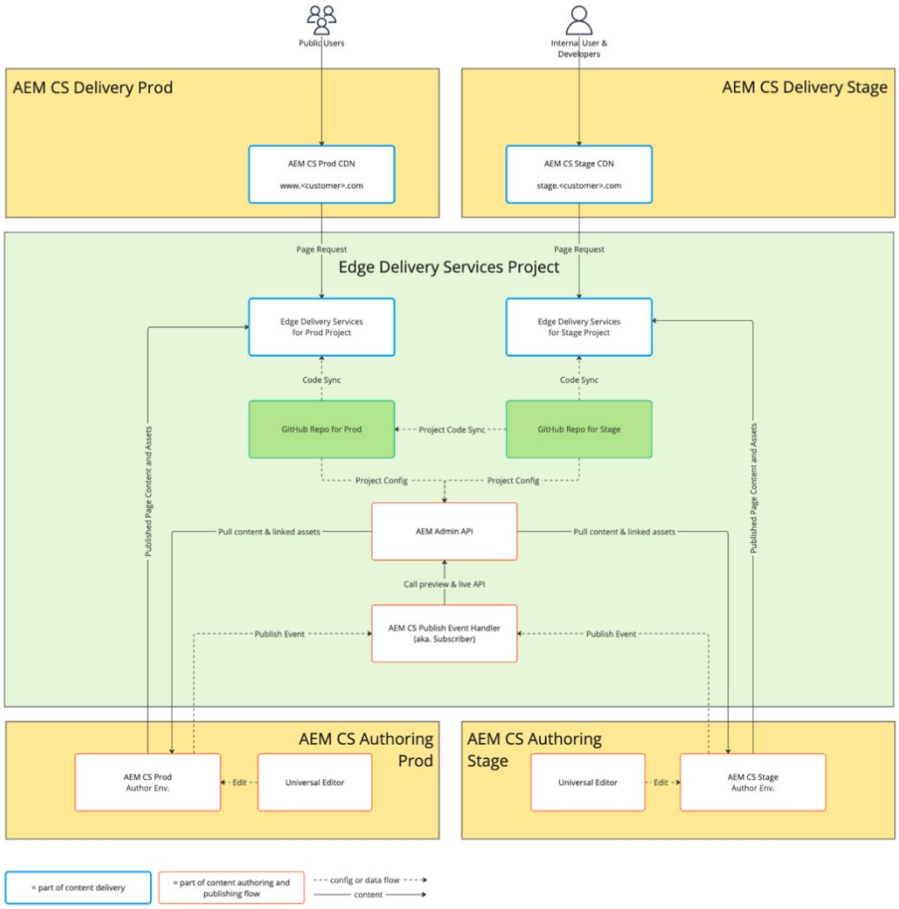
[1] There are minor differences in the markup and DOM, which are necessary to accommodate the requirements for richtext editors.

[2] Publishing to Edge Delivery Services is a two-step process. First the content is previewed, and then published. When using the "publish to preview" feature in AEM the content will only be previewed. This is a key difference in the behavior of the AEM preview and publish tier, where previewing and publishing are separate operations, and Edge Delivery Services, where publishing requires previously previewing.

Project Setup and Architecture

AEM Cloud Services comes at least with a staging and production environment per default, while not strictly necessary we recommend using both for the development of Edge Delivery Services project.

As an Edge Delivery Services project is linked to a single content source, we usually use two repositories: one linked to the stage environment and one linked to the production environment, where the latter is a fork of the former. The major development happens on the stage repository using content from the stage environment, and progress is regularly synchronized to the production repository by updating the fork.



Besides that, we recommend following our [best practices for efficient development on Edge Delivery Services].

Frequently asked Questions

What is better - AEM authoring or document-based authoring?
Edge Delivery Services is a cloud-native delivery tier that is agnostic from the content source. How content is created, authored, and maintained is up to the practitioner's decision. We do usually not make any strict recommendations. While document-based authoring is simple and intuitive when content is created with Word anyway, AEM authoring comes with the benefits of a what-you-see-is-what-you-get authoring experience and the possibility to build on existing investments into the AEM (esp. training of practitioners, utilizing MSM Multi-Site-Manager, Workflows, etc.). The developer experience and hence project velocity is about the same for both.

Can we use Style System, Editable Templates, Experience Fragments, Content Fragments, etc. with Edge Delivery Services projects and AEM Authoring?
Yes and no. We don't aim to achieve feature-parity with AEM Page Editor and Core Components, we aim for feature-parity with document-based authoring instead.

- For the Style System there is already an alternative approach (block options)
- Editable Templates violate one important pattern of Edge Delivery Services - The structured content would need to be de-referenced server-side and hence a change to a template would trigger a publication of every page created from it. This is not desired for Edge Delivery Services. However, any page can be used as editable template. Projects usually create a folder of pages that authors can copy and paste to get started faster.
- Experience Fragments cannot yet be used with Edge Delivery Services. However, a regular page can be used as fragment and loaded as such on a page.
- Content Fragments can be integrated using the AEM headless apis client side.

Other features that are primarily implement the AEM Sites like MSM[3], Translations, Launches, Workflows continue to work as they did in the past also for Edge Delivery Services projects.

Can we continue to build custom server-side components for AEM authoring?
No, server-side extensibility for the rendering stack is not recommended neither supported. AEM implements the stable markup-contract of Edge Delivery Servlets. AEM enables projects out of the box to create and model content in a way that it can be ingested into Edge Delivery Services. Anything that goes beyond that, should be done client side. Other sever-side extension points may still be implemented, for example custom workflow processes or rollout actions.

Can we use AEM authoring with Edge Delivery Services for projects in production?
Yes, with the next AEM CS Sites release this feature will be made available to all customers. Existing projects can be migrated to Edge Delivery Services using the helix-importer utility to migrate the existing content to the new content model of AEM authoring with Edge Delivery Services. The frontend implementation must be rebuild using the core concepts of Edge Delivery Services projects.

---

[3] MSM will only work on page level. Component level inheritance is not yet supported.

Lesson 0 - Setting yourself up for the Lab

Your lab machine is already pre-loaded with several resources you will need:

- Google Chrome
- Visual Studio Code

In addition, you have a dedicated AEM Sandbox.

Your seat number is relevant for the exercise, e.g. exchange “01” (for seat no. 1) in this lab book with your seat number which you will find on your desk!

AEM CS Environment

- Go to url: https://bit.ly/l423-seats. A Google Spreadsheet will open. Find your seat and open the link to AEM in a new tab.
- Login with user: **L423+{xy}@adobeeventlab.com** (replace {xy} with your seat no). Your login should look like **L423+01@adobeeventlab.com** for user “01”.
- Password: **Adobe2024!**

GitHub

You will need to login to GitHub with your personal account. To grant yourself access to your repository, add your private GitHub username in the last column "GitHub Username" of the Google Spreadsheet you opened before. You will receive an E-Mail invitation, which you need to confirm by selecting "View Invitation" and accepting it.

If you don’t have a personal GitHub account, please ask a Teaching Assistant to help you log in, or create one – it is free of charge and only takes 2 minutes.

There is nothing else needed, your git cli is already per-configured. However, if you are asked for credentials when cloning your GitHub repository or pushing to it, use the following credentials: **l423a** with password **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**

Lesson 1 – Getting Started with AEM authoring with Edge Delivery Services

Objectives

1. Create your GitHub project and some content in AEM.
2. Get your project and AEM environment linked to each other.
3. Publish first content from AEM to Edge Delivery Services.

We already created a repository for each attendee upfront to save time. The following steps are meant for users that want to repeat the lessons or share them with co-workers.

You don't need to sign in to GitHub with your personal account. Your lab machine is set up to clone, push and pull from your repository.

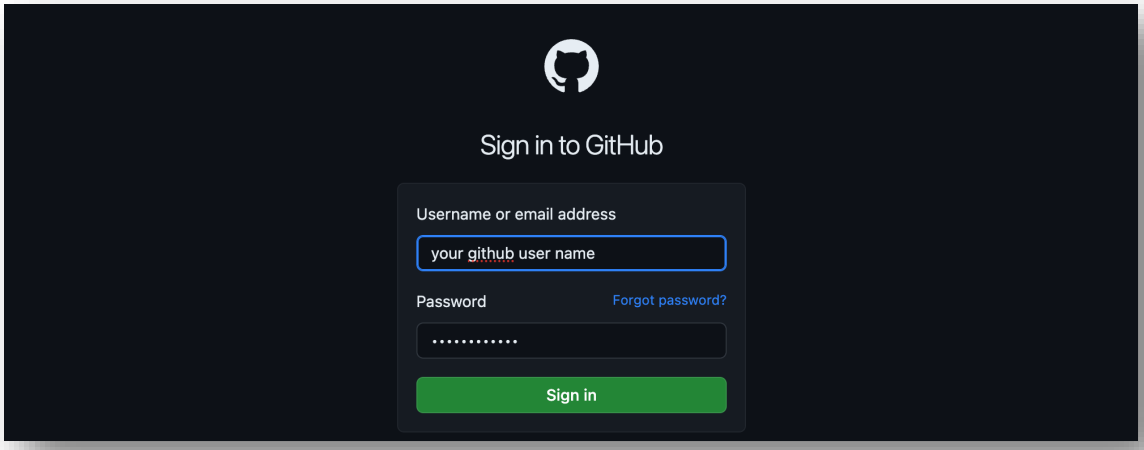
*Please open **https://bit.ly/l423-seats** to find the link to your repository and the link to your AEM author environment.*

Continue with the next steps of Link your Edge Delivery Services Project to AEM

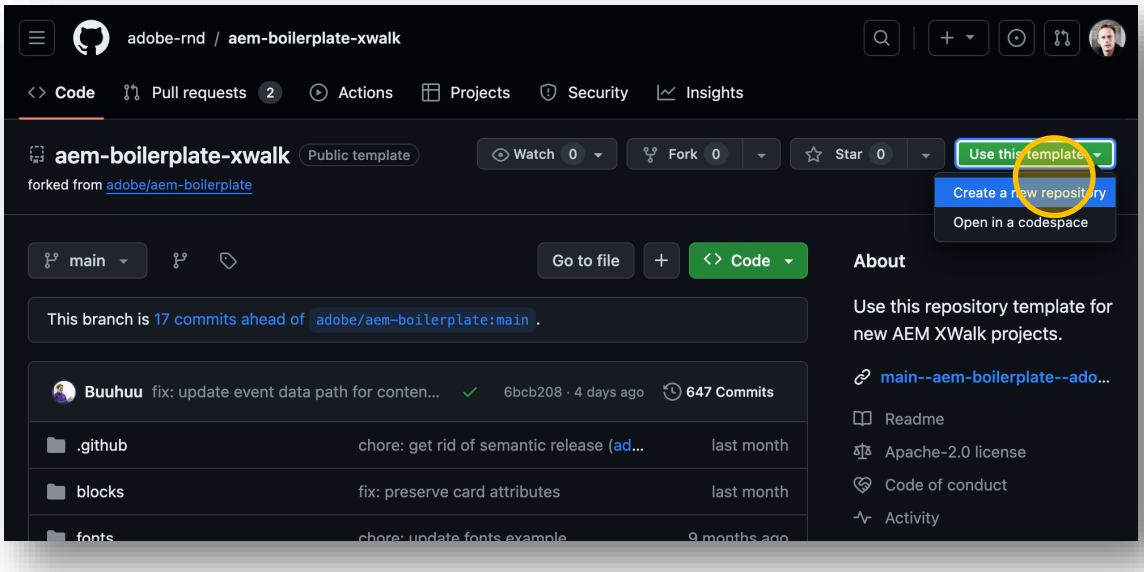
Create your Project

Independent from the content source, e.g. AEM, SharePoint or Google Drive, you will first setup your Edge Delivery Services project in GitHub. In a few minutes, you will be able to create, preview, and publish your own content, add styling, and new blocks. For more info, refer to https://www.aem.live/developer/tutorial.

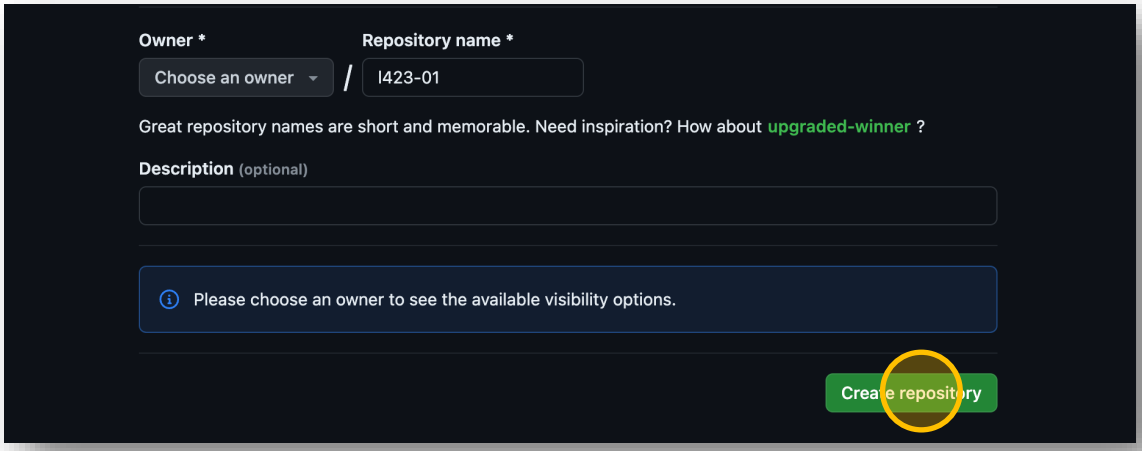
You will open the "AEM authoring with Edge Delivery Services" project template which is the starting point for projects and our lab. Navigate to GitHub, https://GitHub.com/adobe-rnd/aem-boilerplate-xwalk and sign in with your personal GitHub account.



Click "Use this template", and "Create a new repository" .



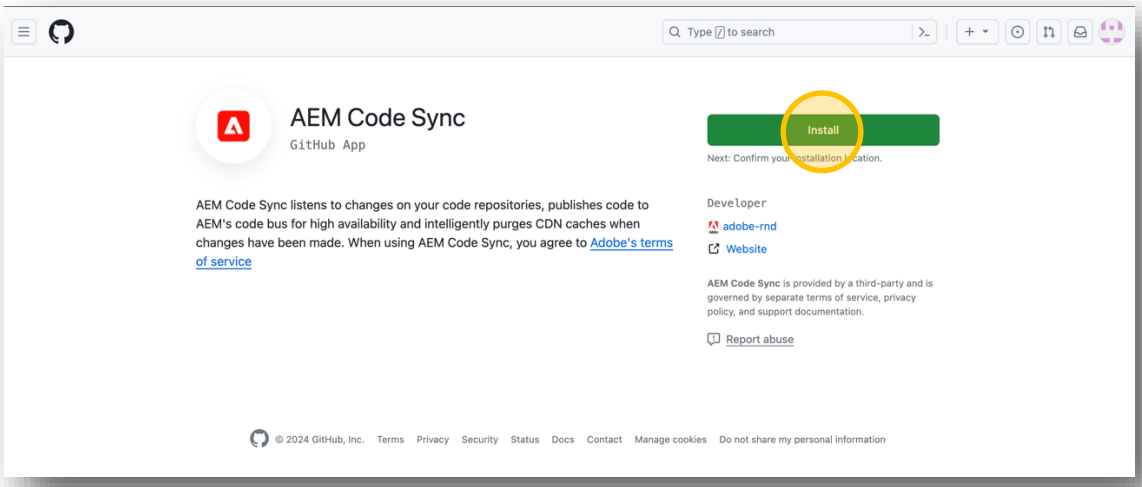
You need to choose an owner and a repository name.



a) Owner: select your own GitHub account (the one you logged in with)
b) Repository name: enter any name, e.g. "*l423-{xy}*" (**replace {xy} with your seat number!** e.g. for seat "01" use *l423-01*). Keep it short as the repository name and owner will be part of the the preview and live urls which are limited in the number of characters.
c) Click "Create repository".

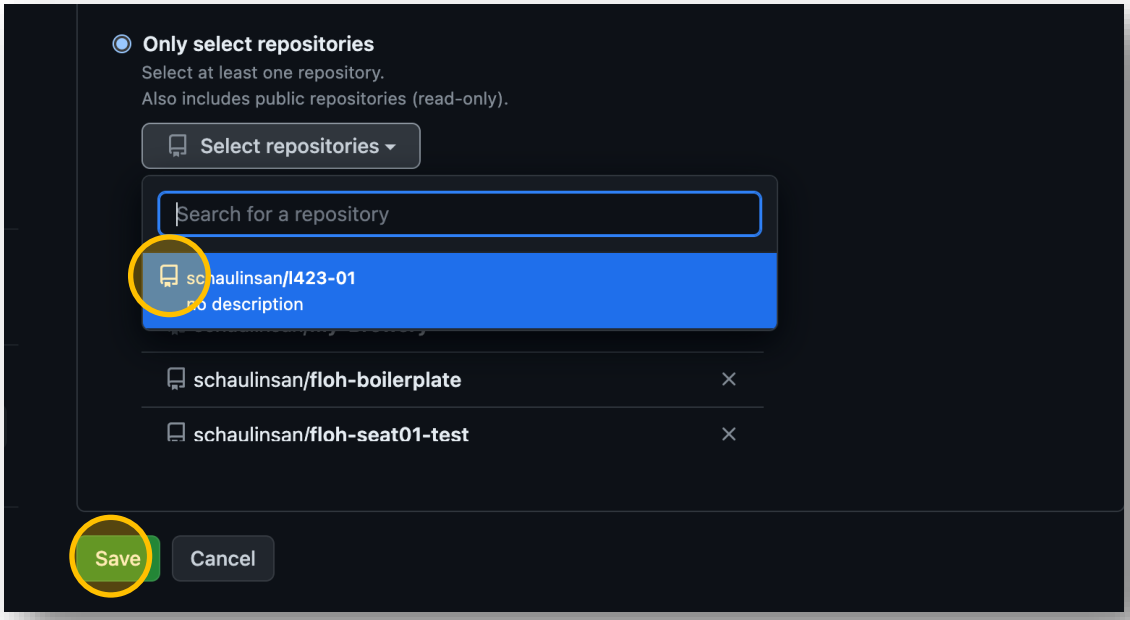
Next, we configure AEM Code Sync. AEM Code Sync listens to code changes, publishes them to the Edge Delivery Services code bus for and intelligently purges CDN caches when changes have been made.

Open a new browser tab and enter https://GitHub.com/apps/aem-code-sync, and click "Install" (or "Configure" if you installed it before).



If you are asked to select the Organization or User to configure AEM Code Sync for select the owner, you created your repository with and click "Configure" again.

Choose "Only selected repositories", and from the dropdown, select the repository you created. Then click "Install" or "Save". You can close this browser window now.


Only select repositories
Select at least one repository.
Also includes public repositories (read-only).
Select repositories
Search for a repository
schaulinsan/l423-01
no description
schaulinsan/floh-boilerplate
schaulinsan/floh-seat01-test
Save
Cancel

Link your Project to AEM

Now we can link our newly created project to AEM as content source. Return to your GitHub repository in your browser e.g. *https://github.com/<your account>/l423-{xy}*. Remember, replace {xy} with your seat number – the repository name you created above in steps before, e.g. *l423 -01.*

Copy the URL from your browser and open a Terminal. Clone your GitHub repository to your machine using:

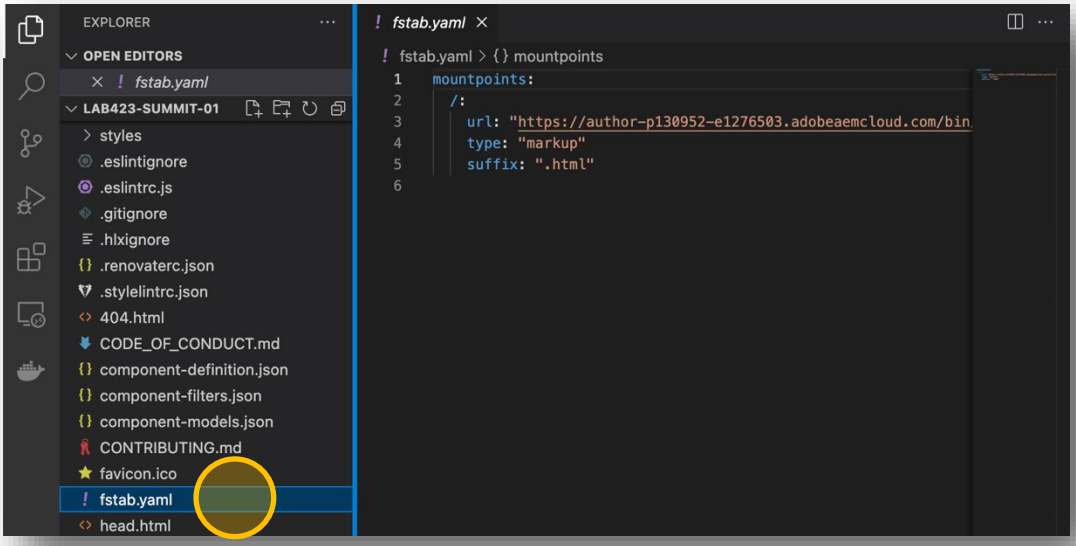
```
git clone https://github.com/<your account>/l423-01
```

Now open the cloned repository in Visual Studio Code using:

```
cd l423-01
code .
```

Where l423-01 is the name of your repository.

On the left hand side, in the Explorer find the "fstab.yaml" file and open it:



The fstab.yaml configures the link from your Edge Delivery Services project to the content of your content source. This could be a SharePoint or Google Drive for document-based authoring, or an AEM author instance for AEM authoring.

Update the mountpoint URL in fstab.yaml. Set the url to https://<aem-author>/bin/franklin.delivery/<owner>/<repository>/main with:

a) <aem-author> ("with the AEM CS author hostname, e.g. "***author-p130952-e1276503.adobeaemcloud.com***" in our example
b) <owner> with the GitHub organization or user the repository was created in, e.g. "***schaulinsan***" in our example.
c) <repository> with the name of the repository, e.g. "***l423-01***" in our example

Before:

fstab.yaml

```
mountpoints:
  /:
    url: "https://author-p15404-e146221-cmstg.adobeaemcloud.com/bin/franklin.delivery/adobe-rnd/aem-boilerplate-xwalk/main"
    type: "markup"
    suffix: ".html"
```

After (Example):

fstab.yaml

```
mountpoints:
  /:
    url: "https://author-p130952-e1276503.adobeaemcloud.com/bin/franklin.delivery/schaulinsan/l423-01/main"
    type: "markup"
    suffix: ".html"
```

If you copy/paste this configuration, be sure you change <aem-author>, <owner> and <repository> accordingly.

---

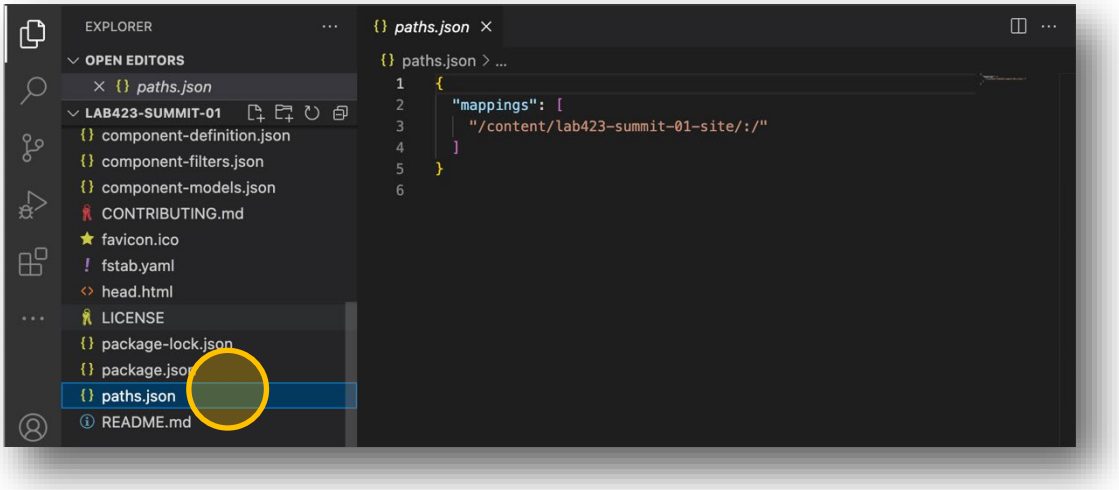
yaml files are case sensitive! *Also be aware of keeping the same indention/ spaces as in the original fstab.yaml file.*

Return to your Terminal and commit the changes and push them to GitHub:

```
git commit -a -m "feat: updated content source location"
git push
```

Next, we will update the path mapping of our project. An AEM environment can be used to create and maintain multiple sites. Usually, each site is stored in a path like */content/site* to physically separate content of one site from another. On the other hand, our Edge Delivery Services project is intended to serve exactly one site and hence a common use case is to remove these parts from the path to get to a public path that visitors will see in the browser.

In Visual Studio Code open the “paths.json” file:



Update the mappings to the AEM site you will later create in AEM CS. You can choose any name for your AEM CS site. For this lab, we recommend that you use the same naming convention as above which is:

a) replace “aem-boilerplate”
b) with “l423-{xy}” where {xy} is your seat number.
Example: “l423-01” for seat 01.

Before:

paths.json

```
{
  "mappings": [
    "/content/aem-boilerplate/:/"
  ]
}
```

After (Example for seat 01):

paths.json

```
{
  "mappings": [
    "/content/l423-01/:/"
  ]
}
```

Again, return to the Terminal, commit, and push your changes:

```
git commit -a -m "feat: updated path mapping"
git push
```

Summary

What you've done so far is

- you created your **Edge Delivery Services GitHub repository** based on the latest template – e.g. "*l423-01*".
- you configured **AEM Code Sync** for pushing code to the Edge Delivery Services code bus.
- you linked your project to your **AEM CS author environment** in **fstab.yaml**– e.g. "*/schaulinsan/l423-01/main*".
- you mapped your to-be-created AEM site's path to your public path **in paths.json** to – e.g. "*/content/l423-01/:/*"

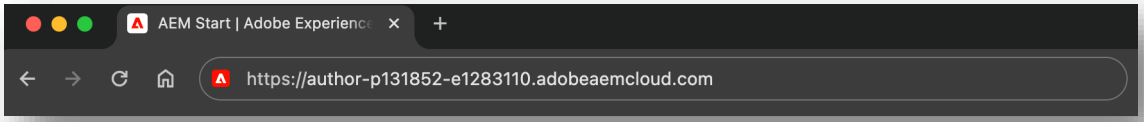
Important: Please note or write down

- your GitHub repository, e.g. https://github.com/schaulinsan/l423-01
- your AEM CS site name, e.g. *l423-01*

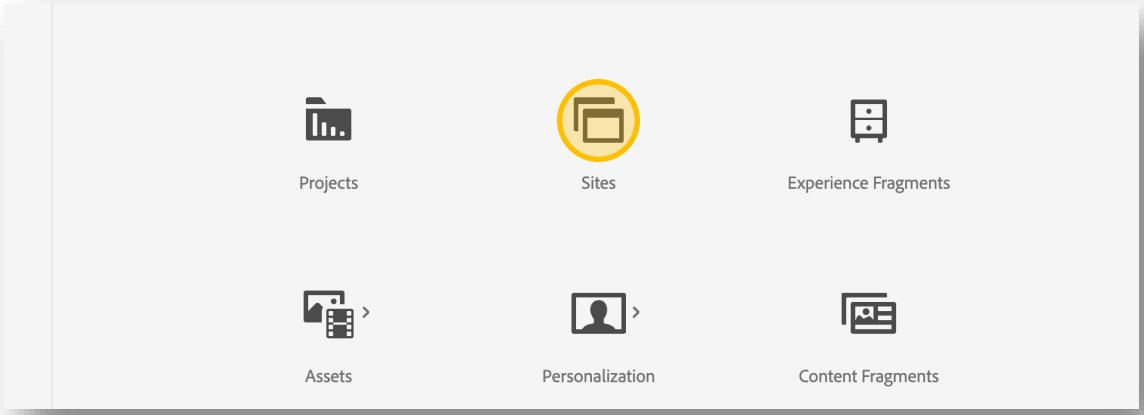
Create a new Site in AEM

We start with a simple site in AEM Sites as a Cloud Service, using a site template that contains some sample content for the boilerplate-like repository we created before. And then configure AEM to publish to Edge Delivery Services.

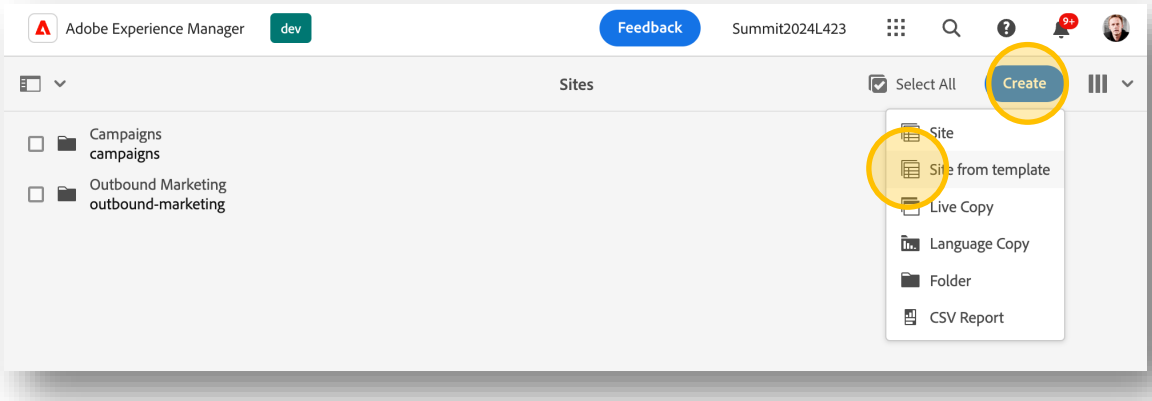
Go to AEM CS author instance, you find the link for your seat in the spreadsheet we opened before https://bit.ly/l423-seats. If you are not logged in, please follow the steps in *AEM CS Environment*.



Navigate to "Sites"



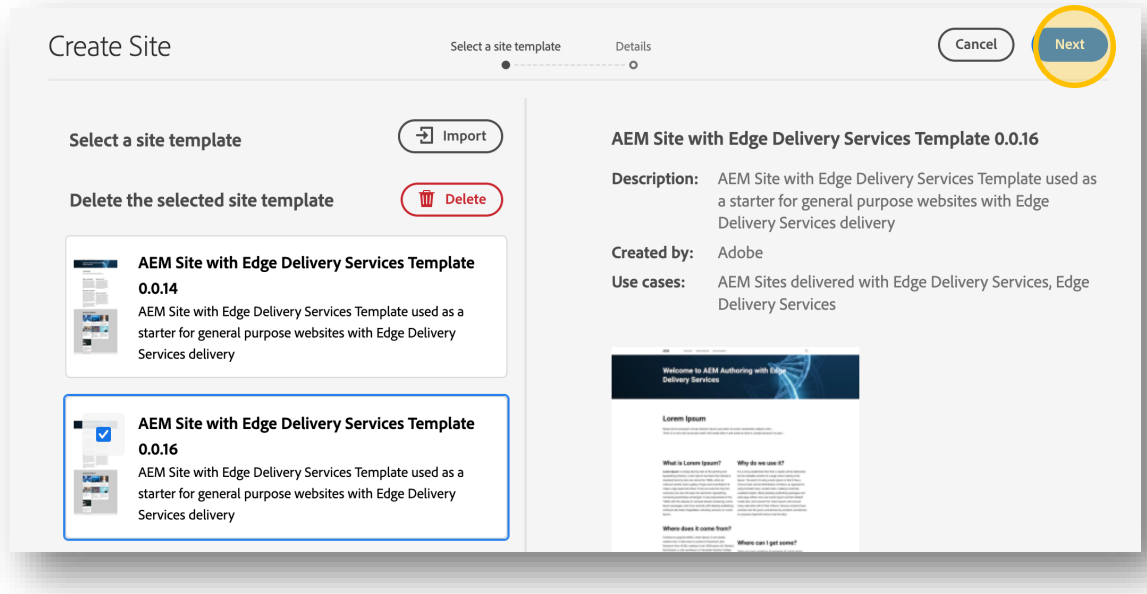
Click "Create" and select "Site from template".



You will see a list all AEM Quick Site Creation templates which got previously imported. We will use the template AEM Sites with Edge Delivery Services.

Select the latest template. On the right side, the details of the template will be shown.

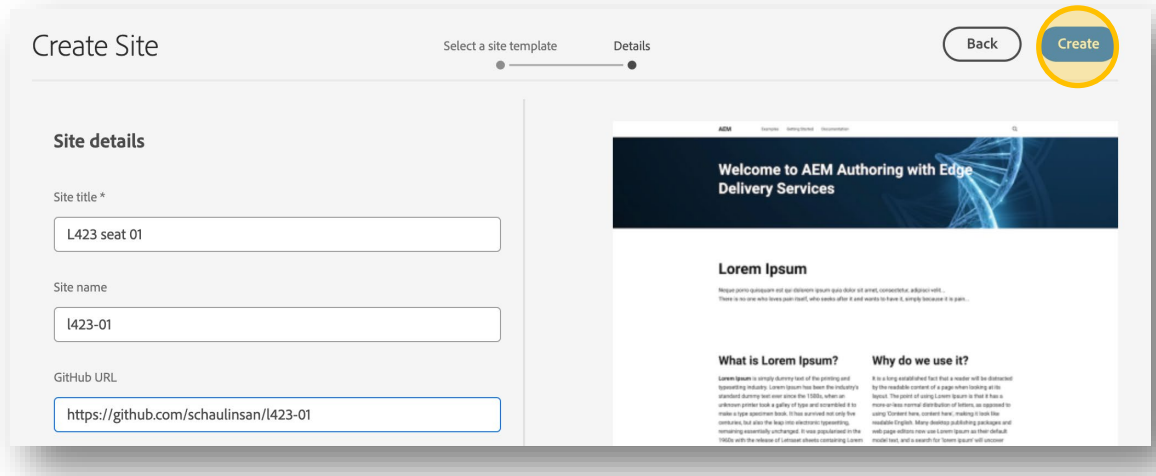
Click “Next”, top right.



Enter a site title, site name and the GitHub URL of the project we previously created.

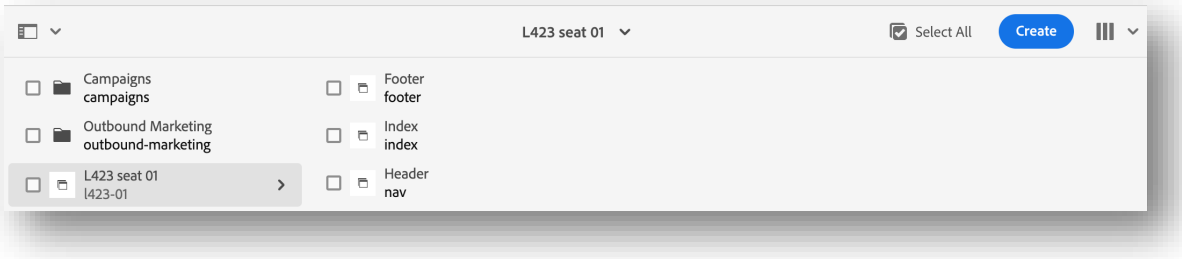
a. Site title: <any text> (e.g. “*L423 seat 01*”)
b. Site name: <site>
   (**important**: This must be the same site name as you previously mapped in the “paths.json” file – in our sample, it’s *l423-01* where “01” represents your seat number)
c. Github URL: https://github.com/<owner>/<repository>
   (**important**: This must be the same GitHub repository name as you mounted in “fstab.yaml” – in our sample, it’s *https://GitHub.com/schaulinsan/l423- 01*)

Click “Create”

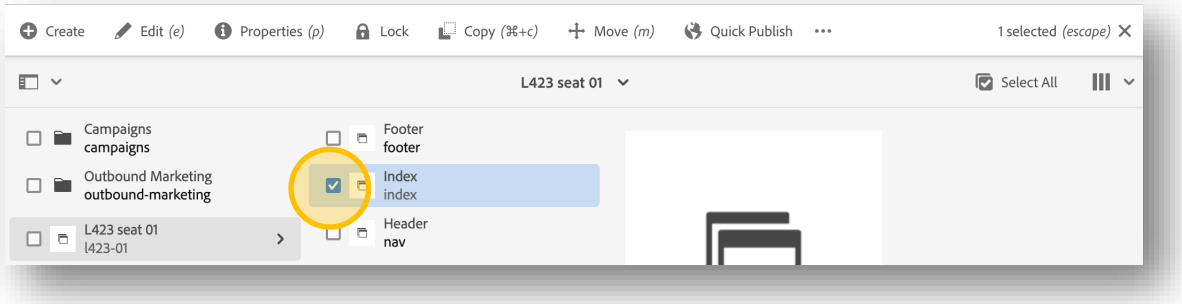


This starts a background job to create your AEM CS site, using the selected template, creating your first AEM CS pages with some sample content.

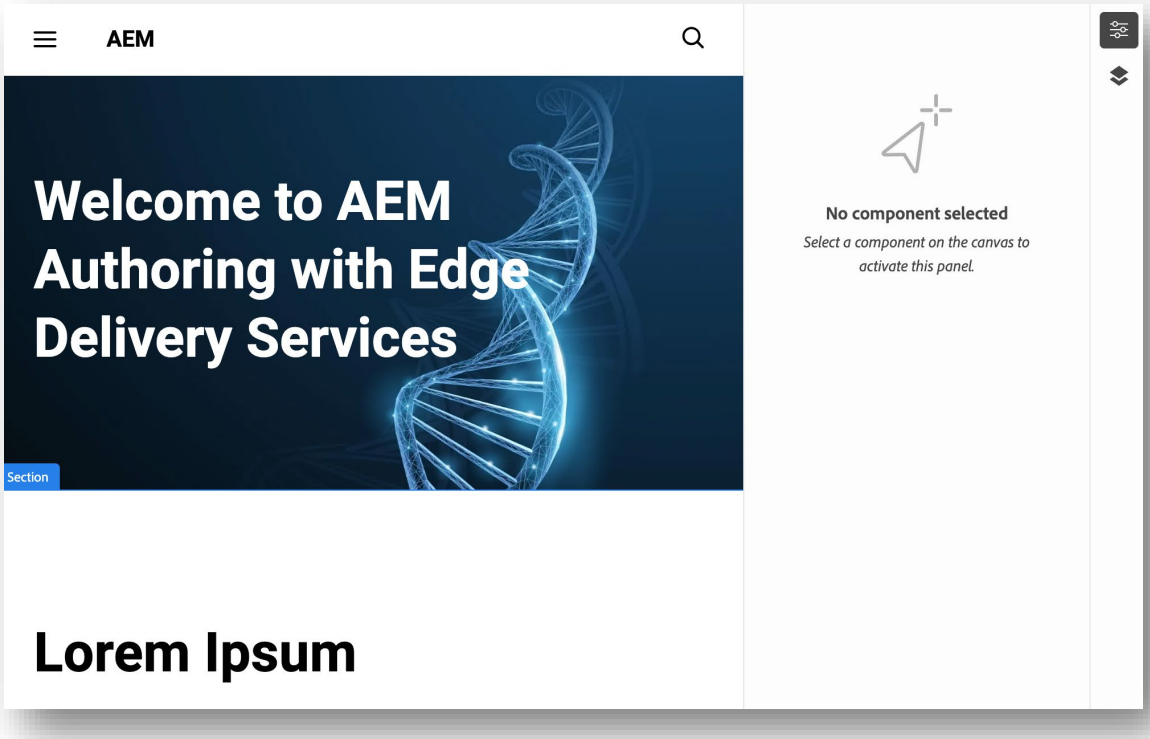
You will return to AEM Sites where you will see your newly created site. If not wait a few seconds and reload the page. The site is created in the background, which may take some time.



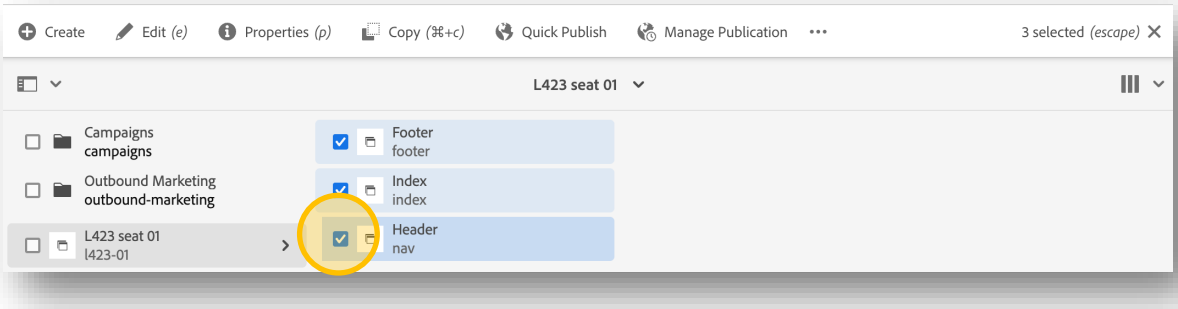
Navigate to your site by clicking on the name. This will list the sample content of the template. Select the page “index” by clicking the checkbox and click “Edit” in the toolbar that appears.



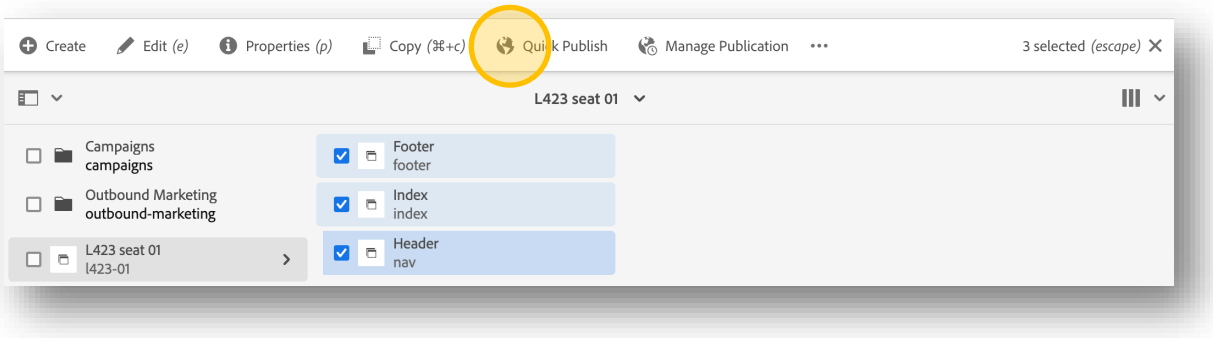
The page opens in a new browser tab in **Universal Editor**. It is fully editable but let’s leave it for now – we will experiment with Universal Editor later.



Close this browser window and return to AEM Sites. Select all the 3 pages of your site “Footer”, “Index”, “Header”.



And publish them by clicking “Quick Publish”. Confirm the prompt.



It will take a few seconds to publish these 3 pages to Edge Delivery Services.

To experience your new AEM Site with Edge Delivery Services open a new tab and navigate to your projects preview or live URL. For more information how about these URLs, please have a look at documentation about the anatomy of a project .

Each code branch of your project has a preview and live URL, either serving previewed content or published content. When publishing content from AEM it will first preview and immediately afterwards publish the content to Edge Delivery Services. So, both the preview and live URL will serve the same content.

Preview URL: https://<branch>--<repo>--<owner>.hlx.page
Live URL: https://<branch>--<repo>--<owner>.hlx.live

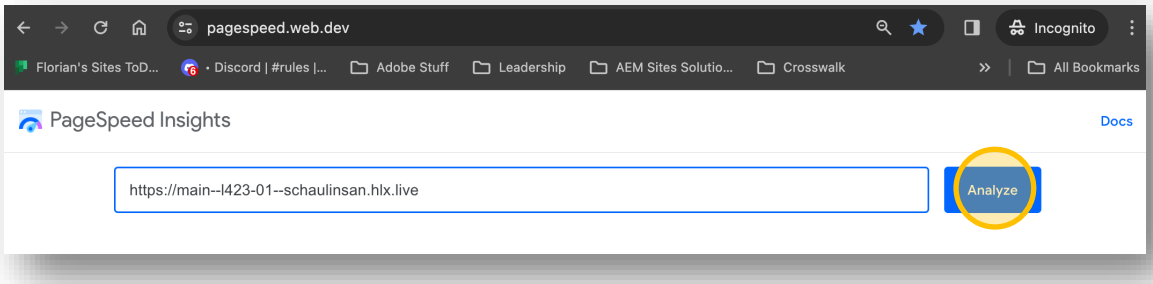
For our example this may be: https://main--l423-01--schaulinsan.hlx.live

---

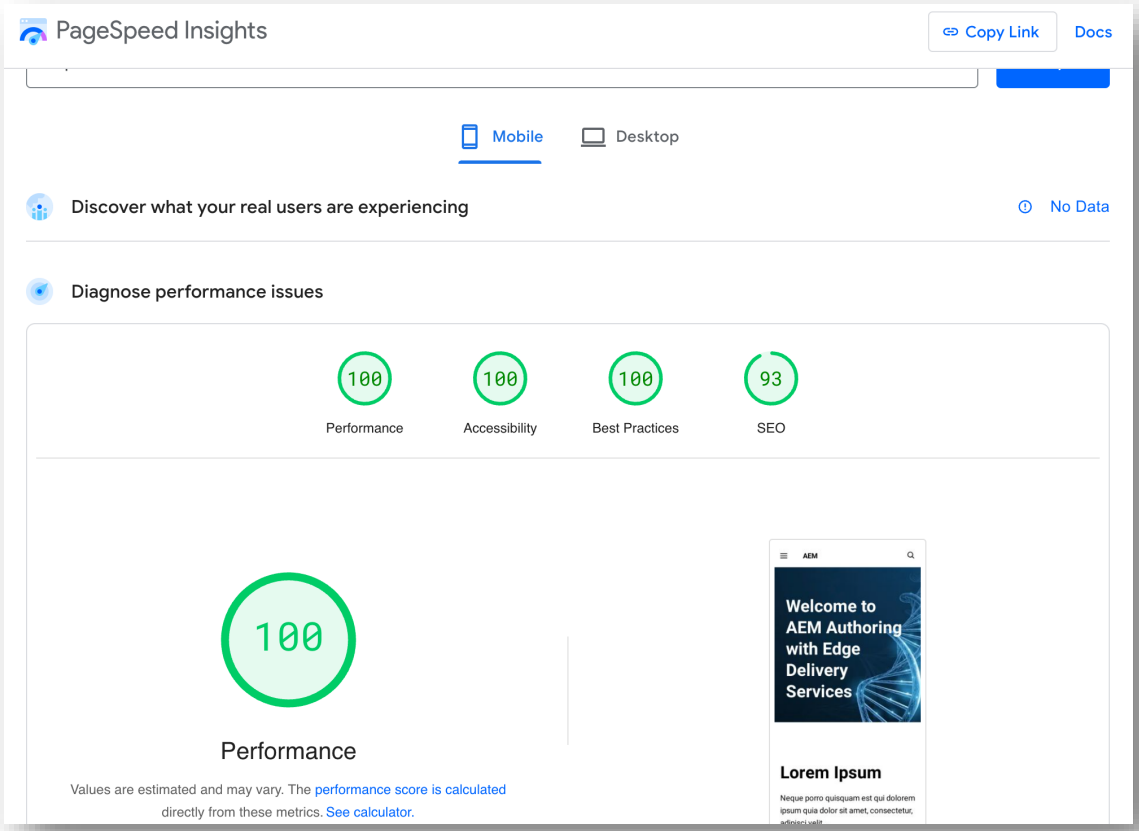
If you use AEM CS publish to preview, the content will be previewed only and will only be served from your projects preview URL.

You can now verify the performance of your new Edge Delivery Services page, e.g. using PageSpeed Insights. With Edge Delivery Services, we attempt 100 LHS and green Core Web Vitals.

Open a new browser window and enter https://pagespeed.web.dev/, paste the URL of your Edge Delivery Services site from your browser's location bar and hit "Analyze".



The report it produces should look like the screenshot below.



Summary

Great job!

What you've done in this last chapter was

- with AEM as a Cloud Services Sites you created your first site, using our latest AEM authoring with Edge Delivery Services template.
- you connected AEM CS with your GitHub repository.
- you experienced Universal Editor authoring.
- you instantly published your page (fast content velocity)
- and you experienced the outstanding performance of your page (fast experience)

Next:

- Get to know Universal Editor – the new authoring experience.

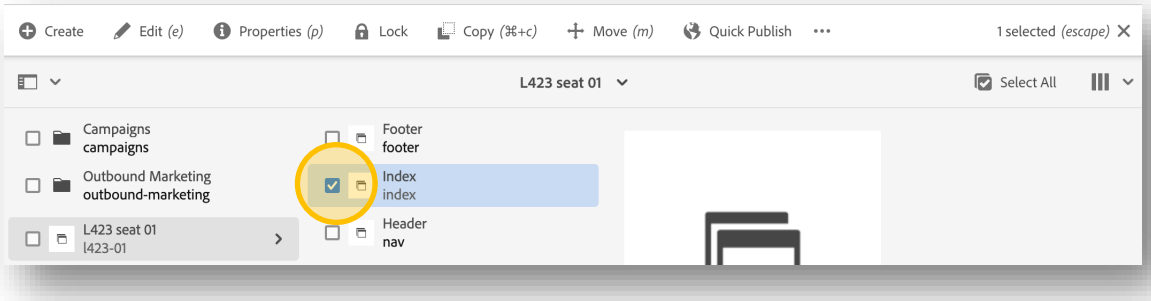
Universal Editor Authoring

You will find public documentation on Experience League, for example Universal Editor Introduction; specific to AEM CS, see Getting Started with the Universal Editor in AEM, and Universal Editor Overview for AEM Developers.

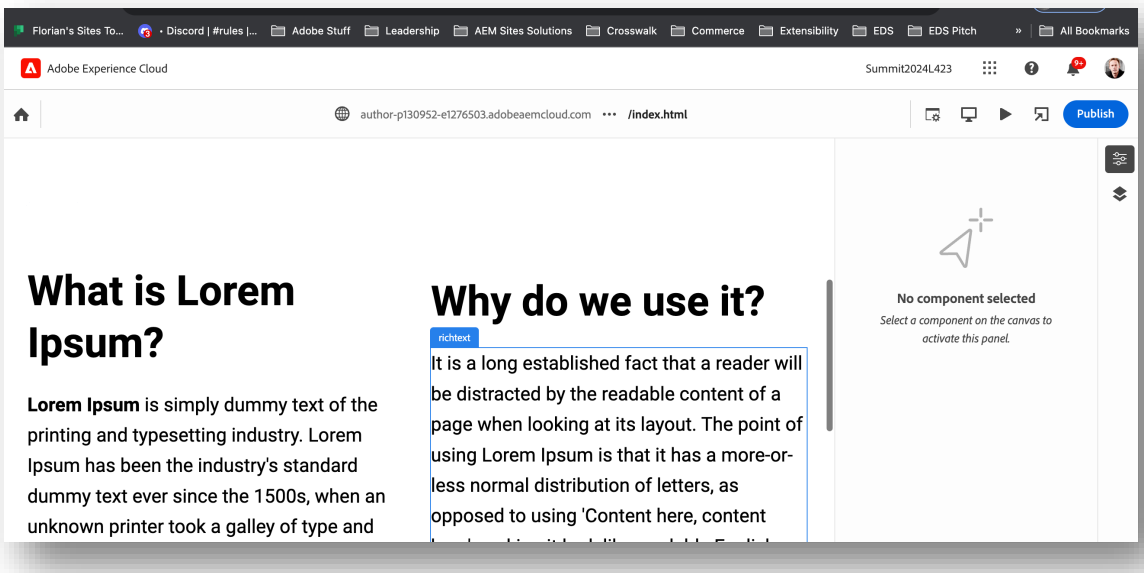
Universal Editor enables what-you-see-is-what-you-get (WYSIWYG) editing of headful and headless sites. We will focus on its usage for creating experiences with AEM authoring with Edge Delivery Services. By decoupling the content editing experience from the content delivery tier, the editor becomes truly universal and flexible allowing the content author to create exceptional experiences and increase content velocity.

Let's do some basic authoring with Universal Editor.

Go back to your AEM CS Sites environment and open the Index page for editing as we did before.

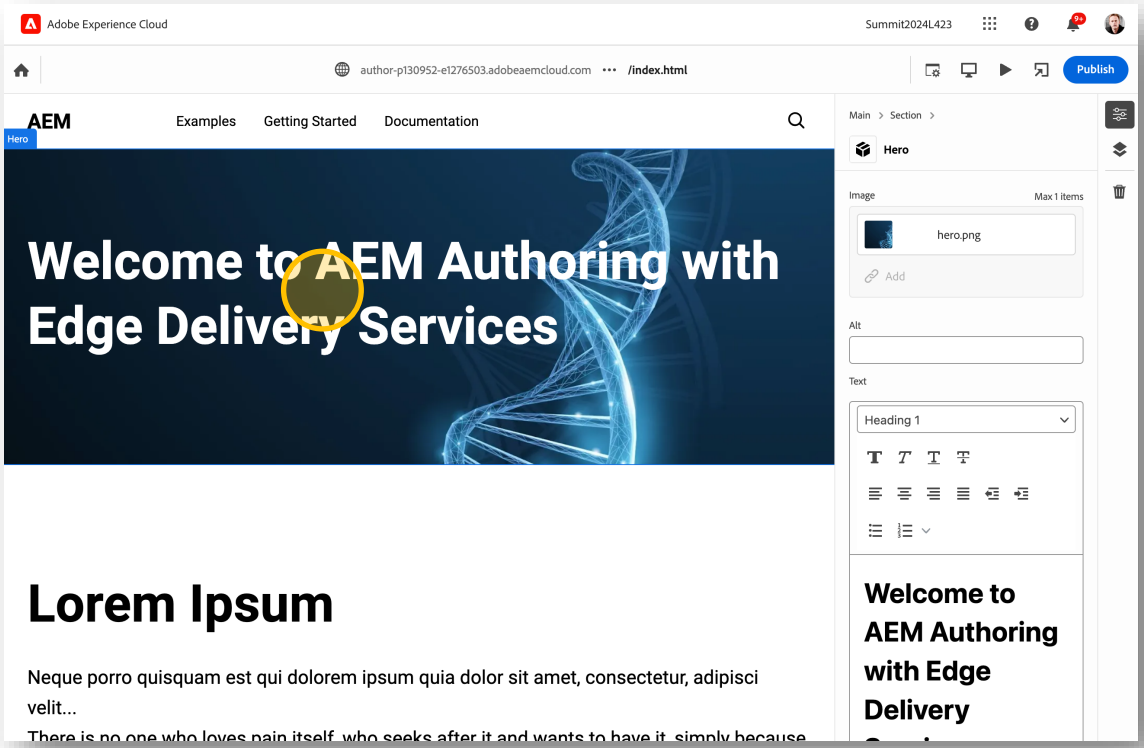


Scroll up and down on the page.



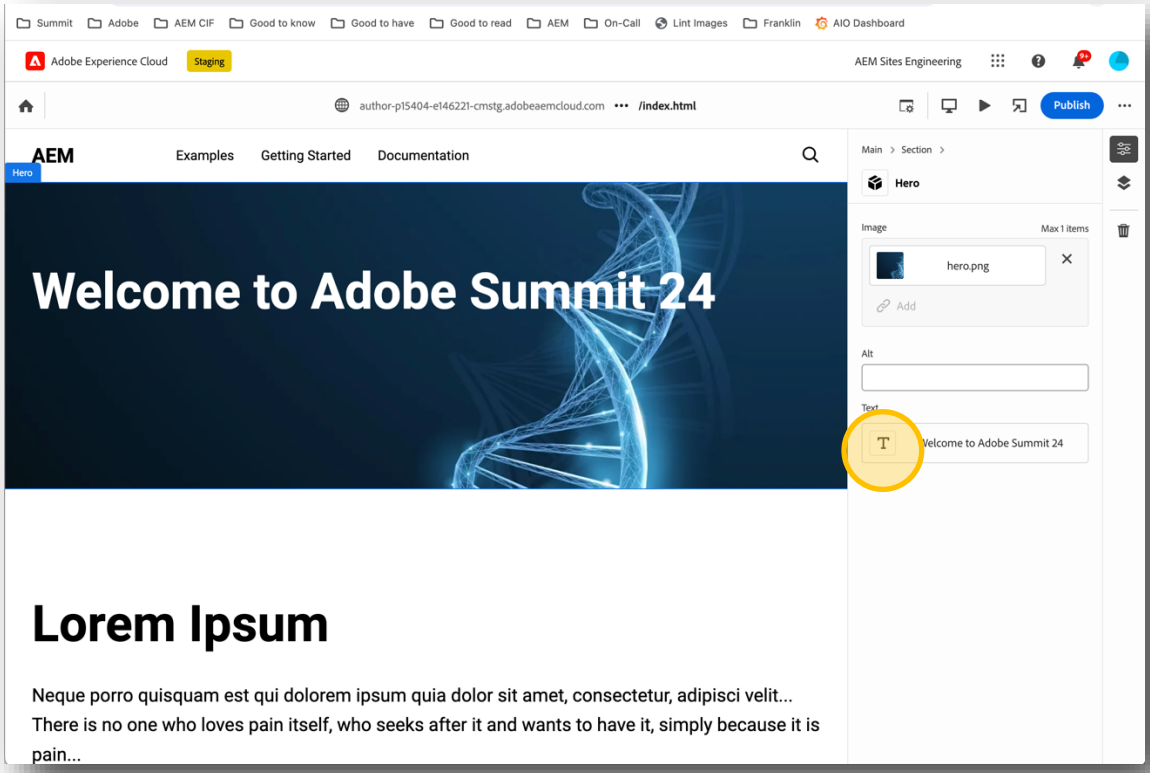
Hover over some content, you will see different content blocks, e.g. richtext. Title, column, hero etc..

Click on the hero banner on the top of the page.

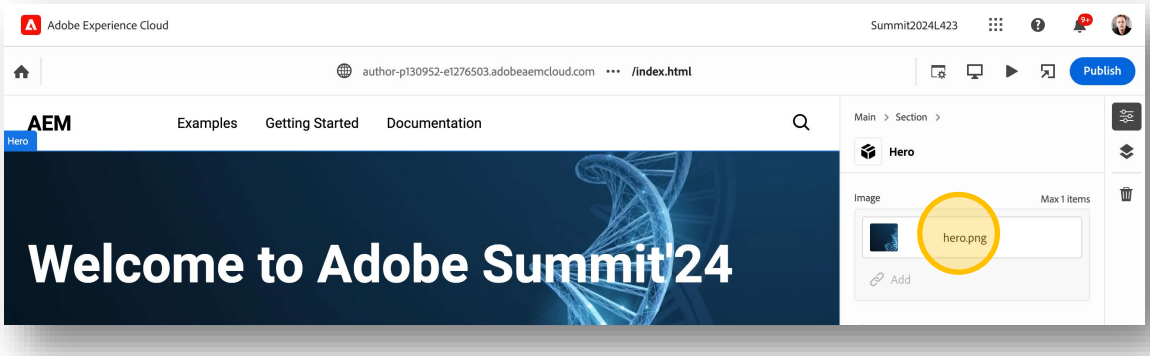


This opens the “Properties Rail” on the right-hand side, showing the properties of the selected “hero” block.

Change hero content to “Welcome to Adobe Summit’24”

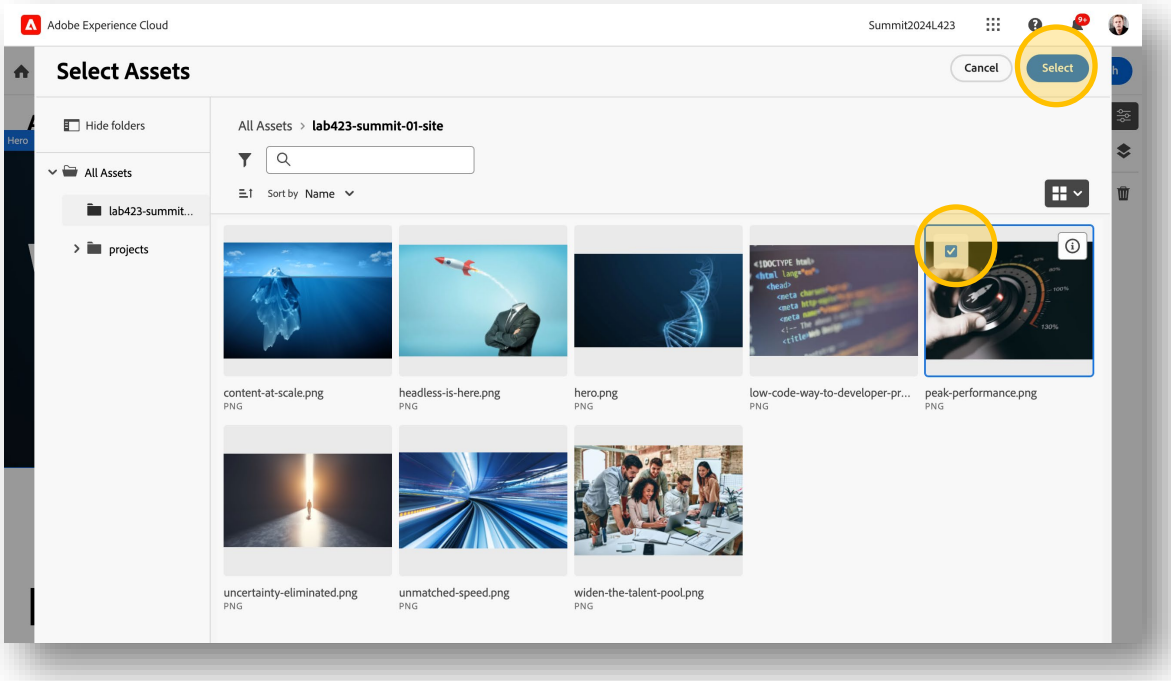


Click on the Image “hero.png”



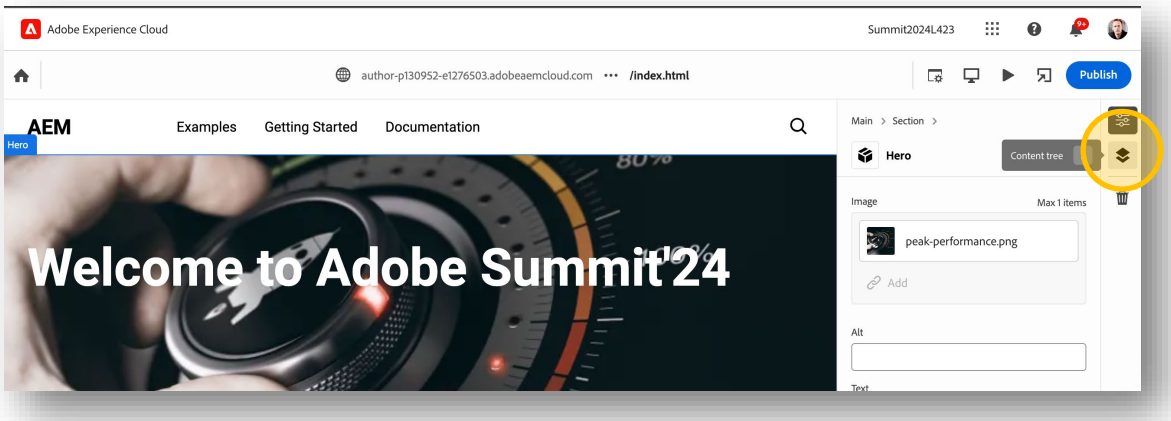
The Asset picker opens, showing all assets which got previously uploaded to your AEM environment. The hero image “hero.png” is pre-selected.

Change the hero image by selecting image “peak-performance.png”

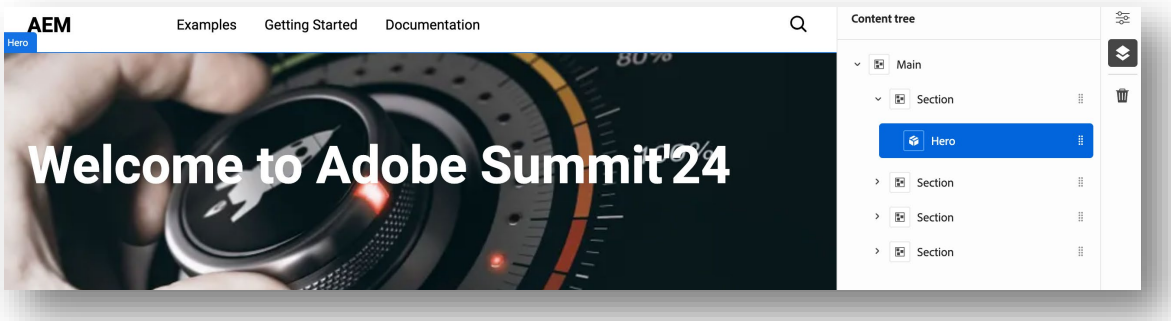


Then click “Select”. You will be directed back to Universal Editor with the hero image updated.

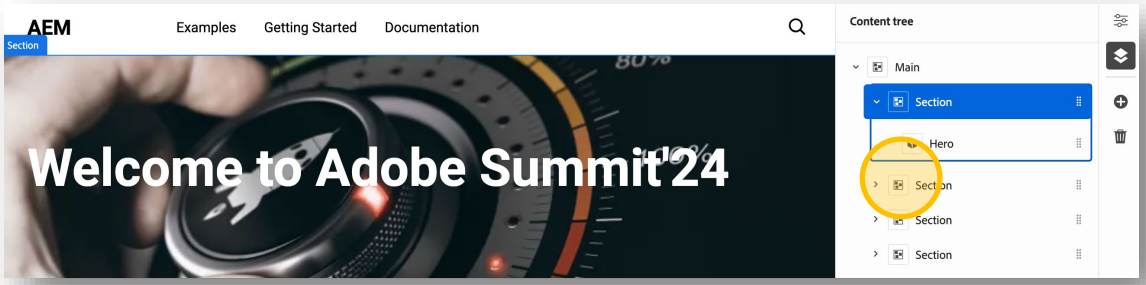
Click the “Content Tree” icon to open the content structure of the page in the rail.



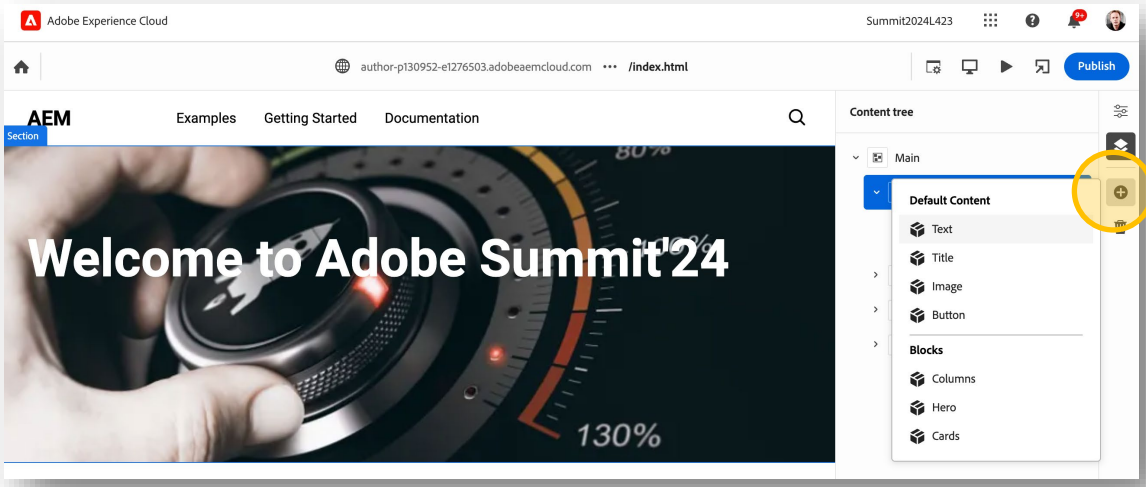
The content tree visualizes how the page is structured in sections, default content and blocks.



Select the second section on the page, the one that comes after the “Hero”.

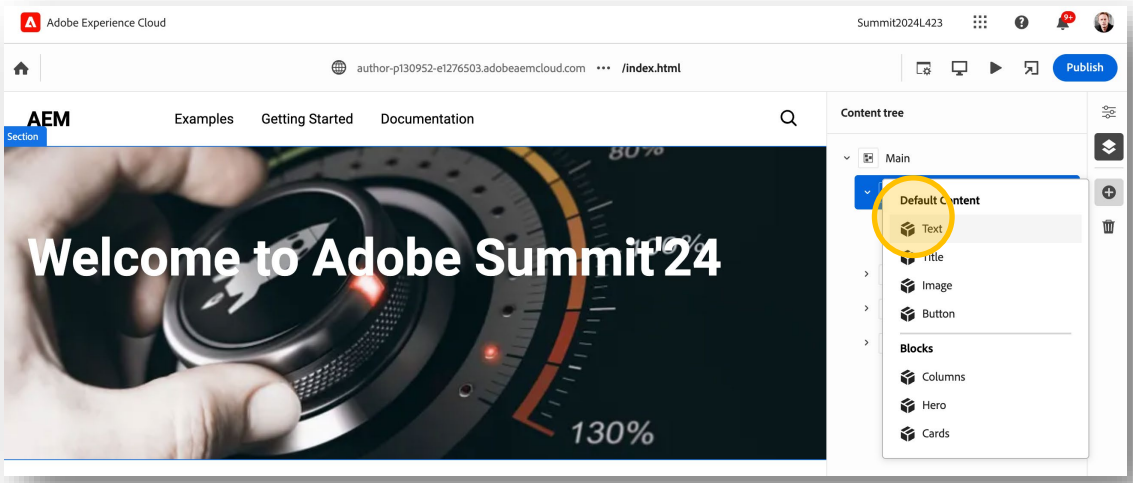


Click on “+” icon which indicates that new content can be added to the section.

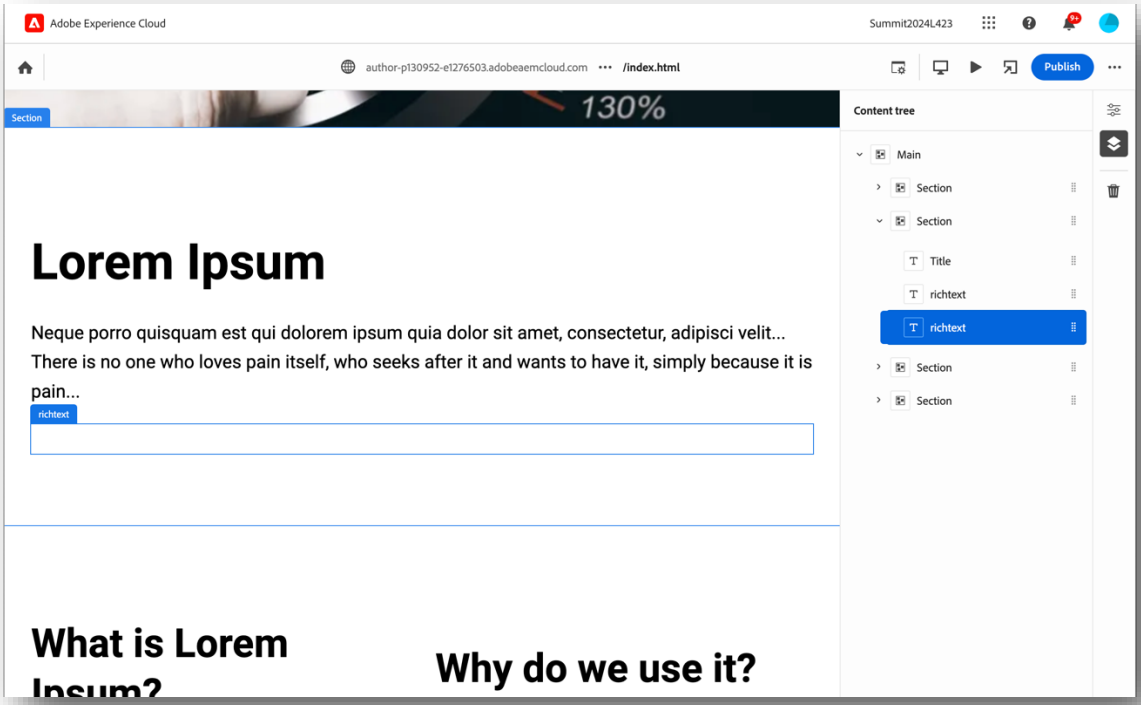


An overlay is shown which list all components that can be added to a section on this site. These components are loaded from your GitHub project – the one you initially created.

Select the “Text” component.



This adds the component “text” at the bottom of the section you selected.

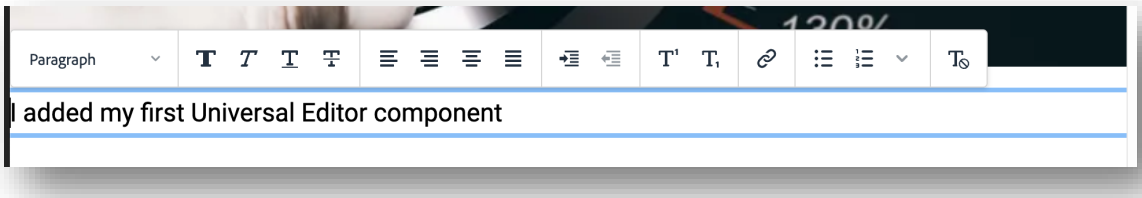


Note that

a) In the component tree, a new **component "Text"** shows up below the other two components in the same section.
b) If selected, an empty **text** is highlighted in the canvas

Doubleclick on the editable richtext. The in-context richtext editor opens.

Enter "I added my first Universal Editor component", then click outside of the editor or press "Tab" to blur the focus. This will automatically persist the content and re-render it server-side as well as updating it client side.



The content is rendered in the editor the same way as if it would be published. You can toggle off the edit overlays of the canvas and navigate the page as an end user would do.

Click preview again from the Universal Editor to update your preview and live URL with the changed content as well.

Take your time to inspect the other components on the page. Note that every element is by default in-context editable. This is a major advantage when comparing the new Universal Editor authoring experience with Page Editor or even document-based authoring.

Summary

What you've done in this last chapter was

- Get to know WYSIWYG authoring with Universal Editor
- Adding components to a page and editing content
- Preview the page

Next:

- Add additional components to your Edge Delivery project by adding a new component definition and model for Universal Editor authoring.

Lesson 2 – Developing for AEM authoring with Edge Delivery Services

Objective:

1. Understand how fast rapid development with EDS is.
2. Understand how code changes can directly be made available to authors.
3. Understand the concepts of setting authoring guard rails.

The process of creating a new block for AEM authoring consists of two steps. First, we have to define the component and a model for it. After that step authors can use the component to create content already. In a second step, we create some sample content and use the AEM simulator cli to implement the block's decoration and style.

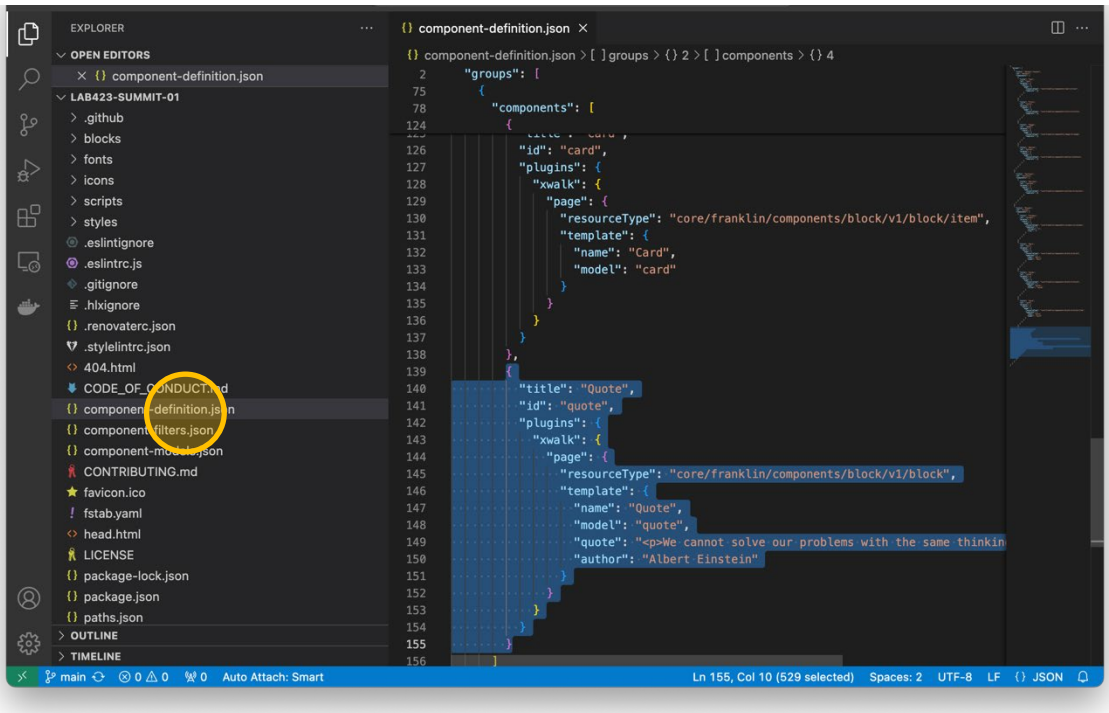
You can find all the files created / modified in this lesson in on your lab machine. Navigate to the folder on your desktop and open the L423 folder, lesson 2.

Component Definition and Model

For doing this, we will add a component "Quote" to

a) component-definition.json
b) component-models.json
c) component-filters.json

Return to Visual Studio Code and open the "component-definition.json" file.

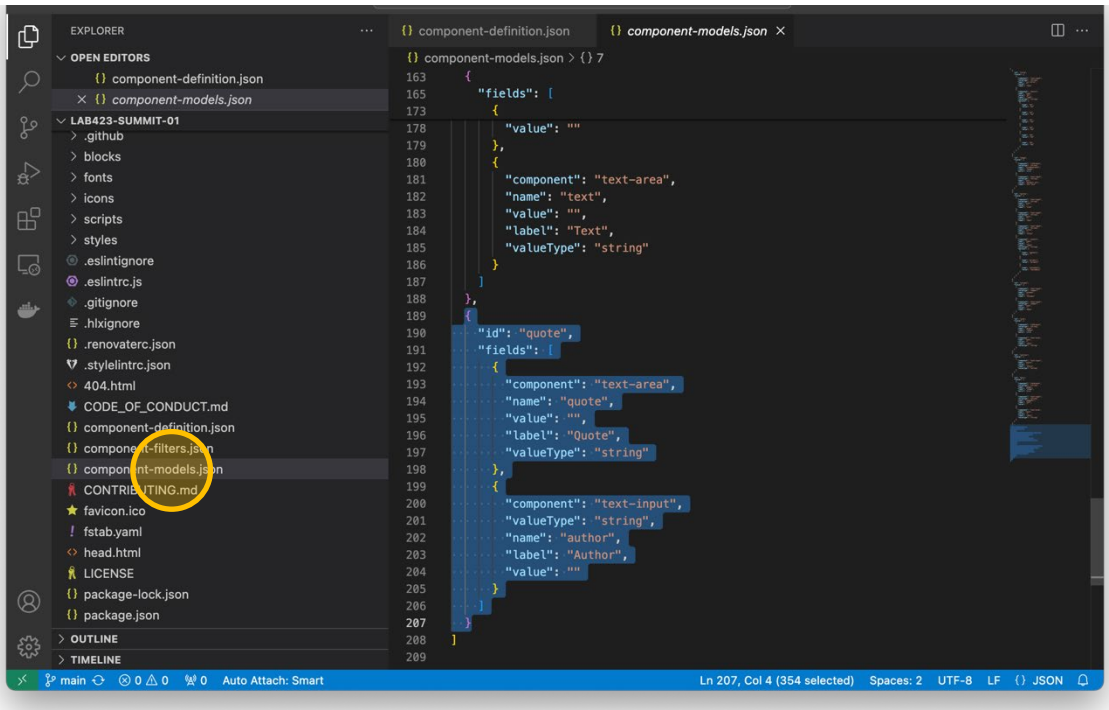


Add the following component definition after line 138. You can copy and paste the code as is. Please make sure to add a comma between the closing curly bracket and the new opening curly bracket.

component-definition.json

```
{
  "title": "Quote",
  "id": "quote",
  "plugins": {
    "xwalk": {
      "page": {
        "resourceType": "core/franklin/components/block/v1/block",
        "template": {
          "name": "Quote",
          "model": "quote",
          "quote": "<p>We cannot solve our problems with the same thinking we used to create
them.</p>",
          "author": "Albert Einstein"
        }
      }
    }
  }
}
```

Next, open the “component-models.json” file

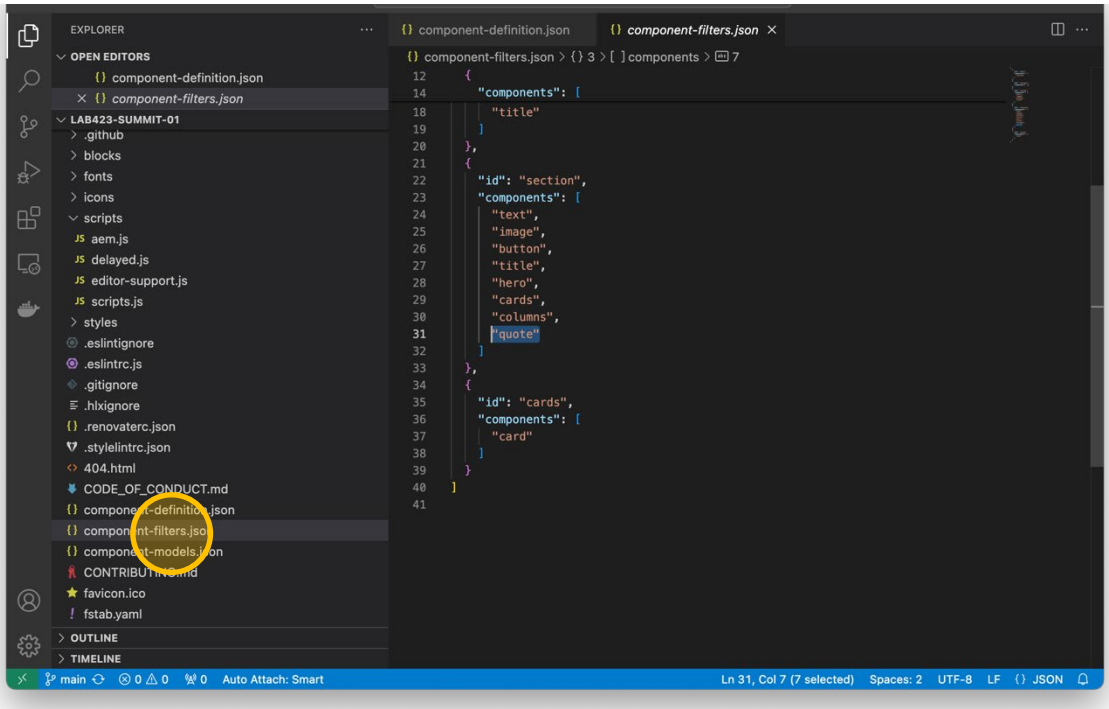


Add the following component model after line 188. You can copy and paste the code as is. Please make sure to add a comma between the closing curly bracket and the new opening curly bracket.

component-models.json

```
{
  "id": "quote",
  "fields": [
    {
      "component": "richtext",
      "name": "quote",
      "value": "",
      "label": "Quote",
      "valueType": "string"
    },
    {
      "component": "text",
      "valueType": "string",
      "name": "author",
      "label": "Author",
      "value": ""
    }
  ]
}
```

Lastly, open the “component-filter.json” file and add the new “quote” component to the list of allowed components for the “section”, by adding it after line 30. Make sure to add a comma at the end of line 30.

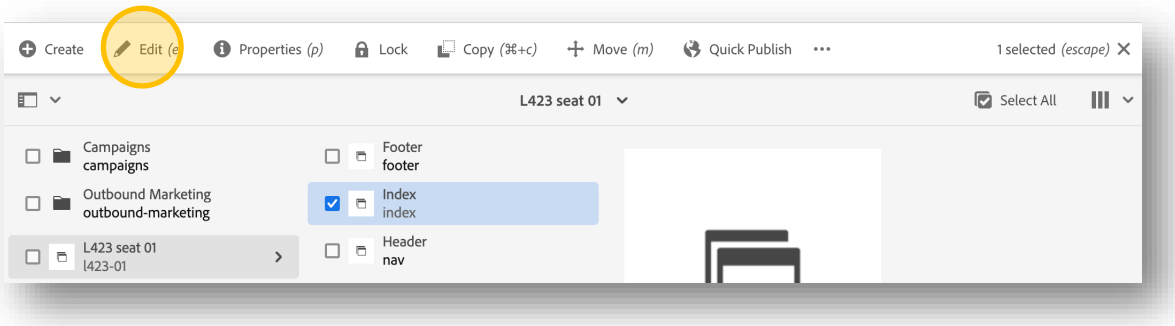


After you saved your changes, return to your Terminal to commit and push your changes to GitHub.

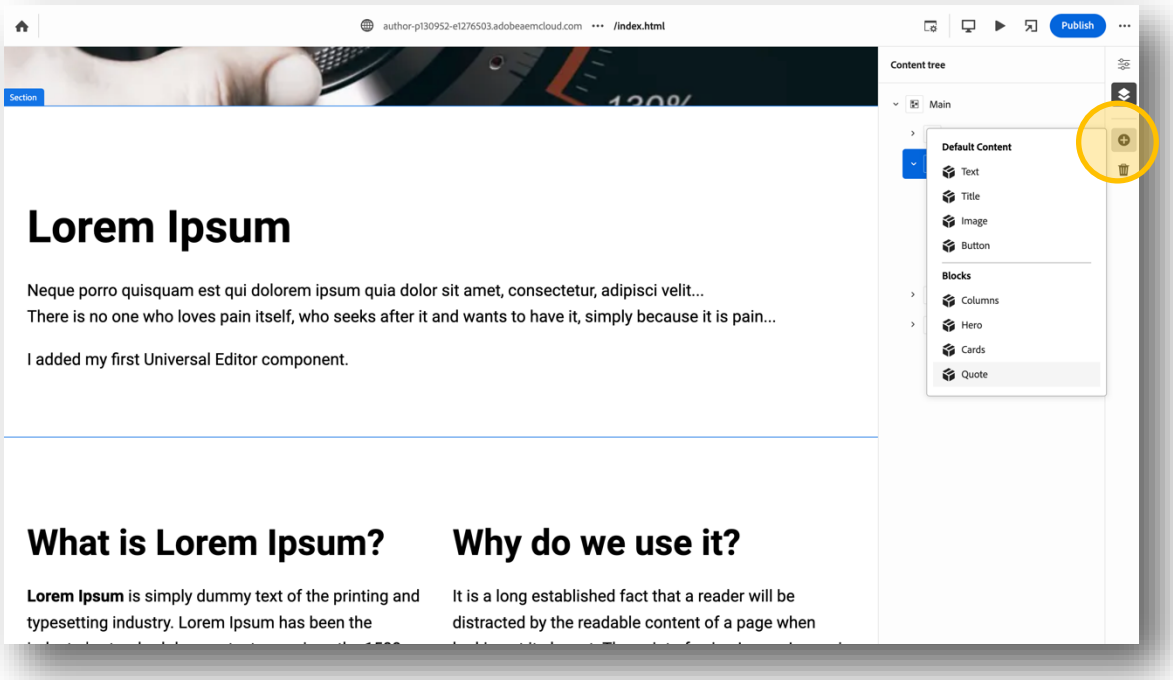
Usually, we would do the following changes on a feature branch and get a review before merging it into main. For simplicity we will directly commit and push our changes to the main branch.

```
git commit -a -m "feat: add quote component"
git push
```

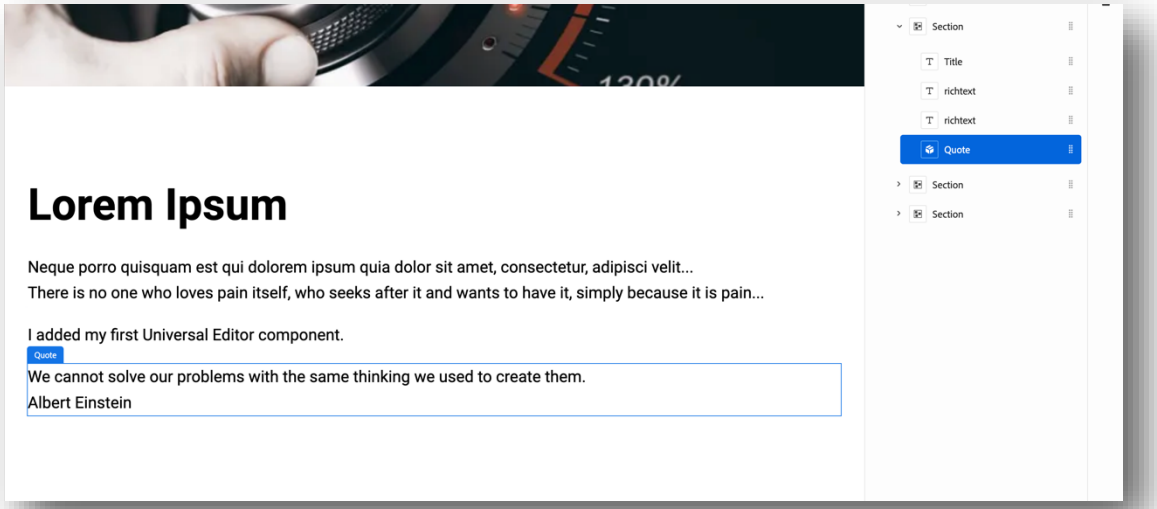
Done! You just added a new component “Quote” to your Edge Delivery Services site. Let’s verify if everything works well in Universal Editor. For this, let’s return to Universal Editor. From AEM Sites select the Index page and click “Edit”.



In Universal Editor select the second section, the one below the hero where you previously created your text component in. Click on the “+” icon to add a new component to that section and select the Quote component.



The Quote block will be added to the section and the section will be reloaded. Afterwards you will see the block rendered below the richtext, with some default content we defined earlier in the “component-definitions.json” file.



There are no block-specific styles applied, as we did not yet create the block.js (to decorate the DOM) and block.css (to style it).

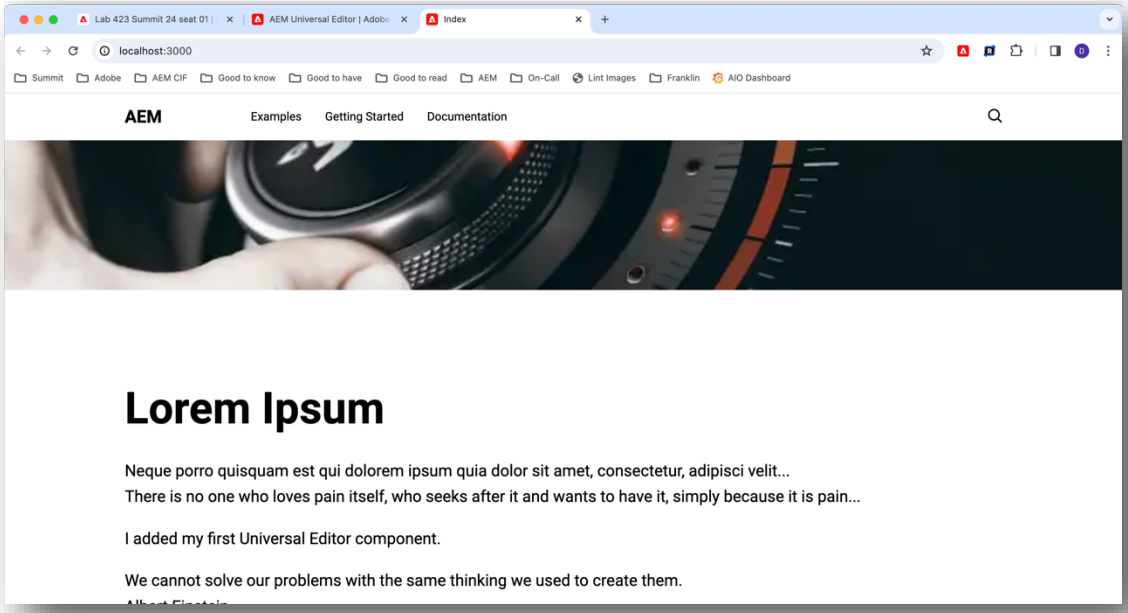
Anyway, publish the page so that we can use the content already for the AEM simulator.

Implement Block Logic and Style

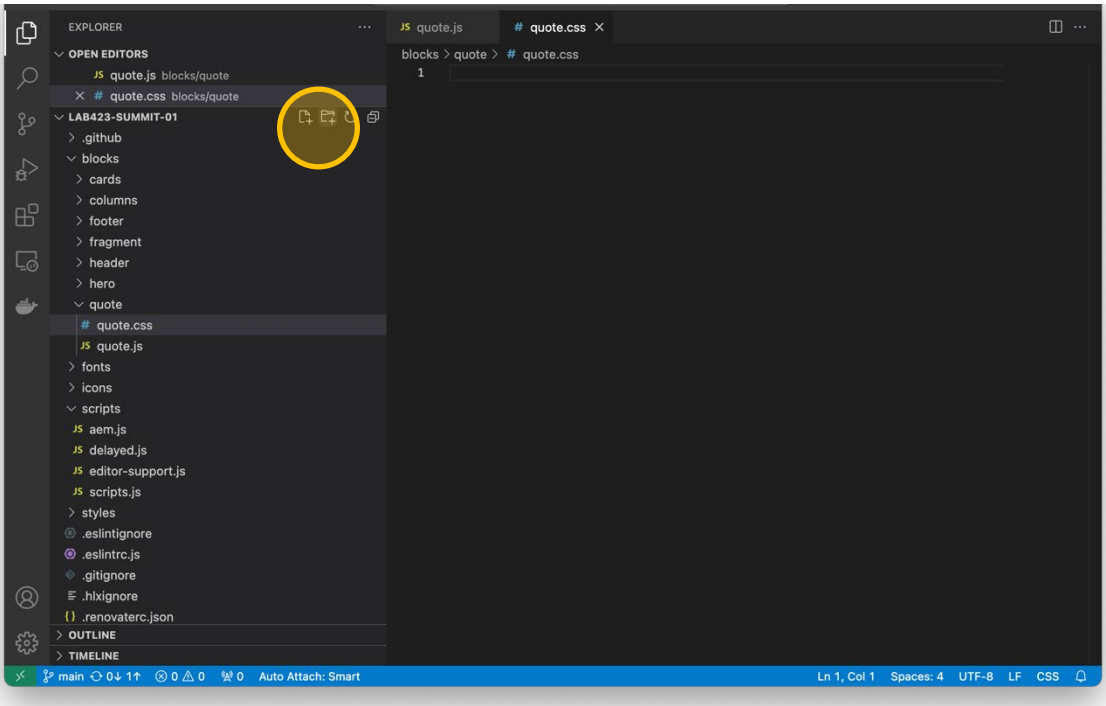
Once published, return to your Terminal, open a new Tab (CMD+T), and run

```
aem up
```

The browser will open with your site being loaded from the AEM simulator on http://localhost:3000. That pulls the code from your local machine and the content from your preview URL. You should see your Quote content on the page. If not try publishing it again and reload the AEM simulator page after a few seconds.



Return to Visual Studio Code and open the “blocks” folder. Create a new folder “quote” and inside that folder two new files “quote.js” and “quote.css”.



Paste the following code in the “quote.js” file. It implements some decoration of the DOM, that wraps the quote in a `<blockquote>` element, instead of the `<div>` rendered server-side. Save the file. Your browser should reload automatically, and your content should have changed slightly, due to the different markup in combination with the boilerplates default styles – if the browser does not reload, reload manually.

quote.js

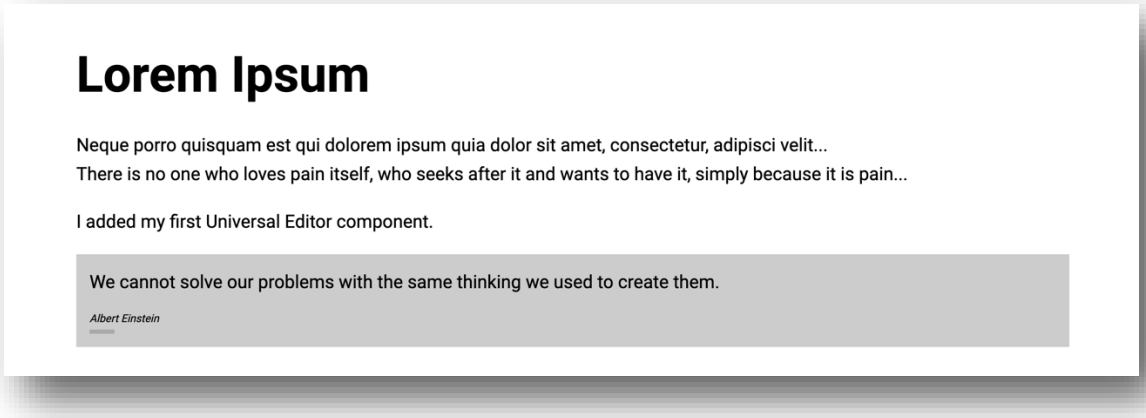
```
import { moveInstrumentation } from '../../scripts/scripts.js';

export default function decorate(block) {
  const [quoteWrapper] = block.children;

  // get the paragraph and its parent
  const par = quoteWrapper.querySelector('p');
  if (par) {
    const parWrapper = par.parentElement;
    // create a new <blockquote> we will wrap it in
    const blockquote = document.createElement('blockquote');
    // move the instrumentation from the paragraph wrapper to the <blockquote> (if any)
    moveInstrumentation(parWrapper, blockquote);
    // replace the paragraph wrapper with the <blockquote>
    parWrapper.replaceWith(blockquote);
    // and append all quote paragraphs to the <blockquote>
    blockquote.append(...parWrapper.children);
  }
}
```

Next, paste the following code in the "quote.css". It implements some basic style for the quote. Save the file. Your browser should apply the styles instantly, and you should see your quote with a light grey background and some updated paddings.

quote.css

```
.block.quote {
    background-color: #ccc;
    padding: 0 0 24px;
    display: flex;
    flex-direction: column;
    margin: 1rem 0;
}

.block.quote blockquote {
    margin: 16px;
    text-indent: 0;
}

.block.quote > div:last-child > div {
    margin: 0 16px;
    font-size: small;
    font-style: italic;
    position: relative;
}

.block.quote > div:last-child > div::after {
    content: "";
    display: block;
    position: absolute;
    left: 0;
    bottom: -8px;
    height: 5px;
    width: 30px;
    background-color: darkgray;
}
```

To recap, the section should look like that in your AEM simulator running on https://localhost:3000.



As the last step, add your files to git, commit and push them. Afterwards your changes should be reflected on your project's live URL – you may have to hard reload the page.

```
git add .
git commit -m "feat: implement quote"
git push
```

Summary

What you've done in this last chapter was

- You added a component "Quote" to your project.
- configured a model to make it editable for the authors.
- and added it to the list of allowed components for any section.
- Then, you created your first block implementation – the quote.js
- and your first stylesheet – quote.css

With this, you've done all basic configuration and development for an Edge Delivery Services project with AEM CS authoring.

Lesson 3 –Advanced Use Cases for AEM authoring with Edge Delivery Services

Objectives

1. Repeat and strengthen what we learned in lesson 2.
2. Learn how to use indexes to build dynamic components.
3. Learn how to work with table-like data and the "Spreadsheet" editor.
4. Learn how AEM and Edge Delivery Services work together seamlessly.

In this lesson we will create a page with details about the author of the quote we previously created. We will index all authors to create a listing page for them as well and we will make sure that all that works well in the context of the Universal Editor.

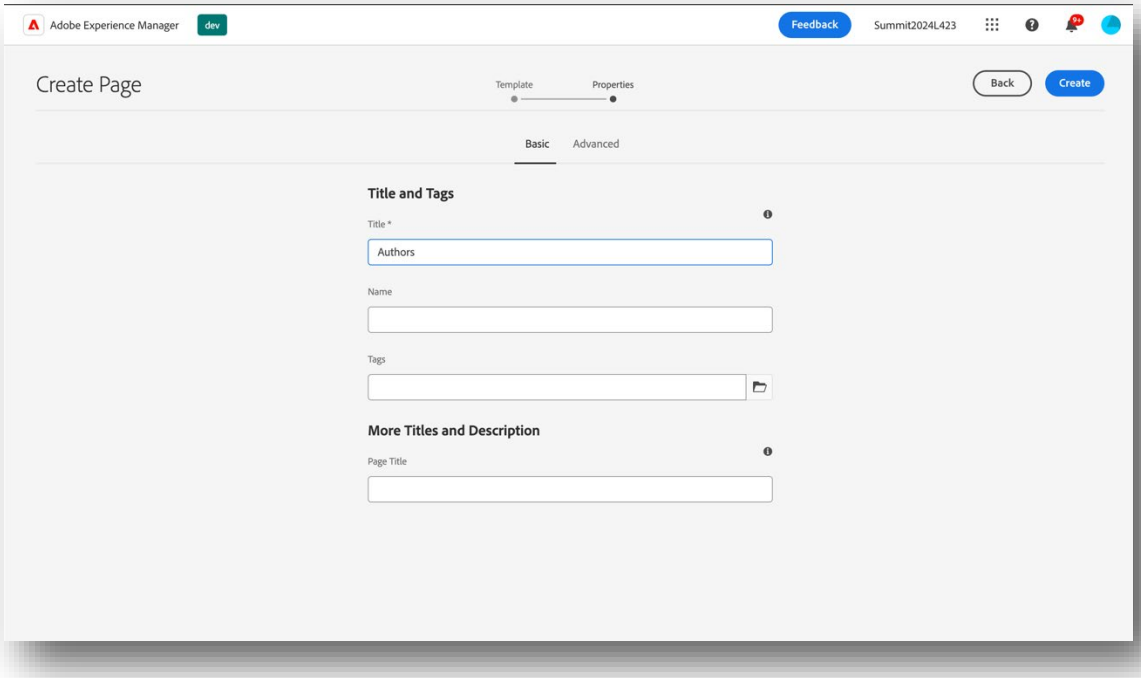
This is an advanced exercise – prerequisite is understanding the previous lessons. Many steps will be described in less detail than before.

You can find all the files created / modified in this lesson in on your lab machine. Navigate to the folder on your desktop and open the L423 folder, lesson 3.

Creating Content for our Famous Quotees

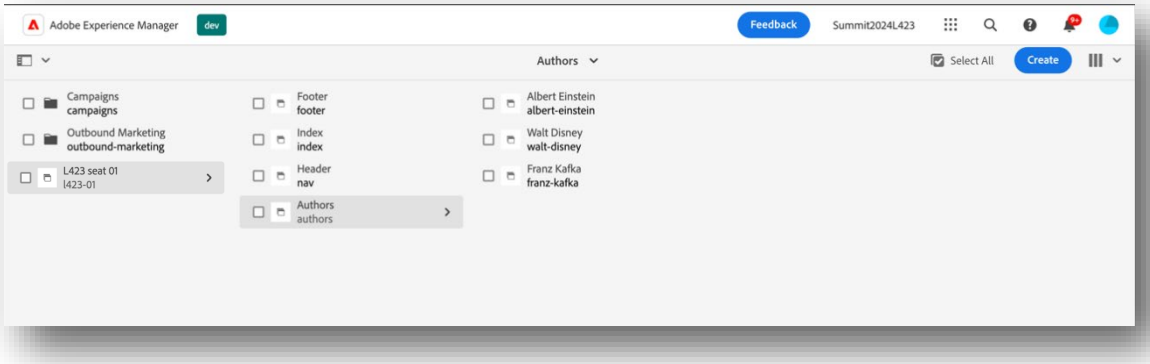
Return to AEM Sites and navigate to the site you created. Create a new page called "Authors" as sibling of the "Index" page we previously created.

- Click on the "Create" button and select "Page".
- In the Wizard, select again "Page" and click "Next".
- Type "Authors" in the "Title" field and click "Create".
- Close the confirmation prompt by clicking "Done" and you will return to the site admin user interface.



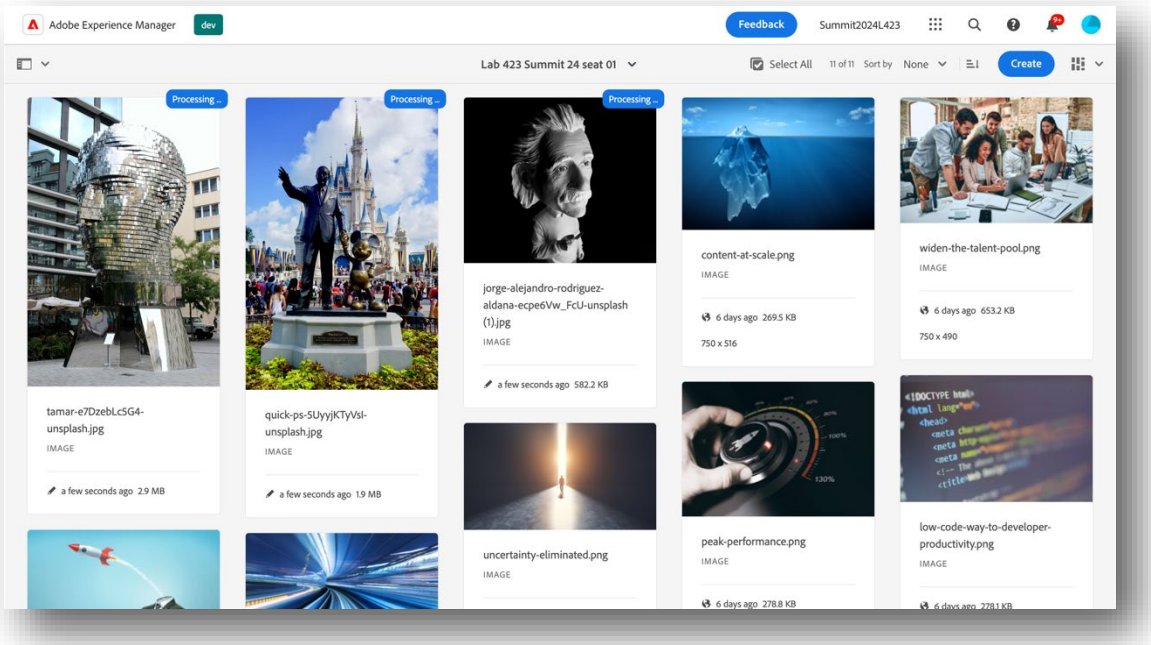
Repeat the same to create 3 pages for 3 quotees of your choice as children of your “Authors” page. Click on the authors page you just created, then hit create, and create 3 pages e.g.

- Albert Einstein
- Walt Disney
- Franz Kafka



Before we start adding content to these pages, let's upload some images to AEM Assets for later usage in this exercise. You can find 3 images free to use without attribution in the Lesson 3 lab folder.

Click on "Adobe Experience Manager" top-left. A menu opens, navigate to Assets, then Files. On the next page you will find a folder for assets of your site – e.g. "l423-01", navigate to it by clicking on it. Now drag and drop the 3 assets from your lab machine – the three .jpg files, into the Assets folder you opened. The assets will upload and start being processed by AEM Assets.

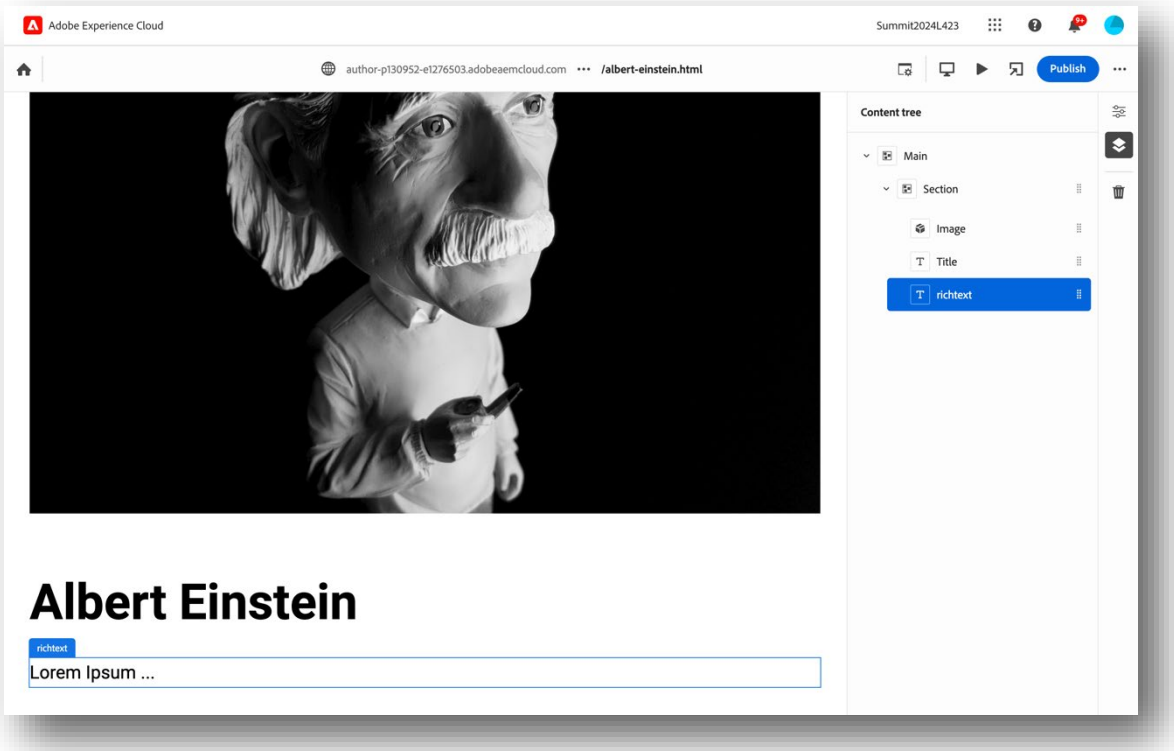


Even though not required, credit for these images goes to

Albert Einstein Photo by Jorge Alejandro Rodríguez Aldana on Unsplash
Walt Disney Photo by Quick PS on Unsplash
Franz Kafka Photo by Tamar on Unsplash

Return to AEM Sites. Click again on “Adobe Experience Manager” top-left and select Sites. Navigate to your site and open one of the pages you created inside “Authors”, e.g. Albert Einstein “checkbox the page and hit “Edit (e)”. A blank page will open in Universal Editor.

Repeat what we learned in Lesson 1 and create an Image, a Title, and a Text component on the page. Fill all of them with some content. Repeat the steps for all of the 3 author pages.

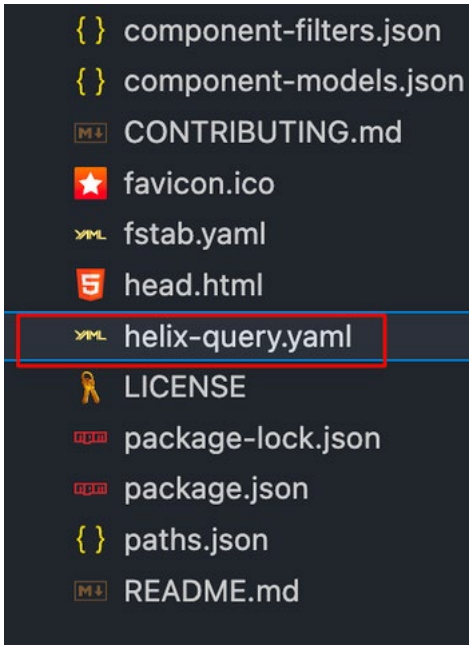


Indexing the Quotees

To access the content we created for our quotees in different places of our site, we have to make them accessible as an API we can consume. Edge Delivery Services provides an indexing service that we can use for this. More details can be found in the Indexing documentation.

Other than for document-based authoring, we do not need to create a Spreadsheet to get started. Instead we will create and update the “helix-query.yaml” file directly in our GitHub repository.

Return to Visual Studio Code and create a new File named “helix-query.yaml” in the root of your project,



paste the content below into the file and save it.

helix-query.yaml

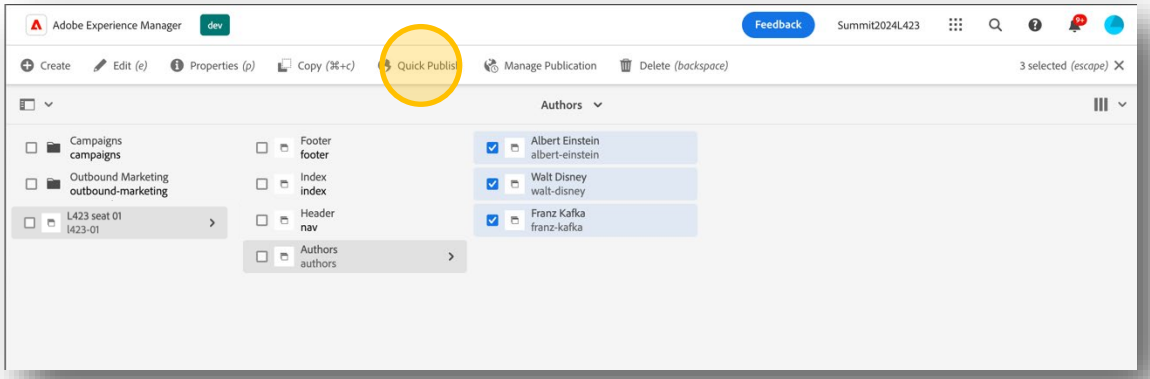
```
version: 1
indices:
  authors:
    include:
      - /authors/*
    target: /authors.json
    properties:
      lastModified:
        select: none
        value: parseTimestamp(headers["last-modified"], "ddd, DD MMM YYYY hh:mm:ss GMT")
      title:
        select: main h1
        value: textContent(el)
      image:
        select: head > meta[property="og:image"]
        value: attribute(el, "content")
```

Now add the index definition to git, commit and push it. Go to the Terminal and run the following commands.

```
git add .
git commit -m "feat: add authors index"
git push
```

Afterwards, every descendant page of your authors page will be indexed in a JSON file on the edge.

Return to AEM Sites and publish the 3 author pages you created. Select them and click “Quick Publish” in the toolbar.



It will take a few seconds until the pages are published and indexed. If everything went well, you can open the index in your browser and you will see the 3 pages with the path, title, image and last modified date extracted.

Open you preview URL for the */authors.json* index, e.g. https://main--l423-01--schaulinsan.hlx.page/authors.json.

Linking the Quote to the Quotee

Now that we have our quotees indexed, lets link our quote to the quotee in the quote block.

Return to Visual Studio Code and open the previously created "quote.js" file – under "blocks", then "quote". Replace its content with the following code and save the file.

quote.js

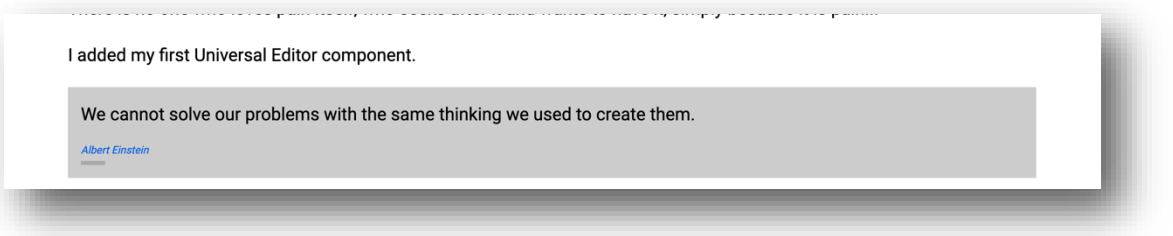
```
import { moveInstrumentation } from '../../scripts/scripts.js';

export default async function decorate(block) {
  const [quoteWrapper, authorWrapper] = block.children;

  // get the paragraph and its parent
  let par = quoteWrapper.querySelector('p');
  if (par) {
    const parWrapper = par.parentElement;
    // create a new <blockquote> we will wrap it in
    const blockquote = document.createElement('blockquote');
    // move the instrumentation from the paragraph wrapper to the <blockquote> (if any)
    moveInstrumentation(parWrapper, blockquote);
    // replace the paragraph wrapper with the <blockquote>
    parWrapper.replaceWith(blockquote);
    // and append all quote paragraphs to the <blockquote>
    blockquote.append(...parWrapper.children);
  }

  // request the /authors.json index
  par = authorWrapper.querySelector('p');
  let quotee = par.textContent.trim();
  let quotees = await fetch('/authors.json');
  if (quotees.ok) {
    // get the returned json as javascript object
    quotees = await quotees.json();
    // find the first entry that's title matches our quotee
    quotee = quotees.data.find(({ title }) => title.toLowerCase() === quotee.toLowerCase());
    if (quotee) {
      const { path } = quotee;
      const a = document.createElement('a');
```

```
      a.href = path;
      a.append(...par.childNodes);
      moveInstrumentation(par, a);
      par.replaceChildren(a);
    }
  }
}
```

Once saved, you will see the AEM Simulator that is still running in the background reload your page, and if the index was successfully created before, the author text will now link to the quotee's detail page. Return to the terminal, commit, and push your changes.

```
git commit -a -m "feat: link quote to quotee"
git push
```

Now open your projects preview or live URL to inspect the homepage with your quote. It will show a link if the authored quote exists in the index.



Now return to AEM Sites and open the Index page for editing again. You will notice that the quote is not linked to the quotee there.

Note: Other than for the preview and live URL, for authoring the AEM author environment renders the page that is opened in Universal Editor. In our case this page has the path /content/<site>/index relative to the author domain author-pXYZ-eXYZ.adobeaemcloud.com. In our "quote.js", we fetch */authors.json* relative to the current domain. That means that we actually try to fetch https://author-pXYZ-eXYZ.adobeaemcloud.com/authors.json, which does not exist as the index is only available on the edge.

This is the case for all resources loaded from Edge Delivery Services into the Universal Editor, including scripts, styles, indexes, fonts, etc. To solve this there are two things involved:

1) A reverse proxy in AEM that forwards requests to resources to Edge Delivery Services
2) A service worker installed on the authorable site that applies the mappings configured in the "paths.json" when requesting domain relative resources.

To make that service worker forward the request to */authors.json* index to the reverse proxy, we have to add a single, so called resource-only mapping rule to the "paths.json" file. Open it in Visual Studio Code, update the mappings as shown below.

paths.json

```
{
  "mappings": [
    "/content/l423-01/:/",
    "/content/l423-01.resource/authors.json:/authors.json"
  ]
}
```

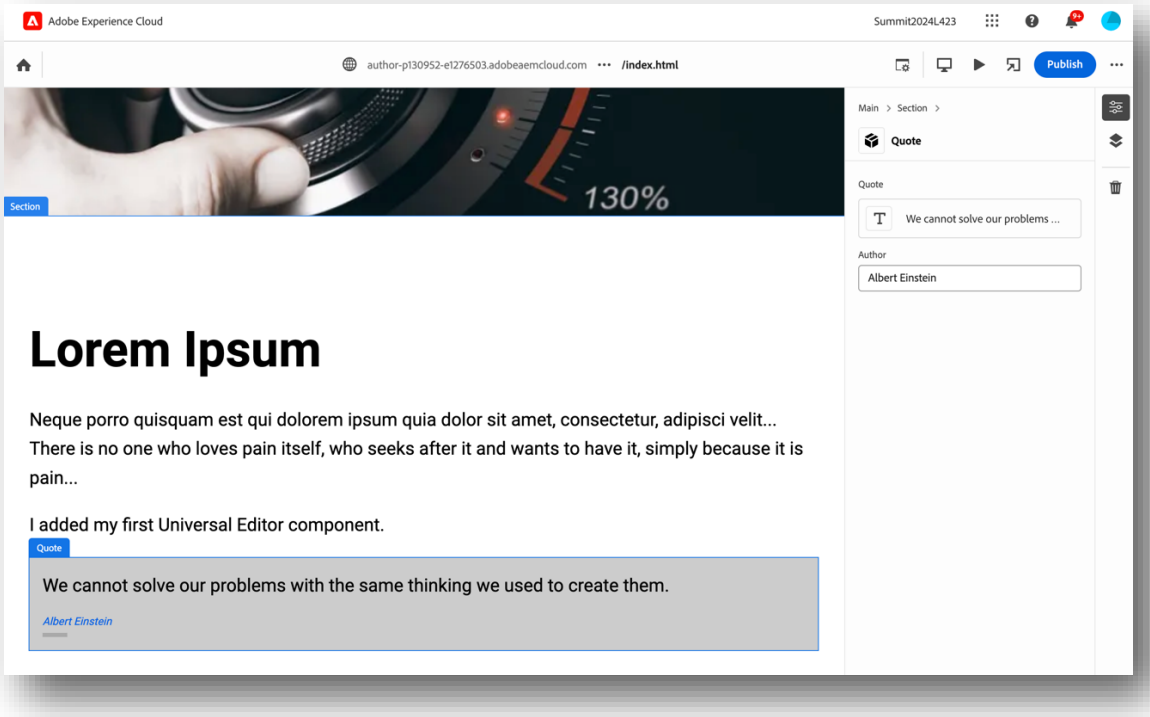
Usually the mappings configured in the “paths.json” file are reversible and do not require repository access, meaning they can be applied to map a path in AEM to a path in Edge Delivery Services and vice versa. This, however, configures a mapping that is only applied when resolving an Edge Delivery Services path in AEM.

The order of the mappings is also important, as they are evaluated bottom-up, meaning they should be ordered from least significant to most significant, and hence we added our mapping to the end of the list.

Return to the Terminal, commit, and push the changes to GitHub.

```
git commit -a -m "fix: add mapping for /authors.json"
git push
```

Reload the Universal Editor page and the quote will link the quotee just fine.



Create the Authors Listing Page

Now that we have our quotes linked to the quotee, we should create a Listing block that allows us to list all our quotees on the "Authors" page. The listing block should be reusable in the sense that the author can configure the index to use.

Return to Visual Studio Code and edit the "component-definitions.json", "component-models.json" and "component-filters.json". Add the shown below

component-definitions.json (line 156ff)

```
{
   "title": "Listing",
   "id": "listing",
   "plugins": {
     "xwalk": {
       "page": {
         "resourceType": "core/franklin/components/block/v1/block",
          "template": {
            "name": "Listing",
            "model": "listing"
          }
       }
     }
   }
}
```

component-models.json (line 208ff)

```
{
  "id": "listing",
  "fields": [
    {
      "component": "text",
      "valueType": "string",
      "name": "index",
      "label": "Index",
      "value": ""
    }
  ]
}
```

component-filters.json (line 23-33)

```
"components": [
  "text",
  "image",
  "button",
  "title",
  "hero",
  "cards",
  "columns",
  "quote",
  "listing"
]
```

Next, create a new folder “listing” in the “blocks” folder, and inside of it a file “listing.js” and “listing.css”. Paste the following file content into these 2 files.

listing.js

```
import { createOptimizedPicture } from '../../scripts/aem.js';

export default async function decorate(block) {
  const link = block.querySelector('a');
  if (link) {
    link.remove();
    let index = link.href;
    const children = [];
    if (index) {
      index = await fetch(index);
      if (index.ok) {
        index = await index.json();
        const lis = index.data.map(({ image, title, path }) => {
          const li = document.createElement('li');
          const picture = createOptimizedPicture(image);
          const a = document.createElement('a');
          a.href = path;
          a.textContent = title;
          li.append(picture, a);
          return li;
        });
        const ul = document.createElement('ul');
        ul.append(...lis);
        children.push(ul);
      }
    }
    block.replaceChildren(...children);
  }
}
```

listing.css

```
.listing ul {
    display: flex;
    flex-flow: row wrap;
    gap: 16px;
    list-style: none;
}

.listing ul li {
    width: 256px;
    position: relative;
}

.listing picture img {
    display: block;
    width: 100%;
    height: 256px;
    object-fit: cover;
}

.listing a {
    position: absolute;
    inset: 0;
    display: flex;
    flex-direction: column-reverse;
    color: white;
    padding: 8px;
    background: linear-gradient(0deg, rgba(0 0 0 / 100%) 0%, rgba(0 0 0 / 50%) 35%, rgba(255 255
255 / 1%) 100%);
}
```

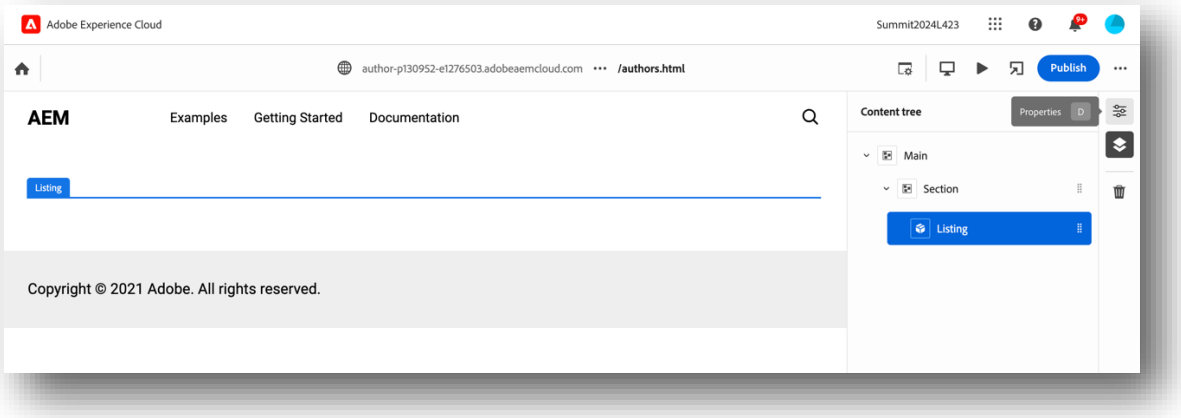
Add all the changes to git, commit them, and push them to GitHub.

```
git add .
git commit -a -m "feat: listing component"
git push
```

Now we implemented the listing component with some basic style, a component definition, and a model. We should be able to add it to our "Authors" page.

Return to AEM Sites, open the "Authors" page in Universal Editor. Select the empty section on the page, click "+" and select the "Listing" component that should be in the list of components now. This will add the Listing to the page.

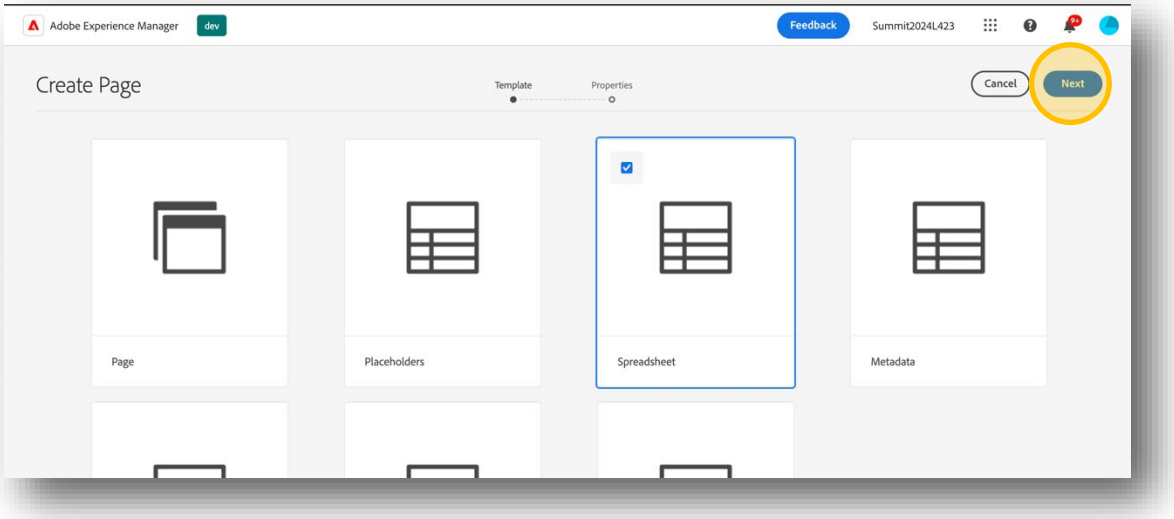
As the listing block, does not render any content if it has no index configured, we must find it in the "Content Tree", select it there and return to the Property Rail to edit it.



Paste the link to the /authors.json index in the "Index" text field. It does not matter if you use the preview or live URL of your site. Blur the focus of the field to save it. The listing block will reload, and it will show the authors indexed. That works, as we configured the reverse proxy mapping before.

Now we want to show a manual curated list of featured quotees instead. To do so we will create a spreadsheet-like page with the data we want to be listed.

Return to AEM Sites and create a new 'Spreadsheet" page as sibling of the "Authors" page.



Call it "Featured Authors" and add the following columns "path", "title", "image". Click "Create" and in the confirmation prompt click "Open". The spreadsheet editor will open.

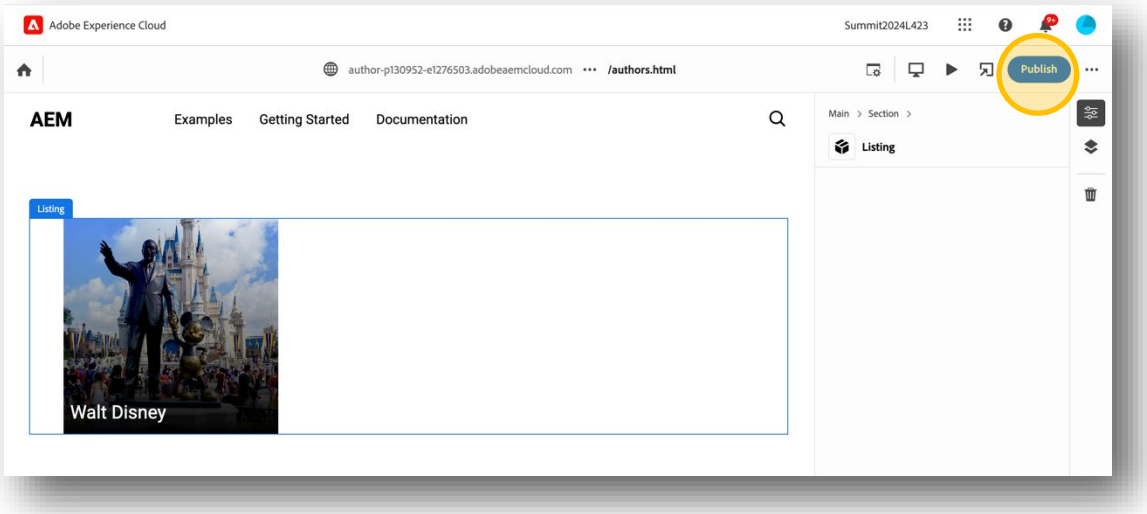
The spreadsheet editor is a light version of an editor that allows to edit tabular data. It is far from being feature complete compared to Excel or Google sheets, and it is not meant to be so. Its primary use case is to maintain the out-of-the-box spreadsheets like bulk metadata, redirects, placeholders, etc., but it also allows to create simple sets of data in an easy reusable way.

Add one featured author to the spreadsheet. Use the path to the page in AEM but copy the image URL from the author's detail page preview or live URL.

path	title	image
/content/lab423-summit-01-site/authors/albert-einstein	Albert Einstein	https://main--lab423-summit-01--schaulinsan.hlx.live/authors/media_1018c965b56036abc18cce0cc8d77ba9eb774de58.jpeg?width=1200&format=pjpg&optimize=medium

From the browser's location bar, copy the path of the spreadsheet, e.g. */content/l423-01/featured-authors*.

Now open the "Authors" page again in Universal Editor and update the Listing component to use the Spreadsheet we just created. Paste the Spreadsheet path into the "index" text field and append a .json extension. Blur the text field and the Listing block saves and reloads.



Now publish the Page and return to AEM Sites.

In order to also publish the "Featured Authors" spreadsheet we need to create another mapping in our "paths.json" file

---

The service that integrates AEM CS with Edge Delivery Services cannot differentiate between a page and a spreadsheet .This is an implementation detail that is going to change soon.

Return to Visual Studio Code and open the "paths.json" file. Update it with the below content.

paths.json

```
{
  "mappings": [
    "/content/l423-01/:/",
    "/content/l423-01/featured-authors.json:/featured-authors.json",
    "/content/l423-01.resource/authors.json:/authors.json"
  ]
}
```

Save the file, return to the Terminal, commit, and push the changes to GitHub.

```
git add .
git commit -a -m "feat: add featured-authors mapping"
git push
```

Now return to AEM Sites and publish the featured authors Spreadsheet.

Open the "Authors" page with the preview URL of your site and it will show the featured author from the spreadsheet we created.

Summary

What you've done in this last chapter was

- You added more content to have detail pages for authors
- You added a listing block to the authors page to list authors
- You configured an index of all authors for this listing page
- You created a manually curated spreadsheet for the listing
- You used either to customize the listing page experience

This was an advanced exercise to illustrate that AEM authoring with Edge Delivery Services supports the same features as Edge Delivery Services with document-based authoring, just in a slightly different way. In general, we aim for a feature parity, and support all features everywhere, no matter the authoring experience preferred by the user.

Lesson 4 – The anatomy of blocks of AEM authoring with Edge Delivery Services

Objectives

1. Understand how AEM renders blocks
2. Understand how blocks are instrumented for Universal Editor
3. Understand which conventions are available to create compelling content models

This lesson is meant to introduce some conventions to enable developers to create compelling content models that work well with AEM authoring and Edge Delivery Serivices. Content models are agnostic to the authoring experience, so the way we structure content should be natural to an author, no matter if he entered the content in AEM, a word document or any other bring-your-own content source.

David's model is a good starting point to learn about the best practices when creating content models for Edge Delivery Services, and the Block Collection gives an wide range of examples. While it seems to related to document-based authoring only at first, it applies for any other content source as well. Word or other document-based authoring interfaces are the baseline for what content model is natural for or intuitive to an author.

Markup and DOM

For Edge Delivery Services the markup and DOM play a dominant role in the developer experience. The DOM is the standard API frontend developers implement against. For Edge Delivery Services projects created from the aem-boilerplate or the aem-boilerplate-xwalk this is the DOM **after** it has been decorated by the aem.js library. The markup on the other hand is rather an internal contract between the helix-html-pipeline and the aem.js library. It is a stable, almost one-to-one representation of that content an author created.

Usually developers do not need to care much about the markup, but to understand how AEM implements the what-you-see-is-what-you-get authoring experiences it is important to understand that when editing a page in Universal Editor the markup served to the browser is rendered by AEM CS, not Edge Delivery Services, but as both implement the same markup loading the scripts from the Edge Delivery Services project onto the page served from AEM works fine.

In general, there are two types of content that can be put in the sections of a page: default content like images, texts, titles, buttons and blocks. The former are straight forward to understand as they exist in the HTML specification as semantic elements. Blocks however are specific to Edge Delivery Services. In document-based authoring they are implicit / free-form structures defined by an author as tables. In AEM authoring, they must be modelled explicitly to map a logical content model (sets of properties) to a physical content model (rendered table-like structures). In the following sections we will learn what kind of table-like structures AEM can render and what use cases they are meant to be used for.

We will do so by building another block – a teaser.

Simple Blocks

Simple blocks are the simplest form of blocks. They render as a table with single column per row, where each row is a property value as defined in the block's model. The order of the properties is respected, and empty values render as empty cells. That makes it very easy to destructure such a block client side.

Return to Visual Studio Code, create a new component definition for the Teaser block and a component model for it. In the first step we will add an image, pre-title, title, description to it.

component-models.json

```
{
  "id": "teaser",
  "fields": [
    {
      "component": "reference",
      "valueType": "string",
      "name": "image",
      "label": "Image",
      "multi": false
    },
    {
      "component": "text",
      "valueType": "string",
      "name": "pretitle",
      "label": "Pre Title",
      "multi": false
    },
    {
      "component": "text",
      "valueType": "string",
      "name": "title",
      "label": "Title",
      "multi": false
    },
    {
      "component": "richtext",
      "valueType": "string",
      "name": "description",
      "label": "Description",
      "multi": false
    }
  ]
}
```

Commit and push the changes to your GitHub project. When you return to AEM Sites, create a new page. Create a new teaser on the page, add an image, some pre-title, title and description. You will see the data being rendered on the page, again without any decoration or styles applied. However, each of the fields is also editable in-context by selecting them on the page. This is possible because the markup of the block is instrumented for Universal Editor. Disable the editables / switch to preview mode in Universal Editor and inspect the page. Select the teaser block and have a look at the DOM. You will see the data-aue-* attributes that are used by the Universal Editor.

When developing for AEM authoring with Edge Delivery Services, developers do not need to write the instrumentation, but must make sure to not remove it accidentally, if they want a block to remain in-context editable.

Teaser DOM

```
<div data-aue-resource="urn:aemconnection:/content/lab423-summit-01-
site/teaser/jcr:content/root/section/block" data-aue-type="component" data-aue-
behavior="component" data-aue-model="teaser" data-aue-label="Teaser" class="teaser block"
data-block-name="teaser" data-block-status="loaded">
  <div>
    <div>
      <p>
        <picture>
          <img src="/content/dam/lab423-summit-01-site/hero.png" data-aue-prop="image"
data-aue-label="image" data-aue-type="media">
        </picture>
      </p>
    </div>
  </div>
  <div>
    <div>
      <p data-aue-prop="pretitle" data-aue-label="pretitle" data-aue-type="text">pre
title</p>
    </div>
  </div>
  <div>
    <div>
      <p data-aue-prop="title" data-aue-label="title" data-aue-type="text">title</p>
    </div>
  </div>
  <div>
    <div>
      <div data-aue-prop="description" data-aue-label="description" data-aue-filter="text"
data-aue-type="richtext">
        <p>Description Paragraph 1</p>
        <p>Description Paragraph 2</p>
      </div>
    </div>
  </div>
</div>
```

Publish the Page, and open and inspect it on the preview URL as well. You will mention that the description is with an additional * * on AEM. This is required to make the richtext work with the paragraphs. The same behavior can be observed for the Text component as well. When decorating blocks, so that they remain in-context editable it is important to keep this special-case in mind. The best practice is to handle cells that contain rich text, and not destructure or remove individual elements from them. If you do so anyway, make sure to remove the instrumentation as well. In general, it is not recommended to destructure / parse a block and replace it with an entirely new DOM, unless for example the block contains some configuration and loads the data it renders from a third-party source.

Key-Value Blocks

For such use cases the block can be made a key-value block, which can be parsed using the *readBlockConfig()* utility from the aem.js library.

Return to Visual Studio Code and open the "components-definitions.json". To the template of the teaser component, add `"key-value": true`. Commit and push the changes to GitHub. Every teaser block will now be a key-value block.

component-definitions.json

```
{
  "title": "Teaser",
  "id": "teaser",
  "plugins": {
    "xwalk": {
      "page": {
        "resourceType": "core/franklin/components/block/v1/block",
        "template": {
          "name": "Teaser",
          "model": "teaser",
          "key-value": true
        }
      }
    }
  }
}
```

Return to AEM and create another Teaser on the previously created page. Add some data and inspect the DOM again. You will notice that now, each row of the block has two cells. The first cell is the property name and the second cell the property value. However, as this format is meant to be parsed and the block being repalced with some dynamic content, the individual cells are not instrumented for in-context authoring.

Remove the teaser from the page, as the key-value format does not fit our use case. Remove the `key-value` attriubte from the component defition as well. Commit and push to GitHub.

Content Modelling

Lets recap the Teaser DOM we looked at earlier. There are a two details that are semantically not correct. First the title is not a heading and second the pretitle, title, description grouped together should make the foreground. To achieve that we will make use of two patterns that help developers to create semanticly content.

Type Inference

Type inference is about detecting the semantic of some value and rendering it accordingly. The following rules implement a similar behavior as document-based authoring:

- If a value starts with a slash and resolves to a resource it is considered a reference
  - ⇒ if the reference is an Asset, render an <img>
  - ⇒ if not render a domain relative link <a>
- If a value starts with http://, https:// or # render a link <a>
- If a value starts with a block element (<p>, <ul>, …) render a richtext
- If a value is an array of texts, concatenate them with a comma

For links and images this raises the question, how additional attributes of these can be authored. For that a naming convention is inplace, to tread any property ending case-sensitive with either Text, Type, Title or Alt as attribute of the respecitve property. More details can be found in the Content Modelling documentation.

Return to Visual Studio Code and add a titleType, imageAlt, cta and ctaText to the teaser model.

component-models.json

```
[
  {
    "component": "reference",
    "valueType": "string",
    "name": "image",
    "label": "Image",
    "multi": false
  },
  {
    "component": "text",
    "valueType": "string",
    "name": "imageAlt",
    "label": "Image Alt",
    "multi": false
  },
  {
    "component": "text",
    "valueType": "string",
    "name": "pretitle",
    "label": "Pre Title",
    "multi": false
  },
  {
    "component": "text",
    "valueType": "string",
    "name": "title",
    "label": "Title",
    "multi": false
  },
  {
    "component": "select",
    "name": "titleType",
    "value": "h1",
    "label": "Title Type",
    "valueType": "string",
    "options": [
      {
        "name": "h2",
        "value": "h2"
      },
      {
        "name": "h3",
        "value": "h3"
      }
    ]
  },
  {
    "component": "richtext",
    "valueType": "string",
    "name": "description",
    "label": "Description",
    "multi": false
  },
  {
    "component": "text",
    "valueType": "string",
    "name": "cta",
    "label": "Link",
    "multi": false
  },
  {
    "component": "text",
    "valueType": "string",
    "name": "ctaText",
    "label": "Link Text",
```

```
    "multi": false
  }
]
```

Return to Universal Editor and edit the previously created teaser block. The new fields should appear. Select a heading type, add a image description and add a link. The block will reload. Inspect the DOM again to see that the title is now rendered as heading, the image has an alternative text and a link is added as new row at the end of the block.

Element Grouping

To group multiple editable prefixing their names with a a group name seprated from the property name by an underscrore.
Return to Visual Studio Code and rename the properties of the teaser model:

- `image` => `background_image`
- `imageAlt` => `background_imageAlt`
- `pretitle` => `foreground_pretitle`
- `title` => `foreground_title`
- `titleType` => `foreground_titleType`
- `description` => `foreground_descriptionType`
- `cta` => `foreground_cta`
- `ctaText` => `forground_ctaText`

Also add a new property to the background to hold a video link and name it `background_video`. Commit and push your changes.

Return to the Univeral Editor, delete the old Teaser, reload and create a new Teaser.

There is an example video and thumbnail in the lesson 4 folder for the lab. Upload it to DAM and publish it.

Keep in mind that Edge Delivery Services only supports videos with up to 300 KB/s bitrate. More details about the limits can be found in the Limits documentation.

Once done inspect the DOM. You will mention a few differences compared to before:

- the block has now 2 rows with one column each
- the content of the properties prefixed with background remain in the first row,
- the content of the properties prefixed with foreground in the second one
- all elements are still instrumented, the description is again wrapped in an additional * *

The content model is now as simple as it would have been created by an author with document-based authoring.

Besides that, the content model can now easily be transposed and used in a container block.

Container Blocks

Now that we understood how to create and model blocks for a single user interface element, we want to explore, how our teaser could be used in a teaser collection, like a carousel or grid.

Container blocks accept exactly one level of children, meaning we can define a Teaser Collection block and add teasers to it. We already know that the single dimension of simple blocks is realised by rendering a single row with a single column for each semantic element of the block or a group of elements. In container container blocks each child is rendered as a row and reach semantic element or group of elements as a column in the row. It is however still possible to add properties to a container block. Those will be rendered the same way as for simple blocks preceding the rows of the children added to the block.

There are a few things that are important to keep in mind:

- the model of all children must have same number of elements / groups of elements, so that the block has a consistent number of columns
- keep the number of columns limitted; make use of element grouping to achieve that
- if the children have block options (classes), make them render in the first column in each row as a comma separated list. Use auto blocking to move the class names to the row's <*div*>

Return to Visual Studio Code, and

- add a component model "teaser-collection" with a single select box named "classes" and the options, "List", "Grid", "Carousel"
- add a component filter "teaser-collection" and add the "teaser" to the list of allowed components.
- add the "teaser-collection" to the allowed components for the section
- add a component definition "teaser-collection" and add the model id and model filter to the component defition's template

component-models.json

```
{
  "id": "teaser-collection",
  "fields": [
    {
      "component": "select",
      "name": "classes",
      "value": "h1",
      "label": "Type",
      "valueType": "string",
      "options": [
        {
          "name": "List",
          "value": "list"
        },
        {
          "name": "Grid",
          "value": "grid"
        },
        {
          "name": "Carousel",
          "value": "carousel"
        }
```

```
      ]
    }
  ]
}
```

component-filters.json

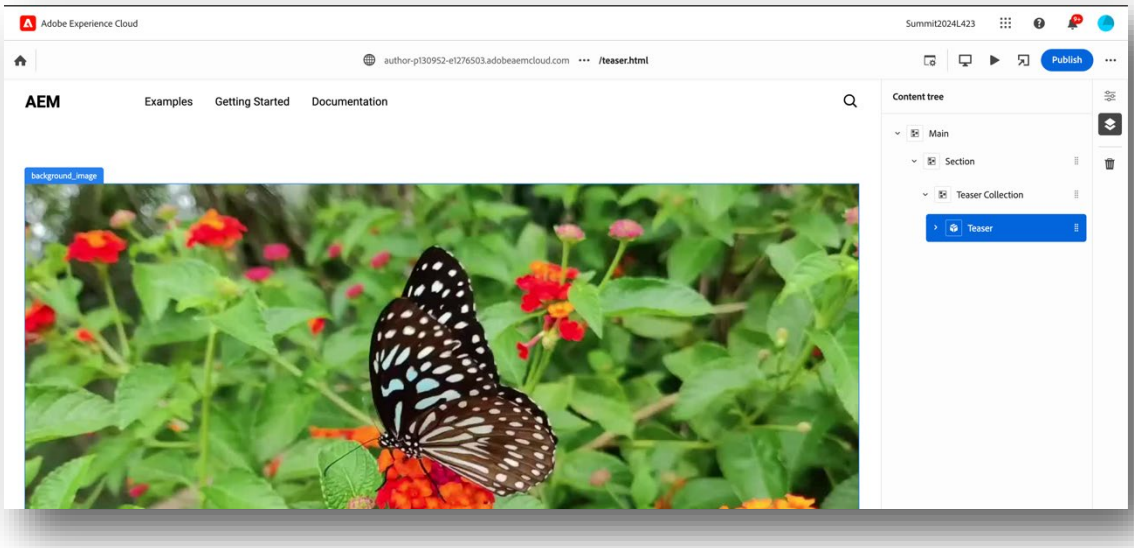
```
{
  "id": "teaser-collection",
  "components": [
    "teaser"
  ]
}
```

component-definition.json

```
{
  "title": "Teaser Collection",
  "id": "teaser-collection",
  "plugins": {
    "xwalk": {
      "page": {
        "resourceType": "core/franklin/components/block/v1/block",
        "template": {
          "name": "Teaser Collection",
          "model": "teaser-collection",
          "filter": "teaser-collection"
        }
      }
    }
  }
}
```

Commit and push those chages to GitHub.

When you now return to Universal Editor you should be able to create a teaser collection block and add teasers to it. The result should look something like this.



If you inspect the DOM again, you will see that the teaser is made our of a single row and the background and foreground group are each a column in that row.
Let's add some styles and decoration to the teaser. Publish the page and open it in the AEM simulator tab (localhost:3000). We want to implement the teaser in a reuseable way. While doing so, keep in mind to preserve the instrumentation added for Universal Editor – ideally in a way that the blocks do not have to hande this detail at all. This should be rather easy with the content model we explicitly designed for this.

teaser.js

```
export function decorateTeaser(teaser) {
  teaser.classList.add('teaser');
  const [background, foreground] = [...teaser.children];

  background.className = 'background';
  // remove the video link and its wrapper
  const videoLink = background.querySelector('a');
  if (videoLink) {
    videoLink.closest('p').remove();
    // TOOD: play video after a timeout when hovering the teaser
  }
  // remove the wrapper around the picture, if any
  const backgroundImg = background.querySelector('picture');
  if (backgroundImg) {
    const backgroundImgPar = backgroundImg.closest('p');
    if (backgroundImgPar) {
      backgroundImgPar.insertAdjacentElement('afterend', backgroundImg);
      backgroundImgPar.remove();
    }
  }
  // add some classes for the pretitle and title to ease styling
  foreground.className = 'foreground';
  const heading = foreground.querySelector('h2,h3');
  if (heading) {
    const pretitle = heading.previousElementSibling;
    if (pretitle) pretitle.className = 'pretitle';
  }
}

export default function decorate(block) {
  const [background, foreground] = [...block.children].map((row) => row.firstElementChild);
  // remove the ancestor <div> and make background and foreground siblings
  background.parentElement.replaceWith(background);
  foreground.parentElement.replaceWith(foreground);
  decorateTeaser(block);
}
```

teaser.css

```
.teaser {
  display: grid;
  position: relative;
}

.teaser > * {
  grid-row: 1;
  grid-column: 1;
}

.teaser .background p,
.teaser .background img {
  margin: 0;
  padding: 0;
  display: block;
}
```

```
.teaser .background img {
  object-fit: cover;
  height: 100%;
  width: 100%;
}

.teaser .background::before {
  display: block;
  position: absolute;
  inset: 0;
  content: "";
  background: linear-gradient(0deg, rgba(0 0 0 / 90%) 0%, rgba(0 0 0 / 50%) 66%, rgba(255 255 255 /
10%) 100%);
}

.teaser .foreground {
  position: relative;
  align-self: self-end;
  color: white;
  padding: 0 10px;
}

.teaser .foreground a {
  margin: 0;
}

.teaser .foreground :where(p,h2,h3) {
  margin: 10px 0;
}

.teaser .foreground .pretitle {
  text-transform: uppercase;
  font-size: .75em;
}
```

teaser-collection.js

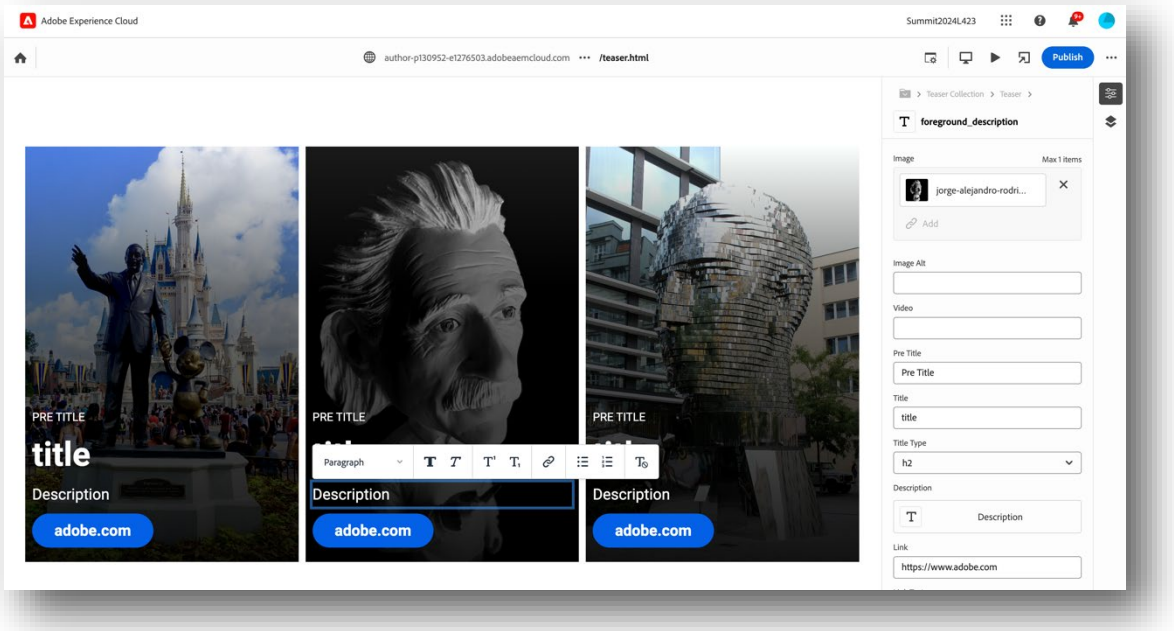
```
import { decorateTeaser } from '../teaser/teaser.js';

export default function decorate(block) {
  [...block.children].forEach(decorateTeaser);
}
```

teaser-collection.css

```
.teaser-collection {
  display: flex;
  flex-direction: row, wrap;
  gap: 10px;
}

.teaser-collection .teaser {
  max-width: calc(33.33% - 20px / 3);
}

.teaser-collection .teaser::after {
  display: block;
  grid-row: 1;
  grid-column: 1;
  content: "";
  padding-top: 100%;
}
```

Commit and push those changes to GitHub and return to AEM.

With 3 configured teaser the teaser collection looks similar as shown below, and each of the individual elements will still be in-context editable.



Summary

Well done, you implemented a semantically correct and visually compelling teaser and teaser collection. You learned:

- what different kinds of blocks AEM supports
- how to use naming conventions and type inference to create semantic HTML
- how to use naming conventions to compelling content-models, that are easy to decorate and easy to style, without any compromises on in-context editing

With that knowledge you should feel comfortable to implement Edge Delivery Services projects with AEM authoring, up to the same level of functionality as document-based authoring would support.

There are a few interesting things, that you can now explore to utilise the setup we worked with. For example, integrating your Edge Delivery Services project with AEM authoring with AEM headless. As we are already in Universal Editor, fetching, rendering and instrumenting content fragments for an mixed headful-headless in-context authoring experience is straight forward as well. Or checkout how to author AEM forms with Unviersal Editor and Edge Delivery Services.

End of document.